CSP00-5084

Department of Electronic and Electrical Engineering

Trends in Embedded Systems

Master Class

Held at Riseholm Campus,
De Montfort University, Lincoln,
10-14 July, 2000.

Professor Wayne Wolf,
Professor S-Y Kung

Department of Electrical Engineering
Princeton University
Princeton, New Jersey USA

DE MONTFORT
UNIVERSITY
LEICESTER

3790.D

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	2000	Conference Proceedings

4. TITLE AND SUBTITLE

Master Class in Current Trends in Embedded DSP Systems

5. FUNDING NUMBERS

F61775-00-WF086

6. AUTHOR(S)

Conference Committee

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

De Montfort University
The Gateway
Leicester LE1 9BH
England

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

EOARD
PSC 802 BOX 14
FPO 09499-0200

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

CSP 00-5086

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

U.S. Government Rights License
This work relates to Department of the Air Force Grant/rContract issued by the European Office Aerospace Research and Development (EOARD). The United States Government has a royalty-free license throughout the world in all copyrightable material contained herein

ꜜTRIBUTION CODE

A

13. ABSTRACT (Maximum 200 words)

The Final Proceedings for Master Class in Current Trends in Embedded DSP Systems, 7 July 2000 - 14 July 2000

The objective of the Master Class is to teach researchers of the latest advances in digital signal processing. Topics include:
a. Embedded systems, where are we now and where are we going?
b. Basic principles of embedded systems.
c. Benefits of digital signal processing (DSP) systems for multimedia applications.
d. Hardware/software co-design issues and solutions.
e. Mapping algorithms to architectures and their implementation in multimedia systems.

14. SUBJECT TERMS

EOARD, Communications, Signal Processing, radar

15. NUMBER OF PAGES

398

16. PRICE CODE

N/A

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

CONTENTS

SECTION ONE

PRESENTER : PROFESSOR WAYNE WOLF

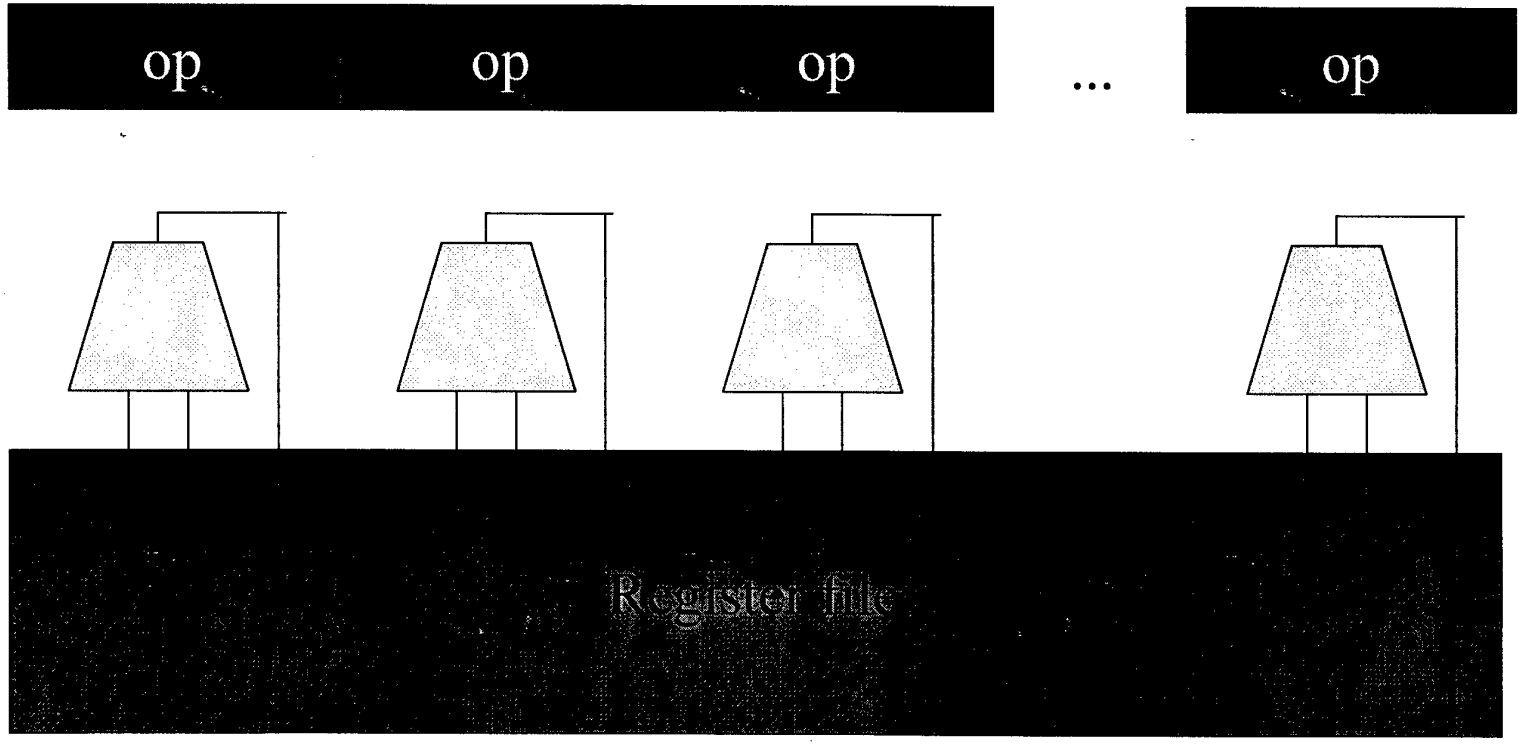
TI TMS320C6000: Overview

- VLIW basics
- C6x architecture
- C6x compilation
- C6x development environment

VLIW characteristics

- Large shared register file.
- Multiple function units.
- Statically scheduled operations.

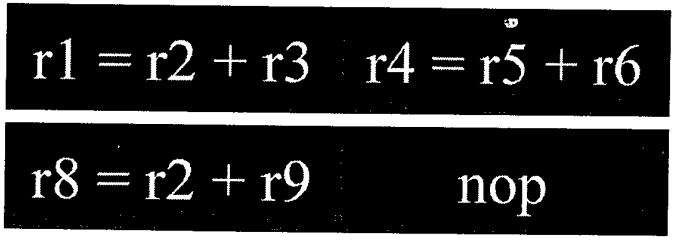
Basic VLIW architecture



VLIW operation scheduling

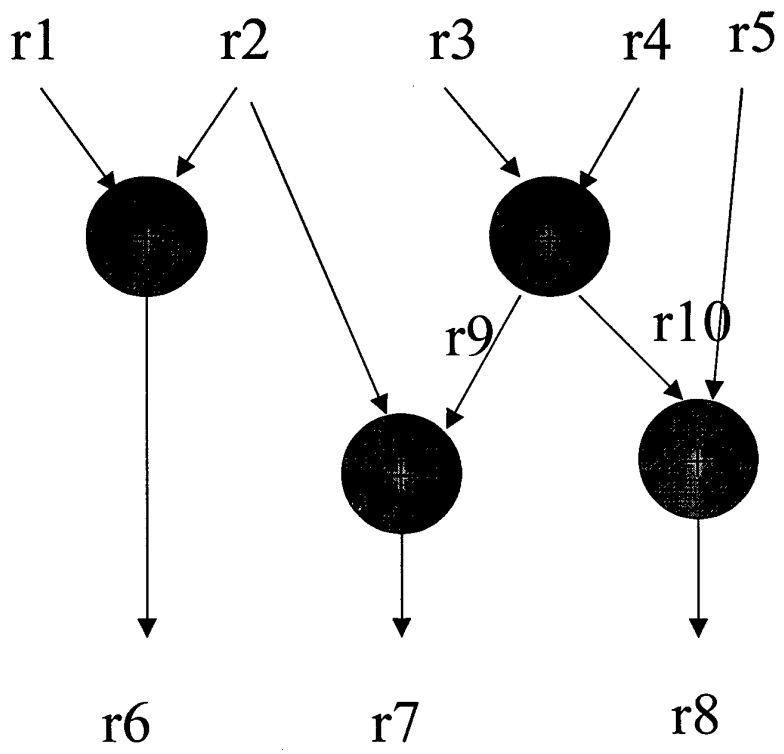
```
r1 = r2 + r3
r4 = r5 + r6
r8 = r2 + r9
```

code



VLIW instructions

Operation scheduling, cont'd



r6 = r1 + r2	nop
r9 = r3 + r4	nop
r7 = r9 + r2	r8 = r10 + r5

or

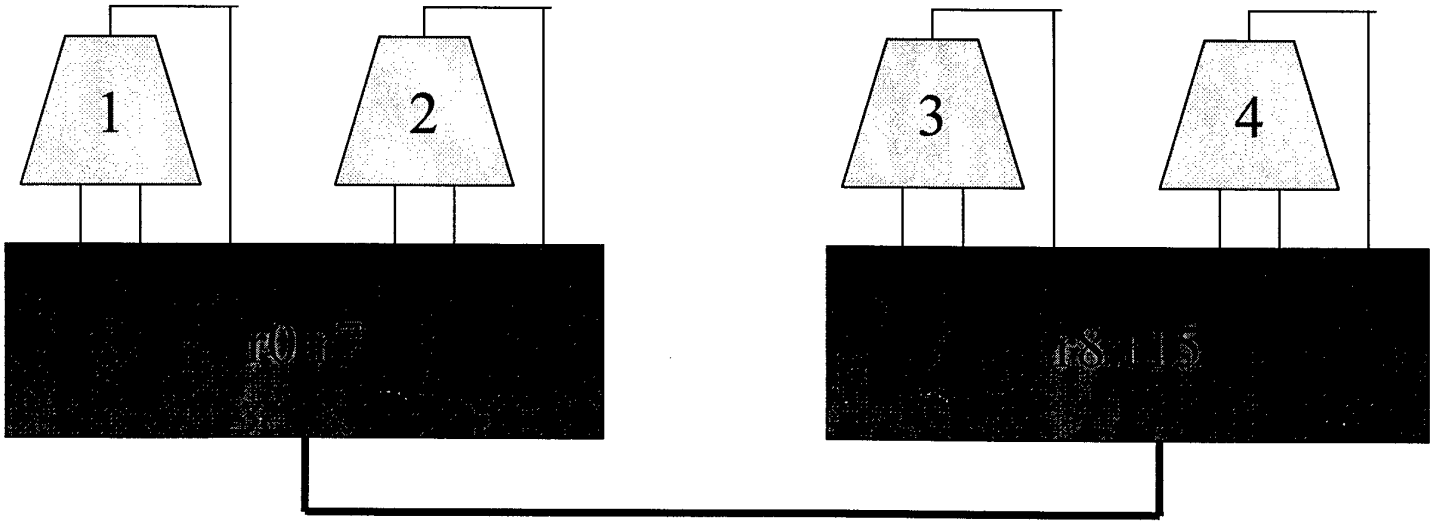
r6 = r1 + r2	r9 = r3 + r4
r7 = r9 + r2	r8 = r10 + r5

Clustered VLIW architectures

- Large register files are too slow.
  - Delay increases with number of ports.
- Functional units are divided into clusters, each sharing a register file.
- Inter-cluster operations require a data transfer between cluster register files.

Clustered operations

r1 = r2 + r3	Move r9,r5	r8 = r9 + r11	Move r0,r10
nop	r4 = r2 + r5	nop	r14 = r10 + r9



Intercluster comm network

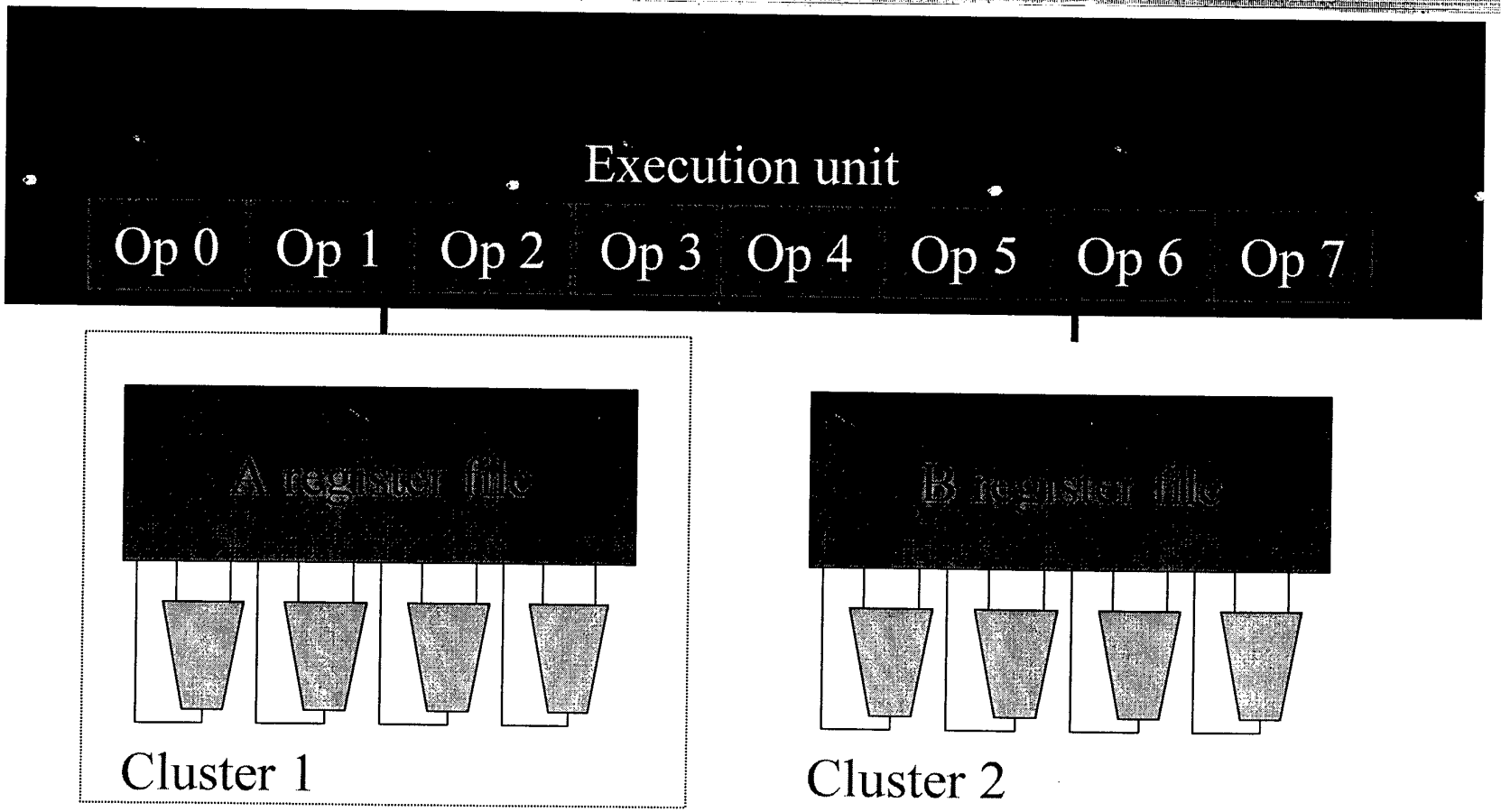
TI C6000 architecture

- VLIW DSP based on VelociTI (TM) architecture
- Varieties:
  - C62xxx, C64xxx fixed-point DSPs
  - C67xxx floating-point DSPs

C6000 CPU

- Two data paths
  - four function units per data path
- 32-bit registers (32 in C62x/C67x, 64 in C64x)
- Control registers

C6x execution unit



C6x function units

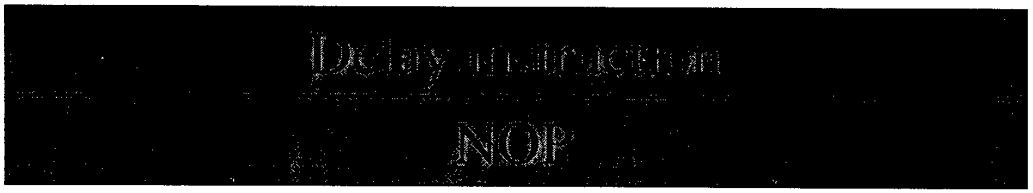
- Each cluster has four function units:
  - S: logical unit with shifter
  - L: logical unit
  - D: data unit
  - M: multiply unit

C6x pipeline

- Pipeline can dispatch eight parallel instructions per cycle.
- All instructions dispatched together travel through the pipe at the same rate.
- No pipeline interlocks.

C6X pipeline, cont'd.

- Delay slot: result is not available for the next instruction. Fill with NOPs:

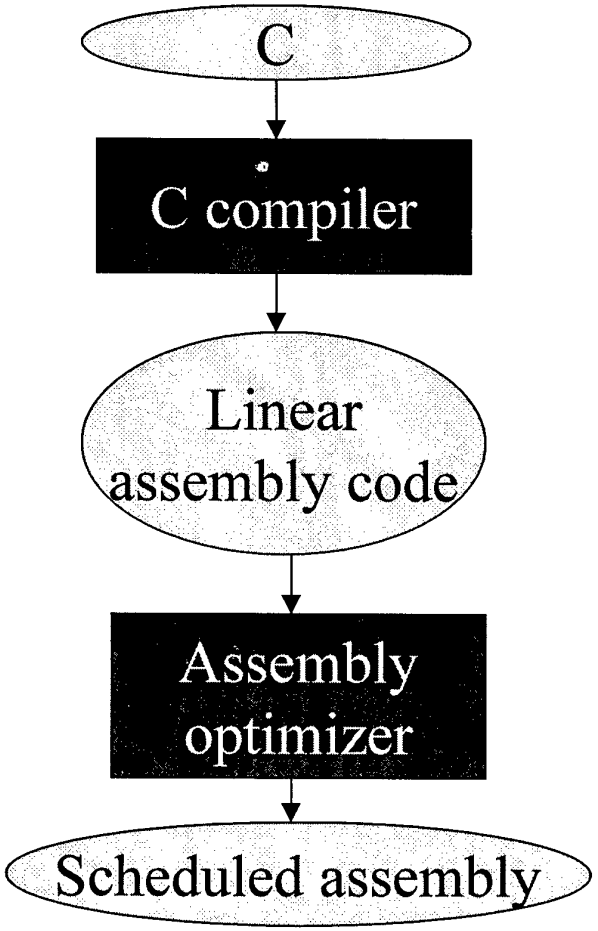


- Some instructions introduce delay slots:
  - 1 cycle: integer multiply
  - 4 cycles: load
  - 5 cycles: branch

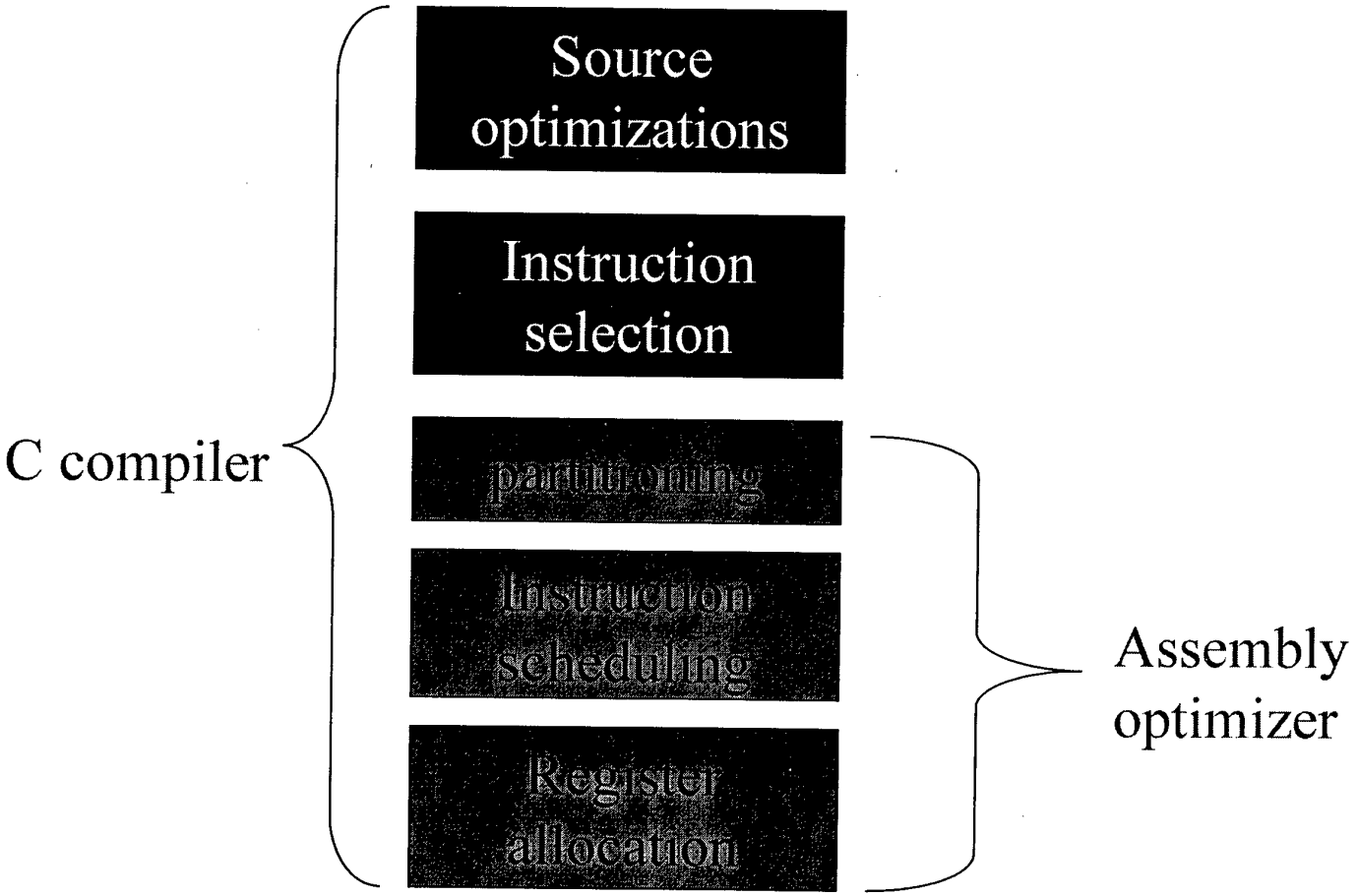
C6X conditional instructions

- All instructions are executed conditionally.
- Conditions controlled by A1, A2, B0, B1, and B2 registers.

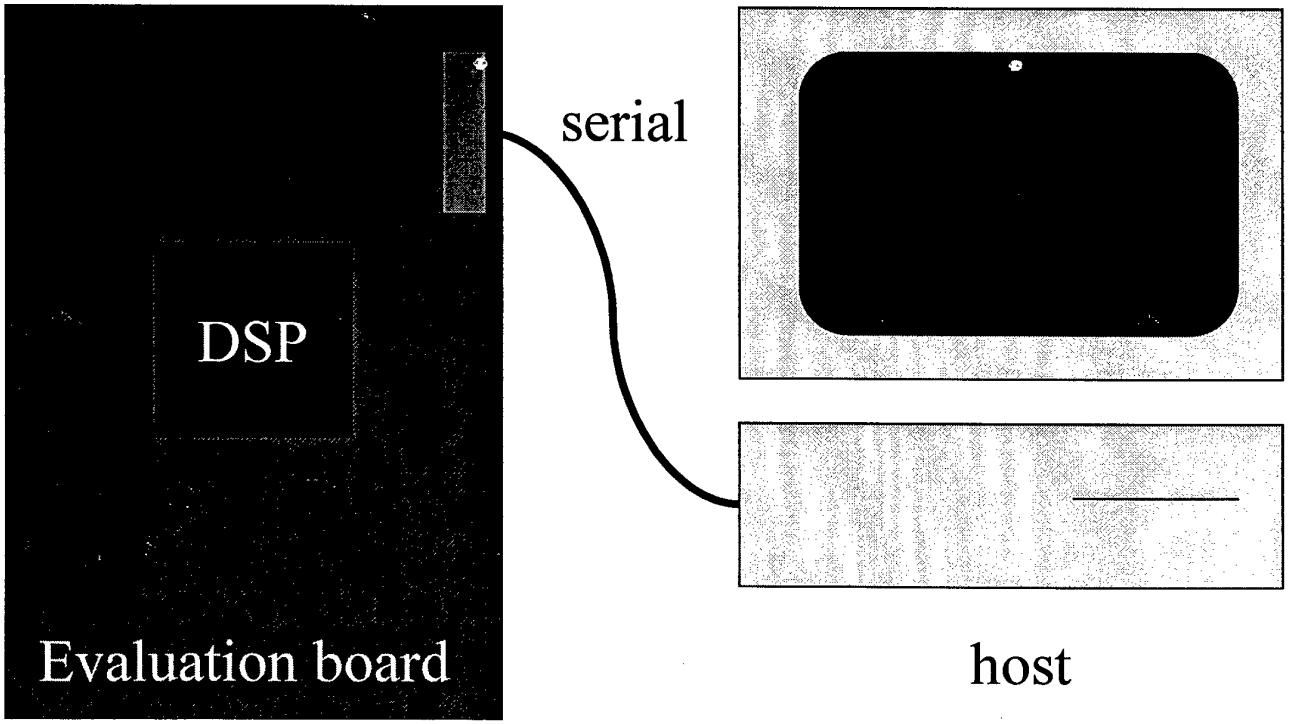
C6x compilation process



6x optimization phases



C6x development environment



Evaluation board software

- Monitor program provides several functions:
  - serial port management
  - program loading
  - program control
  - data inspection

Host software

- Compiler
- Evaluation board communication
- Debugger
  - breakpoints
  - data monitoring
  - profiling
  - timing

Debugging operations

- Breakpoints:
  - substitute instruction that returns to monitor
- Data tracing:
  - substitute instruction that calls data upload routine

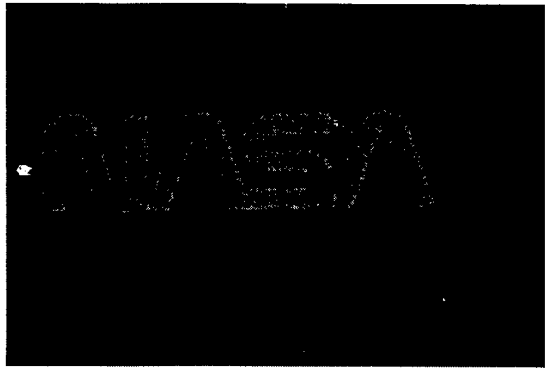
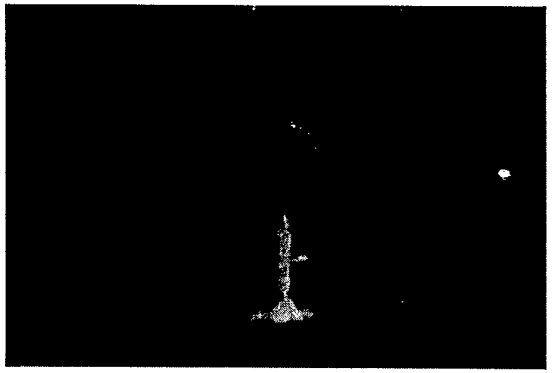
Introduction

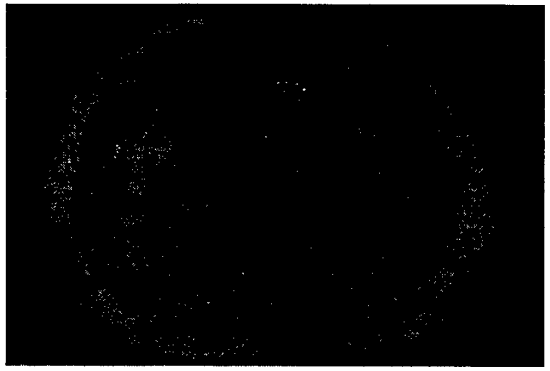
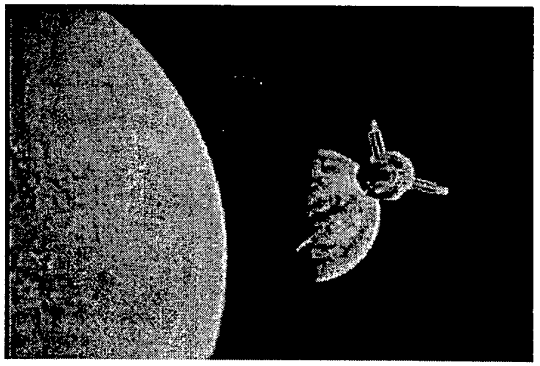
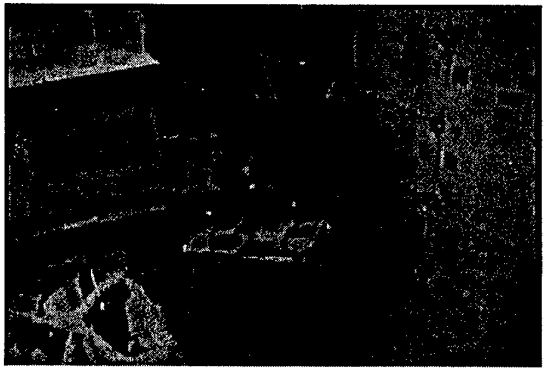
- Multimedia requirements
- Multimedia computing architectures

Multimedia requirements

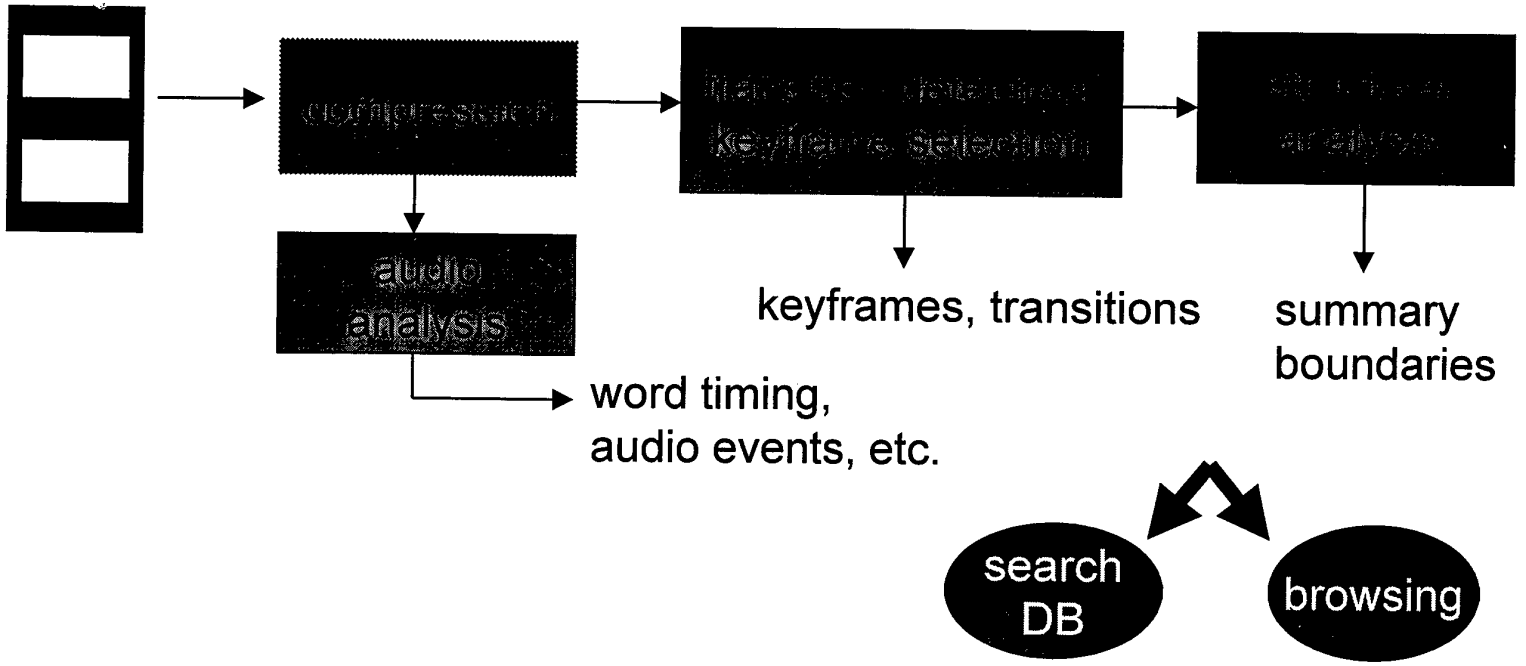
- Today, compression is the dominant application.
- Tomorrow, analysis will be as important:
  - object recognition;
  - summarization;
  - analysis of situations.

Storyboard made of keyframes

Video analysis flow



Transition detection

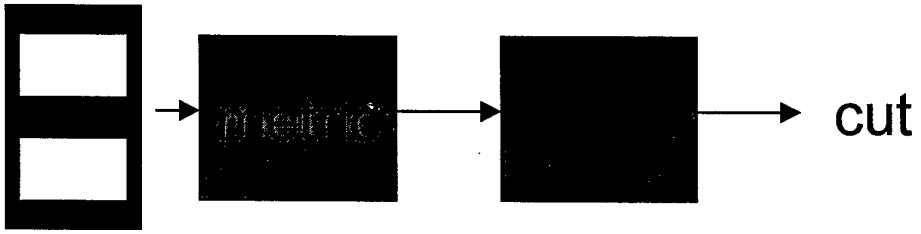
- Types of transitions:
  - cut
  - fade
  - dissolve
  - wipe, etc.
- Information exists in edit decision list, which may not be available to user.

Transition detection techniques

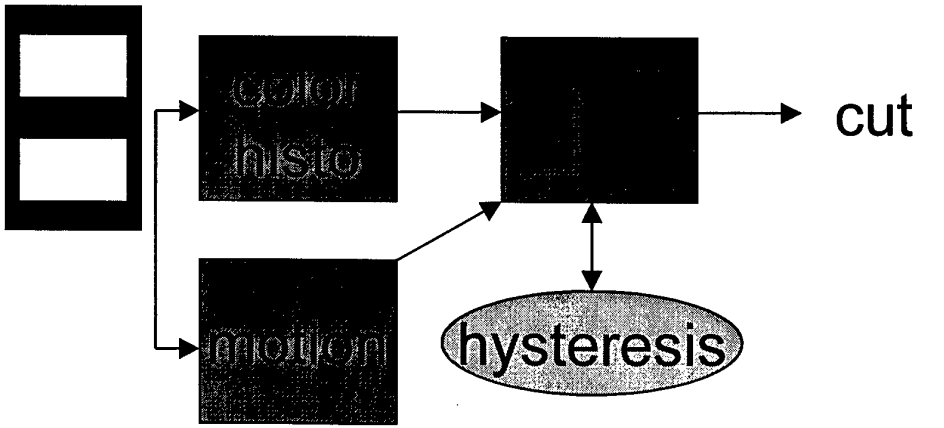
- A large single-frame difference is a cut. A gradual change over a sequence of frames is a fade/dissolve.
- Techniques:
  - frame difference
  - color histogram
  - moment invariants
  - edge detection
  - wavelet
  - optical flow.

Cut detection

- Basic thresholding:

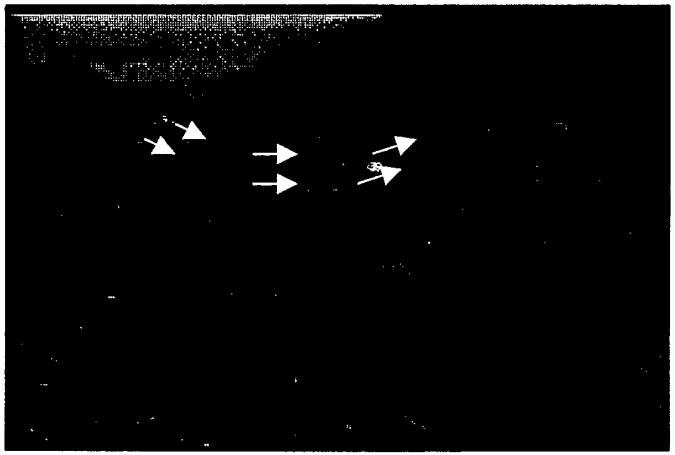


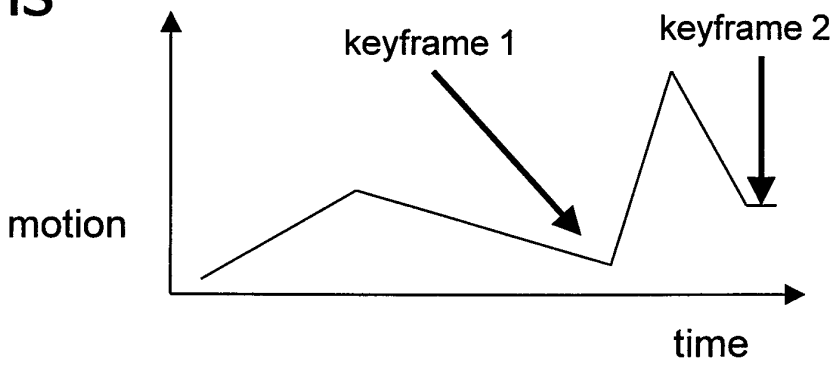
- Two-phase thresholding (Zhang et al., Boreczky and Rowe, Philips and Wolf):



Key frame analysis algorithm

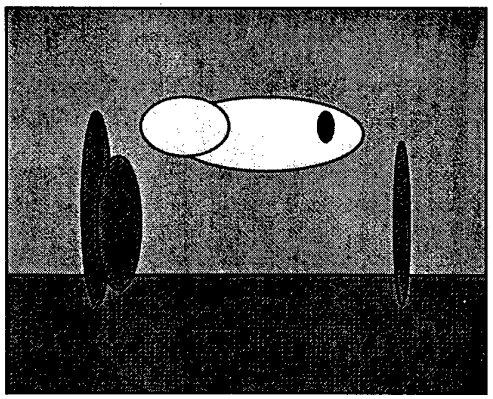
- Compute optical flow.
- Compute sum of magnitudes of optical flow vectors per frame.
- Select key frames at local minima; min/max ratio is user parameter.



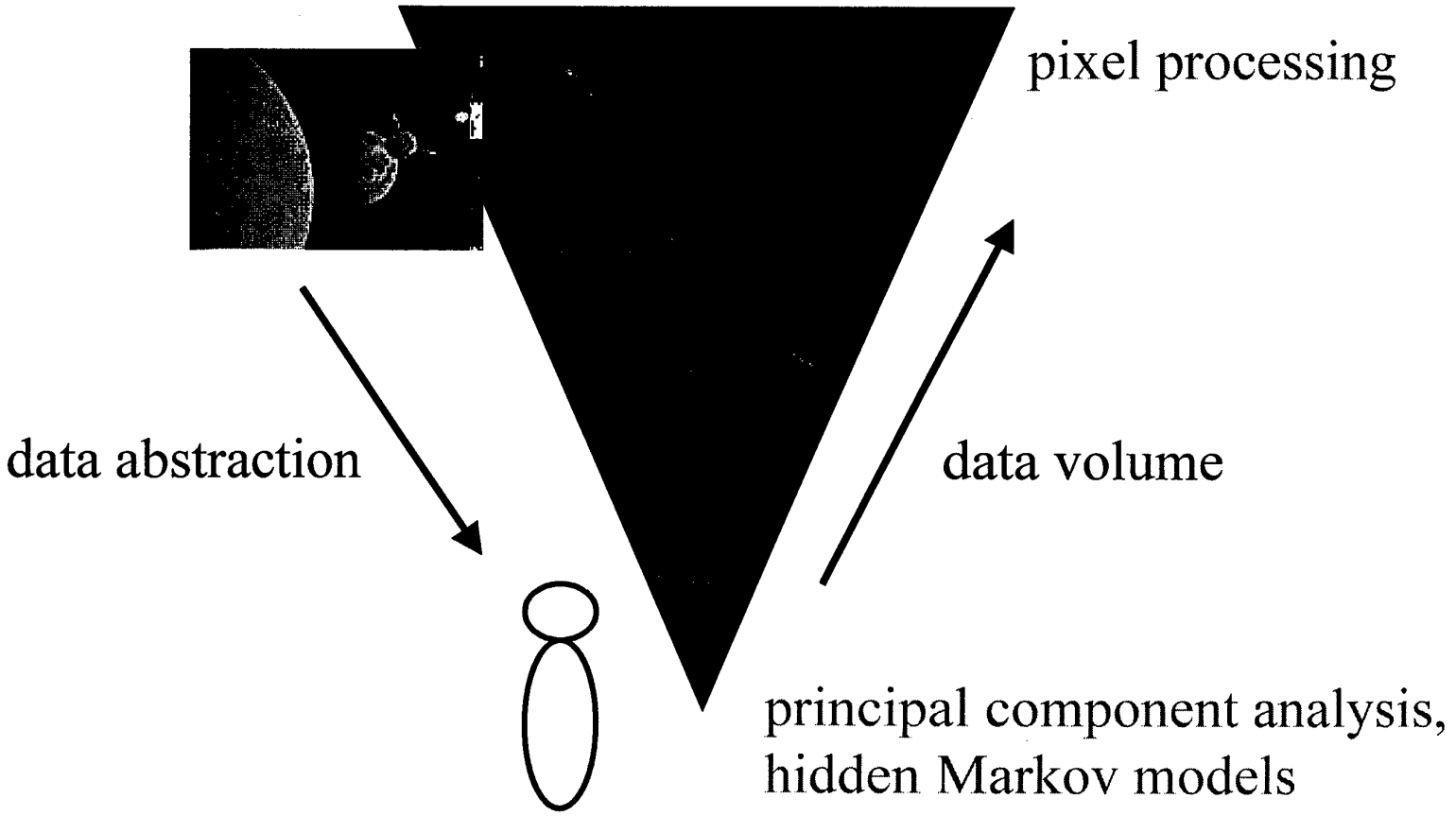


Next-generation video coding

- MPEG-4:
  - object-oriented coding;
  - user-controlled image composition.
- MPEG-7 supports video libraries.



The multimedia processing funnel



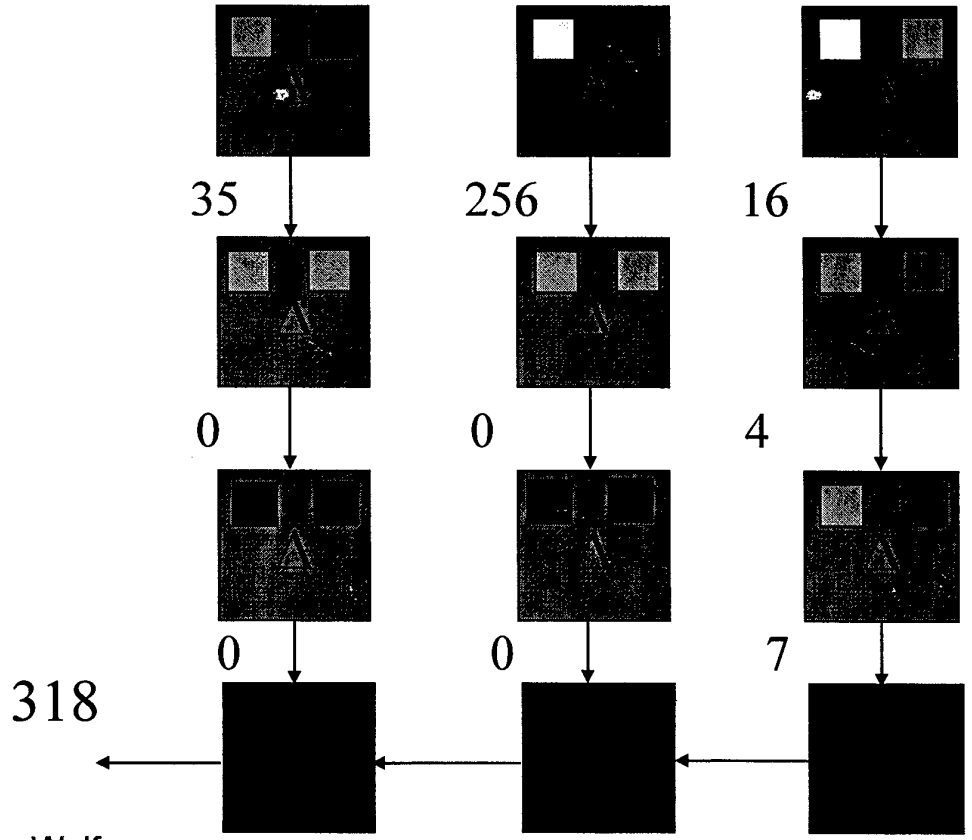
Styles of video processing

- Single-instruction multiple-data (SIMD).
- Heterogeneous multiprocessors.
- Very long instruction word (VLIW) processors.
- Instruction set architecture (ISA) extensions.

SIMD processing

- Broadcast operation to an array of processing elements, each of which has its own data.
- Well-suited to regular, data-oriented operations.

A block correlation architecture



SIMD comparison

- Well suited to certain types of computations.
- Handles only simple computations, not complete systems.

Hetereogeneous multiprocessor design

- Will need accelerators for quite some time to come:
  - power;
  - performance.
- Candidates for acceleration:
  - complex coding and error correction;
  - motion estimation.

Expensive operations

Expensive operations can be speeded up by special-purpose units:

- specialized memory accesses;
- specialized datapath operations.

Special-purpose units may be useful for only certain parameters:

- block size;
- search region size.

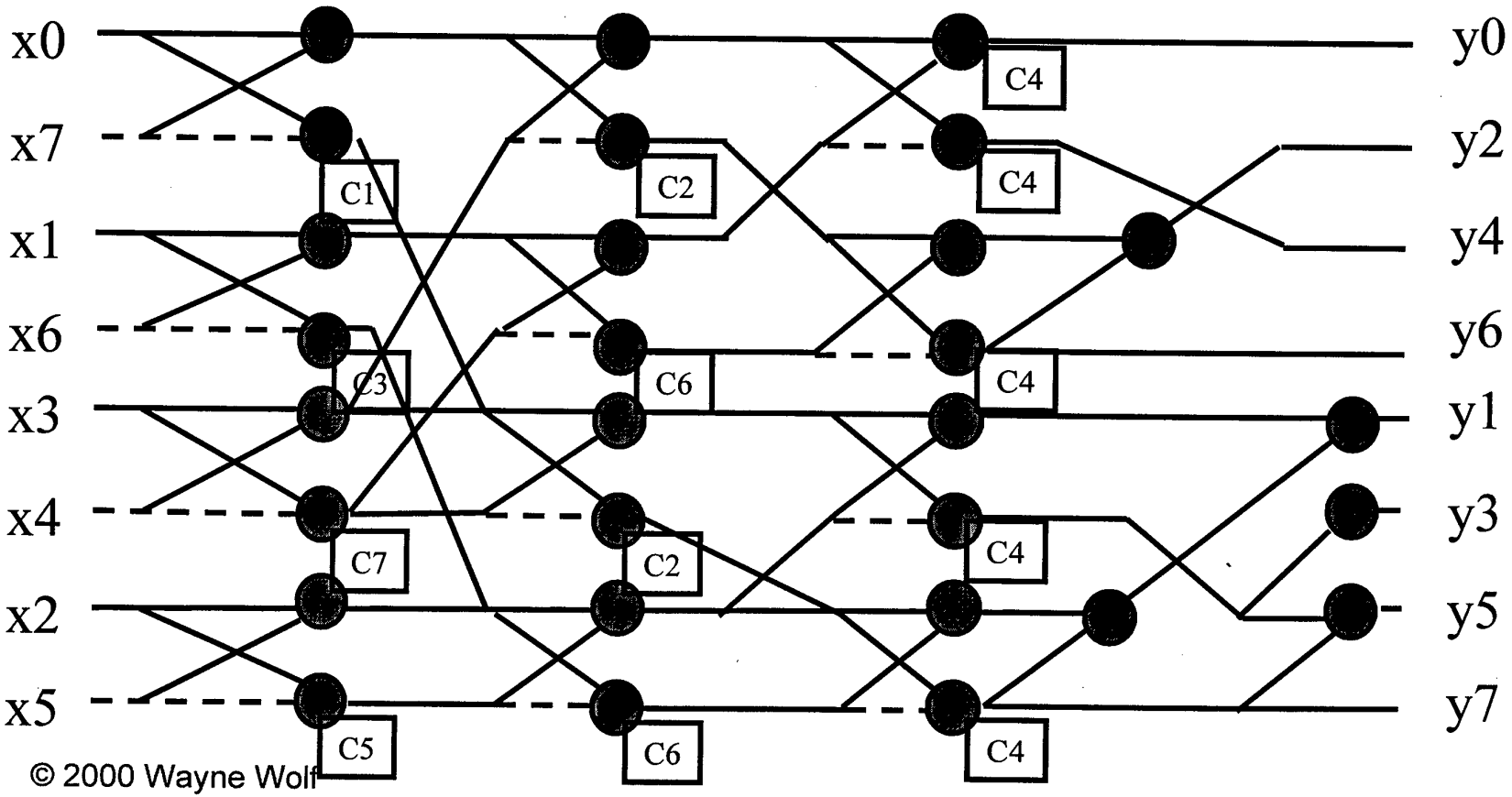
Communication bandwidth

Performance is often limited by communication bandwidth:

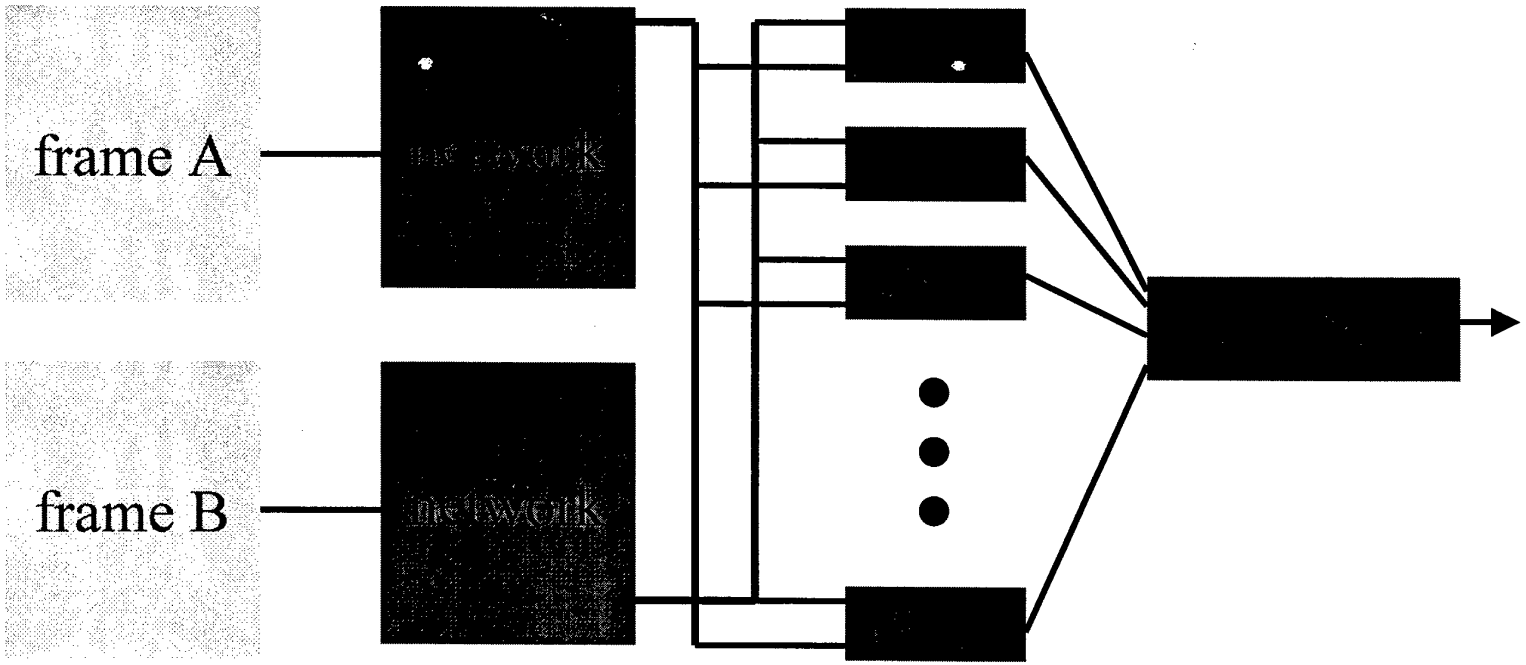
- internal;
- external.

Specialized communication topologies can make more efficient use of available bandwidth.

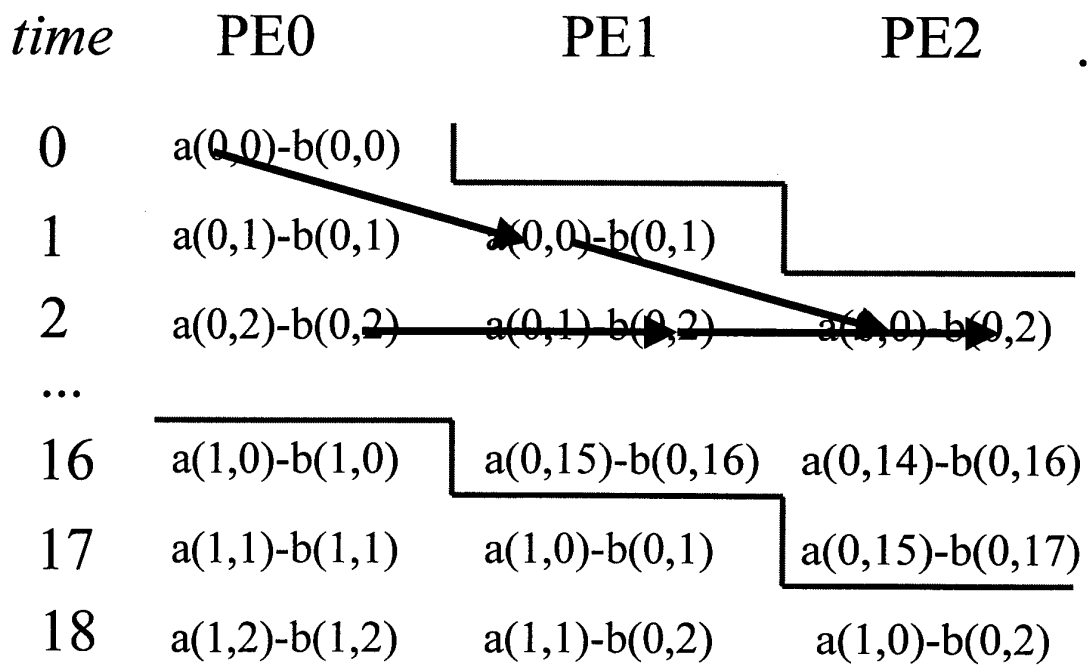
8-point DCT flowgraph (Lee)



Block motion estimation architecture



Data flow in block motion estimation



Heterogeneous multiprocessor comparison

- May make use of multiple SIMD components, CPUs, etc.
- Programmability is often necessary for verification/debugging.
- Hardware utilization is low---blocks cannot be reused for other purposes.

ISA extensions

- Augment instruction set of traditional microprocessor to provide media processing instructions:
  - smaller word sizes
  - operations particular to multimedia (saturation arithmetic)

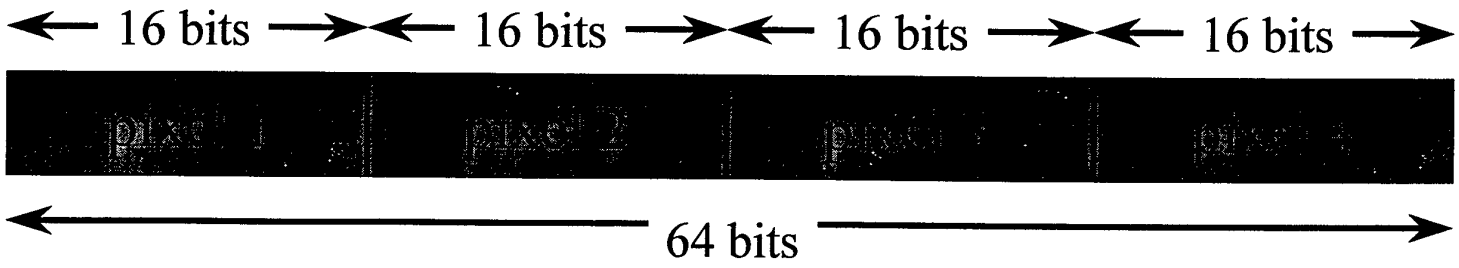
Why ISA extensions

- *Easy*: provide significant parallelism with small changes to architecture.
- *Cheap*: can be implemented with
- *Effective*: provide 2x-4x speedups.

Basic principles of ISA extensions

Split data word into subwords to provide *single instruction multiple data* (*SIMD*) parallelism.

Assemble CPU word from pixels:



Intel MMX

Data operations performed on floating-point registers.

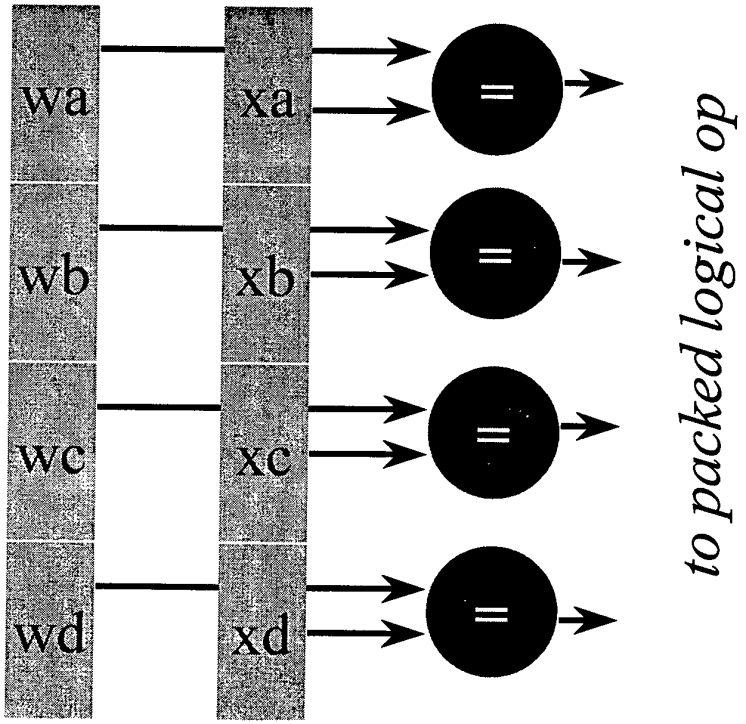
Supports multiple data lengths:

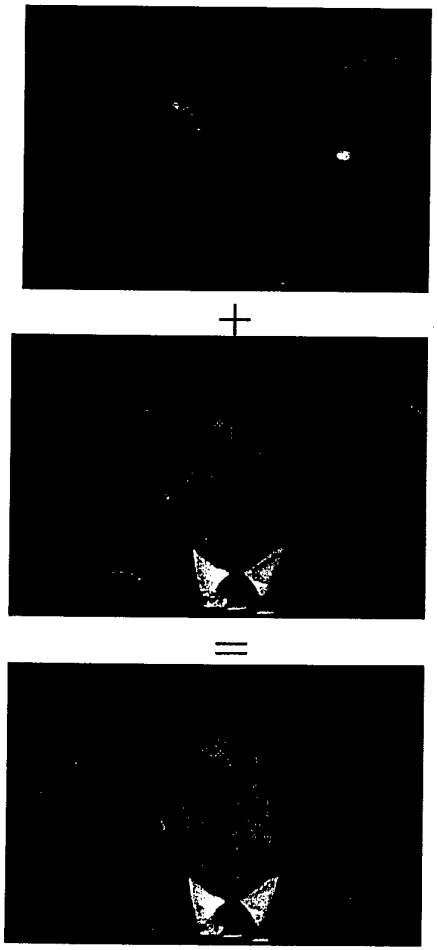
- eight 8-bit words
- four 16-bit words
- two 32-bit words

Provides packed multiply and packed compare instructions.

Packed compare instruction

Used for chromakey:



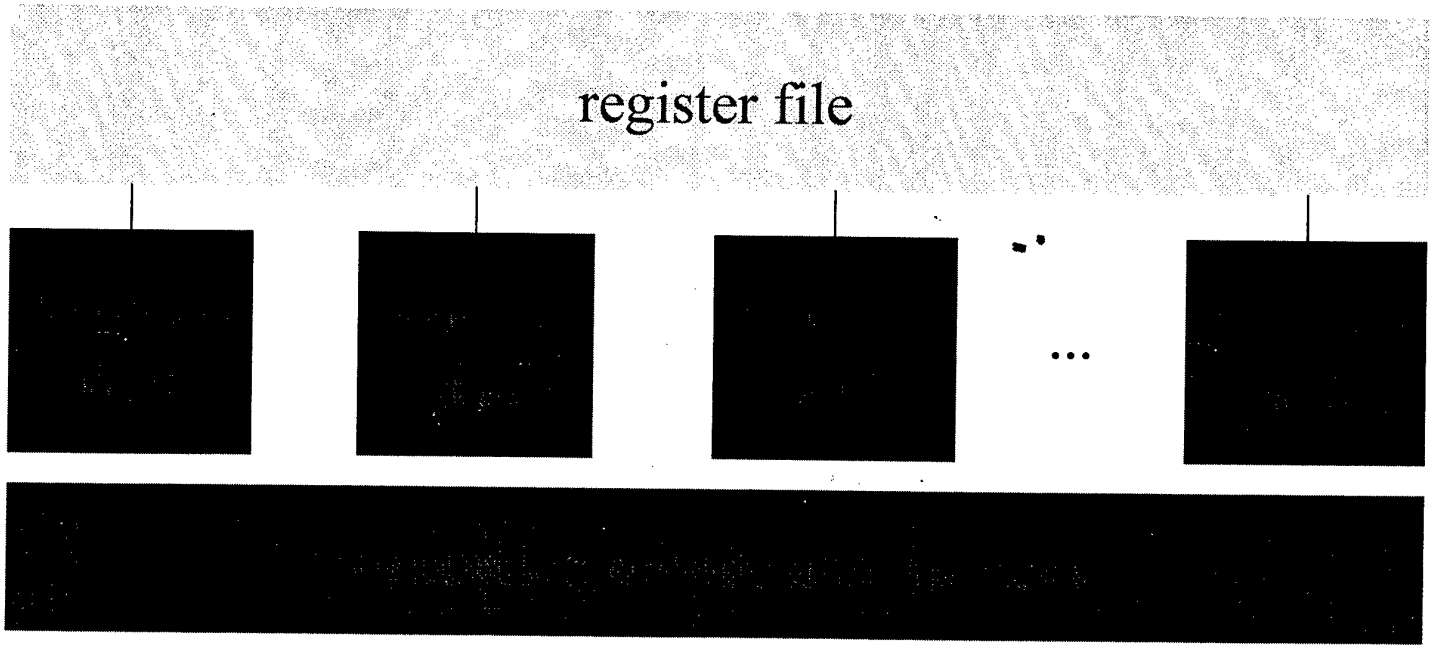


ISA extension comparison

- Significantly improves CPU performance on multimedia at low cost.
- Does not provide radical performance improvements.
- Does not improve memory system.

VLIW architectures

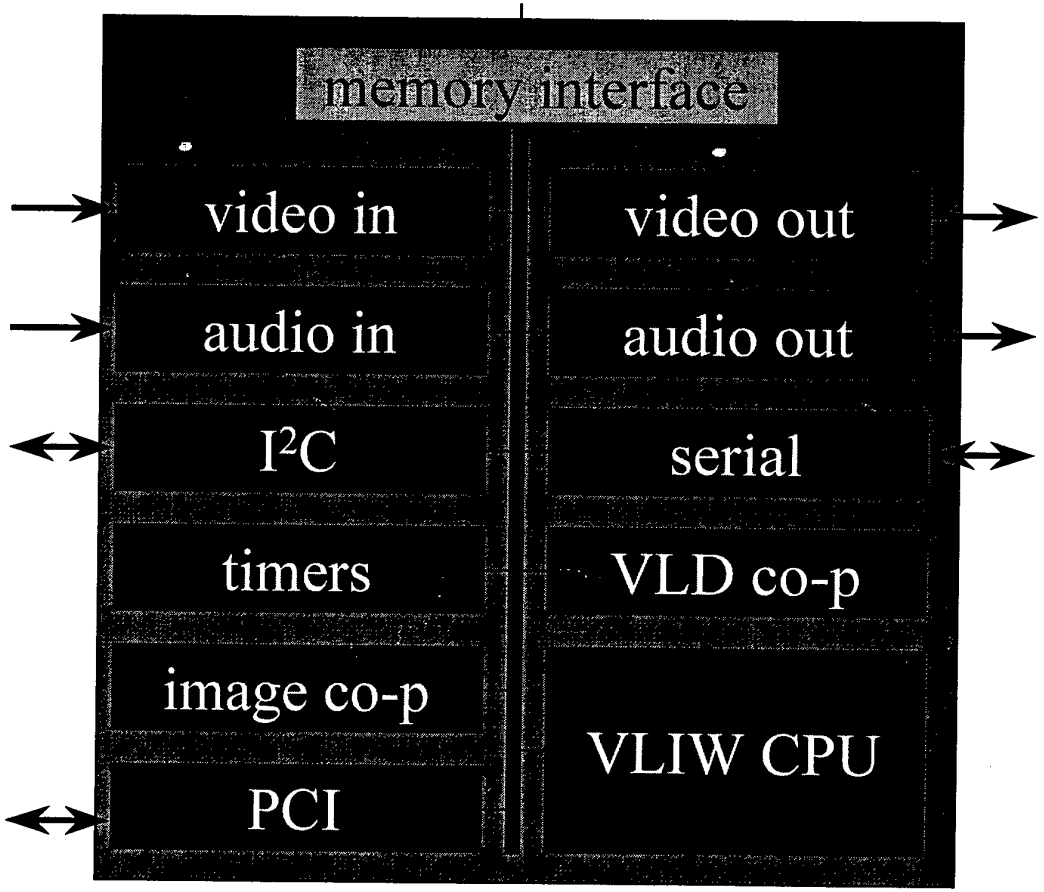
- Parallel function units, shared register file, static scheduling of operations:



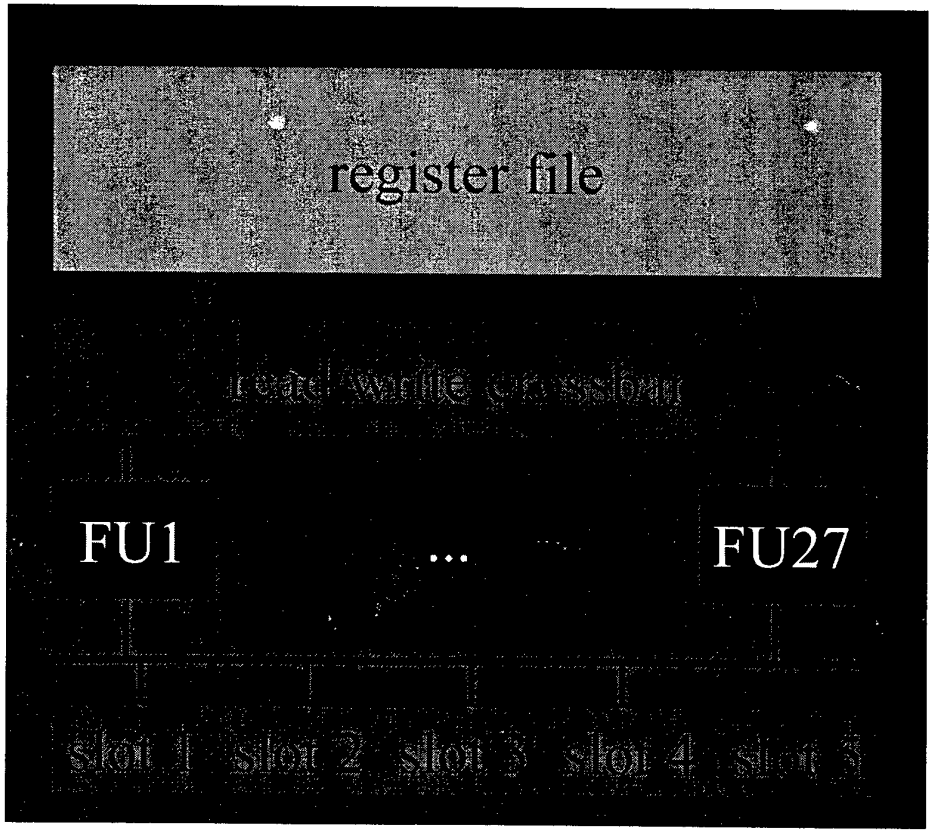
VLIW's popularity

- Invented 20 years ago, popular today:
  - Good compiler technology.
  - Low control overhead.
  - Systems-on-silicon eliminates pinout problems.
- Advantages for video:
  - Embarrassing parallelism with static scheduling opportunities.
  - Less problem with code compatibility.

Trimedia TM-1



TM-1 VLIW CPU



How programmable?

- Depends in part on marketplace and applications:
  - pro: easier to keep up with standards, new applications
  - con: more expensive
- Will probably follow wireless: multiple CPUs with accelerators and I/O.

VLIW comparison

- Well-suited to applications with embarrassing levels of parallelism.
- Relies on compiler to identify parallelism.
- VLIW CPUs often used as components in heterogeneous multiprocessors.

Debugging operations

- Breakpoints:
  - substitute instruction that returns to monitor
- Data tracing:
  - substitute instruction that calls data upload routine

Host software

- Compiler
- Evaluation board communication
- Debugger
  - breakpoints
  - data monitoring
  - profiling
  - timing

C6000 programming

- Loop partitioning.
- Multi-loop code.

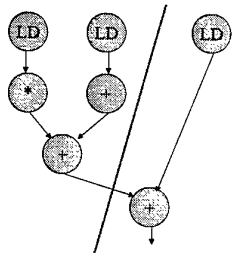
Loop partitioning

- Must partition variables, operations among the two clusters:
  - variable utilization;
  - operation type.

TI hints

- Minimize number of transfers between clusters.
- Equalize number of operations in each cluster.
- Force even numbers of instructions that write to a conditional value on each side.
- Use path directives to force loads/stores onto the proper side.

DAG model



Loop optimizations

- Software pipelining can reduce delay slots.
- Conditionally execute inner loop setup, outer loop operations with inner loop operations.

Program design and analysis

- Optimizing for execution time.
- Optimizing for energy/power.
- Optimizing for program size.

Motivation

- Embedded systems must often meet deadlines.
  - Faster may not be fast enough.
- Need to be able to analyze execution time.
  - Worst-case, not typical.
- Need techniques for reliably improving execution time.

Run times will vary

- Program execution times depend on several factors:
  - Input data values.
  - State of the instruction, data caches.
  - Pipelining effects.

Measuring program speed

- CPU simulator.
  - I/O may be hard.
  - May not be totally accurate.
- Hardware timer.
  - Requires board, instrumented program.
- Logic analyzer.
  - Limited logic analyzer memory depth.

Program performance metrics

- Average-case:
  - For typical data values, whatever they are.
- Worst-case:
  - For any possible input set.
- Best-case:
  - For any possible input set.
- Too-fast programs may cause critical races at system level.

Performance analysis

- Elements of program performance (Shaw):
  - execution time = program path + instruction timing
- Path depends on data values. Choose which case you are interested in.
- Instruction timing depends on pipelining, cache behavior.

Programs and performance analysis

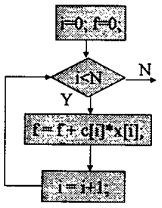
- Best results come from analyzing optimized instructions, not high-level language code:
  - non-obvious translations of HLL statements into instructions;
  - code may move;
  - cache effects are hard to predict.

Program paths

- Consider for loop:

```
for (i=0, f=0, i<N; i++)
  f = f + c[i]*x[i];
```

- Loop initiation block executed once.
- Loop test executed N+1 times.
- Loop body and variable update executed N times.



Instruction timing

- Not all instructions take the same amount of time.
- Instruction execution times are not dependent.
- Execution time may depend on operand values.s

Trace-driven performance analysis

- Trace: a record of the execution path of a program.
- Trace gives execution path for performance analysis.
- A useful trace:
  - requires proper input values;
  - is large (gigabytes).

Trace generation

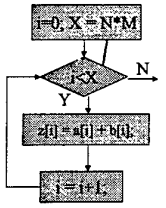
- Hardware capture:
  - logic analyzer;
  - hardware assist in CPU.
- Software:
  - PC sampling.
  - Instrumentation instructions.
  - Simulation.

Loop optimizations

- Loops are good targets for optimization.
- Basic loop optimizations:
  - code motion;
  - induction-variable elimination;
  - strength reduction (x*2 -> x<<1).

Code motion

```
for (i=0; i<N*M; i++)
  z[i] = a[i] + b[i];
```



Induction variable elimination

- Induction variable: loop index.
- Consider loop:

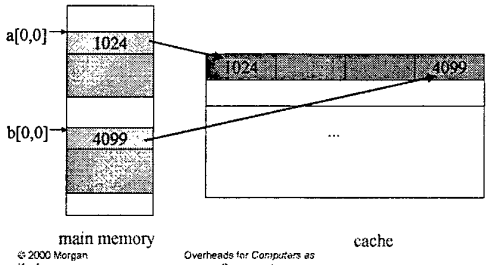
```
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
    z[i,j] = b[i,j];
```

- Rather than recompute i*M+j for each array in each iteration, share induction variable between arrays, increment at end of loop body.

Cache analysis

- Loop nest: set of loops, one inside other.
- Perfect loop nest: no conditionals in nest.
- Because loops use large quantities of data, cache conflicts are common.

Array conflicts in cache



Array conflicts, cont'd.

- Array elements conflict because they are in the same line, even if not mapped to same location.
- Solutions:
  - move one array;
  - pad array.

Performance optimization hints

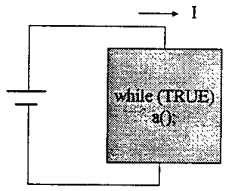
- Use registers efficiently.
- Use page mode memory accesses.
- Analyze cache behavior:
  - instruction conflicts can be handled by rewriting code, rescheudling;
  - conflicting scalar data can easily be moved;
  - conflicting array data can be moved, padded.

Energy/power optimization

- Energy: ability to do work.
  - Most important in battery-powered systems.
- Power: energy per unit time.
  - Important even in wall-plug systems---power becomes heat.

Measuring energy consumption

- Execute a small loop, measure current:



Sources of energy consumption

- Relative energy per operation (Catthoor et al):
  - memory transfer: 33
  - external I/O: 10
  - SRAM write: 9
  - SRAM read: 4.4
  - multiply: 3.6
  - add: 1

Cache behavior is important

- Energy consumption has a sweet spot as cache size changes:
  - cache too small: program thrashes, burning energy on external memory accesses;
  - cache too large: cache itself burns too much power.

Optimizing for energy

- First-order optimization:
  - high performance = low energy.
- Not many instructions trade speed for energy.

Optimizing for energy, cont'd.

- Use registers efficiently.
- Identify and eliminate cache conflicts.
- Moderate loop unrolling eliminates some loop overhead instructions.
- Eliminate pipeline stalls.
- Inlining procedures may help: reduces linkage, but may increase cache thrashing.

Optimizing for program size

- Goal:
  - reduce hardware cost of memory;
  - reduce power consumption of memory units.
- Two opportunities:
  - data;
  - instructions.

Data size minimization

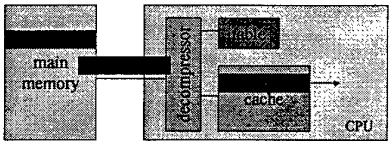
- Reuse constants, variables, data buffers in different parts of code.
  - Requires careful verification of correctness.
- Generate data using instructions.

Reducing code size

- Avoid function inlining.
- Choose CPU with compact instructions.
- Use specialized instructions where possible.

Code compression

- Use statistical compression to reduce code size, decompress on-the-fly:



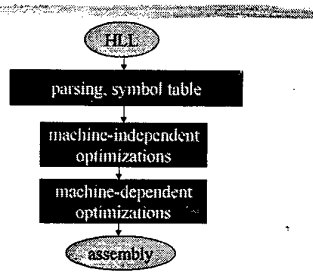
Program design and analysis

- Compilation flow.
- Basic statement translation.
- Basic optimizations.
- Interpreters and just-in-time compilers.

Compilation

- Compilation strategy (Wirth):
  - compilation = translation + optimization
- Compiler determines quality of code:
  - use of CPU resources;
  - memory access scheduling;
  - code size.

Basic compilation phases



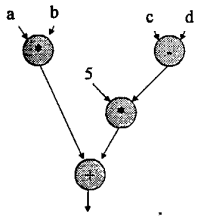
Statement translation and optimization

- Source code is translated into intermediate form such as CDFG.
- CDFG is transformed/optimized.
- CDFG is translated into instructions with optimization decisions.
- Instructions are further optimized.

Arithmetic expressions

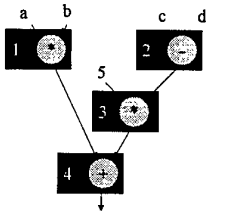
a*b + 5*(c-d)

expression



DFG

Arithmetic expressions, cont'd.



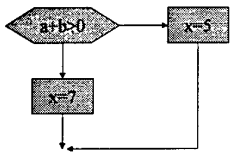
DFG

```
ADR r4,a
MOV r1,[r4]
ADR r4,b
MOV r2,[r4]
ADD r3,r1,r2
ADR r4,c
MOV r1,[r4]
ADR r4,d
MOV r5,[r4]
SUB r6,r4,r5
MUL r7,r6,#5
ADD r8,r7,r3
```

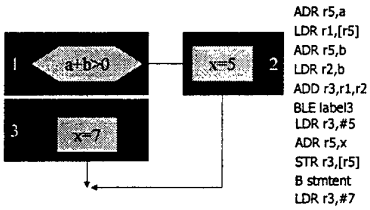
code

Control code generation

```
if (a+b > 0)
  x = 5;
else
  x = 7;
```



Control code generation, cont'd.

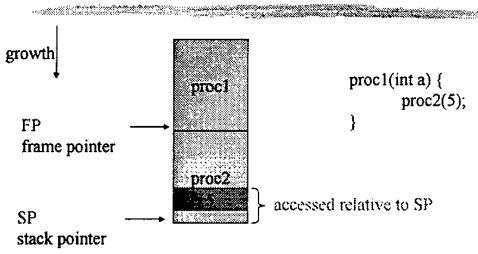



```
ADR r5,a
LDR r1,[r5]
ADR r5,b
LDR r2,b
ADD r3,r1,r2
BLE label3
LDR r3,#5
ADR r5,x
STR r3,[r5]
B stmtent
LDR r3,#7
ADR r5,x
STR r3,[r5]
stmtent ...
```


Procedure linkage

- Need code to:
  - call and return;
  - pass parameters and results.
- Parametres and returns are passed on stack.

Procedure stacks




```
proc1(int a) {
        proc2(5);
}
```


ARM procedure linkage

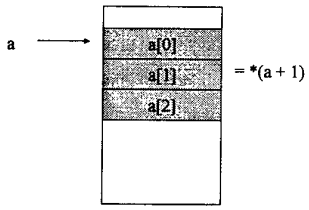
- APCS (ARM Procedure Call Standard):
  - r0-r3 pass parameters into procedure. Extra parameters are put on stack frame.
  - r0 holds return value.
  - r4-r7 hold register values.
  - r11 is frame pointer, r13 is stack pointer.
  - r10 holds limiting address on stack size to check for stack overflows.

Data structures

- Different types of data structures use different data layouts.
- Some offsets into data structure can be computed at compile time, others must be computed at run time.

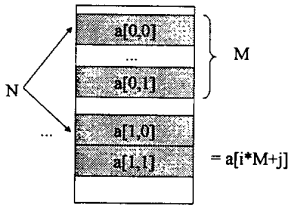
One-dimensional arrays

- C array name points to 0th element:



Two-dimensional arrays

- Column-major layout:

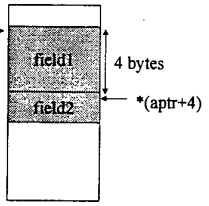


Structures

- Fields within structures are static offsets:

```
struct {
  int field1;
  char field2;
} mystruct;

struct mystruct a, *aptr = &a;
```



Expression simplification

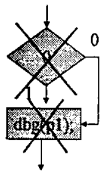
- Constant folding:
  - 8+1 = 9
- Algebraic:
  - a*b + a*c = a*(b+c)
- Strength reduction:
  - a*2 = a<<1

Dead code elimination

- Dead code:

```
#define DEBUG 0
if (DEBUG) dbg(p1);
```

- Can be eliminated by analysis of control flow, constant folding.



Procedure inlining

- Eliminates procedure linkage overhead:

```
int foo(a,b,c) { return a + b - c;}
z = foo(w,x,y);
⇨
z = w + x + y;
```

Loop transformations

- Goals:
  - reduce loop overhead;
  - increase opportunities for pipelining;
  - improve memory system performance.

Loop unrolling

- Reduces loop overhead, enables some other optimizations.

```
for (i=0; i<4 i++)
  a[i] = b[i] * c[i];
```

⇨

```
for (i=0; i<2 i++) {
  a[i*2] = b[i*2] * c[i*2];
  a[i*2+1] = b[i*2+1] * c[i*2+1];
}
```


Loop fusion and distribution

- Fusion combines two loops into 1:

```
for (i=0; i<N; i++) a[i] = b[i] * 5;
for (j=0; j<N; j++) w[j] = c[j] * d[j];
⇨ for (i=0; i<N; i++) {
  a[i] = b[i] * 5; w[i] = c[i] * d[i];
  }
```

- Distribution breaks one loop into two.
- Changes optimizations within loop body.

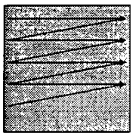
Loop tiling

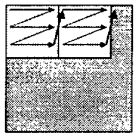
- Breaks one loop into a nest of loops.
- Changes order of accesses within array.
  - Changes cache behavior.

Loop tiling example

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    c[i] = a[i,j]*b[i];
```

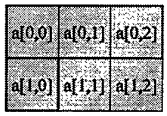
```
for (i=0; i<N; I+=2)
  for (j=0; j<N; j+=2)
    for (ii=0; ii<N; ii++)
      for (jj=0; jj<N; jj++)
        c[ii] = a[ii,jj]*b[ii];
```



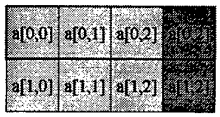


Array padding

- Add array elements to change mapping into cache:



before



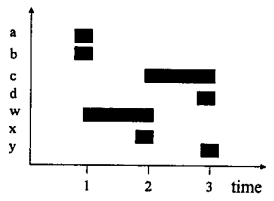
after

Register allocation

- Goals:
  - choose register to hold each variable;
  - determine lifespan of varible in the register.
- Basic case: within basic block.

Register lifetime graph

w = a + b; t=1
x = c + w; t=2
y = c + d; t=3



Instruction scheduling

- Non-pipelined machines do not need instruction scheduling: any order of instructions that satisfies data dependencies runs equally fast.
- In pipelined machines, execution time of one instruction depends on the nearby instructions: opcode, operands.

Reservation table

- A reservation table relates instructions/time to CPU resources.

Time/instr	A	B
instr1	X	
instr2	X	X
instr3	X	
instr4		X

Software pipelining

- Schedules instructions across loop iterations.
- Reduces instruction latency in iteration i by inserting instructions from iteration i+1.

Software pipelining in SHARC

- Example:

```
for (i=0; i<N; i++)
  sum += a[i]*b[i];
```

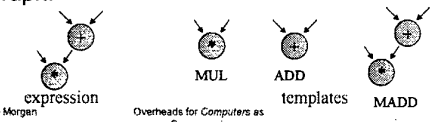
- Combine three iterations:
  - Fetch array elements a, b for iteration i.
  - Multiply a,b for iteration i-1.
  - Compute dot product for iteration i-2.

Software pipelining in SHARC, cont'd

```
/* first iteration performed outside loop */
ai=a[0]; bi=b[0]; p=ai*bi;
/* initiate loads used in second iteration; remaining loads
   will be performed inside the loop */
for (i=2; i<N-2; i++) {
   ai=a[i]; bi=b[i]; /* fetch for next cycle's multiply */
   p = ai*bi; /* multiply for next iteration's sum */
   sum += p; /* make sum using p from last iteration */
}
sum += p; p=ai*bi; sum +=p;
```


Instruction selection

- May be several ways to implement an operation or sequence of operations.
- Represent operations as graphs, match possible instruction sequences onto graph.



Using your compiler

- Understand various optimization levels (-O1, -O2, etc.)
- Look at mixed compiler/assembler output.
- Modifying compiler output requires care:
  - correctness;
  - loss of hand-tweaked code.

Interpreters and JIT compilers

- Interpreter: translates and executes program statements on-the-fly.
- JIT compiler: compiles small sections of code into instructions during program execution.
  - Eliminates some translation overhead.
  - Often requires more memory.

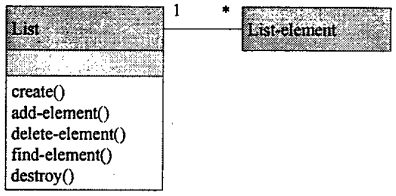
Program design and analysis

- Design patterns
- Representations of programs
- Assembly and linking
- The compilation process

Design patterns

- Design pattern: generalized description of the design of a certain type of program.
- Designer fills in details to customize the pattern to a particular programming problem.

List design pattern



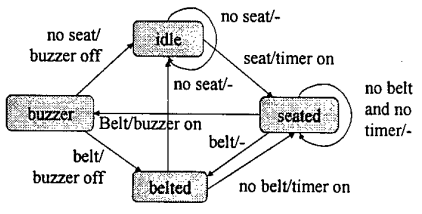
Design pattern elements

- Class diagram
- State diagrams
- Sequence diagrams
- etc.

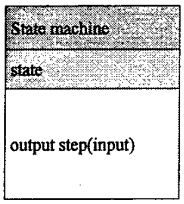
State machine

- State machine is useful in many contexts:
  - parsing user input
  - responding to complex stimuli
  - controlling sequential outputs

State machine example



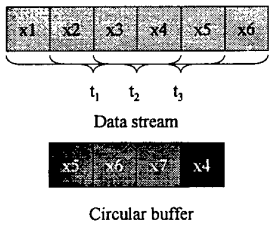
State machine pattern



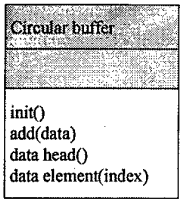
C implementation

```
#define IDLE 0
#define SEATED 1
#define BELTED 2
#define BUZZER 3
switch (state) {
   case IDLE: if (seat) { state = SEATED; timer_on = TRUE; }
        break;
   case SEATED: if (belt) state = BELTED;
                else if (timer) state = BUZZER;
        break;
   ...
}
```


Circular buffer



Circular buffer pattern



Circular buffer implementation: FIR filter

```
int circ_buffer[N], circ_buffer_head = 0;
int c[N]; /* coefficients */
...
int ibuf, ic;
for (f=0, ibuff=circ_buff_head, ic=0;
   ic<N; ibuff=(ibuff==N-1?0:ibuff++), ic++)
f = f + c[ic]*circ_buffer[ibuf];
```


Models of programs

- Source code is not a good representation for programs:
  - clumsy;
  - leaves much information implicit.
- Compilers derive intermediate representations to manipulate and optiize the program.

Data flow graph

- DFG: data flow graph.
- Does not represent control.
- Models basic block: code with no entry or exit.
- Describes the minimal ordering requirements on operations.

Single assignment form

```
x = a + b;
y = c - d;
z = x * e;
y = b + d;
```

original basic block

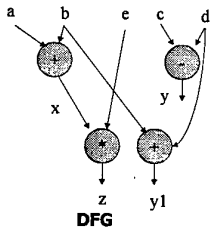
```
x = a + b;
y = c - d;
z = x * e;
y1 = b + d;
```

single assignment form

Data flow graph

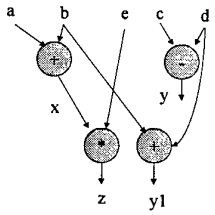
```
x = a + b;
y = c - d;
z = x * e;
y1 = b + d;
```

single assignment form



DFG

DFGs and partial orders



Partial order:

- a+b, c-d, b+d; x*e

Can do first three operations in any order (or in parallel).

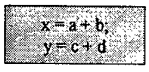
Must do x*e after a+b.

Control-data flow graph

- CDFG: represents control and data.
- Uses data flow graphs as components.
- Two types of nodes:
  - decision;
  - data flow.

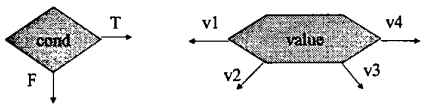
Data flow node

Encapsulates a data flow graph:



Write operations in basic block form for simplicity.

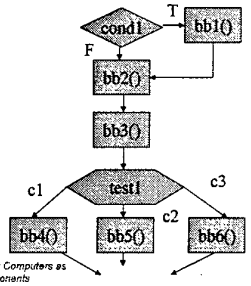
Control



Equivalent forms

CDFG example

```
if (cond1) bb1();
 else bb2();
bb3();
switch (test1) {
   case c1: bb4(); break;
   case c2: bb5(); break;
   case c3: bb6(); break;
}
```



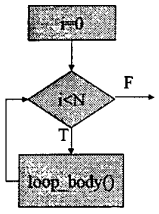
for loop

```
for (i=0; i<N; i++)
   loop_body();
```

for loop

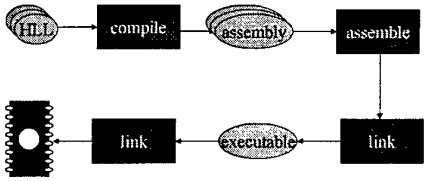
```
i=0;
while (i<N) {
   loop_body(); i++; }
```

equivalent



Assembly and linking

- Last steps in compilation:



Multiple-module programs

- Programs may be composed from several files.
- Addresses become more specific during processing:
  - relative addresses are measured relative to the start of a module;
  - absolute addresses are measured relative to the start of the CPU address space.

Assemblers

- Major tasks:
  - generate binary for symbolic instructions;
  - translate labels into addresses;
  - handle pseudo-ops (data, etc.).
- Generally one-to-one translation.
- Assembly labels:

```
          ORG 100
label1    ADR r4,c
```

Symbol table

```
      ADD r0,r1,r2        xx   0x8
xx    ADD r3,r4,r5        yy   0x10
      CMP r0,r3
yy    SUB r5,r6,r7
```

assembly code symbol table

Symbol table generation

- Use program location counter (PLC) to determine address of each location.
- Scan program, keeping count of PLC.
- Addresses are generated at assembly time, not execution time.

Symbol table example

```
PLC=0x7
PLC=0x7   )D r0,r1,r2        xx   0x8
PLC=0x7   )D r3,r4,r5        yy   0x10
PLC=0x7   IP r0,r3
PLC=0x7
yy →SUB r5,r6,r7
```


Two-pass assembly

- Pass 1:
  - generate symbol table
- Pass 2:
  - generate binary instructions

Relative address generation

- Some label values may not be known at assembly time.
- Labels within the module may be kept in relative form.
- Must keep track of external labels---can't generate full binary for instructions that use external labels.

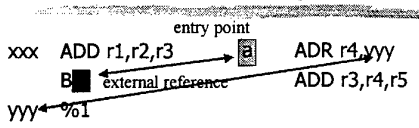
Pseudo-operations

- Pseudo-ops do not generate instructions:
  - ORG sets program location.
  - EQU generates symbol table entry without advancing PLC.
  - Data statements define data blocks.

Linking

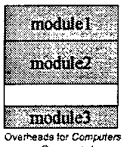
- Combines several object modules into a single executable module.
- Jobs:
  - put modules in order;
  - resolve labels across modules.

Externals and entry points



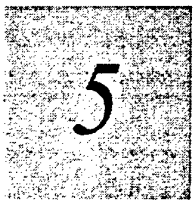
Module ordering

- Code modules must be placed in absolute positions in the memory space.
- Load map or linker flags control the order of modules.



Dynamic linking

- Some operating systems link modules dynamically at run time:
  - shares one copy of library among all executing programs;
  - allows programs to be updated with new versions of libraries.



Program Design and Analysis

5.1 Introduction

In this chapter we will study in detail the process of programming embedded processors. The creation of embedded programs is at the heart of embedded system design. If you are reading this book, you almost certainly have an understanding of programming, but designing and implementing embedded programs is different and more challenging than writing typical workstation or PC programs. Embedded code must not only provide rich functionality, it must also often run at a required rate to meet system deadlines, fit into the allowed amount of memory, and meet power consumption requirements. Designing code that simultaneously meets multiple design constraints is a considerable challenge, but luckily there are techniques and tools that we can use to help us through the design process. Making sure that the program works is also a challenge, but once again methods and tools come to our aid.

Throughout our discussion we will concentrate on high-level programming languages, specifically C. High-level languages were once shunned as too inefficient for embedded microcontrollers, but better compilers, more compiler-friendly architectures, and faster processors and memory have made high-level language programs common. Some sections of a program may

still need to be written in assembly language if the compiler doesn't give sufficiently good results, but even when coding in assembly language it is often helpful to think about the program's functionality in high-level form. Many of the analysis and optimization techniques that we will study are equally applicable to programs written in assembly language.

The next section introduces design patterns as a technique for designing programs. Section 5.3 introduce the control data flow graph as a model for high-level language programs (one that can also be applied to programs written originally in assembly language). We will first make use of that model in Section 5.6 when we talk about methods for analyzing the execution time of a program. Section 5.4 reviews the assembly and linking process and Section 5.5 reviews as background the basic steps in compilation. We talk about optimization techniques specific to embedded computing in the next three sections: performance in Section 5.6, energy consumption in Section 5.7, and size in Section 5.8. In Section 5.9, we discuss techniques for ensuring that the programs you write are correct. We will close with a software modem as a design example in Section 5.10.

In this chapter:

Program design and design patterns.

Models of programs, such as data flow and control flow graphs.

An introduction to compilation methods.

Optimizing programs for speed, size, and power consumption.

How to test programs to verify their correctness.

Example: software modem.

5.2 Program Design

We will start off our study of programs by considering how to design programs. In particular, we will look at one popular technique, the use of design patterns. After introducing design patterns, we will consider a few useful design patterns for embedded computing.

5.2.1 Design Patterns

A **design pattern** is a generalized description of a way to solve a certain class of problems. As a simple example, we could write C code for one implementation of a linked list, but that code would set in concrete the data items available in the list, actions on errors, etc. A design pattern describing the list mechanism would capture the essential components and behaviors of the list without adding unnecessary detail. A design pattern can be described

in UML; it usually takes the form of a collaboration diagram, which shows how classes work together to perform a given function.

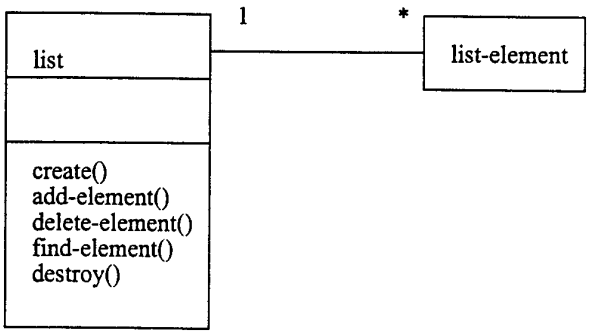


Figure 5-1 A simple description of a design pattern.

Figure 5-1 shows a simple description of a design pattern as a UML class diagram. The diagram defines two classes: *list* to describe the entire list and *list-element* for one of the items in the list. The list class defines the basic operations that you want to do on a list. The details of what goes in the list and so forth can be easily added into this design pattern. A design pattern is parameterized so that it can be customized to the needs of a particular application. A more complete description of the pattern might include:

- state diagrams to describe behavior;
- sequence diagrams to show how classes interact.

Design patterns are primarily intended to help solve mid-level design challenges. A design pattern may include only one class, but it usually describes a handful of classes. It is a rarity for a design pattern to include more than a few dozen classes. A design pattern probably will not provide you with the complete architecture of your system, but they can provide you with the architectures for many subsystems in your design; by stitching together and specializing existing design patterns you may be able to quickly create a large part of your system architecture.

Design patterns are meant to be used in ways similar to how engineers in other disciplines work. A designer can consult catalogs of design patterns to find patterns that seem to fit a particular design problem. The designer can then choose parameters suited to the application and see what that implies for the implementation of the design pattern. The designer can then choose the design pattern that seems to be the best match for the design, parameterize it, and instantiate it.

Design patterns may be of many different types:

- The digital filter is easily described as a design pattern.
- Data structures and their associated actions can be described as design patterns.

- A reactive system that reacts to external stimuli can be described as a design pattern, leaving the exact state transition diagram as a parameter.
- Douglass [Dou99] describes a policy class that describes a protocol that can be used to implement a variety of policies.

5.2.2 Design Patterns for Embedded Systems

In this section, we will consider design patterns for two very different styles of program: the state machine and the circular buffer. State machines are well-suited to **reactive systems** such as user interfaces; circular buffers are useful in digital signal processing.

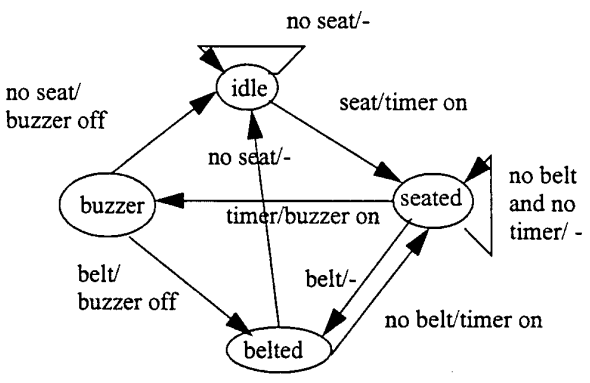
state machine style

When inputs appear intermittently rather than as periodic samples, it is often convenient to think of the system as reacting to those inputs. The reaction of most systems can be characterized in terms of the input received and the current state of the system. This leads one naturally to a **finite-state machine** style of describing the reactive system's behavior. And if the behavior is specified in that way, it is natural to write the program implementing that behavior in a state machine style. The state machine style of programming is also an efficient implementation of such computations. Finite-state machines are usually first encountered in the context of hardware design. Programming Note 5-1 shows how to write a finite-state machine in a high-level programming language.

Programming Note 5-1 A state machine in C

The behavior we want to implement is a simple seat belt controller [Chi94]. The controller's job is to turn on a buzzer if a person sits in a seat and does not fasten the seat belt within a fixed amount of time. This system has three inputs and one output. The inputs are: a sensor for the seat to know when a person has sat down, a seat belt sensor that tells when the belt is fastened; and a timer which goes off when the required time interval has elapsed. The output is the buzzer. Here is a state diagram that describes the seat belt controller's behavior:

inputs / outputs
(- = no action)



illustrator: please use UML state symbol for bubbles

The idle state is in force when there is no person in the seat. When the person sits down, the machine goes into the seated state and turns on the timer. If the timer goes off before the seat belt is fastened, the machine goes into the buzzer state; if the seat belt goes on first, it enters the belted state. When the person leaves the seat, the machine goes back to idle.

To write this in C, we will assume that we have loaded the current values of all three inputs (seat, belt, timer) into variables and will similarly hold the outputs in variables temporarily (timer_on, buzzer_on). We will use a variable named state to hold the current state of the machine and a switch statement to determine what action to take in each state. Here is the code:

```
#define IDLE 0
#define SEATED 1
#define BELTED 2
#define BUZZER 3

switch (state) { /* check the current state */
    case IDLE:
        if (seat) { state = SEATED; timer_on = TRUE; }
        /* default case is self-loop */
        break;
    case SEATED:
        if (belt) state = BELTED; /* won't hear the buzzer */
         else if (timer) state = BUZZER; /* didn't put on belt in time */
        /* default is self-loop */
        break;
    case BELTED:
        if (!seat) state = IDLE; /* person left */
         else if (!belt) state = SEATED; /* person still in seat */
        break;
    case BUZZER:
        if (belt) state = BELTED; /* belt is on---turn off buzzer */
         else if (!seat) state = IDLE; /* no one in seat---turn off buzzer
*/
        break;
}
```

This code takes advantage of the fact that the state will remain the same unless explicitly changed; this makes self-loops back to the same state easy to implement. This state machine may be executed forever in a while(TRUE) loop or periodically called by some other code. In either case, the code must be executed regularly so that it can check on the current value of the inputs and, if necessary, go into a new state.

---

data stream style

The data stream style makes sense for data which comes in regularly and must be processed on-the-fly. The FIR filter of Application Note 2-1 is a classic example of stream-oriented processing: for each sample, the filter must emit one output which depends on the values of the last *n* inputs. In a typical workstation application, we would process the samples over some interval by reading them all in from a file and then computing the results all

at once in a batch process. In an embedded system we must not only emit outputs in real time, but we must also do so using a minimum amount of memory.

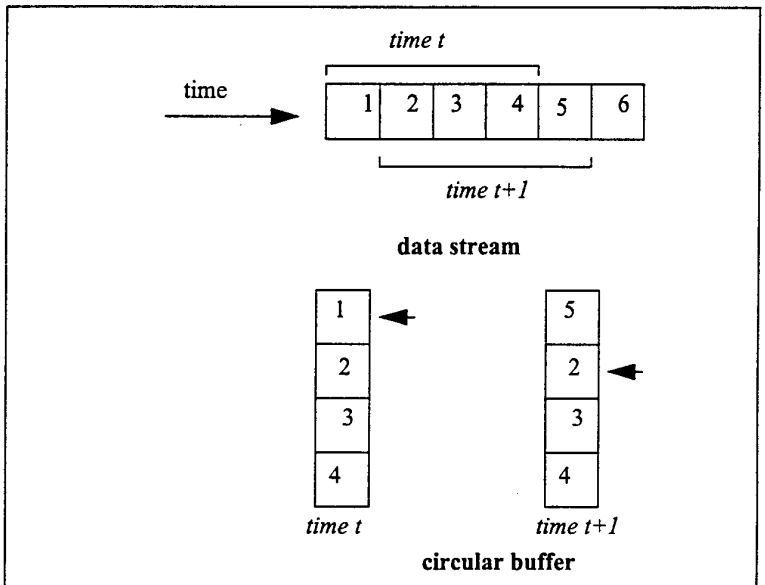


Figure 5-2 A circular buffer for streaming data.

The circular buffer is a data structure which lets us handle streaming data in an efficient way. Figure 5-2 illustrates how a circular buffer stores a subset of the data stream. At each point in time, the algorithm needs a subset of the data stream which forms a window into the stream. That window slides with time as we throw out old values no longer needed and add new values. Since the size of the window does not change, we can use a fixed-size buffer to hold the current data. To avoid constantly copying data within the buffer, we will move the head of the buffer in time. The buffer points to the location at which the next sample will be placed; every time we add a sample, we automatically overwrite the oldest sample, which is the one that needs to be thrown out. When the pointer gets to the end of the buffer, it wraps around to the top. Programming Note 5-2 gives an efficient implementation of a circular buffer.

Programming Note 5-2 A circular buffer implementation of an FIR filter

Here are the declarations for the circular buffer and filter coefficients, assuming that N, the number of taps in the filter, has been previously defined:

```
int circ_buffer[N]; /* circular buffer for data */
int circ_buffer_head = 0; /* current head of the buffer */
int c[N]; /* filter coefficients (constants) */
```

To write C code for a circular buffer-based FIR filter, we need to modify the original loop slightly. Because the 0^{th} element of data may not be in the 0^{th} element of the circular buffer, we have to change the way in which we access the data. One of the implications of this is that we need separate loop indices for the circular buffer and coefficients.

```
int ibuf, /* loop index for the circular buffer */
    ic; /* loop index for the coefficient array */
for (f=0, ibuff=circ_buff_head, ic=0; ic<N; ibuff = (ibuff == N-1 ? 0 :
ibuff++), ic++)
    f = f + c[ic] * circ_buff[ibuf];
```

This code assumes that some other code, such as an interrupt handler, is replacing the last element of the circular buffer at the appropriate times. The statement ibuff = (ibuff == N-1 ? 0 : ibuff++) is a shorthand C way of incrementing ibuff such that it returns to 0 after reaching the end of the circular buffer array.

5.3 Models of Programs

abstractions for programs

In this section, we will develop models for programs that are more general than source code. Why not use the source code directly? First, there are many different types of source code—assembly languages, C code, etc.—but we can use a single model to describe all of them. Once we have such a model, we can perform many useful analyses on the model more easily than we could on the source code. We can use the same model to help us with analyzing performance generating tests for the programs.

Our fundamental model for programs will be the **control-data flow graph (CDFG)**. (We can also model hardware behavior with the CDFG.) As the name implies, the CDFG has constructs which model both data operations (arithmetic and other computations) and control operations (conditionals). Part of the power of the CDFG comes from its combination of control and data constructs. To understand the CDFG, we will start with pure data descriptions, then extend the model to control.

5.3.1 Data Flow Graphs

A **data flow graph** is a model of a program with no conditionals. In a high-level programming language, a code segment with no conditionals—more precisely, with only one entry and exit point—is known as a basic block. Figure 5-1 shows a simple basic block. As the C code is executed, we would enter this basic block at the beginning and execute all the statements.

Before we are able to draw the data flow graph for this code we need to modify it slightly. There are two assignments to the variable x—it appears twice on the left-hand side of an assignment. We need to rewrite the code in **sin-**

```
w = a + b;
x = a - c;
y = x + d;
x = a + c;
z = y + e;
```

Figure 5-1 A basic block in C.

```
w = a + b;
x1 = a - c;
y = x1 + d;
x2 = a + c;
z = y + e;
```

Figure 5-2 The basic block in single-assignment form.

gle-assignment form, in which a variable appears only once on the left-hand side. Since our specification is C code, we will assume that the statements are executed sequentially, so that any use of a variable refers to its latest assigned value. In this case, x is not re-used in this block (presumably it is used elsewhere) so we just have to eliminate the multiple assignment to x. The result is shown in Figure 5-2, where we have used the names x1 and x2 to distinguish the separate uses of x.

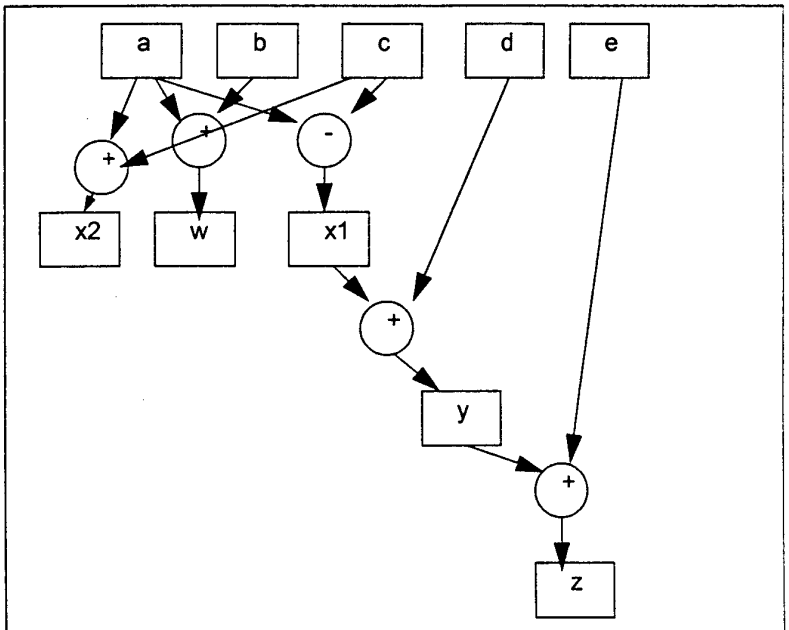


Figure 5-3 An extended data flow graph for our sample basic block.

Single-assignment form is important because it allows us to identify a unique location in the code where each named location is computed. As an introduction to the data flow graph, we will use two types of nodes in the

graph: round nodes will denote operators and square nodes will represent values. The value nodes may be either inputs to the basic block, such as a and b, or variables assigned to within the block, such as w and x1. The data flow graph for our single-assignment code is shown in Figure 5-3. The single-assignment form means that the data flow graph is acyclic—if we assigned to x multiple times, then the second assignment would form a cycle in the graph including x and the operators used to compute x. Keeping the data flow graph acyclic is important in many types of analyses we want to do on the graph. (Of course, it is important to know whether the source code actually assigns to a variable multiple times, since some of those assignments may be mistakes. We will consider the analysis of source code for proper use of assignments in Section 5.9.1.)

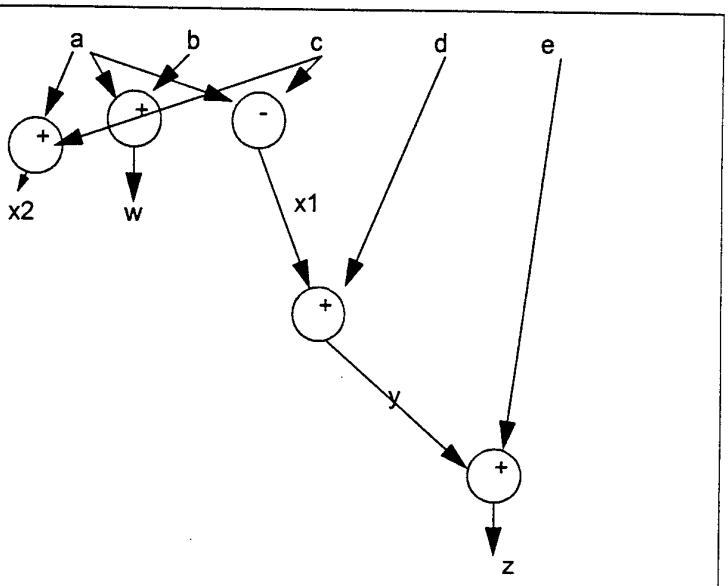


Figure 5-4 Standard data flow graph for our sample basic block.

data flow graph canonical form

The data flow graph is generally drawn in the form shown in Figure 5-4. Here, the variables are not explicitly represented by nodes. Instead, the edges are labeled with the variables they represent. As a result, a variable can be represented by more than one edge. However, the edges are directed and all the edges for a variable must come from a single source. We will use this form for its simplicity and its compactness.

partial ordering

The data flow graph for the code makes the order in which the operations are performed in the C code much less obvious. This is one of the advantages of the data flow graph. We can use it to determine feasible reorderings of the operations, which may help us to reduce pipeline or cache conflicts. We can also use it when the exact order of operations simply doesn't matter. The data flow graph defines a partial ordering of the operations in the basic block: we must ensure that a value is computed before it is used, but there

are in general several possible orderings of evaluating expressions that satisfies this requirement.

5.3.2 Control-Data Flow Graphs

```
if (cond1)
    basic_block_1();
else
    basic_block_2();
basic_block_3();
switch (test1) {
    case c1: basic_block_4(); break;
    case c2: basic_block_5(); break;
    case c3: basic_block_6(); break;
}
```

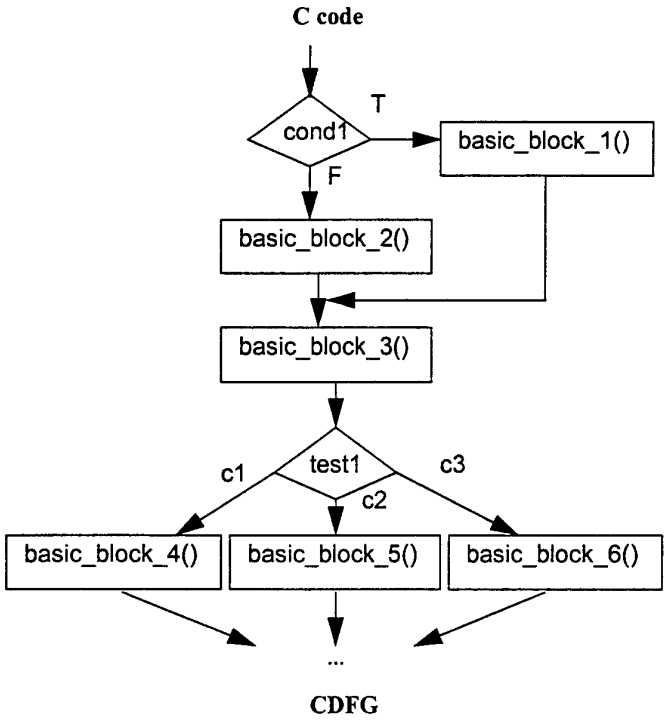


Figure 5-5 C code and its CDFG.

A CDFG uses a data flow graph as an element, adding constructs to describe control. In a basic CDFG, we will have two types of nodes: **decision nodes** and **data flow nodes**. A data flow node encapsulates a complete data flow graph to represent a basic block. We can use one type of decision node to describe all the types of control in a sequential program. (The jump/branch is, after all, the way we implement all those high-level control constructs.)

Figure 5-5 shows some C code with control constructs and the CDFG constructed from it. The rectangular nodes in the graph represent the basic blocks. The basic blocks in the C code have been represented by function calls for simplicity. The diamond-shaped nodes represent the conditionals; the node's condition is given by the label and the edges are labeled with the possible outcomes of evaluating the condition.

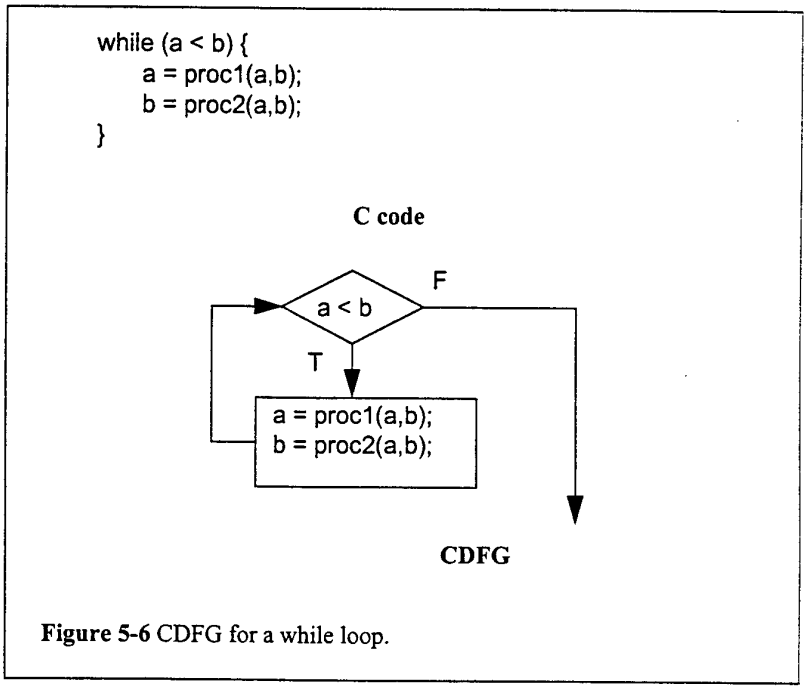


Figure 5-6 CDFG for a while loop.

loops as CDFGs

Building a CDFG for a while loop is straightforward, as shown in Figure 5-6. The while loop consists of both a test and a loop body, each of which we know how to represent in a CDFG. We can represent for loops by remembering that, in C, a for loop is defined in terms of a while loop. The for loop

```
for (i = 0; i < N; i++) {
    loop_body();
}
```

is equivalent to

```
i = 0;
while (i < N) {
    loop_body();
    i++;
}
```

hierarchical representation

For a complete CDFG model, we can use a data flow graph to model each data flow node. The CDFG is then a hierarchical representation: a data flow CDFG can be expanded to reveal a complete data flow graph.

An execution model for a CDFG is very much like the execution of the program it represents. The CDFG does not require explicit declaration of vari-

ables, but we assume that the implementation has sufficient memory for all the variables. We can define a state variable which represents a program counter in a CPU. (When studying a drawing of a CDFG, a finger works well for keeping track of the program counter state.) As we execute the program, we either execute the data flow node or compute the decision in the decision node and follow the appropriate edge, depending on what type of node the program counter points on. The CDFG, even though the data flow nodes may specify only a partial ordering on the data flow computations, is a sequential representation of the program. There is only one program counter in our execution model of the CDFG and operations are not executed in parallel.

The CDFG is not necessarily tied to high-level language control structures. We can also build a CDFG for an assembly language program. A jump instruction corresponds to a non-local edge in the CDFG. Some architectures, such as ARM and many VLIW processors, support predicated execution of instructions, which may be represented by special constructs in the CDFG.

5.4 Assembly and Linking

Assembly and linking are the last steps in the compilation process—they turn a list of instructions into an image of the program's bits in memory. In this section, we will survey the basic techniques required for assembly linking to help us understand the complete compilation process.

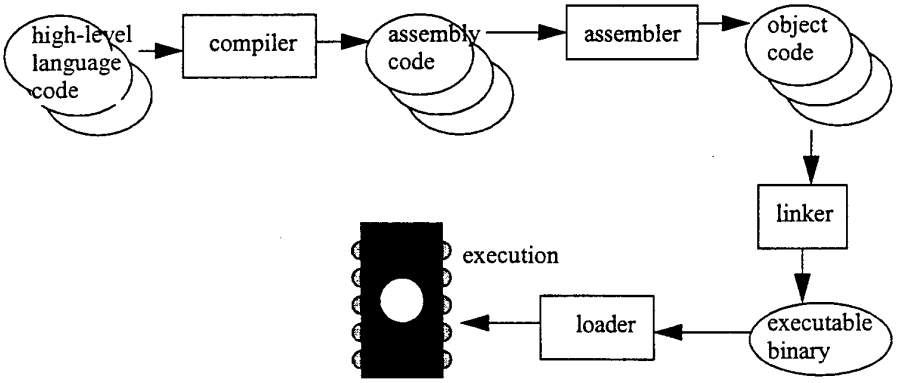


Figure 5-7 The back end of the compilation process.

Figure 5-7 highlights the role of assemblers and linkers in the compilation process. This process is often hidden from us by compilation commands that do everything required to generate an executable program. As the figure shows, most compilers do not directly generate machine code, but instead create the instruction-level program in the form of human-readable assembly language. Generating assembly language rather than binary instructions

frees the compiler writer from details extraneous to the compilation process—not just the instruction format but also the exact addresses of instructions and data. The assembler's job is to translate symbolic assembly language statements into bit-level representations of instructions known as **object code**. The assembler takes care of instruction formats and does part of the job of translating labels into addresses. However, since the program may be built from many files, the final steps in determining the addresses of instructions and data is performed by the linker, which produces an **executable binary** file. That file may not necessarily be located in the CPU's memory, however, unless the linker happens to create the executable directly in RAM. The program that brings the program into memory for execution is called a **loader**.

absolute vs. relative assembly

The simplest form of the assembler assumes that the starting address of the assembly language program has been specified by the programmer. The addresses in such a program are known as **absolute addresses.** However, in many cases, particularly when we are creating an executable out of several component files, we do not want to specify the starting addresses for all the modules before assembly—if we did, we would have to determine before assembly not only the length of each program in memory, but also order in which they would be linked into the program. Most assemblers therefore allow us to use **relative addresses** by specifying at the start of the file that the origin of that assembly language module is to be computed later. Addresses within the module are then computed relative to the start of the module. The linker is then responsible for translating relative addresses into absolute addresses.

5.4.1 Assemblers

When translating assembly code into object code, the assembler must translate opcodes and format the bits in each instruction, and translate labels into addresses.processing labels. In this section, we will review the translation of assembly language into binary.

Labels make the assembly process more complex but they are the most important abstraction provided by the assembler—labels let the programmer (whether that be a human programmer or a compiler generating assembly code) to avoid worrying about the absolute locations of instructions and data. Processing labels requires making two passes through the assembly source code:

1. The first pass scans the code to determine the address of each label.
2. The second pass actually assembles the instructions using the label values computed in the first pass.

The name of each symbol and its address is stored in a **symbol table** that is built during the first pass. The symbol table is built by scanning from the first instruction to the last. (For the moment, we will assume that we know the absolute address of the first instruction in the program; we will consider the general case in Section 5.4.2). During scanning, the current location in memory is kept in a **program location counter** (**PLC**). Despite the similar-

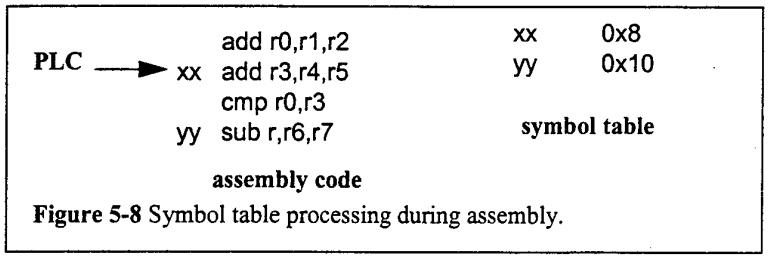


Figure 5-8 Symbol table processing during assembly.

ity in name to a program counter, the PLC is not used to execute the program, only to assign memory locations to labels. The PLC always makes exactly one pass through the program, whereas the program counter will make many passes over code in a loop, for example. So at the start of the first pass, the PLC is set to the program's starting address and the assembler looks at the first line. After examining the line, the assembler updates the PLC to the next location (since ARM instructions are four bytes long, the PLC would be incremented by four) and looks at the next instruction. If the instruction begins with a label, a new entry is made in the symbol table, which includes the label name and its value. The value of the label is equal to the current value of the PLC. At the end of the first pass, the assembler rewinds to the beginning of the assembly language file to make the second pass. During the second pass, when a label name is found, the label is looked up in the symbol table and its value substituted into the appropriate place in the instruction.

specifying program origins

But how do we know the starting value of the PLC? The simplest case is the absolute addressing case. In this case, one of the first statements in the assembly language program is a pseudo-op that specifies the **origin** of the program—the location of the first address in the program. A common name for this pseudo-op (the one used for the ARM, for example) is the ORG statement:

```
ORG   2000
```

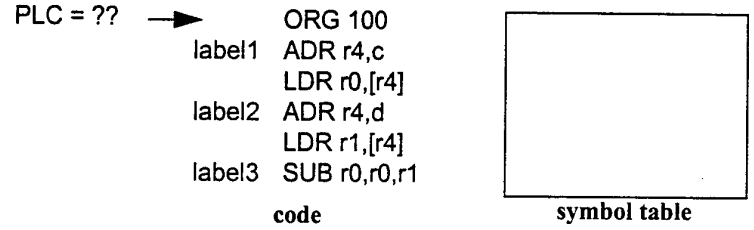
puts the start of the program at location 2000. This pseudo-op accomplishes this by setting the PLC's value to its argument's value, 2000 in this case. Assemblers generally allow a program to have many ORG statements in case instructions or data need to be spread around various spots in memory. Example 5-1 illustrates the use of the PLC in generating the symbol table.

Example 5-1 Generating a symbol table

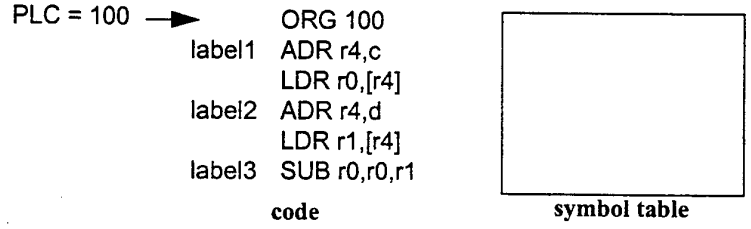
Let's use this simple example of ARM assembly code:

```
        ORG 100
label1  ADR r4,c
        LDR r0,[r4]
label2  ADR r4,d
        LDR r1,[r4]
label3  SUB r0,r0,r1
```

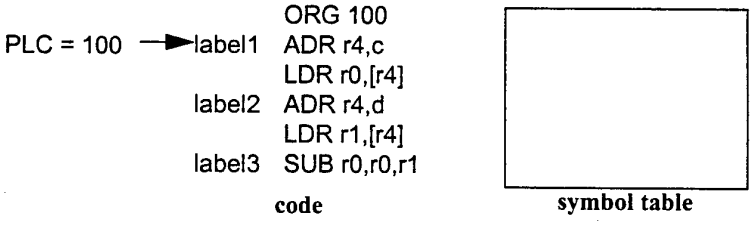
The initial ORG statement tells us the starting address of the program. To start, let's initialize our symbol table to an empty state and put our PLC at the initial ORG statement:



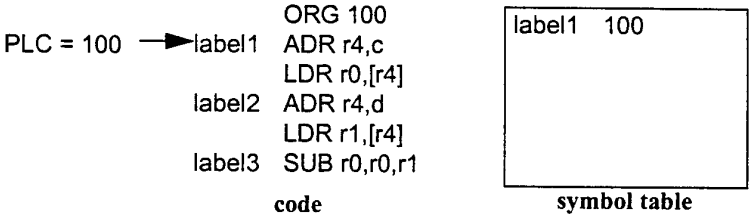
The PLC value shown is at the beginning of this step, before we have processed the ORG statement. The ORG tells us to set the PLC value to 100, which we do:



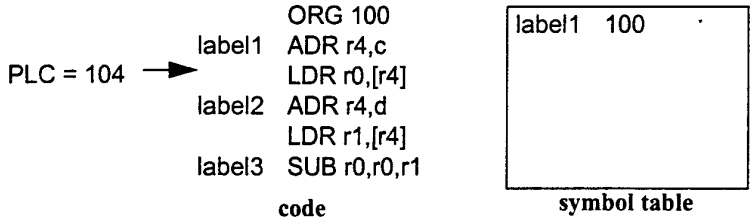
To process the next statement, we move the PLC to point to the next statement. But because the last statement was a pseudo-op that generates no memory values, the PLC value remains at 100:



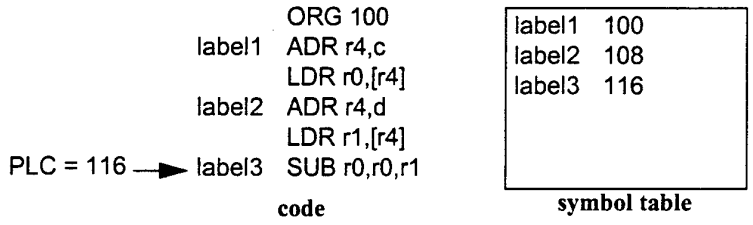
Since there is a label in this statement, we add it to the symbol table, taking its value from the current PLC value:



To process the next statement, we advance the PLC to point to the next line of the program and increment its value by the length in memory of the last line, namely 4:



We continue this process as we scan the program until we reach the end, at which the state of the PLC and symbol table are as shown:



EQU *pseudo-op*

Assemblers allow labels to be added to the symbol table without occupying space in the program memory. A typical name of this pseudo-op is EQU for equate. For example, in the code

```
      ADD r0,r1,r2
FOO   EQU 5
BAZ   SUB r3,r4,#FOO
```

the EQU pseudo-op adds a label named FOO with the value 5 to the symbol table. The value of the BAZ label is the same as if the EQU pseudo-op were not present, since EQU does not advance the PLC. The new label is used in the subsequent SUB instruction as the name for a constant. EQUs can be used to define symbolic values to help make the assembly code more structured.

ARM ADR *pseudo-op*

The ARM assembler supports one pseudo-op that is particular to the ARM instruction set. In other architectures, an address would be loaded into a register (for an indirect access, for example) by reading it from a memory location. ARM does not have an instruction that can load an effective address, so the assembler supplies the ADR pseudo-op to create the address in the register. It does so by using ADD or SUB instructions to generate the address. The address to be loaded can be register-relative, program-relative, or numeric, but it must assemble to a single instruction. More complicated address calculations must be explicitly programmed.

object file format

The assembler produces an object file that describes the instructions and data in binary format. A commonly used object file format, originally developed

for Unix but now used in other environments as well, is known as **COFF** (Common Object File Format). The object file must describe the instructions, data and any addressing information; it usually also carries along the symbol table for later use in debugging.

generating relative code

Generating relative code rather than absolute code introduces some new challenges to the assembly language process. Rather than using an ORG statement to provide the starting address, the assembly code uses a pseudo-op to indicate that the code is in fact relocatable. (Relative code is the default for both the ARM and Sharc assemblers.) Similarly, we must mark the output object file as being relative code. We can initialize the PLC to 0 to denote that addresses are relative to the start of the file. However, when we generate any code that makes use of those labels, we must be careful, since we do not yet know the actual value that must be put into the bits. We must instead generate relocatable code: we use extra bits in the object file format to mark the relevant fields as relocatable; we then insert the label's relative value into the field. The linker must therefore modify the generated code—when it finds a field marked as relative, it uses the absolute addresses that it has generated to replace the relative value with a correct, absolute value for the address. To understand the details of turning relocatable code into absolute executable code, we must understand the linking process described in the next section.

5.4.2 Linking

Many assembly language programs are written as several smaller pieces rather than as one large file. Breaking a large program into smaller files helps delineate program modularity. If the program uses library routines, those will already be pre-assembled and assembly language source code for the libraries may not be available for purchase. A **linker** allows a program to be stitched together out of several smaller pieces. The linker operates on the object files created by the assembler and modifies the assembled code to make the necessary links between files.

labels and linking

Some labels will be both defined and used in the same file. Other labels will be defined in one file but used elsewhere as illustrated in Figure 5-9. In the file in which the label is defined, it is known as an **entry point**. In the file where it is used, it is called an **external reference**. The main job of the loader is to *resolve* external references based on available entry points.As a result of the need to know how definitions and references connect, the assembler passes to the linker not only the object file but also the symbol table. Even if the entire symbol table is not kept for later debugging purposes, it must at least pass the entry points; external references are identified in the object code by their relative symbol identifiers.

the linking process

The linker proceeds in two phases. First, it determines the absolute address of the start of each object file. The order in which object files are to be loaded is given by the user, either by specifying parameters when the loader is run or by creating a **load map** file which gives the order in which files are to be placed in memory. Given the order in which files are to be placed in memory and the length of each object file, it is easy to compute the absolute

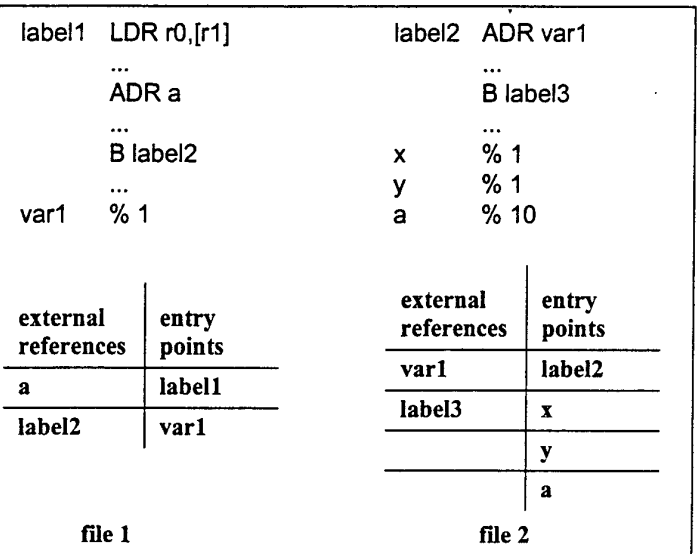


Figure 5-9 External references and entry points.

starting address of each file. At the start of the second phase, the loader merges all the symbol tables from the object files into a single, large table. It then edits the object files to change relative addresses into absolute addresses. This is typically performed by having the assembler write extra bits into the object file to identify the instructions and fields which refer to labels. If a label cannot be found in the merged symbol table, it is undefined and an error message is sent to the user.

placement of data and code

Controlling where code modules are loaded into memory is important in embedded systems. Some data structures and instructions, such as those used to manage interrupts, must be put at precise memory locations for them to work. In other cases, we may have different types of memory installed at different address ranges. For example, if we have EPROM in some locations DRAM in others, we want to make sure that locations that will be written are put in the DRAM locations.

dynamic linking

Workstations and PCs provide **dynamically-linked libraries** and some sophisticated embedded computing environments may provide them as well. Rather than link a separate copy of commonly-used routines such as I/O to every executable program on the system, dynamically-linked libraries allow them to be linked in at the start of program execution. A brief linking process is run just before execution of the program begins; the dynamic linker uses code libraries to link in the required routines. This not only saves storage space but also allows programs that use those libraries to be easily updated. However, it does introduce a delay before the program starts executing.

5.5 Basic Compilation Techniques

It is useful to understand how a high-level language program is translated into instructions. Since implementing an embedded computing system often requires controlling the instruction sequences used to handle interrupts, placement of data and instructions in memory, etc., understanding how the compiler works can help you know when you cannot rely on the compiler. And since many applications are also performance-sensitive, understanding how code is generated can help you meet your performance goals, either by writing high-level code that gets compiled into the instructions you want or by recognizing when you must write your own assembly code.

compilation = translation + optimization

Compilation combines translation and optimization: the high-level language program is translated into the lower-level form of instructions; optimizations try to generate better instruction sequences than would be possible if the brute-force technique of independently translating source-code statements were used. Optimization techniques look at more of the program so that compilation decisions which appear to be good for one statement aren't unnecessarily bad for other parts of the program.

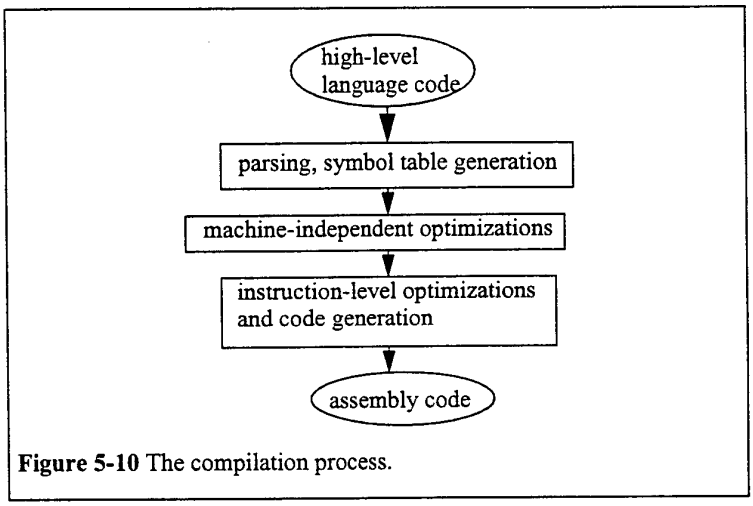


Figure 5-10 The compilation process.

The compilation process is summarized in Figure 5-10. Compilation starts with high-level language code such as C and generally produces assembly code (directly producing object code simply duplicates the functions of an assembler, which is a very desirable stand-alone program to have). The high-level language program is parsed to break it into statements and expressions; a symbol table is also generated which includes all the named objects in the program. Some compilers may then perform some higher-level optimizations which can be viewed as modifying the input high-level language program without reference to instructions. Simplifying arithmetic expressions is one example of a machine-independent optimization. Not all compilers do such optimizations, and compilers can vary widely in which combinations of machine-independent optimizations they do perform. Instruction-level optimizations are aimed at generating code. They may work directly on real instructions or on a pseudo-instruction format which is later mapped onto the

actual instructions of the target CPU. This level of optimization also helps modularize the compiler by allowing code generation to create simpler code that is later optimized. For example, consider this array access code:

```
x[i] = c*x[i];
```

A simple code generator would generate the address for x[i] twice, once for each appearance in the statement. The later optimization phases can recognize this as an example of common expressions that need not be duplicated. While in this simple case it would be possible to create a code generator that never generated the redundant expression, taking into account every such optimization at code generation time is very difficult. We get better code and more reliable compilers by generating simple code first and then optimizing it.

5.5.1 Statement Translation

In this section, we will consider the basic job of translating the high-level language program with little or no optimization. Let's first consider how to translate an expression. A great deal of the code in a typical application consists of arithmetic and logical expressions. Understanding how to compile a single expression, as described in Example 5-2, is a good first step in understanding the complete compilation process.

Example 5-2 Compiling an arithmetic expression

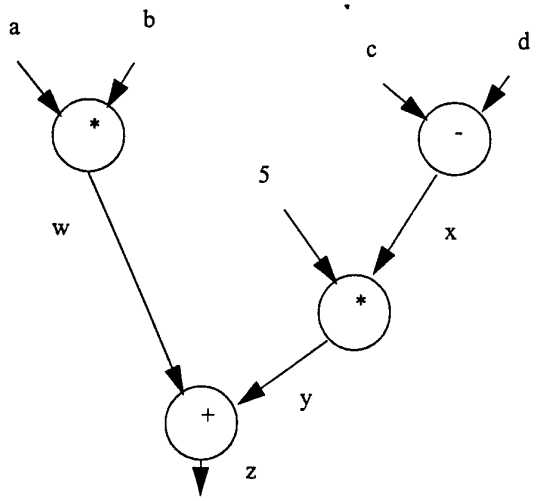
Here is an arithmetic expression:

```
a*b + 5*(c-d)
```

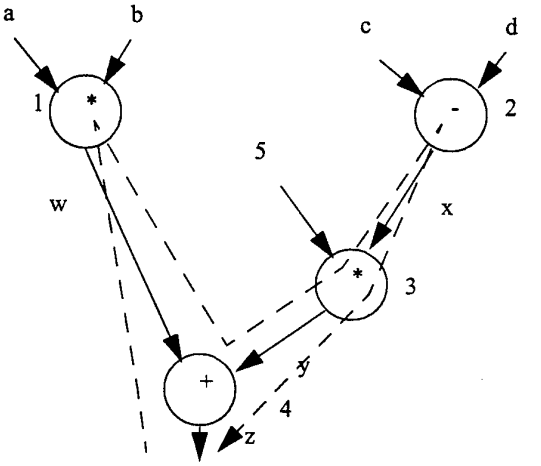
The variable is written in terms of program variables. In some machines we may be able to perform memory-to-memory arithmetic directly on the locations corresponding to those variables. However, in many machines, such as the ARM, we must first load the variables into registers. This requires choosing which registers receive not only the named variables but also intermediate results like (c-d).

code generation by DFG

The code for the expression can be built by walking the data flow graph. The data flow graph for the expression looks like this:



The temporary variables for the intermediate values and final result have been named w, x, y, and z. To generate code, we walk from the tree's root (where z, the final result is generated) by walking the nodes in post-order. During the walk, we generate instructions to cover the operation at every node. The path looks like this:



The nodes are numbered in the order in which code is generated. Since every node in the data flow graph corresponds to an operation which is directly supported by the instruction set, we simply generate an instruction at every node. Since we are making an arbitrary register assignment, we can use up the registers in order starting with r1. The resulting ARM code is

```
; operator 1 (+)
ADR r4,a        ; get address for a
MOV r1,[r4]     ; load a
```

```
ADR r4,b        ; get address for b
MOV r2,[r4]     ; load b
ADD r3,r1,rR2   ; put w into R3
; operator 2 (-)
ADR r4,c        ; get address for c
MOV r4,c        ; load c
ADR r4,d        ; get address for d
MOV r5,[r4]     ; load d
SUB r6,r4,r5    ; put x into R6
; operator 3 (*)
MUL r7,r6,#5    ; operator 3, puts y into R7
; operator 4 (+)
ADD r8,r7,r3    ; operator 4, puts z into R8
```

The equivalent SHARC code looks like this:

```
! operator 1 (+)
R1 = DM(a);             ! load a
R2 = DM(b);             ! load b
R3  = R1 + R2;          ! compute a+b
! operator 2 (-)
R4 = DM(c);             ! load c
R5 = DM(d);             ! load d
R6 = R4 - R5;           ! compute c - d
! operator 3 (*)
R7 = 3;
R8 = R6 * R7;           ! compute y
! operator 4 (+)
R9 = R3 + R8;           ! compute z
```

register allocation

One obvious optimization is to reuse a register whose value is no longer needed. In the case of the intermediate values w, x, and y, we know that they cannot be used after the end of the expression (in another expression, for example) since they have no name in the C program. However, the final result z may in fact be used in a C assignment and the value re-used later in the program. In this case we would need to know when the register is no longer needed to determine its best use.

In the above example, we made an arbitrary allocation of variables to registers for simplicity. When we have large programs with multiple expressions, we must allocate registers more carefully since CPUs have a limited number of registers. We will consider register allocation in Section 5.5.8.

We also need to be able to translate control structures. Since conditionals are controlled by expressions, the code generation techniques of the last example can be used for those expressions, leaving us with the task of generating code for the flow of control itself. Figure 5-11 shows a simple example of changing flow of control in C—an if statement, in which the condition controls whether the true or false branch of the if is taken. Figure 5-11 also shows the control flow diagram for the if statement.

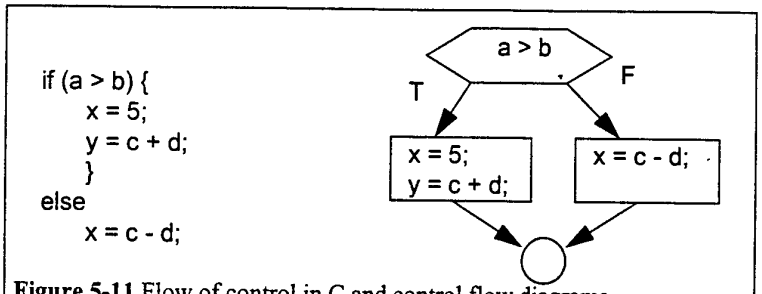


Figure 5-11 Flow of control in C and control flow diagrams.

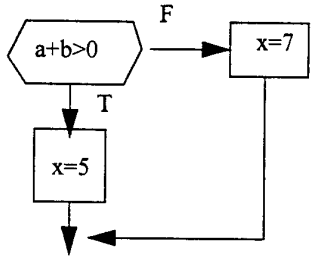
Example 5-3 illustrates how to implement conditionals in assembly language.

Example 5-3 Generating code for a conditional

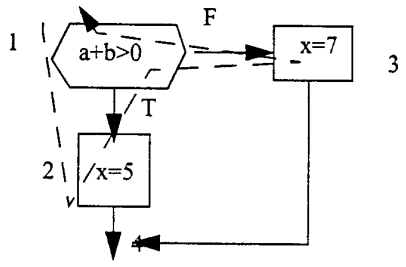
Consider this C statement:

```
if (a+b > 0)
    x = 5;
else
    x = 7;
```

The CDFG for the statement looks like this:



We know how to generate the code for the expressions. We can generate the control flow code by walking the CDFG. Here is one ordered walk through the CDFG:



To generate code, we must assign a label to the first instruction at the end of a directed edge and create a branch for each edge that does not go to the

immediately following instruction. The exact steps to be taken at the branch points depends on the target architecture: on some machines, evaluating expressions generates condition codes that we can test in subsequent branches; on other machines we must use test-and-branch instructions. ARM allows us to test condition codes, so we get this ARM code for the 1-2-3 walk:

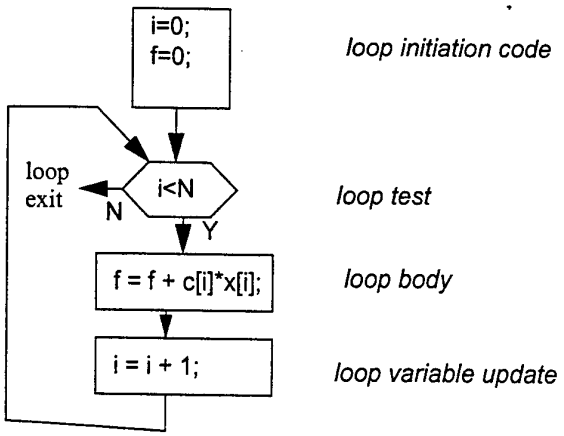
```
                ADR r5, a    ; get address for a
                LDR r1,[r5]  ; load a
                ADR r5, b    ; get address for b
                LDR r2,b     ; load b
                ADD r3,r1,r2
                BLE label3   ; true condition falls through branch
; true case
                LDR r3,#5    ; load constant
                ADR r5, x
                STR r3, [r5] ; store value into x
                B stmtend    ; done with the true case
; false case
label3          LDR r3,#7    ; load constant
                ADR r5, x    ; get address of x
                STR r3,[r5]  ; store value into x
stmtend         ...
```

The 1-2 and 3-4 edges does not require a branch and label because it is straight-line code; the 1-3 and 2-4 edges do require a branch and a label for the target.

Since expressions are generally created as straight-line code, they generally require careful consideration of the order in which the operations are executed. We have much more freedom when generating conditional code because the branches make sure that the flow of control goes to the right block of code. If we walk the CDFG in a different order and lay out the code blocks in a different order in memory, we still get valid code so long as we properly place branches.

loops

Drawing a control flow graph based on the while form of the loop helps us understand how to translate it into instructions:



Web Enhanced

C compilers can generate (using the `-s` flag) assembler source, which some compilers intersperse with the C code. Such code is a very good way to learn about both assembly language programming and compilation. Web Aid 5-Assembly has examples of assembly output from C compilers for the ARM and SHARC.

5.5.2 Procedures

Another major code generation problem is the creation of procedures. Generating code for procedures is relatively straightforward once we know the procedure linkage appropriate for the CPU. At the procedure definition, we generate the code to handle the procedure call and return; at each call of the procedure, we set up the procedure parameters and make the call.

procedure linkage

The CPU's subroutine call mechanism is usually not sufficient to directly support procedures in modern programming languages. We introduced the procedure stack and procedure linkages in Section 2.3.3. The linkage mechanism provides a way for the program to pass parameters into the program and for the procedure to return a value. It also provides help in restoring the values of registers that the procedure has modified. All procedures in a given programming language use the same linkage mechanism (though different languages may use different linkages); the mechanism can also be used to call hand-written assembly language routines from compiled code.

Procedure stacks are typically built to grow down from high addresses; a **stack pointer** (*sp*) defines the end of the current frame, while a **frame pointer** (*fp*) defines the end of the last frame. (The *fp* is technically necessary only if the stack frame can be grown by the procedure during execution.) The procedure can refer to an element in the frame by addressing relative to sp. When a new procedure is called, the *sp* and *fp* are modified to push another frame onto the stack.

ARM procedure linkage

The ARM Procedure Call Standard (APCS) is a good illustration of a typical procedure linkage mechanism. Although the stack frames are in main memory, understanding how registers are used is a key to understanding the mechanism:

- R0-R3 are used to pass parameters into the procedure. r0 is also used to hold the return value. If more than four parameters are required, they are put on the stack frame.
- R4-R7 hold register variables.
- R11 is the frame pointer and R13 is the stack pointer.
- R10 holds the limiting address on stack size, which is used to check for stack overflows.

Other registers have additional uses in the protocol.

5.5.3 Data Structures

The compiler must also translate references to data structures into references into raw memories. In general, this requires address computations. Some of these computations can be done at compile time while others must be done at run time.

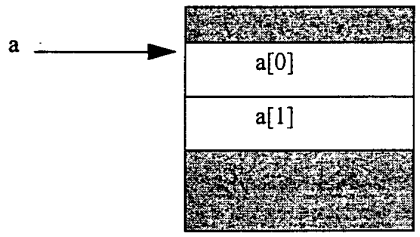


Figure 5-12 Layout of a one-dimensional array in memory.

arrays

Arrays are interesting because the address of an array element must in general be computed at run time, since the array index may change. Let us first consider one-dimensional arrays:

```
a[i]
```

The layout of the array in memory is shown in Figure 5-12: the zero-th element is stored as the first element of the array, the first element directly below, etc. We can create a pointer for the array that points to the array's head, namely a[0]. If we call that pointer aptr for convenience, then we can rewrite the reading of a[i] as

```
*(aptr + i)
```

Two-dimensional arrays are more challenging. There are different possible ways to layout a two-dimensional array in memory, as shown in Figure 5-13. In this form, which is known as **row major**, the inner variable of the array (j in a[i,j]) varies most quickly. (Fortran used a different organization known as

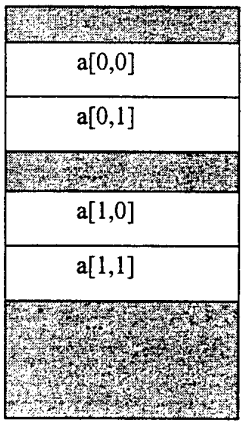


Figure 5-13 Memory layout for two-dimensional arrays.

column major). Two-dimensional arrays also require more sophisticated addressing—in particular, we must know the size of the array. Let us consider the row-major form. If the a[] array is of size $N \times M$, then we can turn the two-dimensional array access into a one-dimensional array access:

a[i,j]

becomes

a[i*M + j]

where the maximum value for j is M-1.

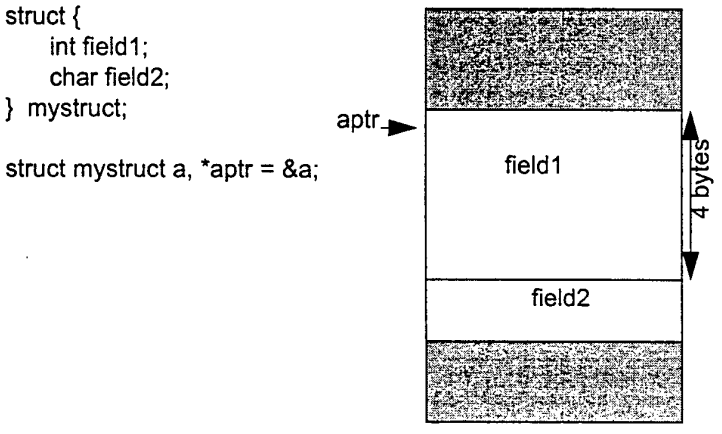


Figure 5-14 C structure layout and access.

C struct

A C struct is easier to address. As shown in Figure 5-14, a structure is implemented as a contiguous block of memory. Fields in the structure can be

accessed using constant offsets to the base address of the structure. In this example, if field1 is four bytes long, then field2 can be accessed as

```
*(aptr + 4)
```

This addition can usually be done at compile time, requiring only the indirection itself to fetch the memory location during execution.

5.5.4 Expression Simplification

Expression simplification is a useful area for machine-independent transformations. We can use the laws of algebra to simplify expressions. Consider this expression:

```
a*b + a*c
```

We can use the distributive law to rewrite it as

```
a*(b+c)
```

Since the new expression has only two operations rather than three for the original form, it is almost certainly cheaper—both faster and smaller. Such transformations make some broad assumptions about the relative cost of operations. In some cases, simple generalizations about the cost of operations may be misleading. For example, a CPU with a multiply-and-accumulate instruction may be able to do a multiply and addition as cheaply as it can do an addition. However, such situations can often be taken care of in code generation.

We can also use the laws of arithmetic to further simplify expressions on constants. Consider this C statement:

```
for (i=0; i<8+1; i++)
```

We can simplify 8+1 to 9 at compile time—there is no need to perform that arithmetic while the program is executing. Why would a program ever have expressions which evaluate to constants? Using named constants rather than numbers is good programming practice and often leads to constant expression. The original form of the for statement could have been

```
for (i=0; i<NOPS+1; i++)
```

where the added 1 takes care of a trailing null character, for example.

```
int foo(a,b,c) { return a + b - c;}

        function definition

z = foo(w,x,y);

        function call


z = w + x - y;

        inlining result
```

Figure 5-15 Function inlining in C.

5.5.5 Dead Code Elimination

Code that will never be executed can be safely removed from the program. The general problem of identifying code that will never be executed is difficult, but there are some important special cases where it can be done.

There are some situations in which programmers will intentionally introduce dead code. Consider this C code fragment:

```
#define DEBUG 0
...
if (DEBUG) print_debug_stuff();
```

In this case, the print_debug_stuff() function is never executed, but the code allows the programmer to override the preprocessor variable definition (perhaps with a compile-time flag) to enable the debugging code. This case is easy to analyze because the condition is the constant 0, which C uses for the false condition. Since there is no else clause in the if statement, the compiler can totally eliminate the if statement, rewriting the CDFG to give a direct edge between the statements before and after the if.

Some dead code may be introduced by the compiler. For example, some optimizations introduce copy statements that copy one variable to another. If uses of the first variable can be replaced by references to the second one, then the copy statement becomes dead code that can be eliminated.

5.5.6 Procedure Inlining

Another machine-independent transformation that requires a little more evaluation is procedure inlining. An inlined procedure does not have a separate procedure body and procedure linkage; rather, the body of the procedure is substituted in place for the procedure call. Figure 5-15 shows an example of function inlining in C. The C++ programming language provides an inline

construct which tells the compiler to generate inline code for a function. In this case, an inlined procedure is always generated in expanded form whenever possible. However, inlining is not always the best thing to do. It does eliminate the procedure linkage instructions. However, in a cached system, having multiple copies of the function body may actually slow down the fetches of these instructions. Inlining also increases code size and memory may be precious.

5.5.7 Loop Transformations

Loops are important program structures—although they are compactly described in the source code, they often take up a large fraction of the computation time. Many techniques have been designed to optimize loops.

A simple but useful transformation is known as loop unrolling, which is illustrated in Example 5-4. Loop unrolling is important because it helps expose parallelism that can be used by later stages of the compiler.

Example 5-4 Loop unrolling

Here is a simple loop in C:

```
for (i=0; i<N; i++) {
    a[i]=b[i]*c[i];
}
```

This loop is executed a fixed number of times, namely N. A straightforward implementation of the loop would create and initialize the loop variable i, update its value on every iteration, and test it to see whether to exit the loop. However, since the loop is executed a fixed number of times, we can generate more direct code. If we let N=4, then we can substitute this C code for the loop:

```
a[0]=b[0]*c[0];
a[1]=b[1]*c[1];
a[2]=b[2]*c[2];
a[3]=b[3]*c[3];
```

This unrolled code has no loop overhead code at all—no iteration variable and no tests. But the unrolled loop has the same problems as the inlined procedure: it may interfere with the cache and it explodes the amount of code required.

We do not, of course, have to fully unroll loops. Rather than unroll the above loop four times, we could unroll it twice. We would then have this code:

```
for (i=0; i<2; i++) {
    a[i*2]=b[i*2]*c[i*2];
    a[i*2+1]=b[i*2+1]*c[i*2+1];
    }
```

In this case, since the all the operations in the two lines of the loop body are independent, later stages of the compiler may be able to generate code that allows them to be executed efficiently on the CPU's pipeline.

Loop fusion combines two or more loops into a single loop. For this transformation to be legal, two conditions must be satisfied. First, the loops must iterate over the same values. Second, the loop bodies must not have any dependencies that would be violated if they are executed together—for example, if the second loop's i^{th} iteration depends on the results of the $i+1^{th}$ iteration of the first loop, the two loops cannot be combined. **Loop distribution** is the opposite of loop fusion—decomposing a single loop into a multiple loops.

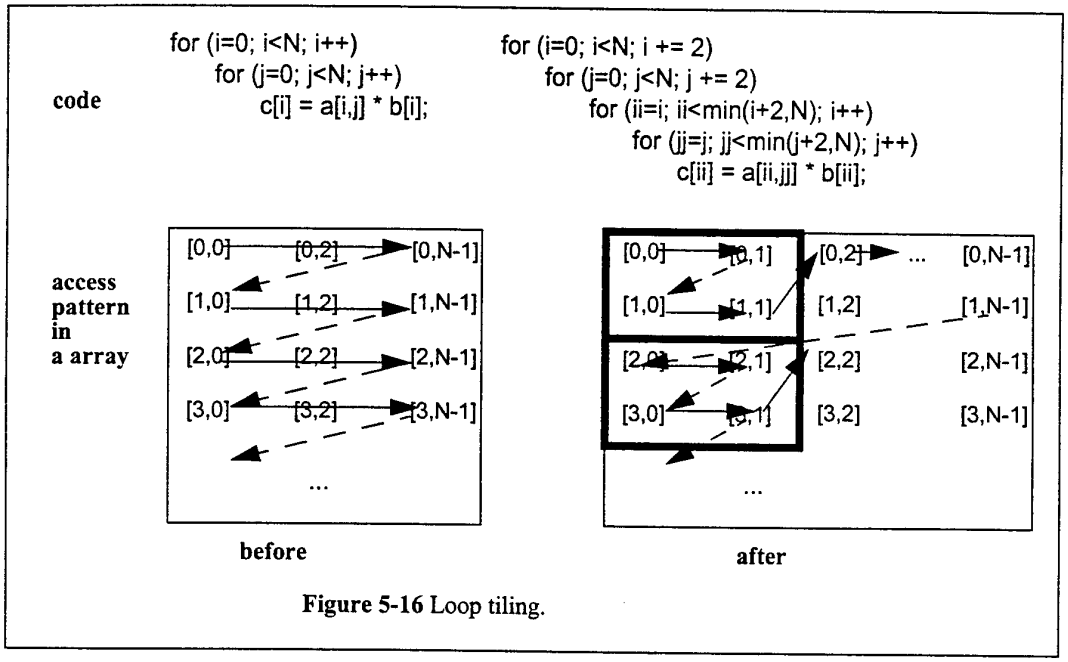


Figure 5-16 Loop tiling.

Loop tiling breaks up a loop into a set of nested loops, with each inner loop performing the operations on a subset of the data. An example is shown in Figure 5-16. Here, each loop is broken up into tiles of size two. Each loop is split into two loops—for example, the inner ii loop iterates within the tile and the outer i loop iterates across the tiles. The result is that the pattern of accesses across the a array is drastically different—rather than walking across one row in its entirety, the code walks through rows and columns following the tile structure. Loop tiling changes the order in which array elements are accessed, thereby allowing us to better control the behavior of the cache during loop execution.

We can also modify the arrays being indexed in loops. **Array padding** adds dummy data elements to a loop in order to change the layout of the array in the cache. Although these array locations will not be used, they do change

how the useful array elements fall into cache lines. Judicious padding can in some cases significantly reduce the number of cache conflicts during loop execution.

5.5.8 Register Allocation

Register allocation is a very important compilation phase. Given a block of code, we want to choose assignments of variables (both declared and temporary) to registers to minimize the total number of required registers. The next example illustrates the importance of proper register allocation.

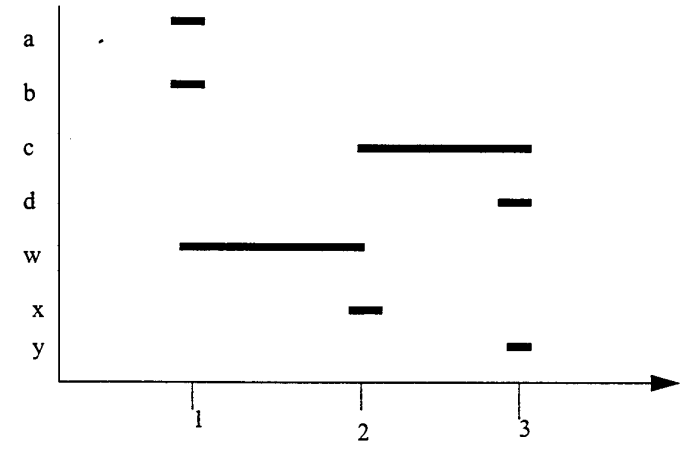
Example 5-5 Register allocation

To keep the example small, we will assume that we can use only four of the ARM's registers. In fact, such a restriction is not unthinkable—programming conventions may reserve certain registers for special purposes and significantly reduce the number of general-purpose registers available.

Consider this C code:

```
w = a + b; /* statement 1 */
x = c + w; /* statement 2 */
y = c + d; /* statement 3 */
```

A naive register allocation, assigning each variable to a separate register, would require seven registers for the seven variables. However, we can do much better by reusing a register once the value stored in the register is no longer needed. To understand how to do this, we can draw a lifetime graph that shows the statements on which each statement is used. Here is a **lifetime graph** in which the x axis is statement number in the C code and the y axis shows the variables:



A horizontal line stretches from the first statement where the variable is used to the last use of the variable; a variable is said to be **live** during this interval.

At each statement, we can determine every variable that is currently in use. The maximum number of variables in use at any statement determines the maximum number of registers required. In this case, statement two requires three registers: c, w, and x. This fits within our four registers limitation. By reusing registers once their current values are no longer needed, we can write code which needs no more than four registers. Here is one register assignment:

a	**r0**
b	**r1**
c	**r2**
d	**r0**
w	**r3**
x	**r0**
y	**r3**

And here is ARM assembly code that uses this register assignment:

```
LDR R0,[p_a]        ; load a into R0 using pointer to a (p_a)
LDR R1,[p_b]        ; load b into R1
ADD R3,R0,R1        ; compute a+b
STR R3,[p_w]        ; w = a + b
LDR R2,[p_c]        ; load c into R2
ADD R0,R2,R3        ; compute c+w, reusing R0 for x
STR R0,[p_x]        ; x = c + w
LDR R0, [p_d]       ; load d into R0
ADD R3,R2,R0        ; compute c+d, reusing R3 for y
STR R3,[p_y]        ; y = c + d
```

If a section of code requires more registers than are available, we must **spill** some of the values out to memory temporarily: after computing some values, we write the values to temporary memory locations, reuse those registers in other computations, and then re-read the old values from the temporary locations to resume work. Spilling registers is bad in several respects: it requires extra CPU time and it uses up both instruction and data memory. It is worth putting effort into register allocation to avoid unnecessary register spills.

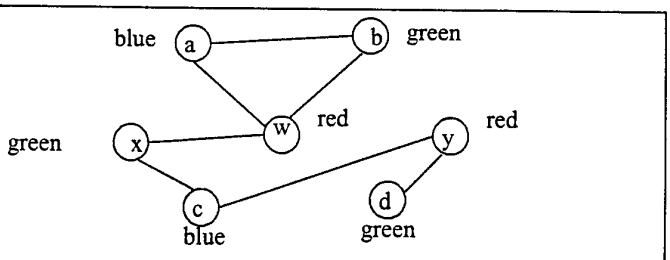


Figure 5-17 Using graph coloring to solve the problem of Example 5-5.

register allocation algorithm

We can solve register allocation problems by building a **conflict graph** and solving a graph coloring problem. As shown in Figure 5-17, each variable in the high-level language code is represented by a node. An edge is added between two nodes if they are both live at the same time. The graph coloring problem is to use the smallest number of distinct colors to color all the nodes such that no two nodes that are directly connected by an edge have the same color. The figure shows a satisfying coloring that uses three colors. Graph coloring is NP-complete, but there are efficient heuristic algorithms which can give good results on typical register allocation problems.

scheduling and register allocation

Lifetime analysis assumes that we have already determined the order in which we will evaluate operations. In many cases, we have freedom in the order in which we do things. Consider this expression:

```
(a + b) * (c - d)
```

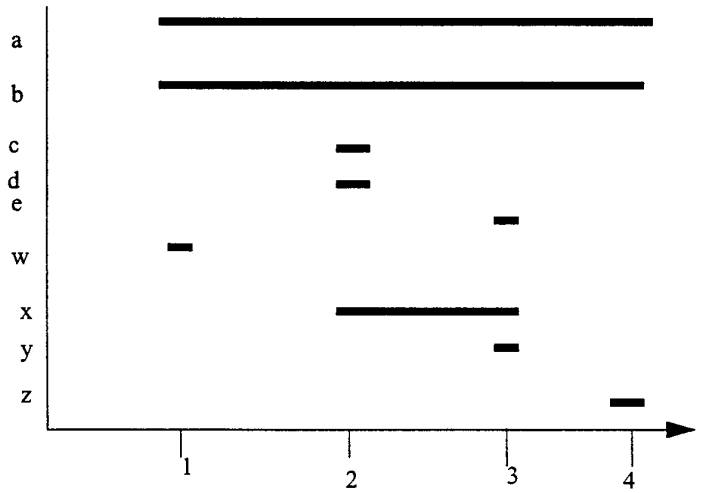
We have to do the multiplication last, but we can do the either the addition or subtraction first. Different orders of loads, stores, and arithmetic operations may also result in different execution times on pipelined machines. If we can keep values in registers without having to re-read them from main memory, we can save execution time and reduce code size as well. The next example illustrates how proper **operator scheduling** can improve register allocation.

Example 5-6 Operator scheduling for register allocation

Here is sample C code fragment:

```
w = a + b; /* statement 1 */
x = c + d; /* statement 2 */
y = x + e; /* statement 3 */
z = a - b; /* statement 4 */
```

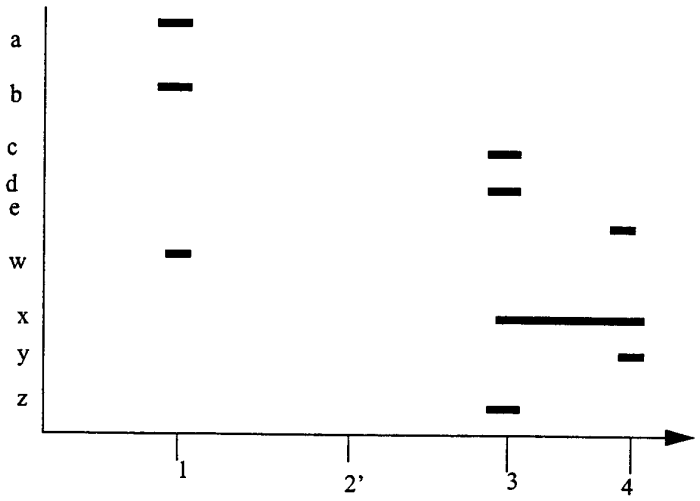
If we compile the statements in the order in which they were written, we get this register graph:



Since w is needed until the last statement, we need five registers at statement 3, even though only three registers are needed for the statement at line 3. If we swap statements 3 and 4 (renumbering them 3' and 4'), reduce our requirements to three registers. Here is the modified C code:

```
w = a + b; /* statement 1 */
z = a - b; /* statement 2' */
x = c + d; /* statement 3' */
y = x + e; /* statement 4' */
```

And here is the lifetime graph for the new code:



Compare the ARM assembly code for these two code fragments. We have written both assuming that we have only four free registers. In the before version, we do not have to write out any values, but we must read a and b twice. The after version allows us to keep all values in registers as long as we need them.

before version

```
LDR r0,a
LDR r1,b
ADD r2,r0,r1
STR r2,w ; w = a + b
LDRr r0,c
LDR r1,d
ADD r2,r0,r1
STR r2,x ; x = c + d
LDR r1,e
ADD r0,r1,r2
STR r0,y ; y = x + e
LDR r0, a ; reload a
LDR r1, b ; reload b
SUB r2,r1,r0
STR r2,z ; z = a - b
```

after version

```
LDR r0,a
LDR r1,b
ADD r2,r1,r0
STR r2,w ; w = a + b
SUB r2,r0,r1
STR r2,z ; z = a - b
LDR r0,c
LDR r1,d
ADD r2,r1,r0
STR r2,x ; x = c + d
LDR r1,e
ADD r0,r1,r2
STR r0,y ; y = x + e
```

5.5.9 Scheduling

We have some freedom to choose the order in which operations will be performed. We can use this to our advantage—for example, we may be able to improve our register allocation by changing the order in which operations are performed, thereby changing the lifetimes of the variables.

We can solve scheduling problems by keeping track of resource utilization over time. We do not have to know the exact microarchitecture of the CPU—all we have to know is that for example, instruction types 1 and 2 both use resource A while instruction types 3 and 4 use resource B. CPU manufacturers generally disclose enough information about the microarchitecture to allow us to schedule instructions even when they do not give a detailed description of the CPU's internals.

time	resource A	resource B
t	X	
t+1	X	X
t+2	X	
t+3		X

Figure 5-18 A reservation table for instruction scheduling.

We can keep track of CPU resources during instruction scheduling using a **reservation table** [Kog81]. Rows in the table represent instruction execution time slots and columns represent resources which must be scheduled. Before scheduling an instruction to be executed at a particular time, we check the reservation table to be sure that all resources needed by that instruction are available at that time. Upon scheduling the instruction, we update the table to note all the resources used by that instruction. A variety of algorithms can be used for the scheduling itself, depending on the types of resources and instructions involved, but the reservation table provides a good summary of the state of an instruction scheduling problem in progress.

software pipelining

We can also schedule instructions to maximize performance. As we know from Section 3.6, when an instruction that takes more cycles than normal to finish is in the pipeline, pipeline bubbles appear that reduce performance. **Software pipelining** is a technique for reordering instructions across several loop iterations to reduce pipeline bubbles. Software pipelining is illustrated in Example 5-7.

Example 5-7 Software pipelining in SHARC

Software pipelining can be illustrated with a small loop on the SHARC. Consider a simple loop for a dot product computation:

```
for (i=0; i<N; i++)
    sum += a[i] * b[i];
```

The SHARC can perform several operations in parallel. However, we can't perform the necessary loads and arithmetic operations on the same cycle.

We want to rewrite the loop so that we perform two loads, an addition, and a multiplication in one iteration. However, because the result of one operation depends on others, we can't do all these operations for the same iteration at the same time. Rather, the loop body will perform operations from three different iterations:

1. The two fetches of the array elements are performed for availability in the next cycle.
2. The multiplication a[i]*b[i] is performed on the operands fetched by the previous loop iteration.
3. The addition into the dot product running sum is performed using the result of the multiplication performed in the previous loop iteration.

When we rewrite the loop, we need to generate special header and trailer code that takes care of the first and last iterations that cannot be pipelined. This C code is designed to show what operations can be performed in parallel on the SHARC:

```
/* first iteration performed outside loop */
ai = a[0]; bi = b[0]; p = ai*bi;
/* initiate loads used in second iteration; remaining
    loads will be performed inside the loop */
for (i=2; i<N-2; i++) {
    ai = a[i]; bi = b[i]; /* fetch for use in next cycle's multiply */
    p = ai*bi; /* multiply for next iteration's sum */
    sum += p; /* compute sum using p computed in last iteration */
}
/* trailer code */
sum += p; p = ai*bi;
sum += p;
```

5.5.10 Instruction Selection

Selecting which instructions to use to implement each operation is non-trivial. There may be several different instructions that can be used to accomplish the same goal, but they may have different execution times and using one instruction for one part of the program may affect what instructions can

be used in adjacent code. While we can't discuss all the problems and methods for code generation here, but a little bit of knowledge helps us envision what the compiler is doing.

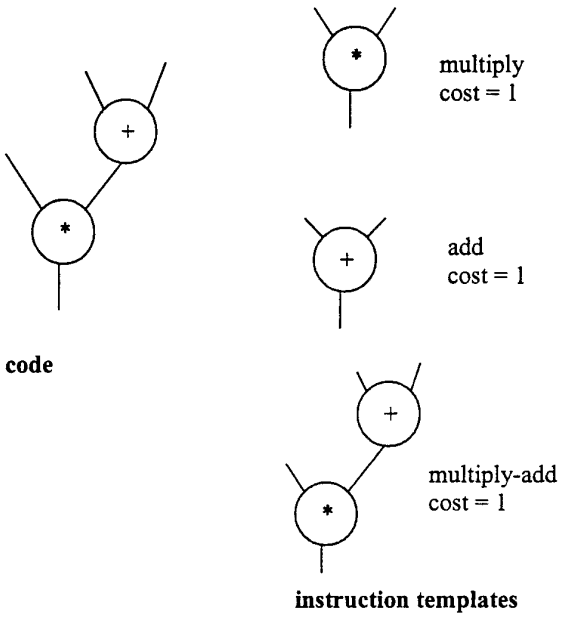


Figure 5-19 Code generation by template matching.

template matching

One useful technique for generating code is **template matching**, illustrated in Figure 5-19. We have a DAG that represents the expression we want to generate code for. In order to be able to match up instructions and operations, we represent instructions using the same DAG representation; we've shaded the instruction template nodes to distinguish them from code nodes. Each node has a cost, which may be simply the execution time of the instruction or may include factors for size, power consumption, etc. In this case, we've shown that each instruction takes the same amount of time and so all have a cost of 1. Our goal is to cover all of the nodes in the code DAG with instruction DAGs—until we have covered the code DAG we haven't generated code for all the operations in the expression. In this case, the lowest-cost covering uses the multiply-add instruction to cover both nodes. If we first tried to cover the bottom node with the multiply instruction, we would find ourselves blocked from using the multiply-add instruction; dynamic programming can be used to efficiently find the lowest-cost covering of trees and heuristics can extend the technique to DAGs.

5.5.11 Understanding and Using Your Compiler

Clearly, the compiler can vastly transform your program during the creation of assembly language. But compilers also differ greatly in what optimiza-

tions they do. Understanding your compiler can help you get the best code out of it.

Studying the assembly language output of the compiler is a good way to learn about what the compiler does. Some compilers will annotate sections of code to help you make the correspondence between the source and assembler output. Starting with small examples that exercise only a few types of statements will help. You can experiment with different optimization levels (the -O flag on most C compilers). You can also try writing the same algorithm in several ways to see how the compiler's output changes.

If you can't get your compiler to generate the code you want, you may need to write your own assembly language. You can do this in two ways: you can write it from scratch or you can modify the output of the compiler. If you write your own assembly code, you must make sure that it conforms to all the compiler conventions, such as procedure call linkage. If you modify the compiler output, you should be sure that you have the algorithm right before you start writing code so that you don't have to repeatedly edit the compiler's assembly language output. You also need to clearly document the fact that the high-level language source is, in fact, not the code used in the system.

5.5.12 Interpreters and JIT Compilers

Programs are not always compiled and then separately executed. In some cases, it may make sense to translate the program into instructions during execution. Two well-known techniques for on-the-fly translation are **interpretation** and **just-in-time (JIT) compilation**. The trade-offs for both techniques are similar. Interpretation or JIT compilation adds overhead—both time and memory—to execution. However, that overhead may be more than made up for in some circumstances. For example, if only parts of the program are executed over some period of time, interpretation or JIT compilation may save memory, even taking overhead into account. Interpretation and JIT compilation also provide added security when programs arrive over the network.

interpretation

An interpreter translates program statements one-at-a-time. The program may be expressed in a high-level language, with Forth being a prime example of an embedded language that is interpreted. An interpreter may also interpret instructions in some abstract machine language. As illustrated in Figure 5-20, the interpreter sits between the program and the machine. It translates one statement of the program at a time. The interpreter may or may not generate an explicit piece of code to represent the statement. Because the interpreter translates only a very small piece of the program at any one time, not much memory is used to hold intermediate representations of the program. A Forth program plus the Forth interpreter is in many cases smaller than the equivalent native machine code.

JIT compilers

JIT compilers have been used for quite some time, but are best known today for their use in Java environments [Cra97]. A JIT compiler is somewhere between an interpreter and a stand-alone compiler. A JIT compiler actually

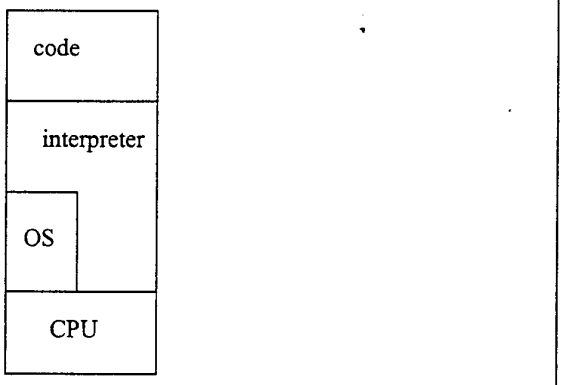


Figure 5-20 Structure of a program interpretation system.

produces executable code segments for pieces of the program. However, it compiles a section of the program (such as a function) only when it knows it will be executed. Unlike an interpreter, it saves the compiled version of the code so that the code does not have to be re-translated the next time it is executed. A JIT compiler saves some execution time overhead relative to an interpreter because it does not translate the same piece of code multiple times, but it also uses more memory for the intermediate representation. The JIT compiler usually generates machine code directly, rather than build intermediate program representation data structures like the CDFG. A JIT compiler also usually performs only simple optimizations as compared to a stand-alone compiler.

5.6 Analysis and Optimization of Execution Time

Because embedded systems must perform functions in real time, we often need to know how fast a program runs. The techniques we use to analyze program execution time are also helpful in analyzing properties like power consumption. In this section, we will study how to analyze programs to estimate their run times. We will also look at how to optimize programs to improve their execution times; of course, optimization relies on analysis.

run times will vary

Note the word *estimate* in the last sentence. While one might hope that execution time of programs could be precisely determined, this is in fact difficult to do in practice. There are several reasons:

- The execution time of a program often varies with the input data values because those values select different execution paths in the program. For example, loops may be executed varying number of times and different branches may execute blocks of varying complexity.
- The cache has a major effect on program performance, and once again, the cache's behavior depends in part on the data values input to the program.

- Execution times may vary even at the instruction level. Floating-point operations are the most sensitive to data values, but the normal integer execution pipeline can also introduce data-dependent variations. In general, the execution time of an instruction in a pipeline depends not only on that instruction but on the instructions around it in the pipeline.

measuring execution speed

We can measure program performance in several ways:

- Some microprocessor manufacturers supply simulators for their CPUs: the simulator runs on a workstation or PC, takes as input an executable for the microprocessor along with input data, and simulates the execution of that program. Some of these simulators go beyond functional simulation to measure the execution time of the program. Simulation is clearly slower than executing the program on the actual microprocessor, but it also provides much greater visibility during execution. Be careful—some microprocessor performance simulators are not 100% accurate and simulation of I/O-intensive code may be difficult.
- A timer connected to the microprocessor bus can be used to measure performance of executing sections of code. The code to be measured would reset and start the timer at its start and stop the timer at the end of execution. The length of the program that can be measured is limited by the accuracy of the timer.
- A logic analyzer can be connected to the microprocessor bus to measure the start and stop times of a code segment. This technique relies on the code being able to produce identifiable events on the bus to identify the start and stop of execution. The length of code that can be measured is limited by the size of the logic analyzer's buffer.

program performance metrics

We are interested in three different types of performance measures on programs:

- **average-case execution time** This is the typical execution time we would expect for typical data. Clearly, the first challenge is defining typical inputs.
- **worst-case execution time** The longest time that the program can spend on any input sequence is clearly important for systems that must meet deadlines. In some cases, the input set that causes the worst-case execution time is obvious, but in many cases it is not.
- **best-case execution time** This measure can be important in multi-rate real-time systems, as we will see in Chapter 6.

First, we will look at the fundamentals of program performance in more detail. We will then consider trace-driven based on executing the program and observing its behavior.

5.6.1 Elements of Program Performance

the program performance equation

The key to evaluating execution time is breaking the performance problem into parts. Program execution time may be seen as [Sha89]

$$execution\ time = program\ path + instruction\ timing$$

The path is the sequence of instructions executed by the program (or its equivalent in the high-level language representation of the program). The instruction timing is determined based on the sequence of instructions traced by the program path; this takes into account data dependencies, pipeline behavior, and caching. Luckily, these two problems can be solved relatively independently.

Although we can trace the execution path of a program through its high-level language specification, it is hard to get accurate estimates of total execution time from a high-level language program. This is because there is not, as we saw in Section 5.5, a direct correspondence between program statements and instructions: the number of memory locations and variables must be estimated, results may be either saved for reuse or recomputed on-the-fly; and other effects. These problems become more challenging as the compiler puts more and more effort into optimizing the program. However, some aspects of program performance can be estimated by looking directly at the C program. For example, if a program contains a loop with a large, fixed iteration bound or if one branch of a conditional is much longer than another, we can get at least a rough idea that these are more time-consuming segments of the program.

Of course, a precise estimate of performance also relies on the instructions to be executed, since different instructions take different amounts of time. (And, to make life even more difficult, the execution time of one instruction can depend on the instructions executed before and after it.) The next example illustrates data-dependent program paths.

Example 5-8 Data-dependent paths in if statements

Here is a pair of nested if statements:

```
if (a || b) { /* test 1 */
    if (c) /* test 2 */
        { x = r * s + t; /* assignment 1 */ }
    else { y = r + s; /* assignment 2 */ }
    z = r + s + u; /* assignment 3 */
} else {
    if (c) /* test 3 */
        { y = r - t; /* assignment 4 */ }
}
```

We've labeled the conditional tests and assignments within each if statement to make it easier to identify paths. What execution paths may be exercised? One way to enumerate all the paths is to create a truth table-like structure. The paths are controlled by the variables in the if conditions, namely *a*, *b*, and *c*. For any given combination of values of those variables we can trace through the program to see which branch is taken at each if and which assignments are performed. For example, when $a = 1$, $b = 0$, and $c = 1$, then test 1 is true and test 2 is true. This means we first perform assignment 1, then assignment 3.

Here are the results for all the controlling variable values:

a	b	c	path
0	0	0	test 1 false, test 3 false: no assignments
0	0	1	test 1 false, test 3 true: assignment 4
0	1	0	test 1 true, test 2 false: assignments 2, 3
0	1	1	test 1 true, test 2 true: assignments 1, 3
1	0	0	test 1 true, test 2 false: assignments 2, 3
1	0	1	test 1 true, test 2 true: assignments 1, 3
1	1	0	test 1 true, test 2 false: assignments 2, 3
1	1	1	test 1 true, test 2 true: assignments 1, 3

Notice that there are only four distinct cases: no assignment, assignment 4, assignments 2 and 3, or assignments 1 and 3. These correspond to the possible paths through the nested ifs; the table adds value by telling us which variable values exercise each of these paths.

Enumerating the paths through a fixed-iteration for loop is seemingly simple. In this code:

```
for (i=0; i<N; i++)
    a[i] = b[i]*c[i];
```

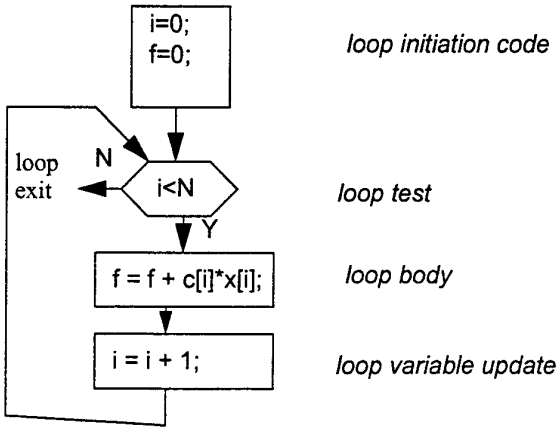
the assignment in the loop is performed exactly N times. However, we can't forget the code executed to set up the loop and to test the iteration variable. Example 5-9 illustrates how to determine the path through a loop.

Example 5-9 Paths in a loop

The loop code for the FIR filter of Application Note 2-1 was:

```
for (i=0, f=0; i<N; i++)
    f = f + c[i] * x[i];
```

By looking at the CDFG for the code we can more easily determine how many times various statements are executed. Here is the CDFG once again:



The CDFG makes it clear that the loop initiation block is executed once, the test is executed N+1 times, and the body and loop variable update are each executed N times.

To measure the longest path length, we must find the longest path through the optimized CDFG since the compiler may change the structure of the control and data flow to optimize the program's implementation. It is important to keep in mind that choosing the longest path through a CDFG as measured by the number of nodes or edges touched may not correspond to the longest execution time. Since the execution time of a node in the CDFG will vary greatly depending on the instructions represented by that node, we must keep in mind that the longest path through the CDFG depends on the execution times of the nodes. In general, it is good policy to choose several of what we estimate are the longest paths through the program and measure the lengths of all of them in sufficient detail to be sure that we have in fact captured the longest path.

instruction timing

Once we know the execution path of the program, we have to measure the execution time of the instructions executed along that path. The simplest estimate is to assume that every instruction takes the same number of clock cycles, which means we need only count the instructions and multiply by the per-instruction execution time to get the program's total execution time. However, even ignoring cache effects, this technique is simplistic:

- **Not all instructions take the same amount of time.** Although RISC architectures tend to provide uniform instruction execution times in order to keep the CPU's pipeline full, even many RISC architectures take different amounts of time to execute certain instructions. Multiple load/store instructions are examples of longer-executing instructions in the ARM architecture. Floating-point instructions show especially wide variances in execution time—while basic multiply and add operations are fast, some transcendental functions can take thousands of cycles to execute.

- **Execution times of instructions are not independent.** The execution time of one instruction depends on the instructions around it. For example, many CPUs use register bypassing to speed up instructions sequences when the result of one instruction is used in the next instruction. As a result, the execution time of an instruction may depend on whether its destination register is used as a source for the next operation (or vise versa).
- **The execution time of an instruction may depend upon operand values.** This is clearly true of floating-point instructions in which a different number of iterations may be required to calculate the result. Other specialized instructions can, for example, perform a data-dependent number of integer operations.

We can handle the first two problems more easily than the third. We can look up instruction execution time in a table; the table will be indexed by opcode and possibly by other parameter values such as which registers are used. To handle interdependent execution times, we can add columns to the table to consider the effects of nearby instructions; since these effects are generally limited by the size of the CPU pipeline, we know that we need to consider a relatively small window of instructions to handle such effects. Handling variations due to operand values is difficult to do without actually executing the program using a variety of data values, given the large number of factors that can affect value-dependent instruction timing. Luckily, these effects are often small. Even in floating-point programs, the majority of the operations are typically additions and multiplications whose execution times have small variances.

caching effects

So far we have not considered the effect of the cache. Because the access time for main memory can be 10 times larger than the cache access time, caching can have huge effects on instruction execution time by changing both the instruction and data access times. Caching performance inherently depends on the program's execution path since the cache's contents depend on the history of accesses. Because of the complexity of this phenomenon, we will consider it in more detail below using both trace-driven and static and dynamic analysis techniques.

5.6.2 Trace-Driven Performance Analysis

A **program trace** (or more succinctly, a **trace**) is a record of the execution path of a program that has been measured during program execution. If we can measure a trace from a running program, we avoid the need to analyze the program's CDFG to determine the execution path. Traces are typically large. Since we usually need to look at the entire execution history of a program, traces can be gigabytes in size. However, because disks to hold the traces are inexpensive and workstations to analyze them are fast and cheap, traces are an appealing way to analyze programs. We have to consider two problems: how to generate a trace and how to use it to determine program performance.

hardware trace generation

A trace can be captured by using hardware methods. For example, a logic analyzer can be attached to the microprocessor bus to capture memory bus traffic. There are two major limitations to this approach. First, the size of hardware trace buffers is usually limited. The hardware must be able to store instructions at execution speed; if we use high-speed RAM for the buffer, we are limited to perhaps several million instructions. This means that we probably cannot capture a trace of the entire program but only a section of its execution. Second, if the CPU has on-chip cache, we will not be able to see internal memory references either to program or data. Alternatively, a microprocessor emulator can be used to trace the PC over time. Although this technique allows us to see what is going on inside the chip and not just behavior on the memory bus, it is too slow to generate long traces. Since the microprocessor must be stopped for probing and probing is very slow compared to normal execution, capturing long traces would require exorbitant amounts of time.

Some CPUs have hardware facilities for automatically generating trace information. For example, the Pentium family microprocessors generate a special bus cycle, a Branch Trace Message, that shows the source and/or destination address of a branch [Col97]. If we record only traces, we can reconstruct the instructions executed within the basic blocks while greatly reducing the amount of memory required to hold the trace.

software trace generation

There are three major methods for generating a trace by software: **PC sampling**, **instrumentation instructions**, and **simulation**. The PC sampling technique uses the operating system or some other facility to periodically interrupt the program and save away the current PC value. This method has the same limitations of any sampling method: if the sampling period is too slow, we may miss important behavior; in particular, the wrong sampling interval can be a multiple of some periodic behavior in the program that could be totally missed. Alternatively, instructions can be added to the program which will write out information to the trace buffer. What instructions are required depends on what information needs to be captured. The simplest case is to add code to the start of each basic block that remembers that program execution went through that point. Traces are often used to analyze cache or superscalar behavior, in which case the instrumentation code may need to save additional information about the state of the CPU. Software-based trace generation methods do add execution overhead time to the program, but this usually does not affect the measurement. Simulation methods interpret the instructions to simulate the CPU; they can generate the required trace information at the same time.

Obtaining a representative trace requires some knowledge of what the program does. Someone needs to supply inputs that properly exercise the program. The users of the program should have knowledge of the types of data typically presented to the program. However, they may not know what data inputs will cause worst-case behavior, so some collaboration between the program designers and users may be necessary. The techniques of Section 5.9 can be used to measure how thoroughly a set of inputs covers the program's control and data flow, giving you an idea of the representativeness your traces.

trace analysis

Once we have a trace, several things can be done with it. At minimum, the trace tells us the execution path of the program. By assigning the proper execution times to the instructions in the trace (taking into account interactions between instructions, etc.) we can determine the total execution time from the trace. In the case of a trace that does not record the complete execution of a program, we will have to infer the trace through the missing pieces, but if the sampling interval was properly chosen the missing program fragments can be easily determined. Traces are often used to analyze cache behavior—it allows us to calculate memory access times throughout the code. This information can be used to select the proper-size cache to be used with a CPU (either by selecting the proper CPU model with an internal cache or by adding external cache). It can also be used to optimize the program to improve its cache behavior.

A trace is generated on a particular CPU. We may want to use the trace to analyze the performance of the program on a different CPU. For example, when selecting which microprocessor to use, we may not want to purchase a complete evaluation and compilation system for each potential CPU just to measure their performance on our programs. There are limits to what we can do in this case, but the trace can still be useful. We can use the trace to determine the execution path through the program, though we have to be careful that the compilers used for the two CPUs perform similar optimizations. We can approximate the execution time on the new CPU, particularly if the instruction sets of the CPUs are relatively simple. Since many RISC processors have similar basic instructions with similar execution times (and because many compilers use only a small fraction of the CPU's instruction set) we can often obtain reasonable estimates of execution time on the new CPU by looking at the trace from another CPU.

5.6.2.1 Loop Optimizations

Loops are important targets for optimization because programs with loops tend to spend a lot of time executing those loops. There are three important techniques in optimizing loops: **code motion**, **induction-variable elimination**, and **strength reduction**.

code motion

Code motion lets us move unnecessary code out of a loop. If a computation's result does not depend on operations performed in the loop body, then we can safely move it out of the loop. Code motion opportunities can arise because programmers may find some computations clearer and more concise when put in the loop body, even though they are not strictly dependent on the loop iterations. A simple example of code motion is also common. Consider this loop:

```
for (i=0; i<N*M; i++) {
    z[i] = a[i] + b[i];
    }
```

The code motion opportunity becomes more obvious when we draw the loop's CDFG as shown in Figure 5-21. The loop bound computation is performed on every iteration during the loop test, even though the result never

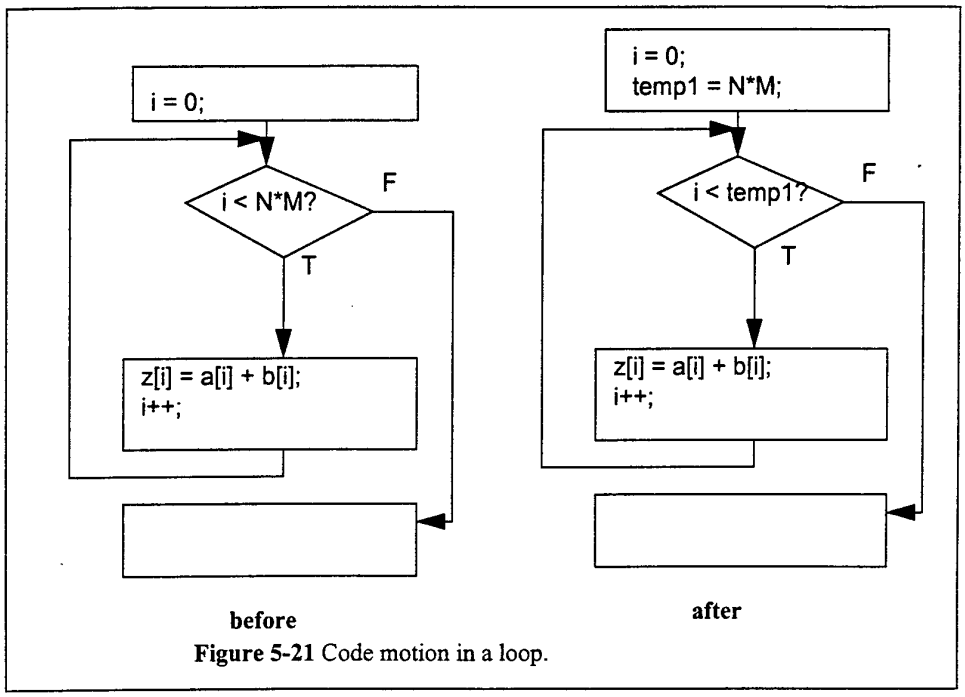


Figure 5-21 Code motion in a loop.

changes. We can avoid $N \times M - 1$ unnecessary executions of this statement by moving it before the loop, as shown in the figure.

induction-variable elimination

An **induction variable** is a variable whose value is derived from the loop iteration variable's value. The compiler often introduces induction variables to help it implement the loop; properly transformed, we may be able to eliminate some and apply strength reduction to others.

A nested loop is a good example of the use of induction variables. Here is a simple nested loop:

```
for (i=0; i<N; i++)
    for (j=0; j<M; j++)
        z[i,j] = b[i,j];
```

The compiler uses induction variables to help it address the arrays. Let us rewrite the loop in C using induction variables and pointers (we will use a common induction variable for the two arrays, even though the compiler would probably introduce separate induction variables and then merge them later):

```
for (i=0; i<N; i++)
    for (j=0; j<M; j++) {
        zbinduct = i*M + j;
        *(zptr + zbinduct) = *(bptr + zbinduct);
        }
```

Here zptr and bptr are pointers to the heads of the z and b arrays and zbinduct is the shared induction variable. However, we do not need to compute

zbinduct afresh each time. Since we are stepping through the arrays sequentially, we can simply add the update value to the induction variable:

```
zbinduct = 0;
for (i=0; i<N; i++) {
    for (j=0; j<M; j++) {
        *(zptr + zbinduct) = *(bptr + zbinduct);
        zbinduct++;
        }
    }
```

This is a form of strength reduction since we have eliminated the multiplication from the induction variable computation.

strength reduction

Strength reduction helps us reduce the cost of a loop iteration. Consider this assignment:

```
y = x * 2;
```

In integer arithmetic, we can use a left shift rather than a multiplication by 2 (so long as we properly keep track of overflows). If the shift is faster than the multiply, we probably want to perform the substitution. This optimization can often be used with induction variables because loops are often indexed with simple expressions. Strength reduction can often be performed with simple substitution rules since there are relatively few interactions between the possible substitutions.

5.6.2.2 Cache Optimizations

A **loop nest** is a set of loops, one inside the other, as illustrated in Figure 5-22. Loop nests occur when we process arrays. A large body of techniques have been developed for optimizing loop nests. Rewriting a loop nest changes the order in which array elements are accessed. This can expose new parallelism opportunities that can be exploited by later stages of the compiler; it can also improve cache performance. In this section we will concentrate on the analysis of loop nests for cache performance.

nested loop model

Let us first consider the examples of Figure 5-22 in more detail. We can represent the set of array locations indexed by the loop by an **iteration space.** Each point in this space is defined by a vector, each component of which is one of the loop's iteration variables. The points in the space represent the iterations actually executed by the program. The top example in the figure accesses every element of the arrays (the a and b matrices have the same access patterns in this case). The lower example accesses bands in the a and b matrices. We can represent any particular point in the iteration space as an **iteration vector**; for example, the vector $i_{3,2} = \langle 3, 2 \rangle$ shown in the figure.

cache miss equations

Ghosh et al [Gho97] developed an analytical technique for describing data cache misses in nested loops. The equations not only describe the set of cache misses but also help us to optimize the program's access patterns. It takes a large number of equations to describe a program, so they are designed to be generated and solved by design tools. Here we will concentrate on the basic principles underlying the equations. By understanding

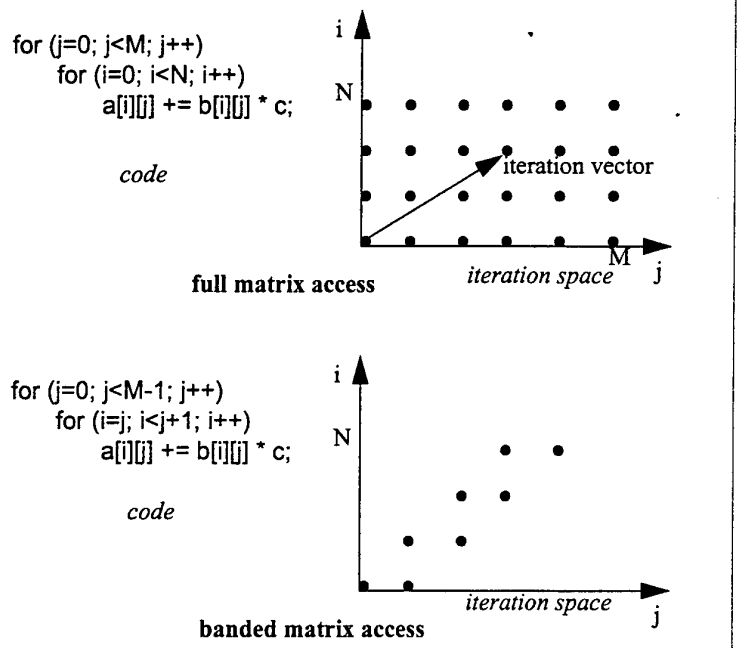


Figure 5-22 Vector representations of nested loops.

these principles, we can apply them by hand in small cases as well as understand what our tools are doing in large cases.

We need to write equations that describe all types of misses: compulsory, capacity, and conflict. The equations are written in terms of **reuse vectors**: if two iterations i_1 and i_2 use the same array location, the reuse vector that defines that reuse is $r = i_2 - i_1$ (assuming that i_2 comes after i_1). We will assume for the moment that our cache is direct mapped; we need to know the cache's line size so that we can compute which cache line each memory reference falls into. Compulsory misses occur in two cases: in the first access along the reuse vector; and when the reuse vector spans more than one cache line. Capacity and conflict misses can be captured in a single form of equation. Two references $\vec{i}$ and $\vec{j}$ conflict if they access distinct memory lines that map to the same cache line:

$$L(\vec{i}) = L(\vec{j}). \qquad \text{(EQ 7-5)}$$

If the cache size is C_s and the line size is L_s, we can rewrite this as

$$mem(\vec{i}) = mem(\vec{j}) + nC_s + b. \qquad \text{(EQ 7-6)}$$

In this equation, n is an integer that represents an arbitrary offset and b is an integer in the range $-L_{off} \le b \le L_s - 1 - L_{off}$, where L_{off} is the offset of the memory reference in the cache line.

We can use these equations for a variety of purposes. For instance, if we want to minimize the number of conflicts between array accesses, we can parameterize the equations in terms of the starting addresses of the arrays and the number of elements in each row (by padding the arrays), then find the values for these parameters that minimize the number of cache conflicts. Example 5-10 illustrates the concepts behind cache miss equations.

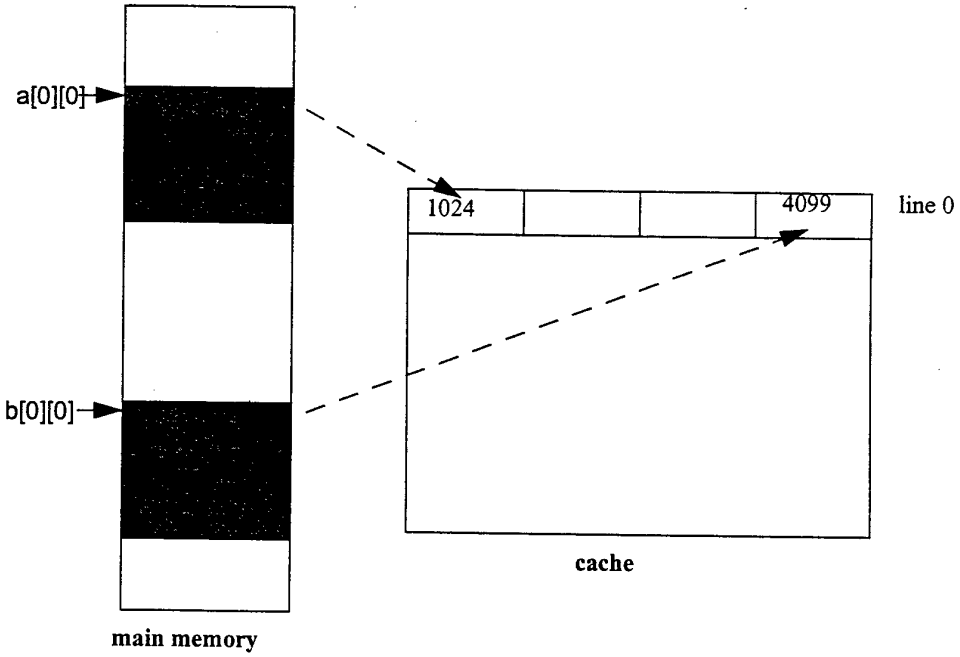
Example 5-10 Data realignment and array padding

We want to optimize the cache behavior of this code:

```
for (j=0; j<M; j++)
    for (i=0; i<N; i++)
        a[i][j] = b[i][j] * c;
```

Let us assume that the a and b arrays are sized with M is 265 and N is 4. Let's also assume that the cache is a 256-line, four-way set-associative cache with four words per line. Even though this code does not reuse any data elements, cache conflicts can cause serious performance problems because they interfere with spatial reuse at the cache line level.

Let's assume that the starting location for a[] is 1024 and the starting location of b[] is 4099. Although a[0][0] and b[0][0] do not map to the same word in the cache, they do map to the same line:



As a result, we see this scenario in execution:

- The access to a[0][0] brings in the first four words of a[].
- The access to b[0][0] replaces a[0] through a[3][0] with b[3][0] and the contents of the three locations before b[].
- When a[1][0] is accessed, the same cache line is again replaced with the first four elements of a[].

Once the a[1][0] access brings that line into the cache, it remains there for the a[2][0] and a[3][0] accesses since the b[] accesses are now on the next line. However, the scenario repeats itself at a[4][0] and every four iterations of the cache.

One way to eliminate the cache conflicts is to move one of the arrays. We do not have to move it far. If we move b's start to 4100, we eliminate the cache conflicts.

However, that fix won't work in more complex situations. Moving one array may only introduce cache conflicts with another array. In such cases, we can use another technique: padding. If we extend each of the rows of the arrays to have four elements rather than three, with the padding word placed at the beginning of the row, we eliminate the cache conflicts. In this case, b[0][0] is located at 4100 by the padding. Although padding wastes memory, it substantially improves memory performance. In complex situations with multiple arrays and sophisticated access patterns, we have to use a combination of relocating arrays and padding them to be able to minimize cache conflicts.

5.6.3 Optimizing For Execution Speed

If you need to improve the speed of your program, you can attack the problem at several levels of abstraction.

system-level analysis

First, make sure that the code really needs to be speeded up. If you are dealing with a large program, it may not be obvious what part of the program is actually using the most time. Profiling the program will help you find hot spots. As we will see in Chapter 6, many systems are built from multiple programs running as communicating processes; a slow program may not actually limit the overall system performance.

algorithm and program structure optimization

You may be able to redesign your algorithm to improve efficiency. Often looking at asymptotic performance is a good guide to efficiency. Doing fewer operations is usually the key to performance. In a few cases, however, brute force may provide the better implementation. A seemingly simple high-level language statement may in fact hide a very long sequence of operations that slows down the algorithm. Using dynamically-allocated memory is one example, since managing the heap takes time but is hidden from the programmer. For example, a sophisticated algorithm that uses dynamic storage may be slower in practice than an algorithm that performs more operations on statically-allocated memory.

coding optimization

Finally, you can look at the implementation of the program itself. Here are some hints on program implementation:

- Try to use registers efficiently. Group accesses to a value together so that it can be brought into a register and kept there.
- Make use of page mode accesses in the memory system whenever possible. Page mode reads and writes eliminate one step in the memory access. You can increase use of page mode by rearranging your variables so that more can be referenced contiguously.
- Analyze cache behavior to find important cache conflicts. Restructure the code to eliminate as many of these as you can:

  - For instruction conflicts, if the offending code segment is small, try to rewrite it to make it as small as possible so that it better fits into the cache; this may require writing in assembly language. For conflicts across larger spans of code, try moving the instructions or padding with NOPs to reduce some conflicts.

  - For scalar data conflicts, move the data values to different locations to reduce conflicts.

  - For array data conflicts, consider either moving the arrays or changing your array access patterns to reduce conflicts.

5.7 Analysis and Optimization of Energy and Power

Power consumption is a particularly important design metric for battery-powered systems since the battery has a very limited lifetime. However, power consumption is increasingly important in systems that run off the power grid—fast chips run hot and controlling power consumption is an important element of increasing reliability and lowering system cost.

How much control do we have over power consumption? Ultimately, if we have some computations that need to be done, we must consume the energy required to perform those computations. However, there are opportunities for saving power:

- We may be able to change the algorithms used to ones that do things in clever ways that consume less power.
- Memory accesses are a major component of power consumption in many applications; by optimizing memory accesses we may be able to significantly reduce power.
- We may be able to turn off parts of the system—subsystems of the CPU, chips in the system, etc.—when we don't need them in order to save power.

measuring energy consumption

The first step in optimizing energy consumption of a program is knowing how much energy the program consumes. It is possible to measure power consumption for an instruction or a small code fragment [Tiw94]. The technique, illustrated in Figure 5-23, executes the code under test over and over in a loop. By measuring the current flowing into the CPU, we are measuring the power consumption of the complete loop, including both the body and other code. By separately measuring the power consumption of a loop with no body (making sure, of course, that the compiler hasn't optimized away the empty loop), we can calculate the power consumption of the loop body

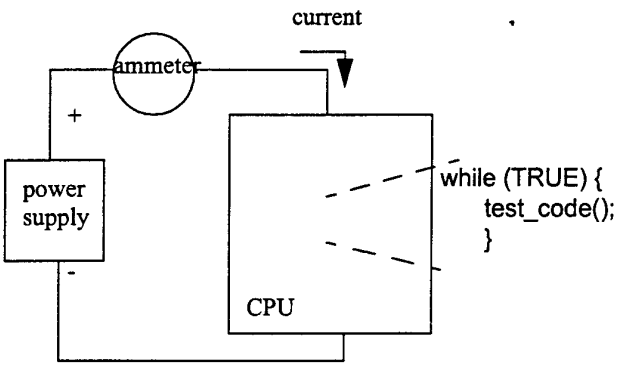


Figure 5-23 Measuring energy consumption for a piece of code.

code as the difference between the full loop and the bare loop.energy cost of an instruction.

5.7.1 Program Energy Consumption and Optimization

factors in energy consumption

A variety of factors contribute to the energy consumption of the program:

- Energy consumption does vary somewhat from instruction to instruction.
- The sequence of instructions does also have some influence.
- The opcode and the locations of the operands also matter.

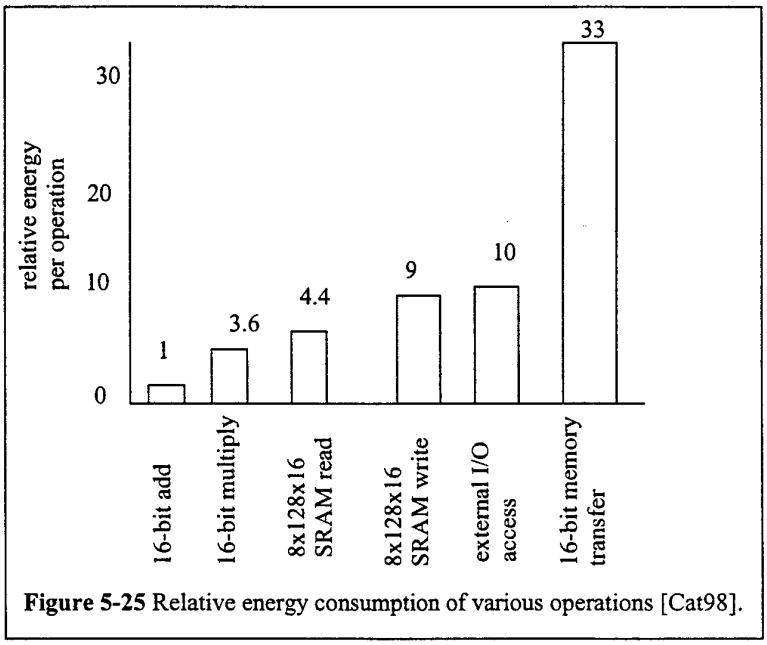


Figure 5-25 Relative energy consumption of various operations [Cat98].

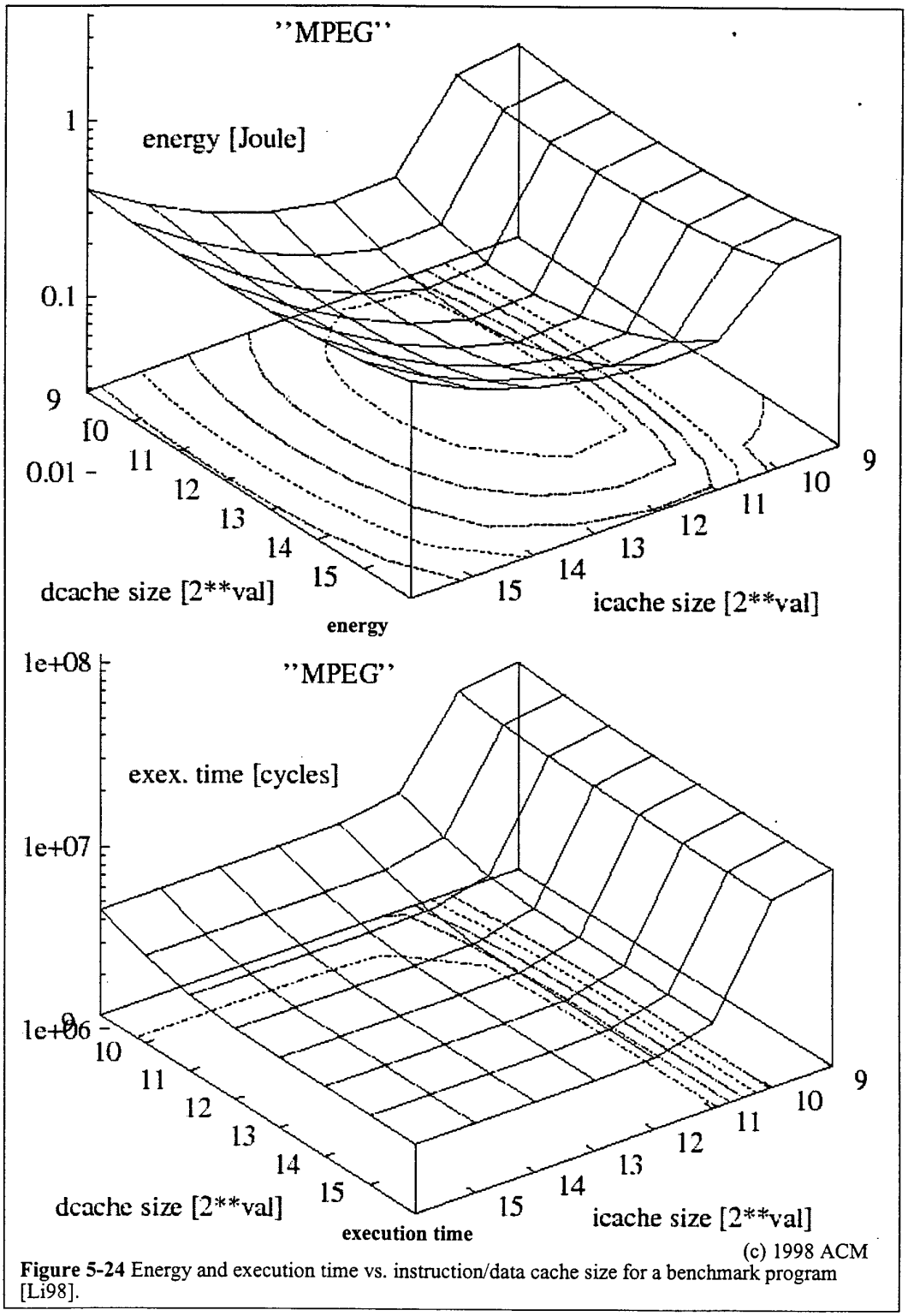


Figure 5-24 Energy and execution time vs. instruction/data cache size for a benchmark program [Li98].

Choosing which instructions to use can make some difference in a program's energy consumption, but concentrating on the instruction opcodes has limited payoffs in most CPUs. The program has to do a certain amount of computation to perform its function. While there may be some clever ways to perform that computation, the energy cost of the basic computation will change only a fairly small amount compared to the total system energy consumption, and usually only after a great deal of effort. We are further hampered in our ability to optimize instruction-level energy consumption because most manufacturers do not provide detailed, instruction-level energy consumption figures for their processors.

memory effects

In many applications, the biggest payoff in energy reduction for a given amount of designer effort comes from concentrating on the memory system. As Figure 5-25 shows [Cat98], memory transfers are by far the most expensive type of operation performed by a CPU: a memory transfer takes 33 times more energy than does an addition. As a result, the biggest payoffs in energy optimization come from properly organizing instructions and data in memory. Accesses to registers are the most energy-efficient; cache accesses more energy-efficient than main memory accesses.

Caches are an important factor in energy consumption: on the one hand, a cache hit saves a costly main memory access; on the other hand, the cache itself is relatively power hungry because it is built from SRAM, not DRAM. If we can control the size of the cache, we want to choose the smallest cache that provides us with the necessary performance. Li and Henkel [Li98] measured the influence of caches on energy consumption in detail. Figure 5-24 breaks down the energy consumption of a computer running MPEG (a video encoder) into several components: software running on the CPU, main memory, data cache, and instruction cache. As the instruction cache size increases, the energy cost of the software on the CPU goes down, but the instruction cache comes to dominate the energy consumption. Experiments like this on several benchmarks show that many programs have sweet spots in energy consumption: if the cache is too small, the program runs slowly and the system consumes a lot of power due to the high cost of main memory accesses; if the cache is too large, the power consumption is high without a corresponding payoff in performance; at intermediate values, the execution time and power consumption are both good.

optimizing programs for energy

How can we optimize a program for low power consumption? The best overall advice is that *high performance = low power*—generally speaking, making the program run faster also reduces energy consumption.

Clearly, the biggest factor that can be reasonably well controlled by the programmer is the memory access patterns. If the program can be modified to reduce instruction or data cache conflicts, for example, the energy required by the memory system can be cut down by quite a bit. The effectiveness of changes such as reordering instructions or selecting different instructions depends on the processor involved, but they are generally less effective than cache optimizations.

Here are a few optimizations that we mentioned originally for performance that are also often useful when improving energy consumption:

- Try to use registers efficiently. Group accesses to a value together so that it can be brought into a register and kept there.
- Analyze cache behavior to find important cache conflicts. Restructure the code to eliminate as many of these as you can:

  - For instruction conflicts, if the offending code segment is small, try to rewrite it to make it as small as possible so that it better fits into the cache; this may require writing in assembly language. For conflicts across larger spans of code, try moving the instructions or padding with NOPs to reduce some conflicts.

  - For scalar data conflicts, move the data values to different locations to reduce conflicts.

  - For array data conflicts, consider either moving the arrays or changing your array access patterns to reduce conflicts.
- Make use of page mode accesses in the memory system whenever possible. Page mode reads and writes eliminate one step in the memory access, saving a considerable amount of power.

Metha et al [Met97] present some additional observations about energy optimization:

- Moderate loop unrolling eliminates some loop control overhead. However, when the loop is unrolled too much, power increases due to the lower hit rates of straight-line code.
- Software pipelining reduces pipeline stalls, thereby reducing the average energy per instruction.
- Eliminating recursive procedure calls where possible saves power by getting rid of function call overhead. Tail recursion can often be eliminated; some compilers do this automatically.

5.8 Analysis and Optimization of Program Size

The memory footprint of a program is determined by the sizes of its data and instructions. Both must be considered to minimize the program's size.

reducing data size

Data is an excellent opportunity for minimizing size because it is most highly dependent on programming style. Inefficient programs often keep several copies of data; identifying and eliminating duplications can lead to significant memory savings usually with little performance penalty. Buffers should be sized carefully—rather than defining a data array to some large size which the program will never attain, determine the actual maximum amount of data held in it and allocate the array accordingly. Data can sometimes be packed, for example by storing several flags in a single word and extracting them by using bit-level operations.

A very low-level technique for minimizing data is to reuse values. For instance, if several constants happen to have the same value, they can be mapped to the same location. Data buffers can often be reused at several different points in the program. This technique must be used with extreme caution, however, since subsequent versions of the program may not use the

same values for the constants. A more generally applicable technique is to generate data on the fly rather than store it. Of course, the code required to generate the data takes up some space in the program, but when complex data structures are involved there may be some net space savings from using code to generate data.

minimizing instruction size

Minimizing the size of the instruction text of a program requires a mix of high-level program transformations and careful instruction selection. Encapsulating functions in subroutines can reduce program size when done carefully—subroutines have overhead for parameter passing which is not obvious from the high-level language code, so there is a minimum-size function body for which a subroutine makes sense. Architectures which have variable-size instruction lengths are particularly good candidates for careful coding to minimize program size, which may require assembly-language coding of key program segments. There may also be cases in which one or a sequence of instructions is much smaller than alternative implementations—for example, a multiply-accumulate instruction may be both smaller and faster than separate arithmetic operations.

When reducing the number of instructions in a program, one important technique is the proper use of subroutines. If the program performs identical operations repeatedly, these operations are natural candidates for subroutines. Even if the operations vary somewhat, a properly parameterized subroutine may be able to be constructed that saves space. Of course, when considering the code size savings, the subroutine linkage code must be counted into the equation—there is extra code not only in the subroutine body but in each call to the subroutine that handles parameters. In some cases, proper instruction selection may reduce code size; this is particularly true in CPUs that use variable-length instructions.

specialized instructions

Some microprocessor architectures support **dense instruction sets**—specially designed instruction sets that use shorter instruction formats to encode the instructions. The ARM Thumb instruction set and the MIPS-16 instruction set for the MIPS architecture are two examples of this sort of instruction set. In many cases, a microprocessor that supports the dense instruction set

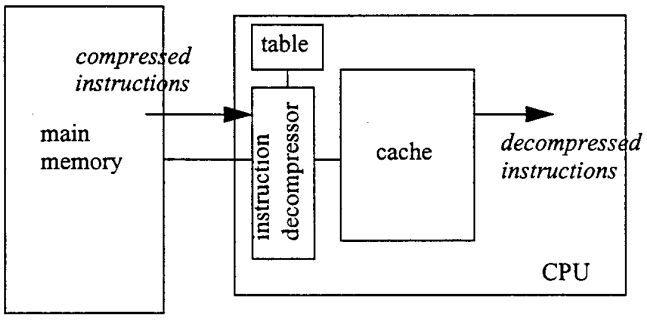


Figure 5-26 Cache-based instruction decompression.

also supports the normal instruction set, though it is possible to build a microprocessor that executes only the dense instruction set. Special compilation modes produce the program in terms of the dense instruction set. Program size of course varies with the type of program, but programs using the dense instruction set are often 70%-80% of the size of the standard instruction set equivalents.

statistical code compression

Another possibility is to use statistical compression algorithms to compress the program after assembly and then decompress it on the fly. Statistical compression views the data to be compressed as a stream of symbols. It relies on the fact that not all symbols or sequences of symbols are equally probable, so that more probable sections of the data are encoded in a small number of bits while less probable sections are encoded with more bits. File compression programs like *gzip* can often reduce a file to 50% of its uncompressed size. However, file compression algorithms cannot be directly applied to dynamic code decompression. First, the algorithm must not force us to decompress the entire program before executing—that would defeat the purpose of code compression since the entire program would reside in memory. We must be able to decompress small sections of the program during execution. This creates particular problems for branches, since the address in a branch refers to an uncompressed location, not the location of the branch target in the compressed program. Second, we must be able to decompress the block of code quickly to avoid slowing down the CPU excessively.

There are several techniques for code compression, but one common one [Wol92a] is illustrated in Figure 5-26. In this scheme, a block of compressed code is fetched from main memory and decompressed into the cache to satisfy a cache miss. Because decompression occurs only on cache misses, it usually occurs relatively infrequently, and the cache is already organized to handle blocks of code. The code is compressed at the end of compilation, after the file has been linked to form an executable; the compression process yields tables that program the decompression unit. A variety of compression algorithms can be used to generate the compressed code that can also yield fast hardware decompression logic.

Application Note 5-1Code compression for PowerPC

IBM has developed a code decompression module for PowerPC processors [Kem98]. Their decompression unit decompresses instructions after they have been fetched from main memory. They found that they could achieve much higher compression ratios by breaking each 32-bit instruction into two 16-bit halves. They use Huffman compression to compress each half, compressing instructions in blocks of 64 bytes. They use an index table to translate uncompressed addresses, such as addresses found in branch instructions, to the new location of the instruction in the compressed code. They also add the K bit to the TLB to indicate whether a page in memory holds compressed instructions. Their method compresses benchmark programs to 55%-65% of their original size.

5.9 Program Validation and Testing

Complex systems need testing to ensure that they work as they are intended to do. But bugs can be subtle, particularly in embedded systems, where specialized hardware and real-time responsiveness make programming more challenging. Fortunately, there are many techniques for software testing available which can help us generate a comprehensive set of tests that can give us high confidence that our system works properly. We will examine the role of validation in the overall design methodology in Section 9.6; here, we will concentrate on nuts-and-bolts techniques for creating a good set of tests for a given program.

The first question we must ask ourselves is how much testing is enough. Clearly we cannot test the program for every possible combination of inputs. Since we must limit the number of tests we do, we naturally ask ourselves what a reasonable standard of thoroughness is. One of the major contributions of software testing is to provide us with standards of thoroughness that make sense. Following these standards does not guarantee that we will find all bugs. But by breaking the testing problem into subproblems and analyzing each subproblem, we can identify testing methods that provide reasonable amounts of testing while keeping the testing time within reasonable bounds.

testing strategies

There are two major types of testing strategies:

- **black-box** methods generate tests without looking at the internal structure of the program;
- **clear-box** (also known as **white-box**) methods generate tests based on the program structure.

In this section we will cover both types of tests starting with clear-box testing, which complement each other by exercising programs in very different ways.

5.9.1 Clear-Box Testing

CDFGs and testing

The control/data flow graph extracted from a program's source code is an important tool in developing clear-box tests for the program. To adequately test the program, we must exercise both its control and data operations.

In order to execute and evaluate these tests, we must be able to control variable values in the program and observe the results of computations, much as in manufacturing test. In general, we may need to modify the program to make it more testable. By adding new inputs and outputs, we can usually substantially reduce the effort required to find a test and to execute the test. Example 5-11 illustrates the importance of observability and controllability in software testing.

Example 5-11 Controlling and observing programs

Let's first consider controllability by looking at an FIR filter with a limiter:

```
firout = 0.0; /* initialize filter output */
/* compute buff*c in bottom part of circular buffer */
for (j=curr, k=0; j<N; j++, k++)
    firout += buff[j] * c[k];
/* compute buff*c in top part of circular buffer */
for (j=0; j<curr; j++, k++)
    firout += buff[j] * c[k];
/* limit output value */
if (firout > 100.0) firout = 100.0;
if (firout < -100.0) firout = -100.0;
```

This code computes the output of an FIR filter from a circular buffer of values, then limits the maximum filter output (much as an overloaded speaker will hit a range limit). If we want to test whether the limiting code works, we must be able to generate two out-of-range values for firout: positive and negative. To do that, we must fill the FIR filter's circular buffer with N values in the proper range. Although there are many sets of values that will work, it will still take time for us to set up the filter output properly for each test.

This code also illustrates an observability problem. If we want to test the FIR filter itself, we want to look at the value of firout before the limiting code executes. We could use a debugger such as dbx to set breakpoints in the code, but this is an awkward way to perform a large number of tests. If we want to test the FIR code independent of the limiting code, we would have to add a mechanism for observing firout independently.

No matter what we are testing, we must do three things in a test:

- provide the program with inputs that exercise the test we are interested in;
- execute the program to perform the test;
- examine the outputs to see whether the test was successful.

Being able to perform this process for a large number of tests entails some amount of drudgery, but that drudgery can be alleviated with good program design that simplifies controllability and observability.

The next task is to determine a set of tests to be performed. We need to perform many different types of tests to be confident that we have identified even a large fraction of the existing bugs. Even if we thoroughly test the program using one criterion, that criterion ignores other aspects of the program. Over the next few pages we will describe several very different criteria for program testing.

execution paths

The most fundamental concept in clear-box testing is the path of execution through a program. We have considered paths for performance analysis; we are now concerned with making sure that a path is covered and determining how to ensure that the path is in fact executed. We want to test the program by forcing the program to execute along chosen paths. We force the execution of a path by giving it inputs that cause it to take the appropriate branches. Execution of a path exercises both the control and data aspects of

the program: the control is exercised as we take branches; both the computations leading up to the branch decision and other computations performed along the path exercise the data aspects.

exercising program paths

Is it possible to execute every complete path in an arbitrary program? The answer is no since the program may contain a while loop that is not guaranteed to terminate. The same is true for any program which operates on a continuous stream of data, since we cannot arbitrarily define the beginning and end of the data stream. If the program always terminates, then there are indeed a finite number of complete paths that can be enumerated from the path graph. This leads us to the next question—does it make sense to exercise every path? The answer to this question is no for most programs, since the number of paths, especially for any program with a loop, is extremely large. However, the choice of an appropriate subset of paths to test requires some thought. The next example illustrates the consequences of two different choices of testing strategies.

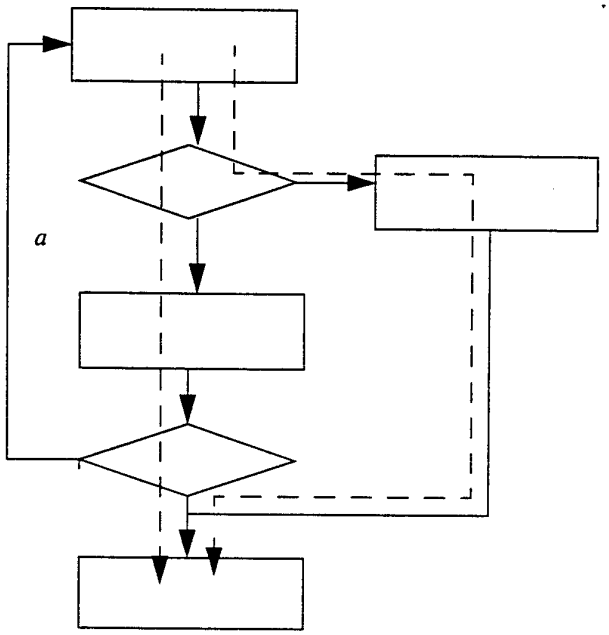
Example 5-12 Choosing which paths to test

Two reasonable choices for a set of paths to test are:

- execute every statement at least once;
- execute every direction of a branch at least once.

These conditions are equivalent for structured programming languages without gotos, but are not the same for unstructured code. Most assembly language is unstructured and state machines may be coded in high-level languages with gotos.

To understand the difference between statement and branch coverage, consider the CDFG below. We can execute every statement at least once by executing the program along two distinct paths:



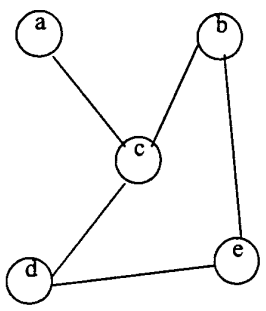
However, this leaves branch *a* out of the lower conditional uncovered. To make sure that we have executed along every edge in the CDFG, we must execute a third path through the program. This path does not test any new statements but it does cause *a* to be exercised.

---

choice of paths

How do we choose a set of paths that adequately covers the program's behavior? Intuition tells us that a relatively small number of paths should be able to cover most practical programs. Graph theory helps us get a quantitative handle on the different paths required. In an undirected graph, we can form any path through the graph from combinations of **basis paths**. (Unfortunately, this property does not strictly hold for directed graphs such as CDFGs, but this formulation still helps us understand the nature of selecting a set of covering paths through a program.) The term basis set comes from linear algebra. Figure 5-27 shows how to evaluate the basis set of a graph. The graph is represented as an **incidence matrix**. Each row and column represents a node; a 1 is entered for each node pair that is connected by an edge. We can use standard linear algebra techniques to find the basis set of the graph. Each vector in the basis set represents a primitive path; we can form new paths by adding together the vectors modulo 2. There is, in general, more than one basis set for a graph.

The basis set property gives us a metric for test coverage: if we cover all the basis paths, we can consider the control flow adequately covered. Although the basis set measure is not entirely accurate since the directed edges of the



graph

$$\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

incidence matrix

$$\begin{array}{c} a \\ b \\ c \\ d \\ e \end{array} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

basis set

Figure 5-27 The matrix representation of a graph and its basis set.

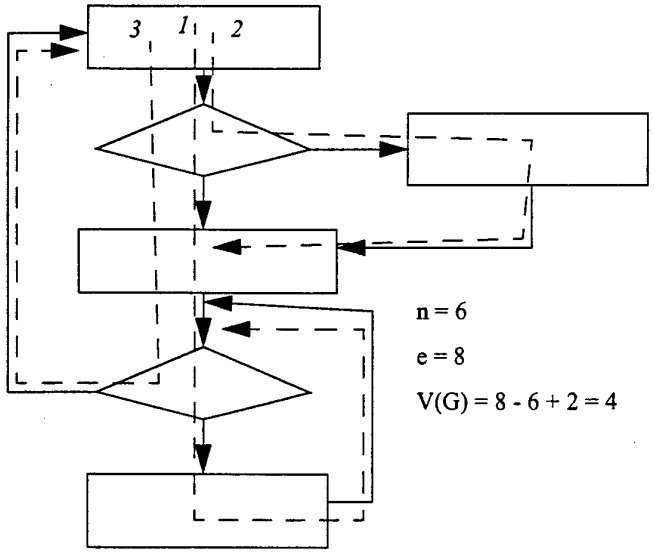


Figure 5-28 Cyclomatic complexity.

CDFG may make some combinations of paths infeasible, it does provide a reasonable and justifiable measure of test coverage.

There is a simple measure, **cyclomatic complexity** [McC76], which allows us to measure the control complexity of a program. Cyclomatic complexity is an upper bound on the size of the basis set that we found in Section 5.6.1. If e is the number of edges in the flow graph, n the number of nodes, and p

the number of components in the graph, then the cyclomatic complexity is given by

$$M = e - n + 2p. \qquad \text{(EQ 4-7)}$$

For a structured program, M can be computed by counting the number of binary decisions in the flow graph and adding 1; if the CDFG has higher-order branch nodes, add b-1 for each b-way branch. In the example of Figure 5-28, the cyclomatic complexity evaluates to 4. There are actually only three distinct paths in the graph, so cyclomatic complexity is in this case an overly conservative bound.

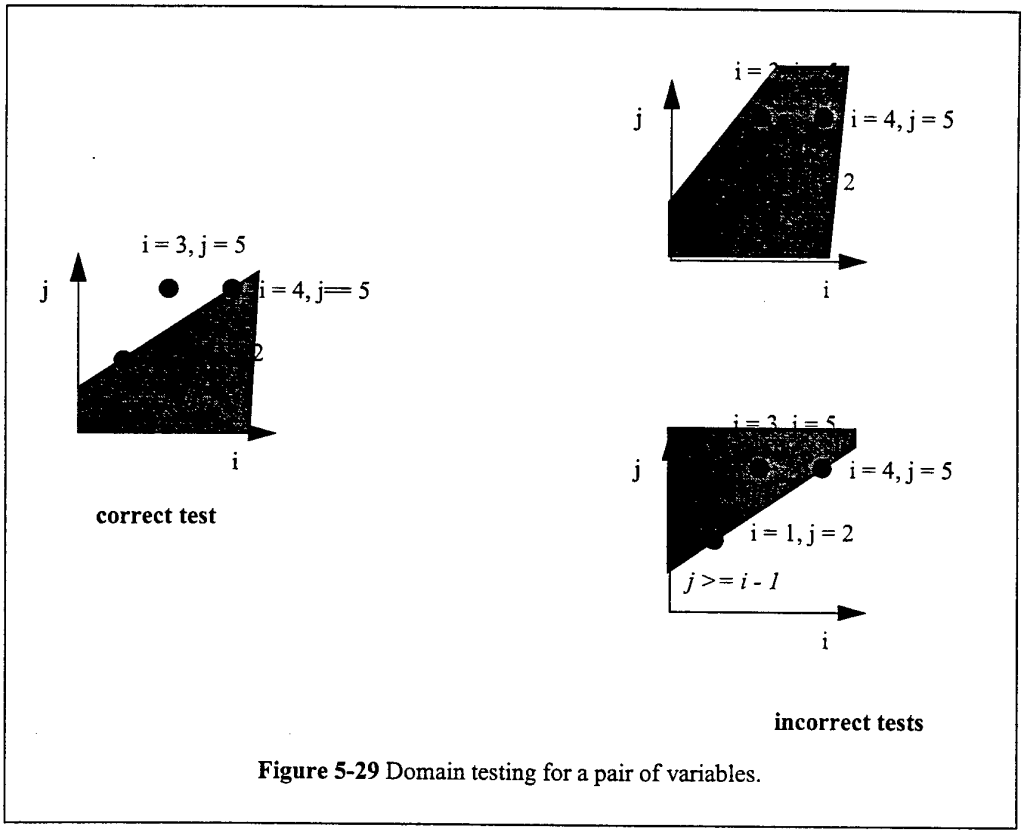


Figure 5-29 Domain testing for a pair of variables.

Another way of looking at control flow oriented testing is to analyze the conditions that control the conditional statements. Consider this if statement:

```
if ((a == b) || (c >= d)) { ... }
```

This complex condition can be exercised in several different ways. If we want to truly exercise the paths through this condition, it is prudent to exercise the conditional's elements in ways related to their own structure, not just the structure of the paths through them. A simple condition testing strategy is known as **branch testing** [Mye79]. This strategy requires the true and false branches of a conditional and every simple condition in the conditional's expression to be tested at least once. Example 5-13 illustrates branch testing.

Example 5-13 Condition testing with the branch testing strategy

Here is the code that we meant to write:

```
if (a || (b >= c)) { printf("OK\n"); }
```

Here is the code that we mistakenly wrote instead:

```
if (a && (b >= c)) { printf("OK\n"); }
```

If we apply branch testing to the code we wrote, one of the tests will use these values: a = 0, b = 3, c = 2 (making a false and b >= c true). In this case, the code should print OK (0 || (3 >= 2) is true) but instead doesn't print (0 && (3 >= 2) evaluates to false). That test picks up the error.

Let's look at another more subtle error that is nonetheless all too common in C. Here is the code we meant to write:

```
if ((x == good_pointer) && (x->field1 == 3)) { printf("got the value\n"); }
```

and here is the bad code we actually wrote:

```
if ((x = good_pointer) && (x->field1 == 3)) { printf("got the value\n"); }
```

The problem here is that we typed = rather than ==, creating an assignment rather than a test. The code x = good_pointer first assigns the value good_pointer to x, then because assignments are also expressions in C, returns good_pointer as the result of evaluating this expression.

If we apply the principles of branch testing, one of the tests we want to use will have x != good_pointer and x->field1 == 3. Whether this test catches the error depends on the state of the record pointed to by good_pointer: if it is equal to 3 at the time of the test, the message will be printed erroneously. Although this test is not guaranteed to uncover the bug, it has a reasonable chance of doing so; one of the reasons to use many different types of tests is to maximize the chance that supposedly unrelated elements will cooperate to reveal the error in a particular situation.

Another more sophisticated strategy for testing conditionals is known as **domain testing** [How82], illustrated in Figure 5-29. Domain testing concentrates on linear inequalities. In the figure, the inequality the program should use for the test is j <= i + 1. We test the inequality with three test points: two on the boundary of the valid region and a third outside the region but between the i values of the other two points. When we make some common mistakes in typing the inequality, these three tests are sufficient to uncover them, as shown in the figure.

A potential problem with path coverage is that the paths chosen to cover the CDFG may not have any important relationship to the program's function. Another testing strategy known as **data flow testing** makes use of **def-use**

analysis (short for definition-use analysis). It selects paths that have some relationship to the program's function.

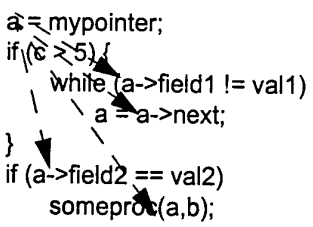


Figure 5-30 Definitions and uses of variables.

The terms def and use come from compilers, which use def-use analysis for optimization [Aho86]. A variable's value is **defined** when an assignment is made to the variable; it is used when it appears on the right-hand side of an assignment (sometimes called a **c-use** for computation use) or in a conditional expression (sometimes called **p-use** for predicate use). A **def-use pair** is a definition of a variable's value and a use of that value. Figure 5-30 shows a code fragment and all the def-use pairs for the first assignment to a. Def-use analysis can be performed on a program using iterative algorithms. Dataflow testing chooses tests that exercise chosen def-use pairs: the test first causes a certain value to be assigned at the definition, then observes the result at the use point to be sure that the desired value arrived there. Frankl and Weyuker [Fra88] have defined criteria for choosing which def-use pairs to exercise to satisfy a well-behaved adequacy criterion.

testing loops

We can write some specialized tests for loops. Since loops are common and often perform important steps in the program, it is worth developing loop-centric testing methods. If the number of iterations if fixed, then testing is relatively simple. However, many loops have bounds that are executed at run time. Consider first the case of a single loop:

```
for (i=0; i<terminate(); i++)
    proc(i,array);
```

It would be too expensive to evaluate the loop for all possible termination conditions, but there are several important cases that we should try at a minimum:

1. Skipping the loop entirely (if that is possible, such as when terminate() returns 0 on its first call).
2. One loop iteration.
3. Two loop iterations.
4. If there is an upper bound n on the number of loop iterations (which may come from the maximum size of an array), some value which is significantly below that maximum number of iterations.
5. Tests near the upper bound on the number of loop iterations: n-1, n, n+1.

We may also have nested loops such as this:

```
for (i=0; i<terminate1(); i++)
    for (j=0; j<terminate2(); j++)
        for (k=0; k<terminate3(); k++)
            proc(i,j,k,array);
```

There are many possible strategies for testing nested loops. One thing to keep in mind is which loops have fixed vs. variable number of iterations. Beizer [Bei90] suggests an inside-out strategy for testing loops with multiple variable iteration bounds. First, concentrate on testing the innermost loop as above; the outer loops should be controlled to their minimum numbers of iterations. After the inner loop has been thoroughly tested, the next outer loop can be tested more thoroughly, with the inner loop executing a typical number of iterations. This strategy can be repeated until the entire loop nest has been tested. Clearly, nested loops can require a large number of tests; it may be worthwhile to insert testing code to allow greater control over the loop nest for testing.

5.9.2 Black-Box Testing

Black-box tests are generated without knowledge of the code being tested. When used alone, black-box tests have a low probability of finding all the bugs in a program. But when used in conjunction with clear-box tests they help provide a well-rounded test set, since black-box tests are likely to uncover errors that are unlikely to be found by tests extracted from the code structure. Black-box tests can really work—one acquaintance of the author, when asked to test an instrument whose front panel was run by a microcontroller, immediately used his hand to depress all the buttons simultaneously. The front panel immediately locked up. This situation could occur in practice if the instrument were set face-down on a table but would be very unlikely to be discovered by clear-box tests.

testing from the spec

One important technique is to take tests directly from the specification for the code under design. The specification should say what outputs are expected for certain inputs. Tests should be created that give specified outputs and test that the results also satisfy the inputs.

We can't test every possible input combination, but some rules of thumb help us select reasonable sets of inputs. When an input can range across a set of values, it is a very good idea to test at the ends of the range. For example, if an input must be between 1 and 10, 0, 1, 10, and 11 are all important values to test. We should be sure to consider tests both within and outside the range: for example, testing values within the range and outside the range. We may want to consider tests well outside the valid range as well as our boundary-condition tests.

tests not from the spec

Random tests form one category of black-box test. Random values are generate with some distribution. The expected values are computed independently of the system, then the test inputs are applied. A large number of tests must be applied for the results to be statistically significant, but the tests are easy to generate.

Another scenario is to test certain types of data values. For example, integer-valued inputs can be generated at interesting values like 0, 1, and values near the maximum end of the data range. Illegal values can be tested as well.

Regression tests form an extremely important category of tests. When tests are created during earlier stages in the system design or for previous versions of the system, those tests should be saved to apply to the later versions of the system. Clearly, unless the system specification changed, the new system should be able to pass old tests. In some cases old bugs can creep back into systems, for example when an old version of a software module is inadvertently installed. In other cases regression tests simply exercise the code in different ways than would be done for the current version of the code and so possibly exercise different bugs.

Some embedded systems, particularly digital signal processing systems, lend themselves to numerical analysis. Signal processing algorithms are frequently implemented with limited-range arithmetic to save hardware costs. Aggressive data sets can be generated to stress the numerical accuracy of the system; these tests can often be generated from the original formulas without reference to the source code.

5.9.3 Evaluating Function Tests

	block	decision	p-use	c-use
TeX	85%	72%	53%	48%
awk	70%	59%	48%	55%

Table 4-1 Code coverage of functional tests for TeX and awk (after Horgan and Mathur [Hor96]).

How much testing is enough? Horgan and Mathur [Hor96] evaluated the coverage of two well-known programs, TeX and awk. They used functional tests for these programs which had been developed over several years of extensive testing. Upon applying those functional tests to the programs, they obtained the code coverage statistics shown in Table 4-1.The columns refer to various types of test coverage: *block* refers to basic blocks, *decision* to conditionals, *p-use* to a use of a variable in a predicate (decision), and *c-use* to variable use in a non-predicate computation. These results are at least suggestive that functional testing does not fully exercise the code and that techniques which explicitly generate tests for various pieces of code are necessary to obtain adequate levels of code coverage.

Methodological techniques are important for understanding the quality of your tests. For example, if you keep track of the number of bugs tested each day, the data you collect over time should show you some trends on the number of errors per page of code to expect on the average, how many bugs are caught by certain kinds of tests, etc. We will talk about methodological approaches to quality control in more detail in Section 9.6.4.

error injection

One interesting method for analyzing the coverage of your tests is **error injection**. First, take your existing code and add bugs to it, keeping track of where the bugs were added. Then run your existing tests on the modified program. By counting the number of added bugs your tests found, you can get an idea of how effective the tests are in uncovering the bugs you haven't yet found. This method assumes that you can deliberately inject bugs that are of similar varieties to those created naturally by programming errors. If the bugs are too easy or too hard to be found or simply require different types of tests, then bug injection's results will not be relevant. Of course, it is essential that you finally use the correct code, not the code with added bugs.

5.9.4 Testing for Performance

Because embedded systems often have real-time deadlines, we must concern ourselves with testing for performance, not just functionality. Performance testing determines whether the required result was generated within a certain amount of time. In many cases, we are interested in the worst-case execution time, though in some cases we may want to verify the best-case or average-case execution time.

performance analysis and testing

Performance analysis is very important here. Performance analysis can tell us what path causes the worst-case (or other case) execution time. From there we can figure out the inputs required and the expected outputs.

Real-time code with deadlines must always terminate—we know that the code must finish for any possible set of inputs. Functional testing can help us determine whether the program actually terminates.

measuring program performance

How do we measure program performance? Functional testing can be performed on any computer in most cases so long as we can compile the code on that platform. Performance testing is not so easy. We may be able to use a simulator to measure performance. However, CPU simulators vary widely in the amount of timing information they give.

Performance can also be measured directly on the hardware. We can use a timer in the system to count execution time: the program resets the timer at the start and checks the timer's value at the end of execution. We can also use logic analyzers so long as the start and end points of the program are visible on the bus.

system interactions

To get truly accurate performance measurements, we must run the program in the environment in which it will operate. The CPU used must, for example, have the same size cache as the target platform will provide. The program should also be run along with other programs that operate concurrently. When several programs (or programs and interrupt drivers) operate concurrently, they can interfere with each other in the cache causing severe performance problems. We will consider the analysis of multiple processes in the cache in more detail in Section 6.8.

5.10 Example: Software Modem

In this section we will design a modem. Low-cost modems generally use specialized chips, but some PCs implement the modem functions in software. Before jumping into the modem design itself, we need to understand some principles of how to transmit digital data over a telephone line. We will then go through a specification and discuss architecture, module design, and testing.

5.10.1 Theory of Operation and Requirements

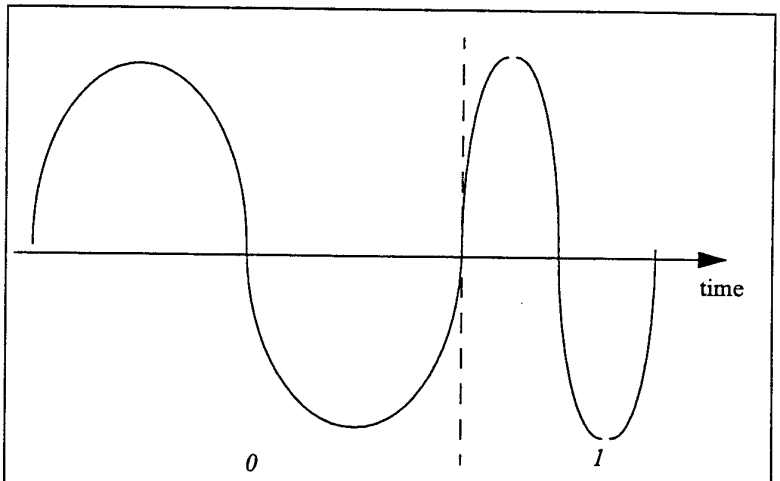


Figure 5-31 Frequency-shift keying.

data transmission

Our modem will use **frequency-shift keying (FSK)**. This technique is the technique used in 1200 baud modems. Keying alludes to Morse code-style keying. The FSK scheme transmits sinusoidal tones, with 0 and 1 assigned to different frequencies. Sinusoidal tones are much better suited to transmission over analog phone lines than are the traditional high and low voltages of digital circuits. 01 bit patterns create the chirping sound characteristic of modems. (Higher-speed modems are backward compatible with the 1200 baud FSK scheme and start a transmission with a protocol to determine what speed and protocol should be used.)

The scheme used to translate this audio input into a bit stream is illustrated in Figure 5-32. The analog input is sampled and the resulting stream is sent to two digital filters (such as an FIR filter): one passes frequencies in the range which represents a 0 and rejects the 1-band frequencies; the other filter does the converse. The outputs of the filters are sent to detectors, which compute the average value of the signal over the past n samples. When the energy goes above a threshold value, the appropriate bit is detected.

We will send data in units of 8-bit bytes. The transmitting and receiving modem agree in advance on the length of time for which a bit will be transmitted (otherwise known as the baud rate). But the transmitter and receiver

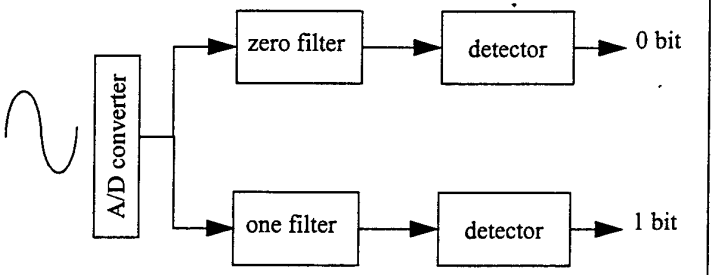


Figure 5-32 The FSK detection scheme.

are physically separated and therefore are not synchronized in any way. The receiving modem does not know when the transmitter has started to send a byte. Furthermore, even when the receiver does detect a transmission, the clock rates of the transmitter and receiver may vary somewhat, causing them to fall out of sync. In both cases, we can reduce the chances for error by sending the waveforms for a longer time.

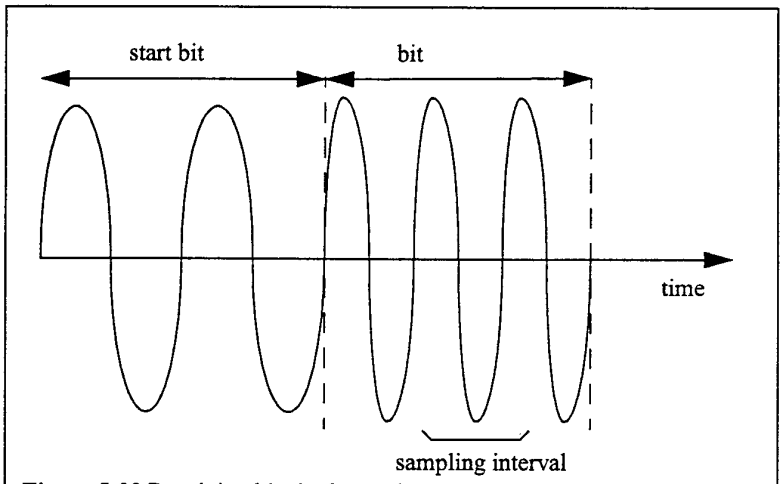


Figure 5-33 Receiving bits in the modem.

The receiver will detect the start of a byte by looking for a start bit, which is always 0. By measuring the length of the start bit, the receiver knows where to look for the start of the first bit. However, since the receiver may have slightly misjudged the start of the bit, it does not immediately try to detect the bit. Instead, it runs the detection algorithm at the predicted middle of the bit.

requirements

Our modem will not implement a hardware interface to a telephone line or software for dialing a phone number. We will assume that we have analog audio inputs and outputs for sending and receiving. We will also run at a much slower bit rate than 1200 baud to simplify the implementation. And we

will not implement a serial interface to a host, but rather put the transmitter's message in memory and save the receiver's result in memory as well. Given those understandings, let's fill out the requirements table.

name:	Modem
purpose:	A fixed baud rate frequency-shift keyed modem.
inputs:	Analog sound input, reset button.
outputs:	Analog sound output, LED bit display.
functions:	*Transmitter:* Sends data stored in microprocessor memory in 8-bit bytes. Sends start bit for each byte equal in length to one bit. *Receiver*: Automatically detects bytes and stores results in main memory. Displays currently received bit on LED.
performance:	1200 baud.
manufacturing cost:	Dominated by microprocessor and analog I/O.
power:	Powered by AC through a standard power supply.
physical size and weight:	Small and light enough to fit on a desktop.

5.10.2 Specification

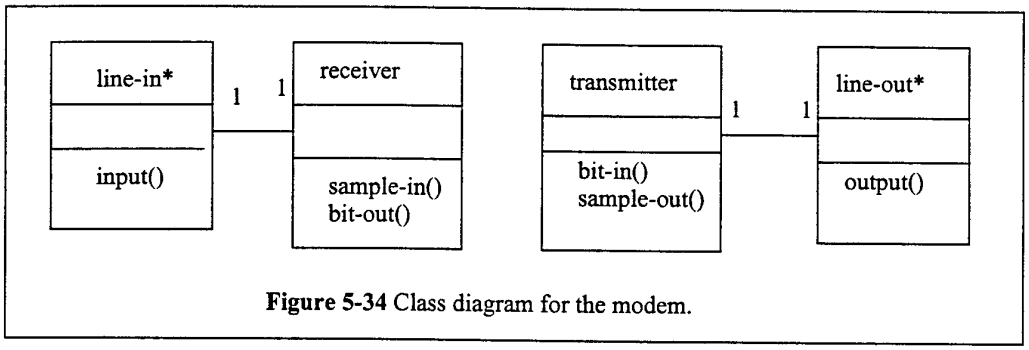
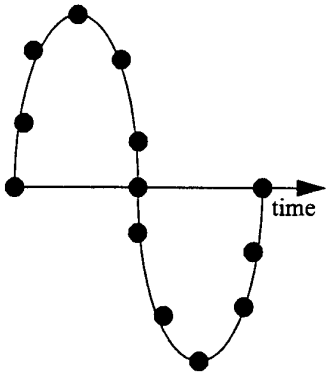


Figure 5-34 Class diagram for the modem.

The basic classes for the modem are shown in Figure 5-34.

5.10.3 System Architecture

The modem consists of one small subsystem—the interrupt handlers for the samples—and two major subsystems—transmitter and receiver. Two sample interrupt handlers are required, one for input and another for output, but they are very simple. The transmitter is simpler so let's consider its software architecture first.




```
float sine_wave[N_SAMP] =
    { 0.0, 0.5, 0.866, 1,
    0.866, 0.5, 0.0, -.05,
    0.866, -1.0, -.866, -0.5,
    0};
```

table

analog waveform and samples

Figure 5-35 Waveform generation by table lookup.

The best way to generate waveforms which keep the proper shape over long intervals is **table lookup**. Software oscillators can be used to generate periodic signals but numerical problems limit their accuracy. Figure 5-35 shows an analog waveform with sample points and the C code for these samples. Table lookup can be combined with interpolation to generate higher-resolution waveforms without excessive memory costs; this is more accurate than oscillators because no feedback is involved. The required number of samples for the modem can be found by experimentation with the analog/digital converter and the sampling code.

The structure of the receiver is considerably more complex. The filters and detectors of Figure 5-33 can be implemented with circular buffers. But that module must feed a state machine which recognizes the bits. The recognizer state machine must use a timer to determine when to start and stop computing the filter output average based upon the starting point of the bit. It must then determine the nature of the bit at the proper interval. It must also detect the start bit and measure it using the counter. The receiver sample interrupt handler is a natural candidate to double as the receiver timer since the receiver's time points are relative to samples.

The hardware architecture is relatively simple: in addition to the analog/digital and digital/analog converters, a timer is required. The amount of memory required to implement the algorithms is relatively small.

5.10.4 Component Design and Testing

The transmitter and receiver can be tested relatively thoroughly on the host platform since the timing-critical code only delivers data samples. The transmitter's output is relatively easy to verify, particularly if the data is plotted. A testbench can be constructed to feed the receiver code sinusoidal inputs and test its bit recognition rate. It is a good idea to test the bit detectors first before testing the complete receiver operation. One potential problem in

host-based testing of the receiver is encountered when library code is used for the receiver function. If a DSP library for the target processor is used to implement the filters, then a substitute must be found or built for the host processor testing. The receiver must then be re-tested when moved to the target system to be sure that it still functions properly with the library code.

Some care must be taken to ensure that the receiver does not run too long and miss its deadline. Since the bulk of the computation is in the filters, it is relatively simple to estimate the total computation time early in the implementation process.

5.10.5 System Integration and Testing

There are two ways to test the modem system: by having the modem's transmitter send bits to its receiver; or by connecting two different modems. The ultimate test is to connect two different modems, particularly two designed by different people to be sure that incompatible assumptions or errors were not made. But single-unit testing, called **loop-back** testing in the telecommunications industry, is simpler and a good first step. Loop-back can be performed in two ways. First, a shared variable can be used to directly pass data from the transmitter to the receiver. Second, an audio cable can be used to plug the analog output to the analog input; in this case it is also possible to inject analog noise to test the resiliency of the detection algorithm.

5.11 Summary

The program is a very fundamental unit of embedded system design—a program usually contains tightly interacting code. Because we care about more than just functionality, we need to understand how programs are created. Because today's compilers do not take directives such as "compile this to run in less than 1 microsecond," we have to be able to optimize programs ourselves for speed, power, and space. Our earlier understanding of computer architecture is critical to our ability to perform these optimizations. We also need to test programs to make sure they do what we want; some of our testing techniques can also be useful in exercising the programs for performance optimization.

What We Learned:

We can use data flow graphs to model straight-line code and CDFGs to model complete programs.

Compilers perform a number of tasks: generating control flow, assigning variables to registers, creating procedure linkages, etc.

Remember the performance optimization equation:

$$execution\ time = program\ path + instruction\ timing$$

Memory and cache optimizations are very important to performance optimization.

Optimizing for power consumption often goes hand-in-hand with performance optimization.

Optimizing programs for size is possible, but don't expect miracles.

Programs can be tested as black boxes (without knowing the code) or as clear boxes (by examining the code structure).

Further Reading

Aho, Sethi, and Ullman [Aho86] wrote a classic text on compilers; Muchnick [Muc97] describes advanced compiler techniques in detail. A paper on the ATOM system [Sri94] is a good description of instrumenting programs for gathering traces. Cramer et al. [Cra97] describe the Java JIT compiler. Li and Malik [Li97d] describe a method for statically analyzing program performance. Banerjee [Ban93,Ban94] describes loop transformations. Two books by Beizer, one on fundamental functional and structural testing techniques [Bei90] and the other on system-level testing [Bei84] give comprehensive introductions to software testing and, as a bonus, are well-written. Lyu [Lyu96] provides a good advanced survey of software reliability. Walsh [Wal97] describes a software modem implemented on an ARM processor.

Questions

Q5-1. Use UML to describe a design pattern for an FIR filter:

a. class diagram;

b. sequence diagram.

Q5-2. Use UML to describe a design pattern for a front panel. The front panel provides displays to output values and two types of inputs: discrete button inputs and continuous knob inputs.

a. Draw a class diagram.

b. Draw sequence diagrams for input and output.

Q5-3. For each basic block given below, rewrite it in single-assignment form, then draw the data flow grpah for that form.

a.

```
x = a + b;
y = c + d;
z = x + e;
```

b.

```
r = a + b - c;
```

```
s = 2 * r;
t = b - d;
r = d + e;
```

c.

```
a = q - r;
b = a + t;
a = r + s;
c = t - u;
```

d.

```
w = a - b + c;
x = w - d;
y = x - 2;
w = a + b - c;
z = y + d;
y = b * c;
```

Q5-4. Draw the CDFG for these code fragments:

a.

```
if (y == 2) r = a + b; s = c - d;
else r = a - c
```

b.

```
x = 1;
if (y == 2) { r = a + b; s = c - d; }
else { r = a - c; }
```

c.

```
x = 2;
while (x < 40) {
    x = foo[x];
}
```

d.

```
for (i=0; i<N; i++)
    x[i] = a[i]*b[i];
```

e.

```
for (i=0; i<N; i++) {
    if (a[i] == 0)
        x[i] = 5;
    else
        x[i] = a[i]*b[i];
}
```

Q5-5. Show the contents of the assembler's symbol table at the end of code generation for each line of these programs:

a.

```
        ORG 200
p1      ADR r4,a
        LDR r0,[r4]
        ADR r4,e
```

```
        LDR r1,[r4]
        ADD r0,r0,r1
        CMP  r0, r1
        BNE q1
p2      ADR r4,e
```

b.

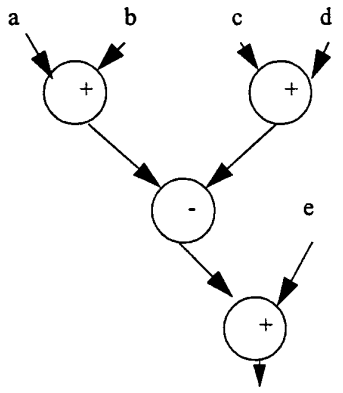
```
        ORG 100
p1      CMP r0,r1
        BEQ x1
p2      CMP r0,r2
        BEQ x2
p3      CMP r0,r3
        BEQ x3
```

Q5-6. Your linker uses a single pass through the set of given object files to find and resolve external references—each object file is processed in the order given, all external references are found, then the previously loaded files are searched for labels that resolve those references. Will this linker be able to successfully load a program with these external references and entry points?

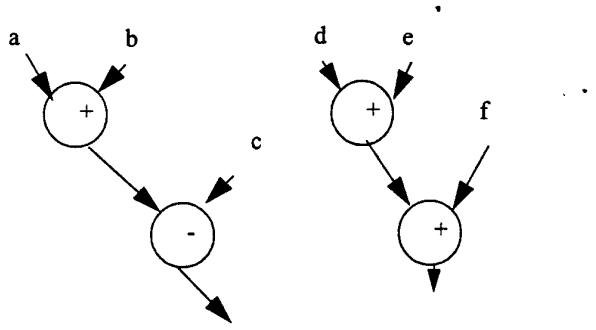
object file	entry points	external references
o1	a, b, c, d	s, t
o2	r, s, t	w, y, d
o3	w, x, y, z	a, c, d

Q5-7. Give the required order of execution of operations in these data flow graphs. If several operations may be performed in arbitrary order, show them as a set: {a+b, c-d}.

a.



b.



Q5-8. Draw the CDFG for this C code before and after applying dead-code elimination to the if statement:

```
#define DEBUG 0
proc1();
if (DEBUG) debug_stuff();
switch (foo) {
    case A: a_case();
    case B: b_case();
    default: default_case();
    }
```

Q5-9. Unroll the loop below

a. two times;

b. three times.

```
for (i=0; i<32; i++)
    x[i] = a[i]*c[i];
```

Q5-10. Can we apply code motion to this example? Explain.

```
for (i=0; i<N; i++)
    for (j=0; j<M; j++)
        z[i,j] = a[i] * b[i,j];
```

Q5-11. For each of the basic blocks of Question 5-3, determine the minimum number of registers required to perform the operations when they are executed in the order shown in the code. (You can assume that all computed values are used outside the basic blocks, so that no assignments can be eliminated.)

Q5-12. For each of the basic blocks of Question 5-3, determine the order of execution of operations that gives the smallest number of required registers. Also state the number of registers required in each case. (You can assume that all computed values are used outside the basic blocks, so that no assignments can be eliminated.)

Q5-13. Draw a data flow graph for the code fragment of Example 5-5. Assign an order of execution to the nodes in the graph so that no more than

four registers are required. Explain how you arrived at your solution using the structure of the data flow graph.

Q5-14. Determine the longest path through each code fragment, assuming that all statements can be executed in equal time and that all branch directions are equally probable.

a.

```
if (i < CONST1) { x = a + b; }
else { x = c - d; y = e + f; }
```

b.

```
for (i=0; i<32; i++)
    if (a[i] < CONST2)
        x[i] = a[i]*c[i];
```

c.

```
if (a < CONST3) {
    if (b < CONST4)
        w = r + s;
    else {
        w = r - s;
        x = s + t;
    }
} else {
    if (c > CONST5) {
        w = r + t;
        x = r - s;
        y = s + u;
    }
}
```

Q5-15. For each of the code fragments of Question 5-14, determine the shortest path through each code fragment, assuming that all statements can be executed in equal time and that all branch directions are equally probable.

Q5-16. The loop given below is executed on a machine that has a 1K word data cache with four words per cache line.

a. How must x and a be placed relative to each other in memory to produce a conflict miss every time the inner loop's body is executed?

b. How must x and a be placed relative to each other in memory to produce a conflict miss one out of four times the inner loop's body is executed?

c. How must x and a be placed relative to each other in memory to produce no conflict misses?

```
for (i=0; i<50; i++)
    for (j=0; j<4; j++)
        x[i][j] = a[i][j]*c[i];
```

Q5-17. Explain why the person generating clear-box program tests should not be the person who wrote the code being tested.

Q5-18. Find the cyclomatic complexity of the CDFGs for each of the code fragments given below.

a.

```
if (a < b) {
    if (c < d)
        x = 1;
    else
        x = 2;
} else {
    if (e < f)
        x = 3;
    else
        x = 4;
}
```

b.

```
switch (state) {
    case A:
        if (x = 1) { r = a + b; state = B; }
        else { s = a - b; state = C; }
        break;
    case B:
        s = c + d;
        state = A;
        break;
    case C:
        if (x < 5) { r = a - f; state = D; }
        else if (x == 5) { r = b + d; state = A; }
        else { r = c + e; state = D; }
        break;
    case D:
        r = r + 1;
        state = D;
        break;
}
```

c.

```
for (i=0; i<M; i++)
    for (j=0; j<N; j++)
        x[i][j] = a[i][j]*c[i];
```

Q5-19. Use the branch condition testing strategy to determine a set of tests for each of these statements.

a.

```
if (a < b || ptr1 == NULL) proc1();
else proc2();
```

b.

```
switch (x) {
case 0: proc1(); break;
case 1: proc2(); break;
```

```
case 2: proc3(); break;
case 3: proc4(); break;
default; dproc(); break;
}
```

c.

```
if (a < 5 && b > 7) proc1();
else if (a < 5) proc2();
else if (b > 7) proc3();
else proc4();
```

Q5-20. Find all the def-use pairs for each code fragment given below.

a.

```
x = a + b;
if (x < 20) proc1();
else {
    y = c + d;
    while (y < 10)
        y = y + e;
}
```

b.

```
r = 10;
s = a - b;
for (i=0; i<10; i++)
    x[i] = a[i] * b[s];
```

c.

```
x = a - b;
y = c - d;
z = e - f;
if (x < 10) {
    q = y + e;
    z = e + f;
}
if (z < y) proc1();
```

Q5-21. For each of the code fragments of Question 5-20, determine values for the variables that will cause each def-use pair to be exercised at least once.

Q5-22. You want to use random tests on an FIR filter program. How would you know when the program under test is executing correctly?

Lab Exercises

L5-1. Compare the source code and assembly code for a moderate-size program. (Most C compilers will give an assembly language listing with the -s flag.) Can you trace the high-level language statements in the assembly code? Can you see any optimizations that can be done on the assembly code?

L5-2. Write C code for an FIR filter. Measure the execution time of the filter, either using a simulator or by measuring the time on a running microprocessor. Vary the number of taps in the FIR filter and measure execution time as a function of the filter size.

L5-3. Generate a trace for a program using software techniques. Use the trace to analyze the program's cache behavior.

L5-4. Measure the power consumption of your microprocessor on a simple block of code.

Q5-23. Generate a set of functional tests for a moderate-size program. Evaluate your test coverage in one of two ways: have someone else independently identify bugs and see how many of those bugs your tests catch (and how many tests they catch that weren't found by the human inspector); inject bugs into the code and see how many of those were caught by your tests.

Processes and operating systems

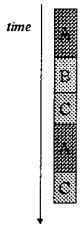
- Motivation for processes
- The process abstraction
- Context switching
- Multitasking
- Processes and UML

Why multiple processes?

- Processes help us manage timing complexity:
  - multiple rates
    - multimedia
    - automotive
  - asynchronous input
    - user interfaces
    - communication systems

Life without processes

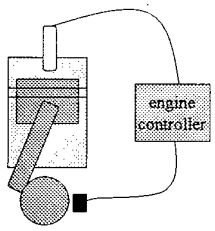
- Code turns into a mess:
  - interruptions of one task for another
  - spaghetti code



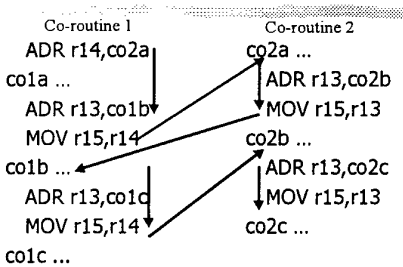

```
A_code();
...
B_code();
...
if (C) C_code();
...
A_code();
...
switch (x) {
  case C: C();
  case D: D();
  ...
```


Example: engine control

- Tasks:
  - spark control
  - crankshaft sensing
  - fuel/air mixture
  - oxygen sensor
  - Kalman filter



Co-routines



Co-routine methodology

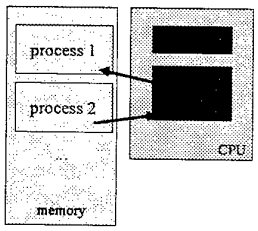
- Like subroutine, but caller determines the return address.
- Co-routines voluntarily give up control to other co-routines.
- Pattern of control transfers is embedded in the code.

Processes

- A process is a unique execution of a program.
  - Several copies of a program may run simultaneously or at different times.
- A process has its own state:
  - registers;
  - memory.
- The operating system manages processes.

Processes and CPUs

- Activation record: copy of process state.
- Context switch:
  - current CPU context goes out;
  - new CPU context goes in.

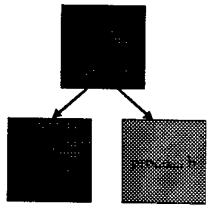


Terms

- Thread = lightweight process: a process that shares memory space with other processes.
- Reentrancy: ability of a program to be executed several times with the same results.

Processes in POSIX

- Create a process with fork:
  - parent process keeps executing old program;
  - child process executes new program.



fork()

- The fork process creates child:

```
childid = fork();
if (childid == 0) {
   /* child operations */
} else {
   /* parent operations */
}
```


execv()

- Overlays child code:

```
childid = fork();
if (childid == 0) {
   execv("mychild",childargs);
   perror("execv");
   exit(1);
}
```

file with child code

Context switching

- Who controls when the context is switched?
- How is the context switched?

Co-operative multitasking

- Improvement on co-routines:
  - hides context switching mechanism;
  - still relies on processes to give up CPU.
- Each process allows a context switch at cswitch() call.
- Separate scheduler chooses which process runs next.

Problems with co-operative multitasking

- Programming errors can keep other processes out:
  - process never gives up CPU;
  - process waits too long to switch, missing input.

Context switching

- Must copy all registers to activation record, keeping proper return value for PC.
- Must copy new activation record into CPU state.
- How does the program that copies the context keep its own context?

Context switching in ARM

- Save old process:

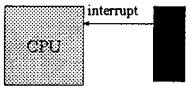
```
STMIA r13,{r0-r14}^
MRS r0,SPSR
STMDB r13,{r0,r15}
```

- Start new process:

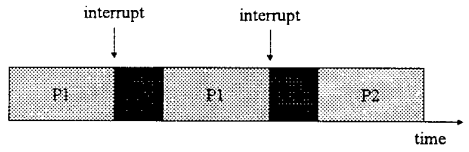
```
ADR r0,NEXTPROC
LDR r13,[r0]
LDMDB r13,{r0,r14}
MSR SPSR,r0
LDMIA r13,{r0-r14}^
MOVS pc,r14
```


Preemptive multitasking

- Most powerful form of multitasking:
  - OS controls when contexts switches;
  - OS determines what process runs next.
- Use timer to call OS, switch contexts:



Flow of control with preemption



Preemptive context switching

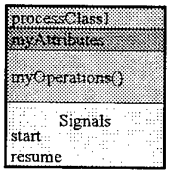
- Timer interrupt gives control to OS, which saves interrupted process's state in an activation record.
- OS chooses next process to run.
- OS installs desired activation record as current CPU state.

Why not use interrupts?

- We could change the interrupt vector at every period, but:
  - we would need management code anyway;
  - we would have to know the next period's process at the start of the current process.

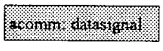
Processes and UML

- A process is an active class---independent thread of control.



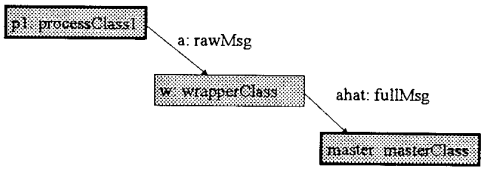
UML signals

- Signal: object that is passed between processes for active communication:



Designing with active objects

- Can mix normal and active objects:



Processes and operating systems

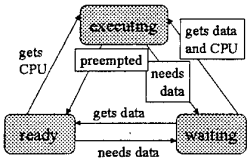
- Operating systems
- Scheduling policies

Operating systems

- The operating system controls resources:
  - who gets the CPU;
  - when I/O takes place;
  - how much memory is allocated.
- The most important resource is the CPU itself.
  - CPU access controlled by the scheduler.

Process state

- A process can be in one of three states:
  - executing on the CPU;
  - ready to run;
  - waiting for data.



Operating system structure

- OS needs to keep track of:
  - process priorities;
  - scheduling state;
  - process activatoin record.
- Processes may be created:
  - statically before system starts;
  - dynamically during execution.

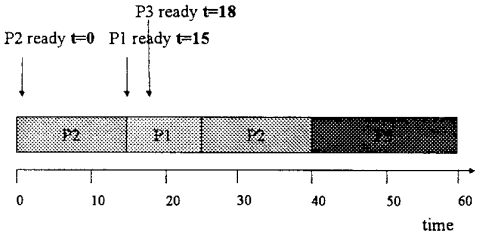
Priority-driven scheduling

- Each process has a priority.
- CPU goes to highest-priority process that is ready.
- Priorities determine scheduling policy:
  - fixed priority;
  - time-varying priorities.

Priority-driven scheduling example

- Rules:
  - each process has a fixed priority (1 highest);
  - highest-priority ready process gets CPU;
  - process continues until done or wait state.
- Processes
  - P1: priority 1, execution time 10
  - P2: priority 2, execution time 30
  - P3: priority 3, execution time 20

Priority-driven scheduling example



Process initiation disciplines

- Periodic process: executes on (almost) every period.
- Aperiodic process: executes on demand.

Timing requirements on processes

- Period: interval between process activations.
- Initiation interval: reciprocal of period.
- Initiation time: time at which process becomes ready.
- Deadline: time at which process must finish.

Timing violations

- What happens if a process doesn't finish by its deadline?
  - Hard deadline: system fails if missed.
  - Soft deadline: user may notice, but system doesn't necessarily fail.

Example: Space Shuttle software error

- Space Shuttle's first launch was delayed by a software timing error:
  - Primary control system PASS and backup system BFS.
  - BFS failed to synchronize with PASS.
  - Change to one routine added delay that threw off start time calculation.
  - 1 in 67 chance of timing problem.

Interprocess communication

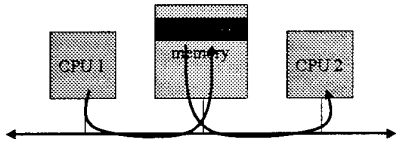
- Interprocess communication (IPC): OS provides mechanisms so that processes can pass data.
- Two types of semantics:
  - blocking: sending process waits for response;
  - non-blocking: sending process continues.

IPC styles

- Shared memory:
  - processes have some memory in common;
  - must cooperate to avoid destroying/missing messages.
- Message passing:
  - processes send messages along a communication channel---no common address space.

Shared memory

- Shared memory on a bus:



Race condition in shared memory

- Problem when two CPUs try to write the same location:
  - CPU 1 reads flag and sees 0.
  - CPU 2 reads flag and sees 0.
  - CPU 1 sets flag to one and writes location.
  - CPU 2 sets flag to one and overwrites location.

Atomic test-and-set

- Problem can be solved with an atomic test-and-set:
  - single bus operation reads memory location, tests it, writes it.
- ARM test-and-set provided by SWP:

```
        ADR r0,SEMAPHORE
        LDR r1,#1
GETFLAG SWP r1,r1,[r0]
        BNZ GETFLAG
```


Critical regions

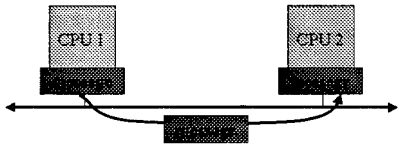
- Critical region: section of code that cannot be interrupted by another process.
- Examples:
  - writing shared memory;
  - accessing I/O device.

Semaphores

- Semaphore: OS primitive for controlling access to critical regions.
- Protocol:
  - Get access to semaphore with P().
  - Perform critical region operations.
  - Release semaphore with V().

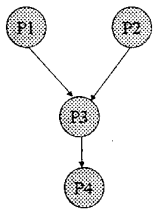
Message passing

- Message passing on a network:



Process data dependencies

- One process may not be able to start until another finishes.
- Data dependencies defined in a task graph.
- All processes in one task run at the same rate.



Other operating system functions

- Date/time.
- File system.
- Networking.
- Security.

Processes and operating systems

- Scheduling policies:
  - RMS;
  - EDF.
- Scheduling modeling assumptions.
- Interprocess communication.
- Power management.

Metrics

- How do we evaluate a scheduling policy:
  - Ability to satisfy all deadlines.
  - CPU utilization---percentage of time devoted to useful work.
  - Scheduling overhead---time required to make scheduling decision.

Rate monotonic scheduling

- RMS (Liu and Layland): widely-used, analyzable scheduling policy.
- Analysis is known as Rate Monotonic Analysis (RMA).

RMA model

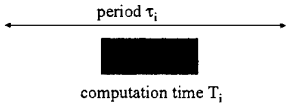
- All process run on single CPU.
- Zero context switch time.
- No data dependencies between processes.
- Process execution time is constant.
- Deadline is at end of period.
- Highest-priority ready process runs.

Rate-monotonic analysis

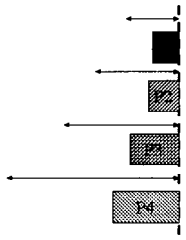
- Response time: time required to finish process.
- Critical instant: scheduling state that gives worst response time.
- Critical instant occurs when all higher-priority processes are ready to execute.

Process parameters

- T_i is computation time of process i; τ_i is period of process i.



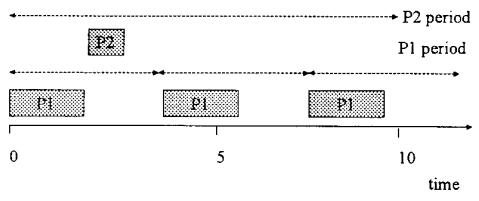
Critical instant



RMS priorities

- Optimal (fixed) priority assignment:
  - shortest-period process gets highest priority;
  - priority inversely proportional to period;
  - break ties arbitrarily.
- No fixed-priority scheme does better.

RMS example



RMS CPU utilization

- Utilization for n processes is
  - $\sum_i T_i / \tau_i$
- As number of tasks approaches infinity, maximum utilization approaches 69%.

RMS CPU utilization, cont'd.

- RMS cannot use 100% of CPU, even with zero context switch overhead.
- Must keep idle cycles available to handle worst-case scenario.
- However, RMS guarantees all processes will always meet their deadlines.

RMS implementation

- Efficient implementation:
  - scan processes;
  - choose highest-priority active process.

Earliest-deadline-first scheduling

- EDF: dynamic priority scheduling scheme.
- Process closest to its deadline has highest priority.
- Requires recalculating processes at every timer interrupt.

EDF analysis

- EDF can use 100% of CPU.
- But EDF may fail to miss a deadline.

EDF implementation

- On each timer interrupt:
  - compute time to deadline;
  - choose process closest to deadline.
- Generally considered too expensive to use in practice.

Fixing scheduling problems

- What if your set of processes is unschedulable?
  - Change deadlines in requirements.
  - Reduce execution times of processes.
  - Get a faster CPU.

Priority inversion

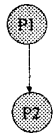
- Priority inversion: low-priority process keeps high-priority process from running.
- Improper use of system resources can cause scheduling problems:
  - Low-priority process grabs I/O device.
  - High-priority device needs I/O device, but can't get it until low-priority process is done.
- Can cause deadlock.

Solving priority inversion

- Give priorities to system resources.
- Have process inherit the priority of a resource that it requests.
  - Low-priority process inherits priority of device if higher.

Data dependencies

- Data dependencies allow us to improve utilization.
  - Restrict combination of processes that can run simultaneously.
- P1 and P2 can't run simultaneously.



Context-switching time

- Non-zero context switch time can push limits of a tight schedule.
- Hard to calculate effects---depends on order of context switches.
- In practice, OS context switch overhead is small.

POSIX scheduling policies

- SCHED_FIFO: RMS
- SCHED_RR: round-robin
  - within a priority level, processes are time-sliced in round-robin fashion
- SCHED_OTHER: undefined scheduling policy used to mix non-real-time and real-time processes.

Interprocess communication

- OS provides interprocess communication mechanisms:
  - various efficiencies;
  - communication power.

Signals

- A Unix mechanism for simple communication between processes.
- Analogous to an interrupt---forces execution of a process at a given location.
  - But a signal is caused by one process with a function call.
- No data---can only pass type of data.

POSIX signals

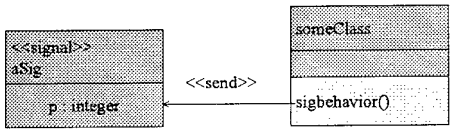
- Must declare a signal handler for the process using sigaction().
- Handler is called when signal is received.
- A signal can be seng with sigqueue():
  sigqueue(destpid,SIGRTMAX-1,sval)

POSIX signal types

- SIGABRT: abort
- SIGTERM: terminate process
- SIGFPE: floating point exception
- SIGILL: illegal instruction
- SIGKILL: unavoidable process termination
- SIGUSR1, SIGUSR2: user defined

Signals in UML

- More general than Unix signal---may carry arbitrary data:



POSIX shared memory

- POSIX supports counting semaphores with _POSIX_SEMAPHORES option.
  - Semaphore with N resources will not block until N processes hold the semaphore.
- Semaphores are given name:
  - /sem1
- P() is sem_wait(), V() is sem_post().

POSIX message-based communication

- Unix pipe supports messages between processes.
- Parent process uses pipe() to create a pipe.
  - Pipe is created before child is created so that pipe ID can be passed to chid.

POSIX pipe example

```
/* create the pipe */
if (pipe(pipe_ends) < 0) { perror("pipe"); break; }
/* create the process */
childid = fork();
if (childid == 0) { /* child reads from pipe_ends[1] */
   childargs[0] = pipe_ends[1];
   execv("mychild",childargs);
   perror("execv");
   exit(1);
}
else { /* parent writes to pipe_ends[0] */ ... }
```


Evaluating performance

- May want to test:
  - context switch time assumptions;
  - scheduling policy.
- Can use OS simulator to exercise process set, trace system behavior.

Processes and caches

- Processes can cause additional caching problems.
  - Even if individual processes are well-behaved, processes may interfere with each other.
- Worst-case execution time with bad behavior is usually much worse than execution time with good cache behavior.

Power optimization

- Power management: determining how system resources are scheduled/used to control power consumption.
- OS can manage for power just as it manages for time.
- OS reduces power by shutting down units.
  - May have partial shutdown modes.

Power management and performance

- Power management and performance are often at odds.
- Entering power-down mode consumes
  - energy,
  - time.
- Leaving power-down mode consumes
  - energy,
  - time.

Simple power management policies

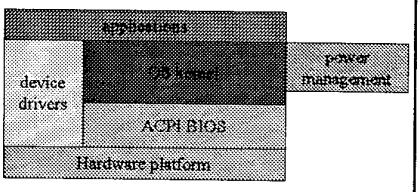
- Request-driven: power up once request is received. Adds delay to response.
- Predictive shutdown: try to predict how long you have before next request.
  - May start up in advance of request in anticipation of a new request.
  - If you predict wrong, you will incur additional delay while starting up.

Probabilistic shutdown

- Assume service requests are probabilistic.
- Optimize expected values:
  - power consumption;
  - response time.
- Simple probabilistic: shut down after time T_{on}, turn back on after waiting for T_{off}.

Advanced Configuration and Power Interface

- ACPI: open standard for power management services.



ACPI global power states

- G3: mechanical off
- G2: soft off
  - S1: low wake-up latency with no loss of context
  - S2: low latency with loss of CPU/cache state
  - S3: low latency with loss of all state except memory
  - S4: lowest-power state with all devices off
- G1: sleeping state
- G0: working state

System design techniques

- Design methodologies.
- Requirements and specification.

Design methodologies

- Process for creating a system.
- Many systems are complex:
  - large specifications;
  - multiple designers;
  - interface to manufacturing.
- Proper processes improve:
  - quality;
  - cost of design and manufacture.

Product metrics

- Time-to-market:
  - beat competitors to market;
  - meet marketing window (back-to-school).
- Design cost.
- Manufacturing cost.
- Quality.

Mars Climate Observer

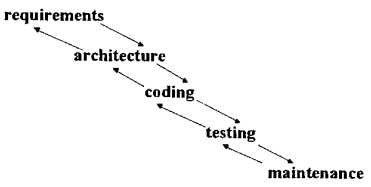
- Lost on Mars in September 1999.
- Requirements problem:
  - Requirements did not specify units.
  - Lockheed Martin used English; JPL wanted metric.
- Not caught by manual inspections.

Design flow

- Design flow: sequence of steps in a design methodology.
- May be partially or fully automated.
  - Use tools to transform, verify design.
- Design flow is one component of methodology. Methodology also includes management organization, etc.

Waterfall model

- Early model for software development:



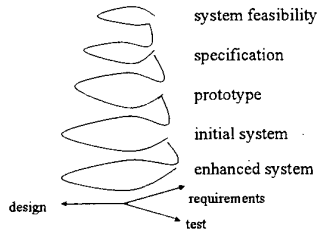
Waterfall model steps

- Requirements: determine basic characteristics.
- Architecture: decompose into basic modules.
- Coding: implement and integrate.
- Testing: exercise and uncover bugs.
- Maintenance: deploy, fix bugs, upgrade.

Waterfall model critique

- Only local feedback---may need iterations between coding and requirements, for example.
- Doesn't integrate top-down and bottom-up design.
- Assumes hardware is given.

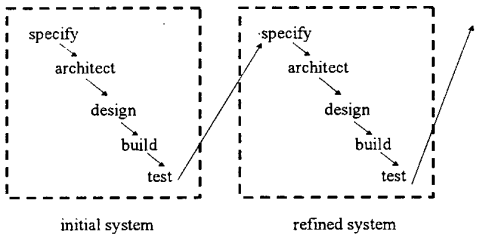
Spiral model



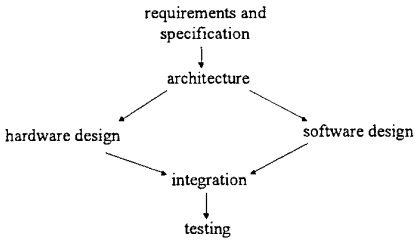
Spiral model critique

- Successive refinement of system.
  - Start with mock-ups, move through simple systems to full-scale systems.
- Provides bottom-up feedback from previous stages.
- Working through stages may take too much time.

Successive refinement model



Hardware/software design flow



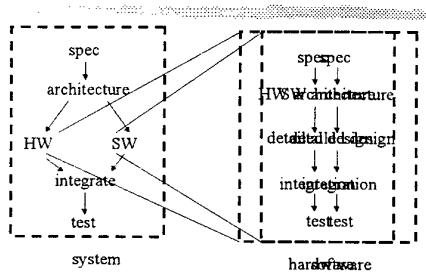
Co-design methodology

- Must architect hardware and software together:
  - provide sufficient resources;
  - avoid software bottlenecks.
- Can build pieces somewhat independently, but integration is major step.
- Also requires bottom-up feedback.

Hierarchical design flow

- Embedded systems must be designed across multiple levels of abstraction:
  - system architecture;
  - hardware and software systems;
  - hardware and software components.
- Often need design flows within design flows.

Hierarchical HW/SW flow



Concurrent engineering

- Large projects use many people from multiple disciplines.
- Work on several tasks at once to reduce design time.
- Feedback between tasks helps improve quality, reduce number of later design problems.

Concurrent engineering techniques

- Cross-functional teams.
- Concurrent product realization.
- Incremental information sharing.
- Integrated product management.
- Supplier involvement.
- Customer focus.

AT&T PBX concurrent engineering

- Benchmark against competitors.
- Identify breakthrough improvements.
- Characterize current process.
- Create new process.
- Verify new process.
- Implement.
- Measure and improve.

Requirements analysis

- Requirements: informal description of what customer wants.
- Specification: precise description of what design team should deliver.
- Requirements phase links customers with designers.

Types of requirements

- Functional: input/output relationships.
- Non-functional:
  - timing;
  - power consumption;
  - manufacturing cost;
  - physical size;
  - time-to-market;
  - reliability.

Good requirements

- Correct.
- Unambiguous.
- Complete.
- Verifiable: is each requirement satisfied in the final system?
- Consistent: requirements do not contradict each other.

Good requirements, cont'd.

- Modifiable: can update requirements easily.
- Traceable:
  - know why each requirement exists;
  - go from source documents to requirements;
  - go from requirement to implementation;
  - back from implementation to requirement.

Setting requirements

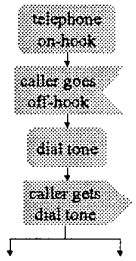
- Customer interviews.
- Comparison with competitors.
- Sales feedback.
- Mock-ups, prototypes.
- Next-bench syndrome (HP): design a product for someone like you.

Specifications

- Capture functional and non-functional properties:
  - verify correctness of spec;
  - compare spec to implementation.
- Many specification styles:
  - control-oriented vs. data-oriented;
  - textual vs. graphical.
- UML is one specification/design language.

SDL

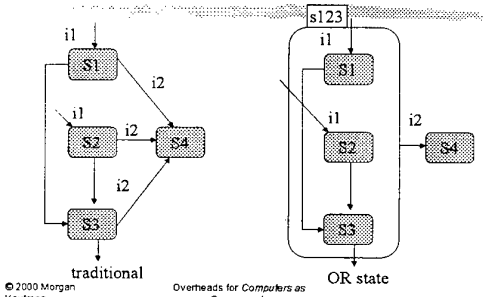
- Used in telecommunications protocol design.
- Event-oriented state machine model.



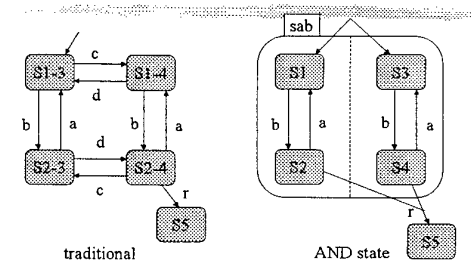
Statecharts

- Ancestor of UML state diagrams.
- Provided composite states:
  - OR states;
  - AND states.
- Composite states reduce the size of the state transition graph.

Statechart OR state



Statechart AND state



AND-OR tables

- Alternate way of specifying complex conditions:

  cond1 or (cond2 and !cond3)

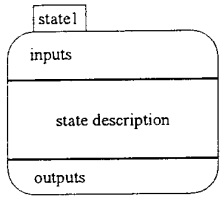
		OR	
	cond1	T	-
AND	cond2	-	T
	cond3	-	F

TCAS II specification

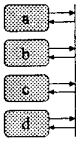
- TCAS II: aircraft collision avoidance system.
- Monitors aircraft and air traffic info.
- Provides audio warnings and directives to avoid collisions.
- Leveson et al used RMSL language to capture the TCAS specification.

RMSL

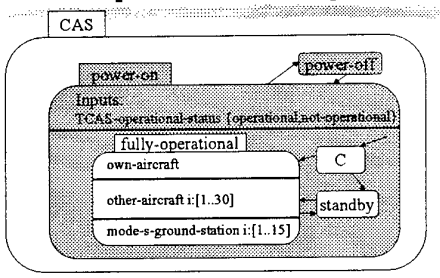
- State description:



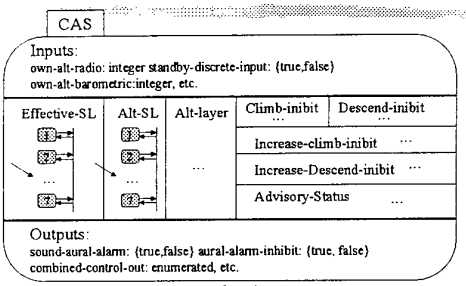
- Transition bus for transitions between many states:



TCAS top-level description



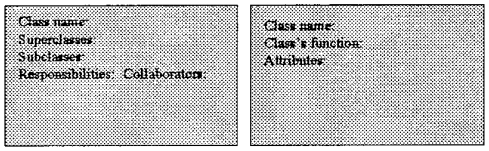
Own-Aircraft AND state



CRC cards

- Well-known method for analyzing a system and developing an architecture.
- CRC:
  - classes;
  - responsibilities of each class;
  - collaborators are other classes that work with a class.
- Team-oriented methodology.

CRC card format



front back

CRC methodology

- Develop an initial list of classes.
  - Simple descriptoin is OK.
  - Team members should discuss their choices.
- Write initial responsibilities/collaborators.
  - Helps to define the classes.
- Create some usage scenarios.
  - Major uses of system and classes.

CRC methodology, cont'd.

- Walk through scenarios.
  - See what works and doesn't work.
- Refine the classes, responsibilities, and collaborators.
- Add class relatoinships:
  - superclass, subclass.

CRC cards for elevator

- Real-world classes:
  - elevator car, passenter, floor control, car control, car sensor.
- Architectural classes: car state, floor control reader, car control reader, car control sender, scheduler.

Elevator responsibilities and collaborators

class	responsibilities	collaborators
Elevator car*	Move up and down	Car control, car sensor, car control sender
Car control*	Transmits car requests	Passenter, floor control reader
Car state	Reads current position of car	Scheduler, car sensor

Quality assurance

- Quality judged by how well product satisfies its intended function.
  - May be measured in different ways for different kinds of products.
- Quality assurance (QA) makes sure that all stages of the design process help to deliver a quality product.

Therac-25 Medical Imager (Leveson and Turner)

- Six known accidents: radiation overdoses leading to death and serious injury.
- Radiation gun controlled by PDP-11.
- Four major software components:
  - stored data;
  - scheduler;
  - set of tasks;
  - interrupt services.

Therac-25 tasks

- Treatment monitor controlled and monitored setup and delivery of treatment in eight phases.
- Servo task controlled radiation gun.
- Housekeeper task took care of status interlocks and limit checks.

Treatment monitor task

- Treat was main monitor task.
  - Eight subroutines.
  - Treat rescheduled itself after every subroutine.

Software timing race

- Timing-dependent use of mode and energy:
  - if keyboard handler sets completion behavior before operator changes mode/energy data, Datent task will not detect the change, but Hand task will.

Software timing errors

- Changes to parameters made by operator may show on screen but not be sensed by Datent task.
- One accident caused by entering mode/energy, changing mode/energy, returning to command linE in 8 seconds.
- Skilled operators typed faster, more likely to exercise bug.

Leveson and Turner observations

- Performed limited safety analysis: guessed at error probabilities, etc.
- Did not use mechanical backups to check machine operation.
- Used overly complex programs written in unreliable styles.

ISO 9000

- Developed by International Standards organization.
- Applies to a broad range industries.
- Concentrates on process.
- Validation based on extensive documentation of organization's process.

CMU Capability Maturity Model

- Five levels of organizational maturity:
  - Initial: poorly organized process, depends on individuals.
  - Repeatable: basic tracking mechanisms.
  - Defined: processes documented and standardized.
  - Managed: makes detailed measurements.
  - Optimizing: measurements used for improvement.

System design techniques

- Quality assurance.

Quality assurance

- Quality judged by how well product satisfies its intended function.
  - May be measured in different ways for different kinds of products.
- Quality assurance (QA) makes sure that all stages of the design process help to deliver a quality product.

Therac-25 Medical Imager (Leveson and Turner)

- Six known accidents: radiation overdoses leading to death and serious injury.
- Radiation gun controlled by PDP-11.
- Four major software components:
  - stored data;
  - scheduler;
  - set of tasks;
  - interrupt services.

Therac-25 tasks

- Treatment monitor controlled and monitored setup and delivery of treatment in eight phases.
- Servo task controlled radiation gun.
- Housekeeper task took care of status interlocks and limit checks.

Treatment monitor task

- Treat was main monitor task.
  - Eight subroutines.
  - Treat rescheduled itself after every subroutine.

Software timing race

- Timing-dependent use of mode and energy:
  - if keyboard handler sets completion behavior before operator changes mode/energy data, Datent task will not detect the change, but Hand task will.

Software timing errors

- Changes to parameters made by operator may show on screen but not be sensed by Datent task.
- One accident caused by entering mode/energy, changing mode/energy, returning to command linE in 8 seconds.
- Skilled operators typed faster, more likely to exercise bug.

Leveson and Turner observations

- Performed limited safety analysis: guessed at error probabilities, etc.
- Did not use mechanical backups to check machine operation.
- Used overly complex programs written in unreliable styles.

ISO 9000

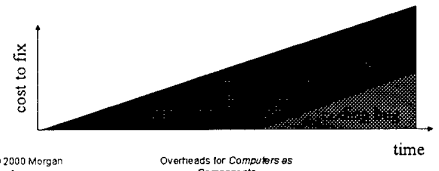
- Developed by International Standards organization.
- Applies to a broad range industries.
- Concentrates on process.
- Validation based on extensive documentation of organization's process.

CMU Capability Maturity Model

- Five levels of organizational maturity:
  - Initial: poorly organized process, depends on individuals.
  - Repeatable: basic tracking mechanisms.
  - Defined: processes documented and standardized.
  - Managed: makes detailed measurements.
  - Optimizing: measurements used for improvement.

Verification

- Verification and testing are important throughout the design flow.
- Early bugs are more expensive to fix:



Requirements and specification verification

- Requirements:
  - prototypes;
  - prototyping languages;
  - pre-existing systems.
- Specifications:
  - usage scenarios;
  - formal techniques.

Design review

- Uses meetings to catch design flaws.
  - Simple, low-cost.
  - Proven by experiments to be effective.
- Use other people in the project/company to help spot design problems.

Design review players

- Designers: present design to rest of team, make changes.
- Review leader: coordinates process.
- Review scribe: takes notes of meetings.
- Review audience: looks for bugs.

Before the deisgn review

- Design team prepares documents used to describe the design.
- Leader recruits audience, coordinates meetings, distributes handouts, etc.
- Audience members familiarize themselves with the documents before they go to the meeting.

Design review meeting

- Leader keeps meeting moving; scribe takes notes.
- Designers present the design:
  - use handouts;
  - explain what is going on;
  - go through details.

Design review audience

- Look for any problems:
  - Is the design consistent with the specification?
  - Is the interface correct?
  - How well is the component's internal architecture designed?
  - Did they use good design/coding practices?
  - Is the testing strategy adequate?

Follow-up

- Designers make suggested changes.
  - Document changes.
- Leader checks on results of changes, may distribute to audience for further review or additional reviews.

Measurements

- Measurements help ground our beliefs:
  - Do our practices really work?
  - Do they work where we think they work?
- Types of measurements:
  - bugs found at different stages of design;
  - bugs as a function of time;
  - bugs in different types of components;
  - how bugs are found.

Hardware/Software Co-synthesis

Wayne Wolf
Dept. of Electrical Engineering
Princeton University

Outline

- The problem
- Models and steps
- Classic co-synthesis: hardware/software partitioning
- Distributed system co-synthesis
- Control-oriented co-synthesis

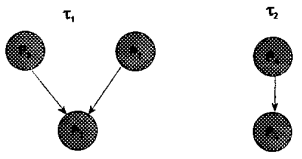
What is co-synthesis?

- Use tools to simultaneously generate hardware architecture and the software which runs on it.
  - Architecture may contain programmable CPUs, ASICs.
- Different styles of co-synthesis use different levels of inputs, synthesis steps, types of results.

Uses for co-synthesis

- Design space exploration:
  - architecture planning;
  - planning of multi-generation products;
  - marketing planning.
- Prototype creation.
- Implementation generation.

Task graph



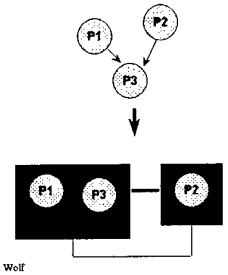
- Can model late arrivals, early departues by adding dummy processes.

Synthesis tasks

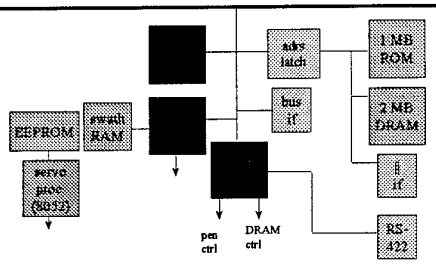
- *Scheduling*: make sure that data is available when it is needed.
- *Allocation*: make sure that processes don't compete for the PE.
- *Partitioning*: break operations into separate processes to increase parallelism, put serial operations in one process to reduce communication.
- *Mapping*: take PE, communication link characteristics into account.

Scheduling and allocation

- Must schedule/allocate
  - computation
  - communication
- Performance may vary greatly with allocation choice.



HP DesignJet architecture



DesignJet design strategy

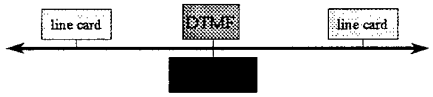
- Put high-rate but simple functions on peripheral processors.
  - Also moves control physically closer.
- Consolidate low-rate background tasks on main CPU.

Problems in scheduling/allocation

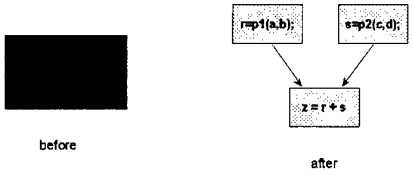
- Can multiple processes execute concurrently?
- Is the performance granularity of available components fine enough to allow efficient search of the solution space?
- Do computation and communication requirements conflict?
- How accurately can we estimate performance?
  - software
  - custom ASICs

TigerSwitch: allocation of DTMF

- Where do we put TouchTone (DTMF) detection?
  - on the CPU;
  - on the line cards;
  - on a co-processor board.



Partitioning example



Problems in partitioning

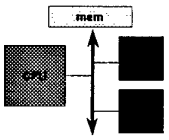
- At what level of granularity must partitioning be performed?
- How well can you partition the system without an allocation?
- How does communication overhead figure into partitioning?

Problems in mapping

- Mapping and allocation are strongly connected when the components vary widely in performance.
- Software performance depends on bus configuration as well as CPU type.
- Mappings of PEs and communication links are closely related.

Hardware-software partitioning

Architectural template: CPU + 1 or more ASICs on a bus:



Properties of classic partitioning algorithms

- Single-rate.
- Single-threaded: CPU waits for ASIC.
- Type of CPU is known; ASIC is synthesized. Closely coupled to high-level synthesis.

Hardware/software partitioning styles

- Two major styles:
- Vulcan starts with all-ASIC solution and moves functions to software to reduce cost (primal method).
- COSYMA starts with all-software solution and moves functions to ASIC to meet performance goal (dual method).

Vulcan

- Gupta and De Micheli: Target architecture: CPU + ASICs on bus
- Break behavior into threads at nondeterministic delay points; delay of thread is bounded
- Software threads run under RTOS; threads communicate via queues

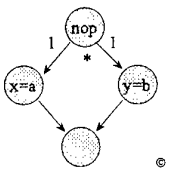
Specification and modeling

- Specified in Hardware C. Spec divided into threads at non-deterministic delay points.
- Hardware properties: size, # clock cycles.
- CPU/software thread properties:
  - thread latency
  - thread reaction rate
  - processor utilization
  - bus utilization

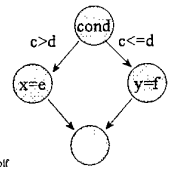
Vulcan thread modeling

Hardware C allows conjunctive, disjunctive execution:

conjunctive: x=a; y=b;



disjunctive: if (c>d) x=e; else y=f;



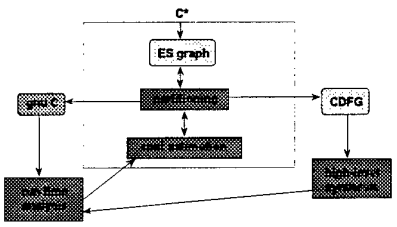
HW/SW allocation

- Start with unbounded-delay threads in CPU, rest of threads in ASIC.
- Optimization:
  - test one thread for move
  - if move to SW does not violate performance requirement, move the thread
  - feasibility depends on SW, HW run times, bus utilization
  - if thread is moved, immediately try moving its successor threads

COSYMA

- Ernst et al.: moves operations from software to hardware.
- Operations are moved to hardware in units of basic blocks.
- Estimates communication overhead based on bus operations and register allocation.
- Hardware and software communicate by shared memory.

COSYMA design flow



Cost estimation

- Speedup estimate for basic block *b*:
  - $\Delta c(b) = w(t_{HW}(b) - t_{SW}(b) + t_{com}(Z) - t_{com}(Z + b)) * It(b)$
  - w = weight, It(b) = # iterations taken on b
- Sources of estimates:
  - Software execution time (t_{SW}) is estimated from source code.
  - Hardware execution time (t_{HW}) is estimated by list scheduling.
  - Communiation time (t_{com}) is estimated by data flow analysis of adjacent basic blocks.

COSYMA optimization

- Goal: satisfy execution time. User specifies maximum number of function units in co-processor.
- Start with all basic blocks in software.
- Estimate potential speedup in moving a basic block to software using execution profiling.
- Search using simulated annealing. Impose high cost penalty for solutions that don't meet execution time.

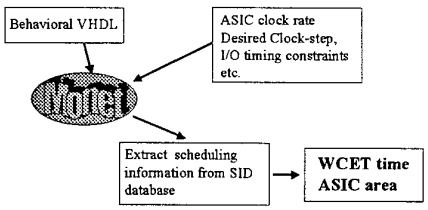
Two-phase optimization

- Inner loop uses estimates to search through design space quickly.
- Outer loop uses detailed measurements to check validity of inner loop assumptions:
  - code is compiled and measured
  - ASIC is synthesized
- Results of detailed estimate are used to apply correction to current solution for next run of inner loop.

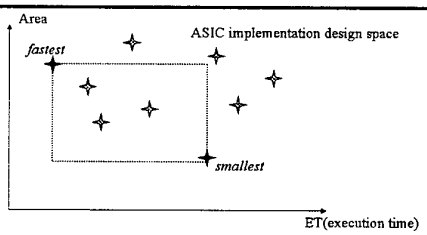
Co-synthesis with custom ASICs

- Xie, Wolf: optimize ASIC to meet requirements of system.
  - How fast does ASIC need to be relative to other system components?
- Requires detailed estimation of ASIC implementations.

ASIC performance analysis

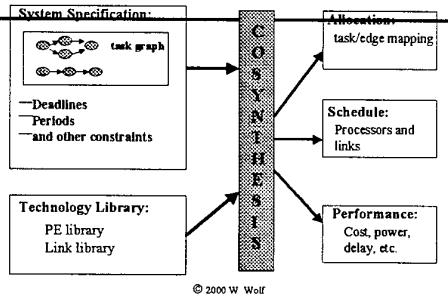


ASIC performance analysis

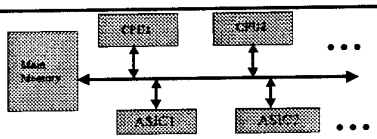


Find out 2 extremes and search intermediate solutions between these 2 extremes.

Problem formulation



Target archtecture



- *Goals :*
  a. Partition the tasks between software and hardware (CPU and ASIC)
  b. choose the number and types of CPU and ASIC such that the deadline is met while the cost is minimized.
  c. return the allocation and scheduling of the tasks on the resultant system.

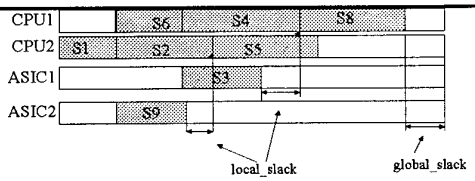
Algorithm outline

1. ***Pre-process and find an initial solution***
2. ***Iteratively reduce ASIC number and CPU cost***
   ASIC_to_CPU procedure
   CPU cost reduction
   Allocation and scheduling
3. ***ASIC cost reduction***
   global_slack
   local_slack

Iterative improvement

1. ***ASIC to CPU***: 2 heuristics to move processes from ASIC to CPU
   a. Difference of ASIC_ET and CPU_ET
   b. Non-Critical path ASIC
2. ***CPU cost reduction***: Reduce the existing CPU architecture cost.
   a. reduce the number of CPU
   b. replace the expensive CPU with a cheaper one
   c. reduce cache cost
3. ***Allocation and scheduling***
   a. static urgency
   b. dynamic urgency

ASIC cost reduction



global_slack=Min(deadline –completion_time)

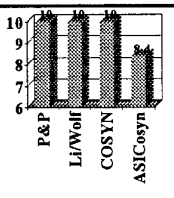
local_slack =Min(start_time_of_successors_nodes) –completion_time_ASIC_j

ASIC cost reduction

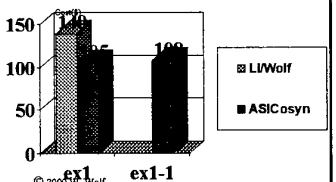
1. Sort ASICs by ***decreasing cost difference D*** (D=cost_of_fastest-cost of smallest)
2. For each ASIC_i in sorted list
   replace the fastest ASIC with the smallest ASIC
   if (meet deadline) use the smallest ASIC
   else keep the fastest
3. Reduce the fastest ASIC cost:
   for each ASIC_j{
   a Set ***ASIC_j_ET=(fastest_ET+global_slack+local_slack_j)***
   b foreach ASIC_i in other ASICs
   { *calculate local_slack_i* and
   set ASIC_i_ET=***fastest_ET+local_slack_i***}
   c call ASIC analysis tool to get the total ASIC cost COST(i)}
4. Select the minimal COST(i) in step 3 and set corresponding ASIC_ET.

Results

PP9 example (ASICosyn uses ASICs)



ex1 example (Li/Wolf)



Distributed system co-synthesis

- Can't take advantage of architectural template:
  - structure;
  - component characteristics.
- Generally multi-rate.

GCLP algorithm

Kalavade and Lee: global criticality/local phase; iterative algorithm.

Global criticality: critical path used to identify nodes to move onto ASIC.

Local phase: identify nodes which can be much more cheaply implemented in one medium than the other to reduce cost.

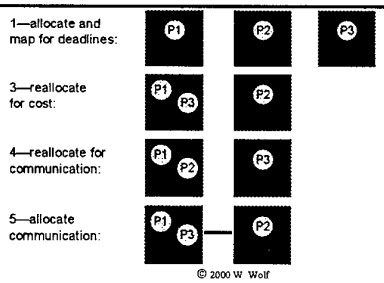
Successive-refinement co-synthesis

- Wolf: scheduling, allocation, and mapping are intertwined:
  - process execution time depends on CPU type selection
  - scheduling depends on process execution times
  - process allocation depends on scheduling
  - CPU type selection depends on feasibility of scheduling
- Solution: allocate and map conservatively to meet deadlines, then re-synthesize to reduce implementation cost.

A heuristic algorithm

1. Allocate processes to CPUs and select CPU types to meet all deadlines.
2. Schedule processes based on current CPU type selection; analyze utilization.
3. Reallocate processes to CPUs to reduce cost.
4. Reallocate again to minimize inter-CPU communication.
5. Allocate communication channels to minimize cost.
6. Allocate devices, to internal CPU devices if possible.

Example



PE cost reduction step

- Step 3 contributes most to minimizing implementation cost. Want to eliminate unnecessary PEs.
- Iterative cost reduction:
- reallocate all processes in one PE;
- pairwise merge PEs;
- balance load in system.
- Repeat until system cost is not reduced.

Co-synthesis with memory hierarchies

Li and Wolf: synthesize with caches.

Process timing: $t_{ideal} < t_{avg} < t_{worst}$.

Cache model: process is in cache or not in cache.

Use EDF scheduling, choosing execution times based on cache state.

Use iterative method to optimize design.

POLIS

Sangiovanni-Vincentelli et al: targets multi-rate reactive systems.

System modeled as network of Co-design FSMs (CFSMs).

Uses zero-delay hypothesis: communication happens instantaneously.

Polis, cont'd

Communiation can be analyzed by forming product of communicating machines.

Partitioning assigns functions to hardware and software.

Library components can be described as CFSMs.

Peformance analysis for reactive systems

Balarin and S-V: validate schedule for reactive system.

Specification is DAG of tasks with priorities and execution times.

Events initiate computation. Fpr every critical event from i to j, min time between 2 executions of i must be >= max time between exectution of i and j.

Compute partial loads to find answer.

Chinook

Borriello et al: targets control-dominated systems.

Major steps:

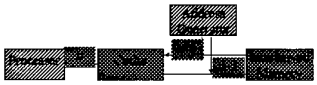
- HW/SW partitioning;
- device driver synthesis and low-level scheduling;
- I/O port allocation and interface synthesis;
- system-level scheduling;
- code generation.

Memory system optimization

- Lin, Wolf: design parallel memory systems and software:
  - transform arrays, loops;
  - design hardware to match.
- Designs to satisfy performance requirement.

Architecture model

- Prefetching and write-back
- Load balancing in the data path
- Assumptions: tractable data streams, fixed cache



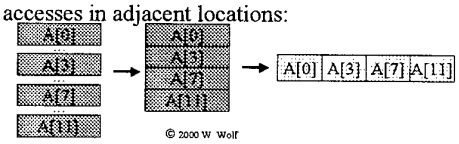
A simplified model

Design framework

1. Improve data access locality
2. Find out the extracted data streams
3. Estimate the minimal configuration
4. Reduce memory access conflicts. Interactively increase the number of memory channels and banks until the execution time is met.

Data access locality

- Varieties of locality:
  - Spatially close: Elements of the fastest changing dimension of array
  - Temporarily close: Iterations of the innermost loop
- Can improve performance by putting local accesses in adjacent locations:



In-dimension Stride Vector

- **Definition**: *In-dimension Stride Vector* (ISV)
  - Distance vector between two iterations that access through the same dimension of an array
- If ISV exists, then switch that dimension to be the fastest changing dimension:

  Transform the loop so that $Td = [0, 0, ..., 0, 1]^T$, where T is the transformation matrix, d is the ISV.

ISV construction

- Each array has its own set of ISVs.
- Each index into an array defines its own ISV space.
- Each ISV space is a candidate to be laid out for adjacent access.

Example

```
do i = 1, N
   do j = 1, N
      do k = 1, N
         c(i,j) += a(i,k)*b(k,j)
enddo
```

ISV: (i, j, k)
Array c:
1st ISV space: *span*{(1, 0, 0), (0, 0, 1)}, 2nd ISV space: *span*{(0, 1, 0), (0, 0, 1)}
Array a:
1st ISV space: *span*{(1, 0, 0), (0, 1, 0)}, 2nd ISV space: *span*{(0, 1, 0), (0, 0, 1)}
Array b:
1st ISV space: *span*{(1, 0, 0), (0, 0, 1)}, 2nd ISV space: *span*{(1, 0, 0), (0, 1, 0)}

Example, cont'd.

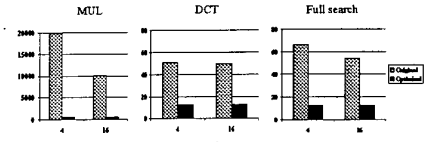
- Loop/data restructurings:
  - data layout: change layout of a, b, c to put fastest-changing dimension in adjacent locations
  - loop transformations: modify loop nest to match new data layout $(1, 0, 0) \Rightarrow (0, 0, 1)$
- Result:

```
do l = 1, N
   do m = 1, N
      do n = 1, N
         c(m,n) += a(l,n)*b(m,l)
enddo
```


Criteria for valid loop transformation

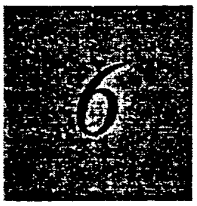
- Dependence vectors are still positive after transformation:
  - $Tp_i > 0$, p_i is dependence vector, $i = 1, 2, ..., M$
- Denotation: d: ISV, $p_{i\parallel}$: projection of p_i on d. $p_{i\perp} = p_i - p_{i\parallel}$. T_*: sub-matrix of T with the last row removed.
- **Lemma:** The transformation is valid if both the following hold:
  1) For those $p_{i\perp}$'s that equal to zero, the corresponding $p_{i\parallel}$'s all suffice $P_{i\parallel} d > 0$
  2) For those $p_{i\perp}$'s that do not equal to zero, then they must suffice $T_* p_{i\perp} > 0$

Experiments



Memory performance

MUL: 256*256 byte matrix multiplication
DCT: fast 8*8 DCT from MSSG on a 352*288 image (2 bytes/pixel)
Full search: full-search motion estimation on a 352*288 image (2 bytes/pixel).
Data Cache: 16KB, 2-way, 32B block. Main memory: 4-way capacity, 32B block



Processes and Operating Systems

6.1 Introduction

Although simple applications can be programmed on a microprocessor by writing a single piece of code, many applications are sophisticated enough that writing one large program does not suffice. When multiple operations must be performed at widely varying times, a single program can easily become too complex and unwieldy. The result is spaghetti code that is too difficult to verify for either performance or functionality.

process, operating system

This chapter studies the two fundamental abstractions which allow us to build complex applications on microprocessors: the **process** and the **operating system (OS)**. Together, these two abstractions let us switch the state of the processor between multiple tasks: the process cleanly defines the state of an executing program, while the operating system provides the mechanism for switching execution between the processes.

timing

These two mechanisms together let us build applications with more complex functionality and much greater flexibility to satisfy timing requirements. It is the need to satisfy complex timing requirements—events happening at widely different rates, intermittent events, etc.—which cause us to use processes and operating systems to build embedded software. Satisfying com-

plex timing tasks can introduce extremely complex control into programs. Using processes to compartmentalize functions and encapsulating in the operating system the control required to switch between processes makes it much easier to satisfy timing requirements with relatively clean control within the processes. We are particularly interested in **real-time operating systems (RTOSs)**—operating systems that provide facilities for satisfying real-time requirements.

POSIX

Throughout this chapter we will use the POSIX operating system [Gal95] as our example real-time operating system. POSIX is a version of the Unix operating system created by a standards organization. POSIX-compliant operating systems are source-code compatible—an application can be compiled and run without modification on a new POSIX platform assuming that the application uses only POSIX-standard functions. While Unix was not originally designed as a real-time operating system, POSIX has been extended to support real-time requirements. Many RTOSs are POSIX-compliant and it serves as a good model for basic RTOS techniques. The POSIX standard has many options; particular implementations do not have to support all options. The existence of features is determined by C preprocessor variables; for example, the FOO option would be available if the _POSIX_FOO preprocessor variable were defined. All these options are defined in the system include file unistd.h.

Linux

The Linux operating system has become increasing popular as a platform for embedded computing. Linux is POSIX-compliant operating system that is available as open source.

Web Enhanced

More information on POSIX, Linux, and other operating systems for embedded computing can be found at Web Aid 6-OS.

We will first use some examples to justify why embedded tasks should often be split into multiple processes. We will then introduce the process abstraction. In Section 6.4, we will study the **context switch**, which is the mechanism used to share one CPU among several processes. Section 6.5 examines operating systems in more detail. Section 6.6 analyzes several different policies for scheduling processes and Section 6.7 looks at interprocess communication mechanisms. In Section 6.8, we will look at some techniques for evaluating performance of operating systems, building upon our understanding of program performance analysis. Section 6.9 looks at operating system-controlled methods for managing system power consumption. Finally, in Section 6.10, we will use a telephone answering machine as an example of a system that is much easier to build out of communicating processes.

In this chapter:

The process abstraction.

Switching contexts between programs.

Real-time operating systems (RTOSs).

Interprocess communication.

Performance analysis and power consumption.

Example: telephone answering machine.

6.2 Multiple Tasks and Multiple Processes

Many (if not most) embedded computing systems do more than one thing—that is, the environment can cause mode changes which cause the embedded system to behave quite differently. For example, when designing a telephone answering machine, we may define recording a phone call and operating the user's control panel as distinct tasks, because they perform logically distinct operations and they must be performed at very different rates. These different **tasks** are part of the system's functionality, but that application-level organization of functionality is often reflected in the structure of the program as well. The need to structure our programs to perform multiple tasks is the motivation behind introducing the notion of a process.

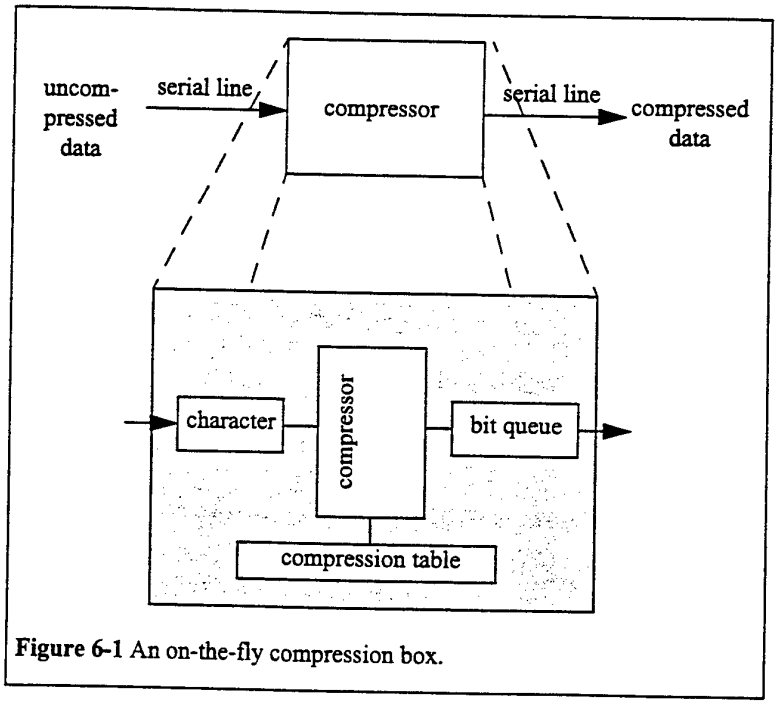


Figure 6-1 An on-the-fly compression box.

To understand why the separation of an application into tasks may be reflected in the program structure, consider how we would build a stand-alone compression unit based on the compression algorithm we implemented in Section 3.8. As shown in Figure 6-1, this device is connected to serial ports on both ends. The input to the box is an uncompressed stream of bytes. The box emits a compressed string of bits on the output serial line, based on a pre-defined compression table. Such a box may be used, for example, to compress data being sent to a modem.

variable data rates

The need to receive and send data at different rates—for example, the program may emit two bits for the first byte, then seven bits for the second byte—will obviously find itself reflected in the structure of the code. It is easy to create irregular, ungainly code to solve this problem; a more elegant solution is to create a queue of output bits, with those bits being removed from the queue and sent to the serial port in 8-bit sets. But beyond the need to create a clean data structure which simplifies the control structure of the code, we must also make sure that we process the inputs and outputs at the proper rates. For example, if we spend too much time packaging and emitting output characters, we may drop an input character. Solving timing problems is a more challenging problem.

asynchronous input

The text compression box provides a simple example of rate control problems. A control panel on a machine provides an example of a different sort of rate control problem, the **asynchronous input**. The control panel of the compression box may, for example, include a compression mode button which disables or enables compression, so that the input text is passed through unchanged when compression is disabled. We certainly don't know when the user will push the compression mode—the button may be depressed asynchronously relative to the arrival of characters for compression.

We do know, however, that the button will be depressed at a much lower rate than characters will be received, since it is not physically possible for a person to repeatedly depress a button at even slow serial line rates. Keeping up with the input and output data while checking on the button can introduce some very complex control code into the program. Sampling the button's state too slowly can cause us to miss a button depression entirely, but sampling it too frequently can cause the machine to incorrectly compress data. One solution is to introduce a counter into the main compression loop, so that a subroutine to check the input button is called once every n times the compression loop is executed. But this solution doesn't work when either the compression loop or the button-handling routine have highly-variable execution times—if the execution time of either varies significantly, it will cause the other to execute later than expected, possibly causing data to be lost. We need to be able to keep track of these two different tasks separately, applying different timing requirements to each; that is the sort of control that processes allow us.

These two examples illustrate how requirements on timing and execution rate can create major problems in programming. When code is written to satisfy several different timing requirements at once, the control structures necessary to get any sort of solution become very complex very quickly. Worse, such complex control is usually quite hard to verify for either functional or timing properties.

6.2.1 Multi-Rate Systems

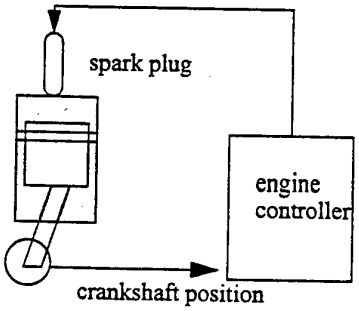
periodic computation

Implementing code which satisfies timing requirements is even more complex when multiple rates of computation must be handled. Multi-rate systems are very common: automobile engines, printers, and telephone PBXs

are all examples of **multi-rate** embedded computing systems. In all these systems, certain operations must be executed periodically, and each operation is executed at its own rate. Application Note 6-1 describes why automobile engines require multi-rate control.

Application Note 6-1: Automotive engine control

The simplest automotive engine controllers, such as the ignition controller for a basic motorcycle engine, perform only one task—timing the firing of the spark plug, taking the place of a mechanical distributor. The spark plug must be fired at a certain point in the combustion cycle, but to obtain better performance, the phase relationship between the piston's movement and the spark should change as a function of engine speed. Using a microcontroller which senses the engine crankshaft position allows the spark timing to vary with engine speed. Firing the spark plug is a periodic process (but note that the period depends on the engine's operating speed).



The control algorithm for a modern automobile engine is much more complex, making the need for microprocessors that much greater. Automobile engines must meet strict requirements (mandated by law in the United States) on both emissions and fuel economy. On the other hand, the engines must still satisfy customers not only with performance, but with ease of starting in extreme cold and heat, low maintenance, etc.

Automobile engine controllers use additional sensors, including the gas pedal position and an oxygen sensor used to control emissions. They also use a multi-mode control scheme: for example, one mode may be used for engine warm-up, another for cruise, yet another for climbing steep hills, etc. The larger number of sensors and modes increases the number of discrete tasks that must be performed. The highest-rate task is still firing the spark plugs. The throttle setting must be sampled and acted upon regularly, though not as frequently as the crankshaft setting and the spark plugs. The oxygen sensor responds much more slowly than the throttle, so adjustments to the fuel/air mixture suggested by the oxygen sensor can be computed at a much lower rate.

period, rate

The **period** of a process is the time between successive executions. For example, the period of a digital filter is defined by the time interval between

successive input samples. The process's **rate** is the inverse of its period. In a multi-rate system, each process executes at its own distinct rate.

When a machine's processes have periods which are simply related, designing the code to execute all the processes at the proper rates may not be too difficult. For example, if one process operates at half the rate of the other, the faster process's execution loop can use the counter trick to call the inner process on every other loop iteration. However, if the least-common multiple of the periods is large, this solution becomes very unwieldy. The automobile engine poses an interesting problem because some of the computation rates are not constant—several processes, such as firing the spark plugs, have rates which depend on the engine's rotation speed.

6.2.2 Early Multitasking Technology: The Co-Routine

The **co-routine** was a programming technique commonly used in the early days of embedded computing to handle multiple processes. The co-routine structure is not used very frequently today, though still has its uses, but it provides a useful example of the problems inherent in trying to devise complex control between multiple processes without using processes.

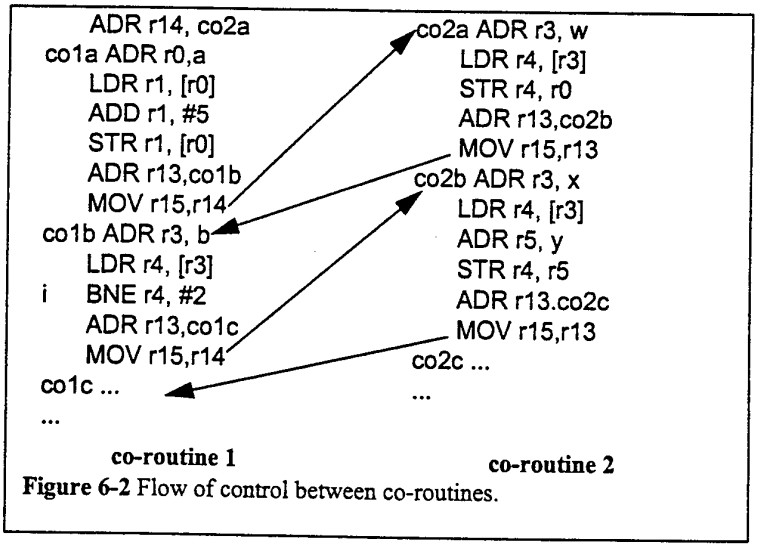


Figure 6-2 Flow of control between co-routines.

The basic co-routine structure, which is best understood in assembly language, is illustrated in Figure 6-2. The ARM code in the co-routines isn't meant to be meaningful computations; what is important in this example is the structure of the jumps between the co-routines. Each co-routine has several different entry points; we use the labels here only for convenience. The co-routine generalizes the subroutine register usage so that we can return to either co-routine at the point from which we left it: we use r14 to hold the return address for co-routine 2 and r13 for co-routine 1's return address. When one routine wants to transfer control to the other co-routine, it loads up its return address in the proper register, then sets the PC to the other co-

routine's return value. The co-routines, of course, must be careful not to destroy values in the registers needed by the other co-routines.

The co-routine structure lets us implement more general kinds of flow of control than is possible with only subroutines; the identification of co-routine entry points provides us some hooks for non-hierarchical calls and returns within the program. However, the co-routine does not do nearly enough to help us construct complex programs with significant timing properties. The co-routine in general does very little to simplify the design of code which satisfies timing requirements. Furthermore, even without timing constraints, co-routines can be ugly to work with: it is often hard to trace through the flow of control between co-routines, particularly when the system is built from three or more co-routines.

6.3 Processes

The process is the first tool we need to help us tame complex systems—it provides a fundamental abstraction for dealing with multiple simultaneous operations. While the procedure helps us organize source code into manageable units, the process lets us organize executable code into similarly manageable units. Once we have divided the system into processes that interact cleanly, we can apply operating system techniques to manage the interactions between the processes.

A process is simply a unique execution of a program. That is, a process is defined both by its code and its data. Two copies of a single program, executing on their own data, constitute two distinct processes. The data set for a process includes not only the CPU registers, but also its main memory, since the process's results may be stored in both. (The need to consider all data in a process is strong motivation for avoiding self-modifying code. Processes executing self-modifying code must each have their own copy of the code. If a program does not modify its own instructions, on the other hand, all the processes spawned from it can share a single copy of the instructions.)

We can imply from the above that a process is executed sequentially, since it is written in terms of CPU instructions. Furthermore, a CPU may execute at most one process at a time. However, because we have identified the complete state of the process, we can cause the CPU to stop executing one process and start executing another: by changing the program counter to the new process's code and similarly moving its data into position in the registers and main memory, we cause a context switch to the new process. We will study how to implement context switching in detail in Section 6.4.

Figure 6-3 illustrates processes co-habitating on a CPU. The currently-executing process (process 2 in this case) has the program counter pointing within its own code. The main memory contents for the processes in this example all co-exist within main memory. (We could store non-executing process code and data on a disk, as is done in most general-purpose systems, but many embedded systems do not require the cost and complexity of a disk.) In addition to the code and data for each process, we need to keep a separate record of the state of each process. The record for a process is often

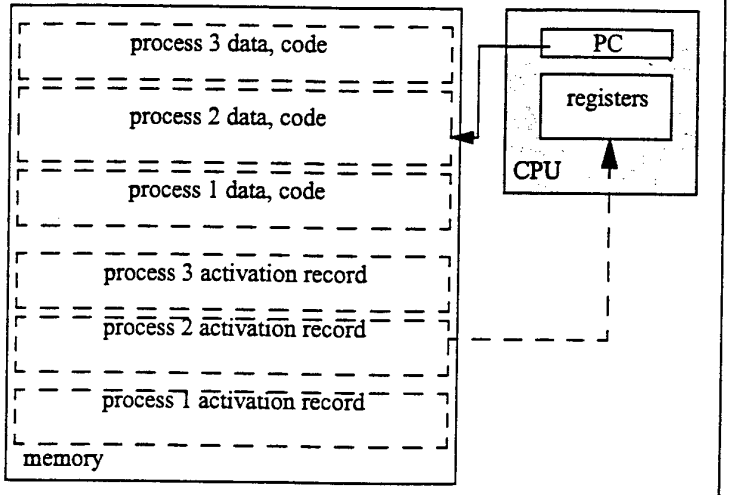


Figure 6-3 Several processes on a single CPU.

called its **activation record**, since it contains the data used to re-activate the process. The state for process 2 was copied into the CPU from its activation record at the beginning of the process's current execution. The other, non-executing processes have their own activation records which record the CPU internal state when those processes were stopped. That state will be used to reactivate the processes at a later date.

One commonly-used form of process in embedded systems is the **lightweight process**, also known as a **thread**. Lightweight processes have their own distinct sets of values for CPU registers, but co-habitate the same main memory space. As a result, one lightweight process may inadvertently destroy the data of another lightweight process executing on the machine. Lightweight processes are commonly used in embedded computing to avoid the cost and complexity of memory management units that provide strict separation between memory spaces. General-purpose computing platforms typically run heavyweight processes with full protection of process memory spaces because they may run user code which has not been tested for correctness and safety. A memory management unit can map physical memory into distinct logical memory spaces, ensuring that processes do not interfere with each other. However, since embedded computing platforms generally run only a few programs which can be tested before deployment, it usually makes sense to use a lightweight process model and save the cost of the MMU.

reentrancy

A program is said to be **reentrant** if it can be executed several times without reloading it from some pristine copy. An example of a non-reentrant program is one which first reads some global variable, then writes it—if the program were restarted, it would see a different value for that global variable. Non-reentrant programs are prone to obscure bugs that can be found only when the program is executed multiple times. A process does not have to be

written as a reentrant program, but the system will be easier to debug if all processes are reentrant.

processes in POSIX

Creating a process in POSIX requires somewhat cumbersome code although the underlying principle is elegant. In POSIX, a new process is created by making a copy of an existing process. The copying process creates two different processes both running the same code. The complication comes in ensuring that one process runs the code intended for the new process while the other process continues the work of the old process.

A process makes a copy of itself by calling the fork() function. That function causes the operating system to create a new process (the child process) which is a nearly exact copy of the process that called fork() (the parent process). They both share the same code and the same data values with one exception, the return value of fork(): the parent process is returned the process ID number of the child process, while the child process gets a return value of 0. We can therefore test the return value of fork() to determine which process is the child:

```
childid = fork();
if (childid == 0) { /* must be the child */
    /* do child process here */
}
```

However, it would be clumsy to have both processes have all the code for both the parent and child processes. POSIX provides the exec facility for overloading the code in a process. We can use that mechanism to overlay the child process with the appropriate code. There are several versions of exec; execv() takes a list of arguments to the process in the form of an array, just as would be accepted by a typical UNIX program from the shell. So here is the process call with an overlay of the child's code on the child process:

```
childid = fork();
if (childid == 0) { /* must be the child */
    execv("mychild",childargs);
    perror("execv");
    exit(1);
}
```

The execv() function takes as argument the name of the file that holds the child's code and the array of arguments. It overlays the process with the new code and starts executing it from the main() function. In the absence of an error, execv() should never return. The code that follows the call to perror() and exit(), take care of the case where execv() fails and returns to the parent process. The exit() function is a C function that is used to leave a process; it relies on an underlying POSIX function is called _exit().

The parent process should use one of the POSIX wait functions before calling exit() for itself. The wait functions not only return the child process's status, in many implementations of POSIX they make sure that the child's resources (namely memory) are freed. So we can extend our code as follows:

```
childid = fork();
if (childid == 0) { /* must be the child */
```

```
    execv("mychild",childargs);
    perror("execl");
    exit(1);
    }
else { /* is the parent */
    parent_stuff(); /* execute parent functionality */
    wait(&cstatus);
    exit(0);
    }
```

The parent_stuff() function performs the work of the parent function. The wait() function waits for the child process; the function sets the integer cstatus variable to the return value of the child process.

the POSIX process model

POSIX does not implement lightweight processes. Each POSIX process runs in its own address space and cannot directly access the data or code of other processes. We will see in Section 6.7.4 that POSIX must supply a mechanism for shared memory to allow processes to communicate in this way.

6.4 Context Switching

The context switch is the mechanism for moving the CPU from one executing process to another. Clearly, the context switch needs to be bug-free: a process that does not look at a real-time clock should not be able to tell that it was stopped and then restarted. Furthermore, we want context switching to be fast—time spent by the CPU on context switching is time not spent on the computations for which the system is built.

6.4.1 Co-operative Multitasking

Before we study the most general form of context switching, preemptive multitasking, let us consider a more restricted form and its limitations. In a co-operative multitasking system, one process gives up the CPU to another voluntarily; this voluntarism can introduce severe bugs.

A context switch in a co-operative multitasking system can be implemented as a something that looks like a procedure call, but is not a standard procedure call because it does not immediately return to the caller. As shown in Figure 6-4, each process contains calls to a context-switch function (switch() in this case). cswitch() does not start a new procedure, but rather copies in the state of another process. When that process makes its own call to cswitch(), the system may go back to executing the original process, or to executing yet another process. A separate **scheduler** determines which process will be executed next. The scheduler first saves the state of the process that called the scheduler, then determines which process to call next, and finally sets the CPU state to the new process. This structure is similar to the co-routine structure of Section 6.2.2. The main difference is that, rather than directly calling the other process, we use the scheduler to determine what process runs next. This gives us much more flexibility in choosing the order

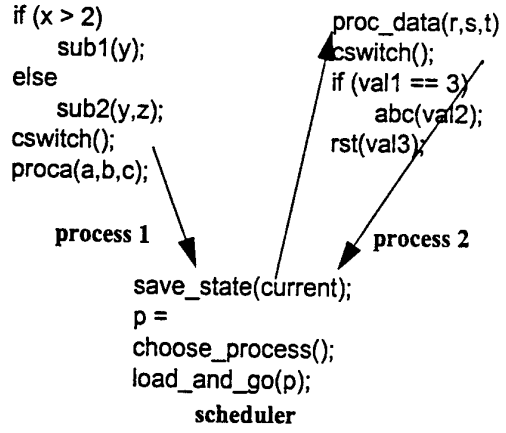


Figure 6-4 A set of cooperatively multitasking processes.

in which processes run, but it still means that a process will continue to execute until it voluntarily turns over control to another process.

The process states are stored in process activation records in main memory. CPU's procedure call mechanism is a starting point for the context switch operation:

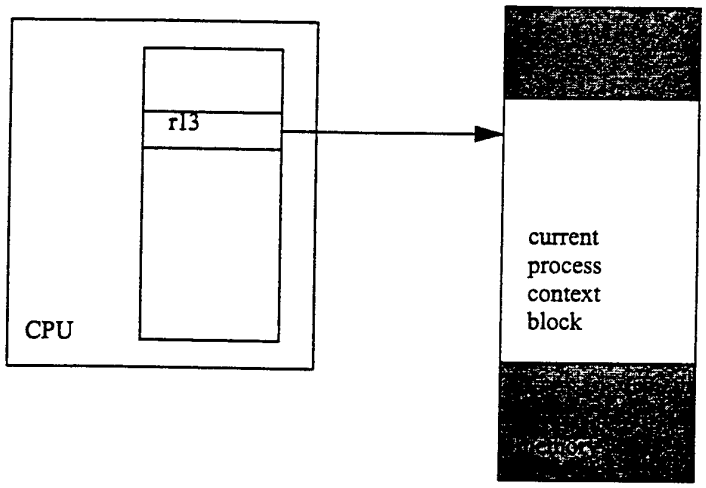
- The call to cswitch() uses standard procedure call mechanisms to generate a return address.
- cswitch() copies the procedure call state, as well as other registers not saved by the procedure call, to a process state record somewhere in memory.
- cswitch(), after deciding which process will execute next, can copy that process's state record into the locations normally used for procedure call state.
- Executing a standard procedure call will then return control to the selected process.

Example 6-1 describes in more detail how to implement cswitch() on the ARM.

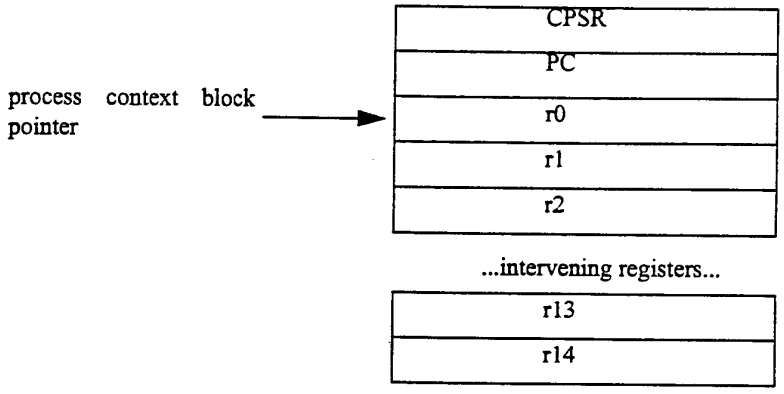
Example 6-1 Co-operative context switching for the ARM

In this co-operative context switching mechanism for the ARM, we will assume that we have lightweight processes. Our main job, therefore, is to save the user-mode processor registers and restore their values from another process.

The standard way to maintain a process's context [Jag96] is to store a pointer to the current process's context in r13. Here is how the CPU's registers relate to the process context block:



Here is the format we will use to save the registers in a context block (the reason for this order of appearance of registers will become apparent in a moment):



For simplicity, we will assume that the next process's context block is pointed to by a variable named NEXTPROC.

To better understand the code, let us separately consider the steps required for saving the old context and restoring the new one. Here is the code for saving the state of the old process:

```
STMIA  r13, {r0-r14}^   ; save registers up from [r13]
MRS    r0, SPSR         ; get status register
STMDB r13, {r0, r15}    ; save status register and PC
```

Remember that r13 always holds the pointer to the current process's context block. The store multiple instruction (STMIA) stores all the user registers into the space pointed to by r13. The IA suffix causes the address to be incremented after each register is stored, causing the registers to be stored in ascending order. The ^ directive sets the S bit in the STM instruction, which causes the user-mode registers to be saved. The next instruction (MRS) gets the status register and puts it in r0. The STMDB instruction saves the status register (copied into r0) and the PC into the context block.

We can get the new process's state by loading r13 first, then using that to load all the other registers:

```
ADR    r0, NEXTPROC ; get address for pointer
LDR    r13, [r0]    ; get the pointer to the next context block
LDMDB  r13, {r0, r14} ; get status register and PC
MSR    SPSR, r0     ; set the status register
LDMIA  r13, {r0-r14}^ ; get the registers
MOVS   pc, r14      ; restore the status register
```

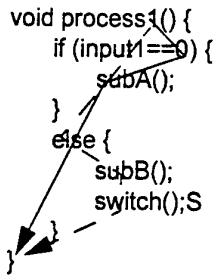
If we have a floating-point co-processor, we must save and restore its registers as well.

Co-operative multitasking has been used in both embedded computing and in general-purpose computing systems. However, it is prone to bugs which cause the system to become totally inoperable. The fact that control of the CPU must be voluntarily given up by a process means that simple programming mistakes can cause the system to lock up, failing to respond to input and to be totally inoperable.

The next example illustrates a simple case in which program bugs break the co-operative multitasking mechanism. Even logical bugs are not necessary to cause problems in co-operative multitasking. If a process executes for too long before passing control back to the scheduler, the system's real-time characteristics are at risk. Co-operative multitasking can work well, but it does not provide a mechanism to overcome bugs introduced by the individual processes that don't properly obey the co-operative multitasking protocol.

Example 6-2 A buggy co-operative multitasking system

As a first example, consider this program in a co-operative multitasking system:



This process is defined as a procedure; once the process is executed once, it will be executed again, since systems assume that most processes execute repeatedly. The if statement in this process handles different cases for input1, which is presumably set by some input device through another process. If input1 is zero, then a context switch is not executed by this process—switch() is called along one paths of execution for process1() (namely the dotted path), but not for every path (the solid path being an exception). Depending on what other processes are defined and the manner in which the operating system selects processes, failing to allow context switching within process1() may or may not cause the system to lock up.

That case is relatively simple and potentially harmless. But other flow-of-control problems can cause the system to completely lock up. Consider this code:

```
process2() {
    x = global1; /* global1 is an input to the process */
    while (x < 500)
        x = aproc(global2); /* subroutine does its work */
    switch();
}
```

At first glance, it may appear that switch() is called along every execution path of process2(). However, the flow of control depends on the value for x computed by the procedure aproc(). Depending on what the values of global1 and global2 passed into the process and the function performed by aproc(), it is quite possible that x is always less than 500 and that the while loop executes forever. In this case, unless aproc() calls switch() itself, the operating system will never get a chance to switch contexts because process2() will never exit.

This example illustrates the danger of subroutines in co-operative multitasking systems—the properties of the subroutine, either inherent or in combination with the calling code, can cause the flow of control in the program to be restricted in non-obvious ways that prevent context switches from occurring. The implementation-hiding property of the subroutine is, in this case, a distinct disadvantage.

6.4.2 Preemptive Multitasking

The interrupt is an ideal mechanism on which to build context switching for preemptive multitasking. Like the procedure call, the interrupt mechanism is designed to save state and then invisibly return control to an executing program. However, interrupts force the CPU to transfer control to the operating system. Not only does preemption reduce the consequence of programming errors, but we will also see in later sections that preemption allows CPU time to be allocated more efficiently between multiple processes with deadlines.

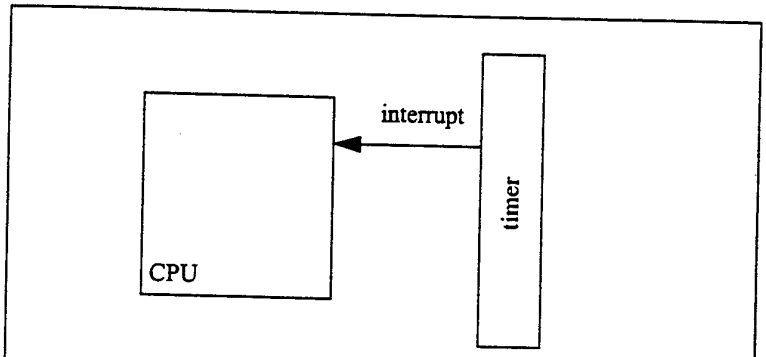


Figure 6-5 A timer supplying interrupts for preemptive scheduling.

timer-driven preemption

Figure 6-5 shows the basic hardware configuration for preemptive multitasking. A timer generates periodic interrupts to the CPU. The interrupt handler for the timer calls the operating system, which saves the previous process's state in an activation record, selects the next process to execute and switches the context to that process.

An interrupt-driven, preemptive context switch follows a general flow similar to that of the procedure-driven, co-operative task switch:

- The interrupt causes the flow of control to switch to the interrupt handler, which saves a minimum amount of state.
- The interrupt state is saved in a process state record, along with the remaining registers for the process.
- After choosing the process to run next, the operating system copies its state into the areas used by the interrupt return.
- Returning from the interrupt causes the chosen process to resume execution at the point where it was previously interrupted.

We can implement preemptive context switches using the same basic techniques shown in Example 6-1. The only difference between the two is the triggering event: voluntary release of the CPU in the case of co-operative, timer interrupt in the case of preemptive.

6.4.3 Processes and Object-Oriented Design

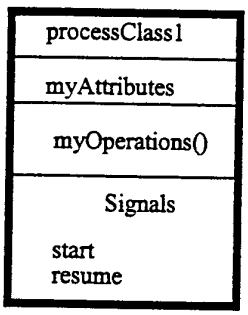


Figure 6-6 An active class in UML.

We need to design systems with processes as components. In this section, we will survey the ways we can describe processes in UML and how to use processes as components in object-oriented design.

active classes and objects

UML often refers to processes as **active objects**—objects that have independent threads of control. The class that defines an **active object** is known as an **active class**. Figure 6-6 shows an example of a UML active class. It has all the normal characteristics of a class, including a name, attributes, and operations. It also provides a set of signals that can be used to communicate with the process. A signal is an object that is passed between processes for asynchronous communication; we will describe signals in more detail in Section 6.7.

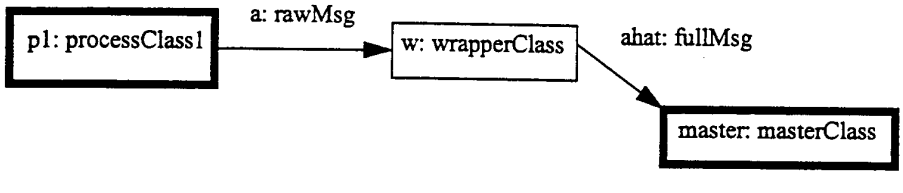


Figure 6-7 A collaboration diagram with active and normal objects.

We can mix active objects and normal objects when describing a system. Figure 6-7 shows a simple collaboration diagram in which an object is used as an interface between two processes: *p1* uses the *w* object to manipulate its data before the data is sent to the *master* process.

6.5 Operating Systems

The operating system's fundamental job is to allocate resources in the computing system among programs that request them. Naturally, the CPU is scarcest resource, so scheduling the CPU is the most important job of the operating system. In this section, we will consider the structure of operating systems, how they schedule processes to meet performance requirements, and how they provide services beyond CPU scheduling.

The operating system is an important abstraction because it vastly simplifies the control code required to coordinate processes. Rather than write spaghetti code to coordinate all the processes, scheduling is centralized in a single algorithm which coordinates computation and the transfer of the CPU state. The simplification of control structure comes from applying a single control strategy for all processes are loaded into the CPU. In theory, one can find schedules for a particular application which are more efficient than the general policy of an OS. However, in practice it is difficult to ensure that such special-purpose schedules work when they are crafted by hand. Even when CAD tools are used to create application-specific schedules, the operating system provides a reliable mechanism for switching between processes and implementing the schedule.

6.5.1 Process State and Scheduling

The first job of the operating system is to determine what process runs next. The work of choosing the order of running processes is known as scheduling.

illustrator: please use UML state symbol for bubbles

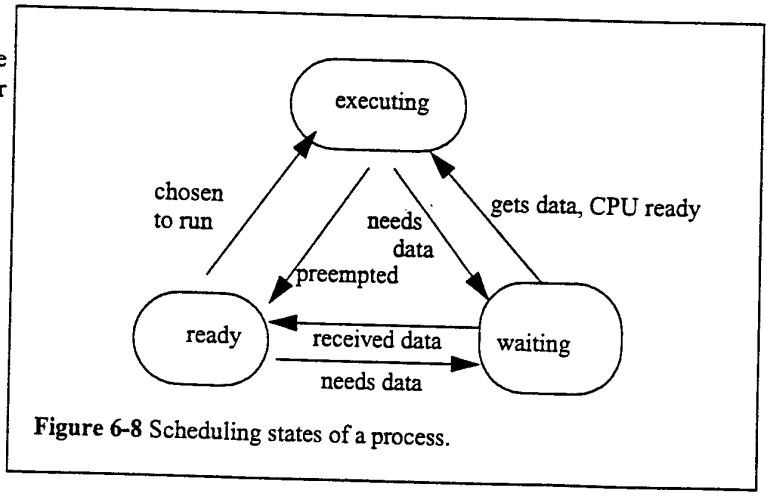


Figure 6-8 Scheduling states of a process.

scheduling states

The operating system considers a process to be in one of three basic **scheduling states**: **waiting**, **ready**, or **executing**. There is at most one process actually executing on the CPU at any time. (If there is no useful work to be done, an idling process may be used to perform some null operation.) Any process that could execute is in the ready state; the operating system chooses

among the ready processes to select the next executing process. A process may not, however, always be ready to run: it may be waiting for data from an I/O device or from another process; it may also be set to run from a timer which has not yet expired. Such processes are in the waiting state. Figure 6-8 shows the possible transitions between states available to a process. A process goes into the waiting state when it needs data that it has not yet received or when it has finished all its work for this period. A process goes into the ready state when it receives its required data and when it enters a new period. A process can go into the executing state only when it has all its data, is ready to run, and the scheduler selects the process as the next process to run.

process priorities

A common way to choose the next executing process is based on process priorities. Each process is assigned a priority, an integer-valued number. The next process to be chosen to execute is the process in the set of ready processes which has the highest-valued priority. The priority embodies the rules used to select running processes. The seeming simplicity of using a number to determine the next running process hides some important complications, however: are priorities fixed or can they change during execution? How are priorities determined? We will consider these problems in much more detail in Section 6.5.4. In the meantime, a simple example shows how priorities can be used to schedule processes.

Example 6-3 Priority-driven scheduling

For this example, we will adopt some simple rules:

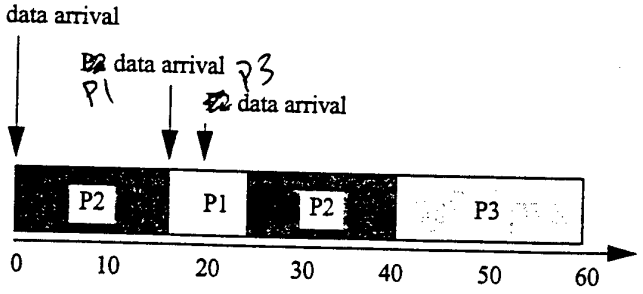
- Each process has a fixed priority that does not vary during the course of execution. (More sophisticated scheduling schemes do, in fact, change the priorities of processes to control what happens next.)
- The ready process with the highest priority (with 1 as the highest priority of all) is selected for execution.
- A process continues execution until it completes or it is preempted by a higher-priority process.

Let's define a simple system with three processes:

process	priority	execution time
P1	1	10
P2	2	30
P3	3	20

In addition to describing the properties of the processes in general, we need to know the environmental setup. We will assume that P2 is ready to run when the system is started, P1's data arrives at time 15, and P3's data arrives at time 18.

Once we know the process properties and the environment, we can use the priorities to determine what process is running throughout the complete execution of the system:



When the system begins execution, P2 is the only ready process, so it is selected for execution. At time 15, P1 becomes ready; it preempts P2 and begins execution since it has a higher priority. And since P1 is the highest priority process in the system, it is guaranteed to execute until it finishes. P3's data arrives at time 18, but it cannot preempt P1. Even when P1 finishes, P3 is not allowed to run. P2 is still ready and has higher priority than P3. Only after both P1 and P2 finish can P3 execute.

6.5.2 Structure of the Operating System

The process activation record can be generalized to include all the data required by the operating system to run the process:

- the process priority;
- the process scheduling state;
- the starting address of the process.

The data structures used to support scheduling depend in part on whether processes can be created during execution. If the system runs a completely fixed set of processes, the process state records can be kept in an array; if processes can be created on-the-fly during execution, the process state records must be kept in a linked list. While it may initially seem that most embedded computing systems would not create processes on the fly, there are in fact a number of situations in which it is convenient and efficient to do so. When some events occur aperiodically and infrequently, it may be reasonable to create a process to handle the event when it occurs—see Application Note 6-2 for an example. This is particularly true when all combinations of events are unlikely to occur, since creating processes only when needed will save memory.

Application Note 6-2: Telephone calls as processes

A telephone switching system must keep track of some basic information about the call: the two lines connected by the call, billing information such as the length of the call, any special features such as conference calling, etc. It is therefore convenient to think of each call as a process that is created when the call is initiated and destroyed when the call is complete. Not only does this mean that processes are created and destroyed during the execution of the system, but there may also be a large number of active processes at any one time. A large telephone switch may be capable of handling 10,000 calls. Any implementation that uses processes to represent calls must therefore be able to manipulate processes very efficiently.

Most switching systems use a call process only to keep track of basic information, not to switch the actual call data. Since each call requires high-rate data transfers (4 kHz for voice), activating even one process at that rate would be impractical, and activating all the processes at that rate would be impossible.

The operating system generally executes in protected mode if the microprocessor supports protected-mode execution. Even if all the code executing in the system was tested by the designers, running in protected mode gives an extra measure of protection against programming bugs which might cause one process to interfere with another.

operating system call in ARM and SHARC

The ARM processor uses the SWI (software interrupt) instruction to provide operating system access. This instruction puts the CPU into the supervisor mode; the exception vector table is used to direct execution to the proper place in the operating system. SWI takes one argument that can be used as a parameter to the operating system, generally which operating system function is requested (input, output, etc.). If the processes do not share a common address space, the supervisor must save the MMU state as well. The SHARC does not have a supervisor mode and so does not support an equivalent instruction.

6.5.3 Timing Requirements on Processes

Processes can have several different types of timing requirements imposed on them by the application. The timing requirements on a set of processes strongly influences the type of scheduling which is appropriate. A scheduling policy must define the timing requirements which it uses to determine whether a schedule is valid. Before studying scheduling proper, we will outline the types of process timing requirements that are useful in embedded system design.

initiation time

Figure 6-9 illustrates different ways in which we can define two important requirements on processes: **initiation time** and **deadline**. The initiation time for a process is the time at which it goes from the waiting to the ready state. An aperiodic process is by definition initiated by an event, such as external data arriving or some data computed by another process. The initiation time is generally measured from that event, though the system may want to make the process ready at some interval after the event itself. For a periodically-

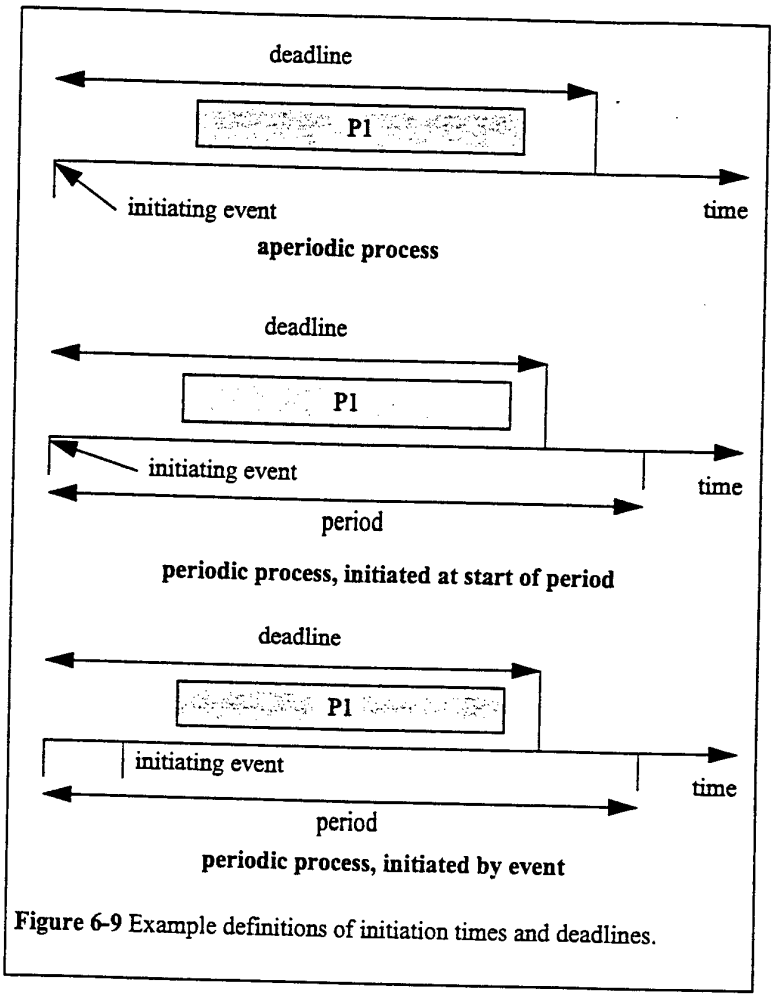


Figure 6-9 Example definitions of initiation times and deadlines.

executed process, there are two common possibilities. In simpler systems, the process may become ready at the beginning of the period. More sophisticated systems may set the initiation time at the arrival time of some data, at some time after the start of the period.

deadline

A deadline specifies when a computation must be finished. The deadline for an aperiodic process is generally measured from the initiation time, since that is the only reasonable time reference. The deadline for a periodic process may in general occur at some time other than the end of the period. As we will see in Section 6.6.1, some scheduling policies make the simplifying assumption that the deadline occurs at the end of the period.

rates

Rate requirements are also fairly common. A rate requirement specifies how quickly processes must be initiated. The inverse of the rate is the period, which is also called the **initiation interval**. The most common case for periodic processes is for the initiation interval to be equal to the period. However, pipelined execution of processes allows the initiation interval to be less than the period. Figure 6-10 illustrates process execution in a system with

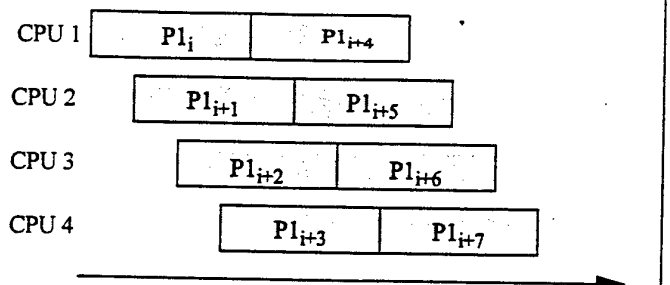


Figure 6-10 A sequence of processes with a high initiation rate.

four CPUs. The various execution instances of program P1 have been subscripted to distinguish their initiation times. In this case, the initiation interval is equal to one-fourth of the period. It is possible for a process to have an initiation rate less than the period even in single-CPU systems: if the process execution time is significantly less than the period, it may be possible to initiate multiple copies of a program at slightly offset times.

timing violations

What happens when a process misses a deadline? The practical effects of a timing violation depend on the application: the results can be catastrophic in an automotive control system, whereas a missed deadline in a telephone system may cause a temporary silence on the line. The system can be designed to take a variety of actions when a deadline is missed. Safety-critical systems may try to take compensatory measures such as approximating data or switching into a special safety mode. Systems for which safety is not as important may take simple measures to avoid propagating bad data, such as inserting silence in a phone line, or may completely ignore the failure.

Even if the modules are functionally correct, their timing behavior can introduce major execution errors. Application Note 6-3 describes a timing problem in Space Shuttle software that caused the delay of the first launch of the Shuttle.

Application Note 6-3A Space Shuttle Software Error

Garman [Gar81] describes a software problem that delayed the first launch of the U. S. Space Shuttle. No one was hurt and the launch proceeded after the computers were reset. However, this bug was serious and unanticipated.

The Shuttle's primary control system was known as the Primary Avionics Software System (PASS). It used four computers to monitor events, with the four machines voting to ensure fault tolerance. Four computers allowed one machine to fail while still leaving three operating machines to vote, such that a majority vote would still be possible to determine operating procedures. If at least two machines failed, control was to be turned over to a fifth computer called the Backup Flight Control System (BFS). The BFS used the same computer, requirements, programming language, and compiler, but it was developed by a different organization than the one that built PASS to

ensure that methodological errors did not cause simultaneously failure of both systems. The switchover from PASS to BFS was controlled by astronauts.

During normal operation, the BFS would listen to the operation of the PASS computers so that it could keep track of the state of the Shuttle. However, BFS would stop listening when it thought that PASS was compromising data fetching. This would prevent PASS failures from inadvertently destroying the state of the BFS. PASS used an asynchronous, priority-driven software architecture. If high-priority processes take too much time, the operating system can skip or delay lower-priority processing. The BFS, in contrast, used a time-slot system that allocated a fixed amount of time to each process. Since the BFS monitors the PASS, it could get confused by temporary overloads on the primary system. As a result, the PASS was changed late in the design cycle to make its behavior more amenable to the backup system.

On the morning of the launch attempt, the BFS failed to synchronize itself with the primary system. It saw the events on the PASS system as inconsistent and therefore stopped listening to PASS behavior. It turned out that all the PASS and BFS processing had been running late relative to telemetry data. This occurred because the system incorrectly calculated its start time.

After a great deal of analysis of system traces and software, it was determined that a few minor changes to the software had caused the problem. First, about two years before the incident, a subroutine used to initialize the data bus was modified. Since this routine was run prior to calculating the start time, it introduced additional, unnoticed delay into that computation. About a year later, a constant was changed as an attempt to fix that problem. As a result of these changes, there was a 1 in 67 probability for a timing problem. When this occurred, almost all the computations on the computers would occur a cycle late, leading to the observed failure. The problems were hard to detect in testing since they required running through all initialization code; many tests start with a known configuration to save the time required to run the setup code. The changes to the programs were also not obviously related to the final changes in timing.

6.5.4 Interprocess Communication

Processes often need to communicate with each other. **Interprocess communication mechanisms** are provided by the operating system as part of the process abstraction.

In general, a process can send a communication in one of two ways: **blocking** or **non-blocking**. After sending a blocking communication, the process goes into the waiting state until it receives a response. Non-blocking communication allows the process to continue execution after sending the communication. Both types of communication are useful.

There are two major styles of interprocess communication: **shared memory** and **message passing**. The two are logically equivalent: given one, you can

build an interface which implements the other. However, some programs may be easier to write using one than the other. In addition, the hardware platform may make one more easier to implement or more efficient than the other.

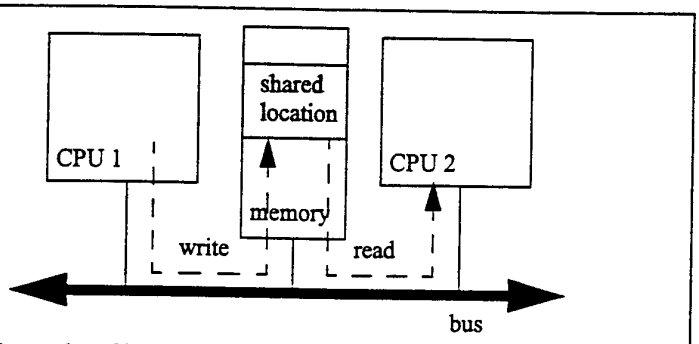
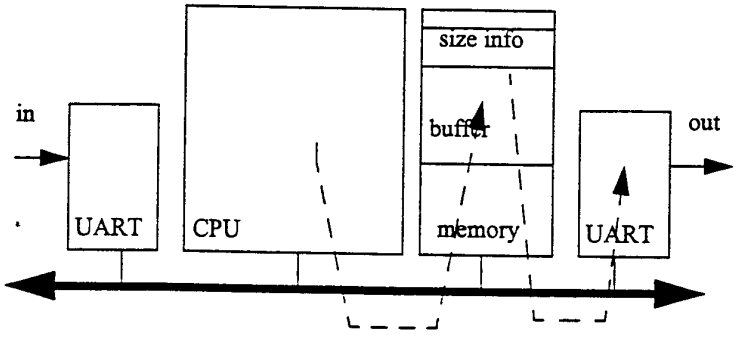


Figure 6-11 Shared memory communication implemented on a bus.

Figure 6-11 illustrates how shared memory communication works in a bus-based system. Two components, such as two CPUs, communicate through a shared memory location. The software on CPUs 1 and 2 has been designed to know the address of the shared location. If, as in the figure, CPU 1 wants to send data to CPU 2, it writes to the shared location. CPU 2 then reads the data from that location. The read and write operations are standard and can be encapsulated in a procedural interface. Example 6-4 describes the use of shared memory as a practical communication mechanism.

Example 6-4 Elastic buffers as shared memory

The text compressor of Application Note 3-4 provides a good example of a shared memory. The text compressor uses the CPU to compress incoming text, which is then sent on a serial line by a UART:



The input data arrives at a constant rate and is easy to manage. But the output data, since it is consumed at a variable rate, requires an elastic buffer. The CPU and output UART share a memory area: the CPU writes compressed characters into the buffer and the UART removes them as necessary

to fill the serial line. Because the number of bits in the buffer changes constantly, the compression and transmission processes need additional size information. In this case, coordination is simple: the CPU writes at one end of the buffer and the UART reads at the other end. The only challenge is to make sure that the UART does not overrun the buffer.

As an application of shared memory, let us consider the situation of Figure 6-11 in which programs on two CPUs want to communicate through a shared memory block. There must be a flag which tells one CPU when the data from the other CPU is ready. The flag, an additional shared data location, has a value of 0 when the data is not ready and 1 when the data is ready. CPU 1, for example, would write the data, then set the flag location to 1. If the flag is used only by CPU 1, then the flag can be implemented using a standard memory write operation. If the same flag is used for bi-directional signalling between CPUs 1 and 2, care must be taken. Consider this scenario:

1. CPU 1 reads the flag location and sees that it is 0.
2. CPU 2 reads the flag location and sees that it is 0.
3. CPU 1 sets the flag location to 1 and writes data to the shared location.
4. CPU 2 erroneously sets the flag to 1 and overwrites the data left by CPU 1.

race condition

This scenario is caused by a critical timing race between the two programs. To avoid such problems, the microprocessor bus must support an atomic **test-and-set** operation, which is available on a number of microprocessors. The test-and-set operation first reads a location, then sets it to a specified value. It returns the result of the test. If the location was already set, then the additional set has no effect but the test-and-set instruction returns a false result. If the location was not set, the instruction returns true and the location is in fact set. The bus supports this as an **atomic** operation that cannot be interrupted. Programming Note 6-1 describes a test-and-set operation in more detail.

semaphore: P and V

A test-and-set can be used to implement a **semaphore**, which is a language-level synchronization construct. For the moment, let's assume that the system provides one semaphore that is used to guard access to a block of protected memory. Any process that wants to access the memory must use the semaphore to make sure that no other process is actively using it. The semaphore names by tradition are P() to gain access to the protected memory and V() to release it:

```
/* some non-protected operations here */
P(); /* wait for semaphore */
/* do protected work here */
V(); /* release semaphore */
```

The P() operation uses a test-and-set to repeatedly test a location that holds a lock on the memory block. The P() operation does not exit until the lock is available; once it is available, the test-and-set automatically sets the lock. Once past the P() operation, the process can work on the protected memory

block. The V() operation resets the lock, allowing other processes access to the region by using the P() function.

Programming Note 6-1 Test-and-set operations

The SWP (Swap) instruction is used in the ARM to implement atomic test-and-set:

```
SWP    Rd,Rm,Rn
```

The SWP instruction takes three operands: the memory location pointed to

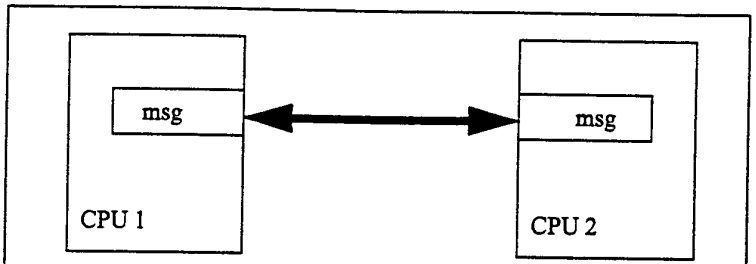


Figure 6-12 Message passing communication.

by *Rn* is loaded and saved into *Rd*; the value of *Rm* is then written into the location pointed to by *Rn*. When *Rd* and *Rn* are the same register, the instruction swaps the register's value and the value stored at the address pointed to by *Rd/Rn*. For example, the code sequence

```
        ADR r0, SEMAPHORE      ; get semaphore address
        LDR r1, #1
GETFLAG SWP r1,r1,[r0]         ; test and set the flag
        BNZ GETFLAG            ; no flag yet, try again
HASFLAG ...
```

first loads the constant 1 into *r1* and the address of the semaphore FLAG1 into register *r2*, then reads the semaphore into r0 and writes the 1 value into the semaphore. The code then tests whether the semaphore fetched from memory is zero; if it was, the semaphore was not busy and we can enter the critical region which begins with the HASFLAG label. If the flag was non-zero, we loop back to try to get the flag once again.

message passing

Message passing communication complements the shared memory model. As shown in Figure 6-12, each communicating entity has its own message send/receive unit. The message is not stored on the communications link, but rather at the senders/receivers at the endpoints. In contrast, shared memory communication can be seen as a memory block used as a communication device, in which all the data is stored in the communication link/memory.

Applications in which units operate relatively autonomously are natural candidates for message passing communication. For example, a home control system has one microcontroller per household device—lamp, thermostat,

faucet, appliance, etc. The devices must communicate relatively infrequently; furthermore, their physical separation is great enough that we would not naturally think of them as sharing some central pool of memory. Passing communication packets between the devices is a natural way to describe coordination between these devices. Message passing is the natural implementation of communication in many 8-bit microcontrollers that do not normally operate with external memory. We will discuss message passing in more detail in Chapter 8.

We almost always need to pass data between processes. There are two distinct cases to consider: communicating processes which execute at the same rate, and those which communicate but operate at different rates.

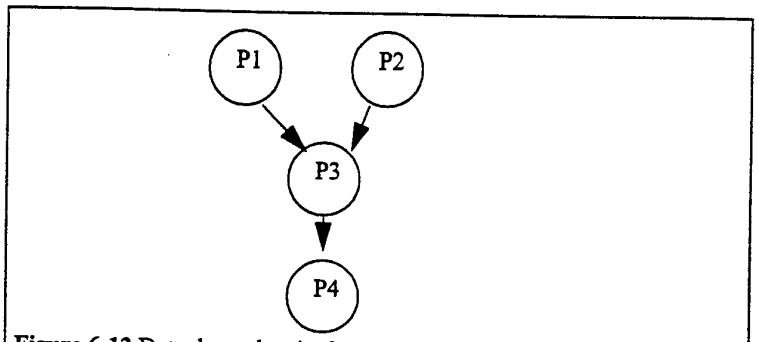


Figure 6-13 Data dependencies between processes.

data dependencies

When communicating processes run at the same rate, we express their relationship using data dependencies. Figure 6-13 shows a set of processes with data dependencies between them. Before a process can become ready, all the processes on which it depends must complete and send their data to it. The data dependencies define a partial ordering on process execution: P1 and P2 can execute in any order (or in interleaved fashion) but must both complete before P3; P3 must complete before P4. All processes must finish before the end of the period. The data dependencies must form a directed acyclic graph (DAG)—a cycle in the data dependencies is difficult to interpret in a periodically-executed system.

A set of processes with data dependencies is known as a **task graph**. Although the terminology for elements of a task graph vary from author to author, we will consider a component of the task graph (a set of nodes connected by data dependencies) as a **task**, and the complete graph as the **task set**.

Communication between processes which run at different rates cannot be represented by data dependencies because there is not a one-to-one relationship between data coming out of the source process and going into the destination process. Nevertheless, communication between processes of different rates is very common. Figure 6-14 illustrates the communication required between three elements of an MPEG audio/video decoder. Data comes into the decoder in the system format, which multiplexes audio and video data. The system decoder process demultiplexes the audio and video data and distributes it to the appropriate processes. Multi-rate communication is neces-

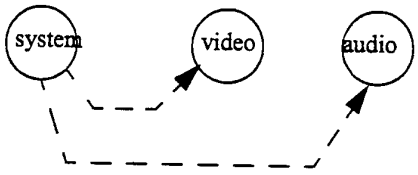


Figure 6-14 Communication between processes at different rates.

sarily one way—for example, the system process writes data to the video process, but a separate communication mechanism must be provided for any communication from the video process back to the system process.

6.5.5 Other Operating System Functions

Beyond scheduling processes, operating systems may provide a wide variety of other services. The role of the operating system may be viewed as managing shared resources. The CPU is clearly shared among the processes; I/O devices rate a close second as shared resources requiring management. The operating system will generally supply a driver for each device and standard means for communicating with the devices. Network connections are one class of I/O which is becoming increasingly important in embedded computing systems, which may talk either on a local-area network or on the Internet. Advanced embedded operating systems may also provide file system services, with named files accessible for reading and writing. The file system may be kept in RAM, and not necessarily on a disk; the file system is a convenient means for organizing large data sets, even if mass storage devices are not built into the hardware platform. An embedded operating system must also provide powerful debugging facilities which allow system designers to evaluate the state of the system during execution. Debugging facilities are frequently built to take advantage of host PC systems which provide easy-to-use interfaces based on raw data supplied to the host PC by the embedded OS.

6.6 Scheduling Policies

A **scheduling policy** defines how processes are selected for promotion from the ready state to the running state. Every multitasking operating system implements some scheduling policy. Choosing the right scheduling policy not only ensures that the system will meet all its timing requirements, it also has a profound influence on the CPU horsepower required to implement the system's functionality.

As we will see in this section, scheduling policies vary widely in the generality of the timing requirements they can handle and the efficiency with which they use the CPU. **Utilization** is one of the key metrics in evaluating a scheduling policy. We will see that some types of timing requirements for a

set of processes implies that we cannot utilize 100% of the CPU's execution time on useful work, even ignoring context switching overhead. However, some scheduling policies can deliver higher CPU utilizations than others even for the same timing requirements. The best policy depends on the required timing characteristics of the processes being scheduled.

In addition to utilization, we must also consider **scheduling overhead**—the execution time required to choose the next execution process, which is incurred in addition to any context-switching overhead. In general, the more sophisticated the scheduling policy, the more CPU time it takes during system operation to implement it. And in general, we achieve higher theoretical CPU utilization by applying more complex scheduling policies with higher overheads. The final decision on a scheduling policy must take into account both theoretical utilization and practical scheduling overhead.

real-time scheduling in POSIX

POSIX supports real-time scheduling in the _POSIX_PRIORITY_SCHEDULING resource. Not all processes have to run under the same scheduling policy. The sched_setscheduler() function is used to determine a process's scheduling policy and other parameters:

```
#include <sched.h>
int i, my_process_id;
struct sched_param my_sched_params;

...
i =
sched_setscheduler(my_process_id,SCHED_FIFO,&sched_params)
;
```

This call tells POSIX to use the SCHED_FIFO policy for this process, along with some other scheduling parameters.

We'll see how the scheduling methods supported by POSIX relate to the range of possible scheduling mechanisms.

6.6.1 Rate-Monotonic Scheduling

Rate-monotonic scheduling (**RMS**) [Liu73], introduced by Liu and Layland, was one of the first scheduling policies developed for real-time systems and is still very widely used. We say that RMS is a **static scheduling policy** because it assigns fixed priorities to processes. It turns out that these fixed priorities are sufficient to efficiently schedule the processes in many situations.

The theory underlying RMS is known as **rate-monotonic analysis** (**RMA**). This theory uses a relatively simple model of the system:

- All processes run periodically on a single CPU.
- Context switching time is ignored.
- There are no data dependencies between processes.
- The execution time for a process is constant.
- All deadlines are at the ends of their periods.

- The highest-priority ready process is always selected for execution.

The major result of rate-monotonic analysis is that a relatively simple scheduling policy is optimal: priorities are assigned by rank order of period, with the process with the shortest period being assigned the highest priority. This fixed-priority scheduling policy is the optimum assignment of static priorities to processes in that it gives the highest CPU utilization while ensuring that all processes meet their deadlines.

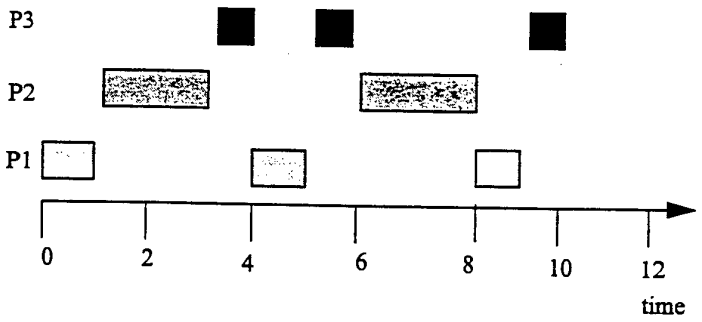
The next example illustrates rate-monotonic scheduling.

Example 6-5 Rate-monotonic scheduling

Here is a simple set of processes and their characteristics:

process	execution time	period
P1	1	4
P2	2	6
P3	3	12

Applying the principles of RMA, we give P1 the highest priority, P2 the middle priority, and P3 the lowest priority. To understand all the interactions between the periods, we need to construct a timeline equal in length to the least-common multiple of the process periods, which is 12 in this case. The complete schedule for the least-common multiple of the periods is called the **unrolled schedule**.



All three periods start at time zero. P1's data arrives first; since it is both the highest-priority process, it can start to execute immediately. After one time unit, P1 finishes and goes out of the ready state until the start of its next period. At time 1, P2 starts executing as the highest-priority ready process. At time 3, P2 finishes and P3 starts executing. P1's next iteration starts at time 4, at which point it interrupts P3. P3 gets one more time unit of execution between the second iterations of P1 and P2, but P3 doesn't get to finish until after the third iteration of P1.

Consider a different set of execution times for these processes, keeping the same deadlines:

process	execution time	period
P1	2	4
P2	3	6
P3	3	12

In this case, we can show that there is no feasible assignment of priorities that guarantees scheduling. Even though each process alone has an execution time significantly less than its period, combinations of processes can require more than 100% of the available CPU cycles. For example, during one twelve time-unit interval, we must execute P1 three times, requiring six units of CPU time, P2 twice costing six units of CPU time, and P3 one time, requiring 3 units of CPU time. The total of $6+6+3 = 15$ units of CPU time is more than the 12 time units available, clearly exceeding the available CPU capacity.

analysis

Liu and Layland proved that the RMA priority assignment is optimal using critical-instant analysis. We define the **response time** of a process as the time at which the process finishes. The **critical instant** for a process to be the instant during execution at which the task has the largest response time. It is easy to prove that the critical instant for any process P, under the RMA model, occurs when it is ready and all higher-priority processes are also ready—if we change any higher-priority process to waiting, then P's response time can only go down.

We can use critical-instant analysis to determine whether there is any feasible schedule for the system; in the case of the second set of execution times in Example 6-5, there was no feasible schedule. Critical-instant analysis also implies that priorities should be assigned in order of period. Let the periods and computation times of two processes P1 and P2 be τ_1, τ_2 and T_1, T_2, with $\tau_1 < \tau_2$. We can determine generalize the result of Example 6-5 to show the total CPU requirements for the two processes in two cases. In the first case, let P1 have the higher priority. Then we must in the worst case execute P2 once during its period and as many iterations of P1 as fit in the same interval. Since there are $\lfloor \tau_2/\tau_1 \rfloor$ iterations of P1 during a single period of P2, the required constraint on CPU time, ignoring context-switching overhead, is

$$\left\lfloor \frac{\tau_2}{\tau_1} \right\rfloor T_1 + T_2 \leq \tau_2 . \qquad \text{(EQ 6-1)}$$

If, on the other hand, we give higher priority to P2, then critical-instant analysis tells us that we must execute all of P2 and all of P1 in one of P1's periods in the worst case:

$$T_1 + T_2 \leq \tau_1 . \qquad \text{(EQ 6-2)}$$

There are cases where the first relationship can be satisfied but the second cannot, but there are no cases where the second relationship can be satisfied and the first cannot. We can inductively show that the process with the shorter period should always be given higher priority for process sets of arbitrary size. It is also possible to prove that RMS always provides a feasible schedule if such a schedule exists.

CPU utilization

The bad news is that, although RMS is the optimal static-priority schedule, it does not allow the system to use 100% of the available CPU cycles. The total CPU **utilization** for a set of n tasks is

$$U = \sum_{i=1}^{n} \frac{T_i}{\tau_i}. \qquad \text{(EQ 6-3)}$$

It is possible to show that, for a set of two tasks under RMS scheduling, the CPU utilization U will be no greater than $2(2^{1/2} - 1) \cong 0.83$. In other words, the CPU will be idle at least 17% of the time. This idle time is due to the fact that priorities are assigned statically; we will see in the next section that more aggressive scheduling policies can improve CPU utilization. When there are m tasks, the maximum processor utilization is

$$U = m(2^{1/m} - 1) \qquad \text{(EQ 6-4)}$$

As m approaches infinity, the CPU utilization asymptotically approaches $\ln 2 = 0.69$—the CPU will be idle 31% of the time. We can use processor utilization U as an easy measure of the feasibility of an RMS scheduling problem.

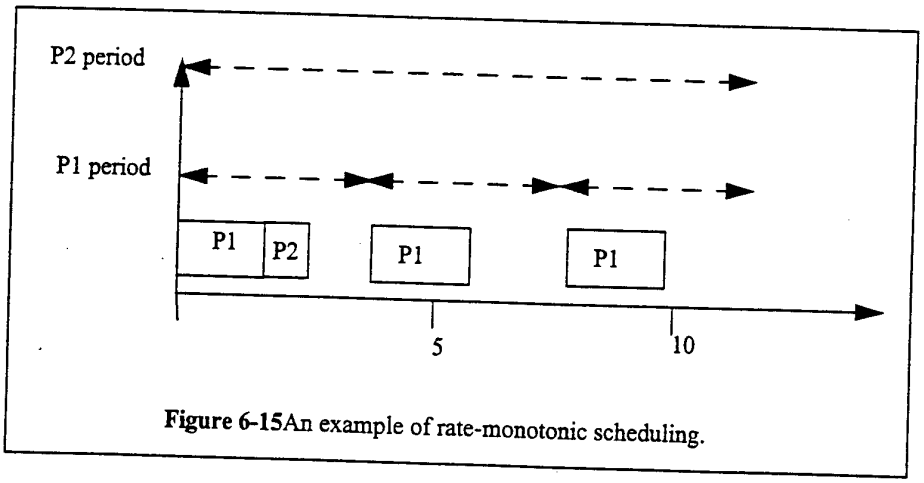


Figure 6-15 An example of rate-monotonic scheduling.

Figure 6-15 shows an example of an RMS schedule for a system in which P1 has a period of 4 and an execution time of 2 and P2 has a period of 12 and an execution time of 1. The execution of P2 is preempted by P1 during P2's first period and P2's second. The least-common multiple of the periods of the processes is 12, so the CPU utilization of this set of processes is

$[(2 \times 3) + (1 \times 12)]/12 = 0.58333$, which is within the feasible utilization of RMS.

```
/* processes[] is an array of process activation records,
   stored in order of priority, with processes[0] being
   the highest-priority process */
Activation_record processes[NPROCESSES];

void RMA(int current) { /* current = currently-executing pro-
cess */
    int i;
    /* turn off current process (may be turned back on) */
    processes[current].state = READY_STATE;
    /* find process to start executing */
    for (i=0; i<NPROCESSES; i++)
        if (processes[i].state == READY_STATE) {
            /* make this the running process */
            processes[i].state == EXECUTING_STATE;
            break;
        }
    }
```

Figure 6-16 C code for rate-monotonic scheduling.

implementation

The implementation of RMS is very simple. Figure 6-16 shows C code for an RMS scheduler which is run at the operating system's timer interrupt. The code merely scans through the list of processes in priority order and selects the highest-priority ready process to run. Because the priorities are static, the processes can be sorted by priority in advance, before the system starts executing. As a result, this scheduler has an asymptotic complexity of $O(n)$, where n is the number of processes in the system. (This code assumes that processes are not created dynamically; if dynamic process creation is required, the array can be replaced by a linked list of processes, but the asymptotic complexity remains the same.) The RMS scheduler has both low asymptotic complexity and low actual execution time, which helps minimize the discrepancies between the zero-context-switch assumption of rate-monotonic analysis and the actual execution of an RMS system.

POSIX supports rate-monotonic scheduling in the SCHED_FIFO scheduling policy. The name of this policy is unfortunately misleading. It is a strict priority-based scheduling scheme in which a process runs until it is preempted or terminates. The term FIFO simply refers to the fact that, within a priority, processes run in first-come first-served order.

We already saw the sched_setscheduler() function that allows a process to set a scheduling policy. Two other useful functions allow a process to determine the minimum and maximum priority values in the system:

```
minval = sched_get_priority_min(SCHED_RR);
maxval = sched_get_priority_max(SCHED_RR);
```

The sched_getparams() function returns the current parameter values for a process and sched_setparams() changes the parameter values:

```
int i, mypid;
struct sched_param my_param;

mypid = getpid();
i = sched_getparam(mypid,&my_params);
my_params.sched_priority = maxval;
i = sched_setparam(mypid,&my_params);
```

Whenever a process changes its priority, it is put at the back of the queue for that priority level. A process can also explicitly move itself to the end of its priority queue with a call to the sched_yield() function.

6.6.2 Earliest-Deadline-First Scheduling

Earliest-deadline-first (EDF) is another well-known scheduling policy. It is a dynamic-priority scheme: it changes process priorities during execution based on initiation times. As a result, it can achieve higher CPU utilizations than can RMS. POSIX does not currently support EDF, but it still serves as a useful comparison to RMS.

The EDF policy is also very simple: it assigns priorities in order of deadline. The highest-priority process is the one whose deadline is nearest in time; the lowest-priority process is the one whose deadline is farthest away. Clearly, priorities must be recalculated at every completion of a process. However, the final step of the OS during the scheduling procedure is the same as for RMS: the highest-priority ready process is chosen for execution.

The next example illustrates EDF scheduling in practice.

Example 6-6 Earliest-deadline-first scheduling

Consider this set of processes:

process	execution time	period
P1	1	3
P2	1	4
P3	2	5

The least-common multiple of the periods is 60, and the utilization is $\frac{1}{3}+\frac{1}{4}+\frac{1}{5} = 0.98333333$. This utilization is too high for RMS, but it can be

handled with an earliest-deadline first schedule. Because the schedule is so long, let's write it as a table:

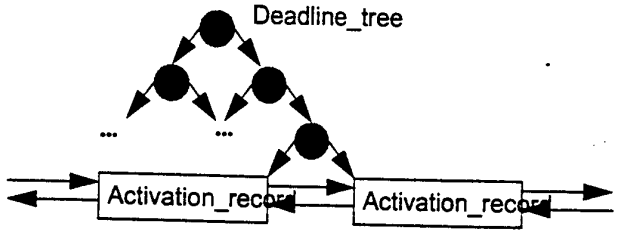
time	running process	deadlines
0	P1	
1	P2	
2	P3	P1
3	P3	P2
4	P1	P3
5	P2	P1
6	P1	
7	P3	P2
8	P3	P1
9	P1	P3
10	P2	
11	P3	P1, P2
12	P3	
13	P1	
14	P2	P1, P3
15	P1	P2
16	P2	
17	P3	P1
18	P3	
19	P1	P2, P3
20	P2	P1
21	P1	
22	P3	
23	P3	P1, P2
24	P1	P3
25	P2	
26	P3	P1
27	P3	P2
28	P1	
29	P2	P1, P3
30	P1	
31	P3	P2
32	P3	P1
33	P1	

time	running process	deadlines
34	P2	P3
35	P3	P1, P2
36	P1	
37	P2	
38	P3	P1
39	P1	P2, P3
40	P2	
41	P3	P1
42	P1	
43	P3	P2
44	P3	P1, P3
45	P1	
46	P2	
47	P3	P1, P2
48	P3	
49	P1	P3
50	P2	P1
51	P1	P2
52	P3	
53	P3	P1
54	P2	P3
55	P1	P2
56	P2	P1
57	P3	
58	P3	
59	idle	P1, P2, P3

There is one time slot left at the end of this unrolled schedule, which is consistent with our earlier calculation that the CPU utilization is 59/60.

utilization

Liu and Layland showed EDF can achieve 100% utilization [Liu73]. A feasible schedule exists if the CPU utilization (calculated in the same way as for RMA) is less than or equal to 1. They also showed that when an EDF system is overloaded and misses a deadline, it will run at 100% capacity for some period before the deadline is missed. Distinguishing between a system that is running just at capacity and one that is about to miss a deadline is difficult, making the use of very high deadlines problematic.



data structure

```
/* linked list, sorted by deadline */
Activation_record *processes;
/* data structure for sorting processes */
Deadline_tree *deadlines;

void expired_deadline(Activation_record *expired) {
    remove(expired); /* remove from the deadline-sorted list */
    add(expired,expired->deadline); /* add at new deadline */
}

void EDF(int current) { /* current = currently-executing process */
    int i;
    /* turn off current process (may be turned back on) */
    processes->state = READY_STATE;
    /* find process to start executing */
    for (alink = processes; alink != NULL; alink = alink->next_deadline)
        if (processes->state == READY_STATE) {
            /* make this the running process */
            processes->state == EXECUTING_STATE;
            break;
        }
    }
```

code

Figure 6-17 C code for earliest-deadline-first scheduling.

implementation

The implementation of EDF is more complex than the RMS code. Figure 6-17 outlines one way to implement EDF. The major problem is keeping the processes sorted by time-to-deadline—since the times-to-deadlines for the processes change during execution, we cannot pre-sort the processes into an array, as we could for RMS. To avoid re-sorting the entire set of activation records at every change, we can build a binary tree to keep the sorted records and incrementally update the sort. At the end of each period, we can move the activation record to its new place in the sorted list by deleting it from the tree, then adding it back to the tree, using standard tree manipulation techniques. We must update process priorities by traversing them in sorted order,

so the incremental sorting routines must also update the linked list pointers that let us traverse the records in deadline order. (The linked list lets us avoid traversing the tree to go from one node to another, which would require more time.) After putting in the effort to building the sorted list of activation records, selecting the next executing process is done in a manner similar to that of RMS. However, the dynamic sorting adds complexity to the entire scheduling process. Each update of the sorted list requires $O(\log n)$ steps. The EDF code is also significantly more complex than the RMS code.

6.6.3 RMS vs. EDF

Which scheduling policy is better: RMS or EDF? That depends on your criteria. EDF can extract higher utilization out of the CPU it may be difficult to diagnose the possibility of an imminent overload. Because the scheduler does take some overhead to make scheduling decisions, a factor that is ignored in the schedulability analysis of both EDF and RMS, running a scheduler at very high utilizations is somewhat problematic. RMS achieves lower CPU utilization but is easier to ensure that all deadlines will be satisfied. In some applications, it may be OK for some processes to occasionally miss deadlines. For example, a set-top box for video decoding is not a safety-critical application and the occasional display artifacts caused by missing deadlines may be acceptable in some markets. But there are other domains in which missing deadlines is unacceptable.

What if your set of processes is unschedulable and you need to guarantee that they complete their deadlines? There are several possible ways to solve this problem:

- Get a faster CPU. That will reduce execution times without changing the periods, giving you lower utilization. This will require you to redesign the hardware, but is often feasible—you are rarely using the fastest CPU available.
- Redesign the processes to take less execution time. This requires knowledge of the code and may or may not be possible.
- Rewrite the specification to change the deadlines. This is unlikely to be feasible, but may be in a few cases where some of the deadlines were initially made tighter than necessary.

6.6.4 A Closer Look at Our Modeling Assumptions

Our analyses of RMS and EDF have made some strong assumptions. These assumptions have made the analysis much more tractable, but the predictions of analysis may not hold up in practice. Since a mis-prediction may cause a system to miss a critical deadline, it is important to at least understand the consequences of these assumptions.

priority inversion

In all of the above discussions, we have assumed that each process is totally self-contained. However, that is not always the case: a process may need some system resource, such as an I/O device or the bus, to complete its

work. Scheduling the processes without considering the resources those processes require can cause **priority inversion**, in which a low-priority process blocks execution of a higher-priority process by keeping hold of its resource. The next example illustrates

Example 6-7 Priority inversion

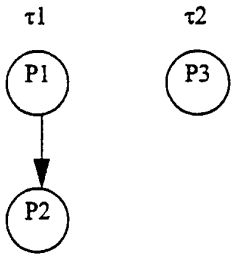
Consider a system with two processes: the higher-priority P1 and the lower-priority P2. Each uses the microprocessor bus to communicate to peripherals. When P2 executes, it requests the bus from the operating system and receives it. If P1 becomes ready while P2 is using the bus, the operating system will preempt P2 for P1, leaving P2 with control of the bus. When P1 requests the bus, it will be denied the bus, since P2 already owns it. Unless P1 has a way to take the bus from P2, the two processes may deadlock.

The most common method for dealing with priority inversion is to promote the priority of any process when it requests a resource from the operating system. The priority of the process temporarily becomes higher than that of any other process which may use the resource. This ensures that the process will continue executing once it has the resource so that it can finish its work with the resource, return it to the operating system, and allow other processes to use it. Once the process is done with the resource, its priority is demoted to its normal value.

Rate-monotonic scheduling assumes that there are no data dependencies between processes. The next example shows that knowledge of data dependencies can help use the CPU more efficiently.

Example 6-8 Data dependencies and scheduling

Data dependencies imply that certain combinations of process activations can never occur. Consider this simple example [Mal95]:



task graph

task	deadline
τ1	10
τ2	8

task rates

process	CPU time
P1	2
P2	1
P3	4

execution times

We know that P1 and P2 cannot execute at the same time, since P1 must finish before P2 can begin. Furthermore, we also know that because P3 has a higher priority, it will not preempt both P1 and P2 in a single iteration: if P3 preempts P1, then P3 will complete before P2 begins; if P3 preempts P2, then it will not interfere with P1 in that iteration. Because we know that some combinations of processes cannot be ready at the same time, we know that our worst-case CPU requirements are less than would be required if all processes could be ready simultaneously.

non-zero context switch time

One important simplification we have made is that contexts can be switched in zero time. On the one hand, this is clearly wrong: we must execute instructions to save and restore context and we must execute additional instructions to implement the scheduling policy. On the other hand, context switching can be implemented efficiently—context switching need not kill performance. The effects of non-zero context-switching time must be carefully analyzed in the context of a particular implementation to be sure that the predictions of an ideal scheduling policy are sufficiently accurate.

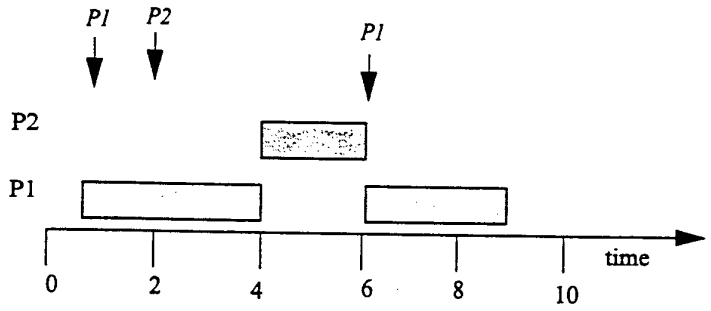
The next example shows that context switching can, in fact, cause a system to miss a deadline.

Example 6-9 Scheduling and context-switching overhead

Here is a set of processes and their characteristics:

process	execution time	deadline
P1	3	5
P2	3	10

First, let us try to find a schedule assuming that context-switching time is zero. Here is a feasible schedule for a sequence of data arrivals that meets all the deadlines:



Now let us assume that the total time to initiate a process, including context switching and scheduling policy evaluation, is one time unit. It is easy to see that there is no feasible schedule for the above data arrival sequence, since we require a total of $2T_{P1} + T_{P2} = 2 \times (1+3) + (1+3) = 11$ time units to execute one period of P2 and two periods of P1.

In this example, overhead was a large fraction of the process execution time and of the periods. In most real-time operating systems, a context switch requires only a few hundred instructions, with only slightly more overhead for a simple real-time scheduler like RMS. These small overhead times are less likely to cause serious scheduling problems. Any problems are most likely to manifest themselves in the highest-rate processes, which are often the most critical in any case. Completely checking that all deadlines will be met with non-zero context-switching time requires checking all possible schedules for processes and including the context-switch time at each preemption or process initiation. However, assuming an average number of context switches per process and computing CPU utilization can give at least an estimate of how close the system is to CPU capacity.

polling vs. interrupts

Rhodes and Wolf [Rho97] developed a CAD algorithm for implementing processes that computes exact schedules to accurately predict the effects of context switching. Their algorithm selects between interrupt-driven and polled implementations for processes on a microprocessor. Polled processes introduce less overhead, but they do not respond to events as quickly as do interrupt-driven processes. Furthermore, adding interrupt levels to a microprocessor usually incurs some cost in added logic, so we do not want to use interrupt-driven processes when they are not necessary. Their algorithm computes exact schedules for the processes including the overhead for poll-

ing or interrupts as appropriate, then uses heuristics to select implementations for the processes. The major heuristic starts with all processes implemented as polled mode, then changes processes that miss deadlines to use interrupts; some iterative improvement steps try various combinations of interrupt-driven processes to eliminate deadline violations. These heuristics minimize the number of processes implemented with interrupts.

Another important assumption we have made so far is that process execution time is constant. As we saw in Section 5.6, this is definitely not the case—both data-dependent behavior and caching effects can cause large variations in run times. The techniques for bounding the cache-based performance of a single program do not work when multiple programs are in the same cache. The state of the cache depends on the product of the states of all the programs executing in the cache, making the state space of the multiple-process system exponentially larger than that for a single program. We will discuss this problem in more detail in Section 6.8.

6.6.5 Other POSIX Scheduling Policies

In addition to SCHED_FIFO, POSIX supports two other real-time scheduling policies: SCHED_RR and SCHED_OTHER. SCHED_RR is a combination of real-time and interactive scheduling techniques: within a priority level, the processes are timesliced. Interactive systems must ensure that all processes get a chance to run, so time is divided into quanta. Processes get the CPU in multiple of quanta. SCHED_RR allows each process to run for a quantum of time, then gives the CPU to the next process to run for its quantum. The length of the quantum can vary with priority level.

The SCHED_OTHER is defined to allow non-real-time processes to intermix with real-time processes. The precise scheduling mechanism used by this policy is not defined. It is used to indicate that the process does not need a real-time scheduling policy.

Remember that different processes in a system can run with different policies, so some processes may run SCHED_FIFO while others run SCHED_RR.

6.7 Interprocess Communication Mechanisms

In this section, we will consider interprocess communication mechanisms in more detail.

6.7.1 Signals

We said in Section 6.5.4 that the two major forms of interprocess communication are shared memory and message passing. However, Unix supports another, very simple communication mechanism—the **signal**. A signal is simple because it does not pass any data beyond the existence of the signal

itself. A signal is analogous to an interrupt, but it is entirely a software creation. A signal is generated by a process and transmitted to another process by the operating system.

	name	description
termination	SIGABRT	abort process
	SIGTERM	terminate process
exceptions	SIGFPE	floating-point exception
	SIGHUP	terminal hang-up
	SIGILL	illegal instruction exception
	ISGINT	interactive termination
	SIGKILL	unavoidable process termination
	SIGPIPE	write to pipe with no readers
	SIGQUIT	abnormal termination
	SIGSEGV	memory access exception
	SIGALRM	real-time clock expired
user-defined	SIGUSR1	user-defined
	SIGUSR2	user-defined

Figure 6-18 Required POSIX signals.

POSIX defines some basic signal types that are listed in Figure 6-18. Some of these signals (SIGFPE, SIGILL, SIGSEGV) are used to abstract CPU exceptions to the operating system. Some (SIGALRM, SIGPIPE) relate to operating system services. Several of them (SIGABRT, SIGHUP, SIGINT, SIGKILL, SIGQUIT, SIGTERM) are various ways to terminate a process. The SIGUSR1 and SIGUSR2 signals are not used directly by POSIX and are free for application-specific purposes.

These signals have a fairly limited repertoire of functions—it would be difficult to build a rich set of communicating processes based on error and process termination signals. That point is driven home by the way in which a process generates a signal, namely the kill() function:

```
kill(0,SIGABRT);
```

sends the SIGABRT signal to all the child processes. Any signal type can be sent by the kill command.

However, a process can control what happens when it receives a signal by declaring a function to handle a signal of a given type. (A control-C in an interactive Unix process usually corresponds to the SIGINT signal. A program that cannot be killed by control-C has told the operating system that it wants to ignore SIGINT signals.) The process declares a handler for a signal

using the sigaction() function. The process uses the sigaction data structure to describe which signals it wants the handler to be called for. For example, we can define a handler for the SIGUSR1 function:

```
#include <signal.h>

extern void usr1_handler(int); /* declaration of SIGUSR1 handler */
struct sigaction act, oldact;
int retval;

/* set up the descriptor data structure */
act.sa_flags = 0;
sigemptyset(&act.sa_mask); /* initialize the signal set to empty */
act.sa_handler = usr1_handler; /*add SIGUSR1 handler to the set */
/* tell the operating system about this handler */
retval = sigaction(SIGUSR1,&act,&oldact);
    /* oldact has old action set */
```

These signals have a long history in UNIX and POSIX but are not particularly well-suited to real-time communication. (The use of the kill() function to send these signals certainly suggests that these signals have one primary use.) Therefore, the POSIX.4 version defines a _POSIX_REALTIME_SIGNALS facility that is much better suited to real-time communication.

The real-time signals are identified by number in the range SIGRTMIN and SIGRTMAX. Although they are named in all capitals, the normal UNIX convention for constants, these values can in fact change during execution. Therefore, it is a good idea to refer to signals relative to these bounds—for example, SIGRTMIN+1.

The sigqueue() function is used to send a signal:

```
if (sigqueue(destpid,SIGRTMAX-1,sval) < 0) { /* error */ }
```

This code sends the signal SIGRTMAX-1 to process number destpid. The function returns a negative value if an error occurs sending the signal. As the name of the function implies, signals can be queued—if two processes send a third process signals of the same type, they are both received in the order in which they were sent. However, queueing must be turned on using the SA_SIGINFO bit in the sa_flags field of the sigaction structure.

6.7.2 Signals in UML

A UML signal is actually a generalization of the Unix signal. While a Unix signal carries no parameters other than a condition code, a UML signal is an object. As such, it can carry parameters as object attributes. Figure 6-19 shows the use of a signal in UML. The *sigbehavior()* behavior of the class is responsible for throwing the signal, as indicated by <<*send*>>. The signal object is indicated by the <<*signal*>> stereotype.

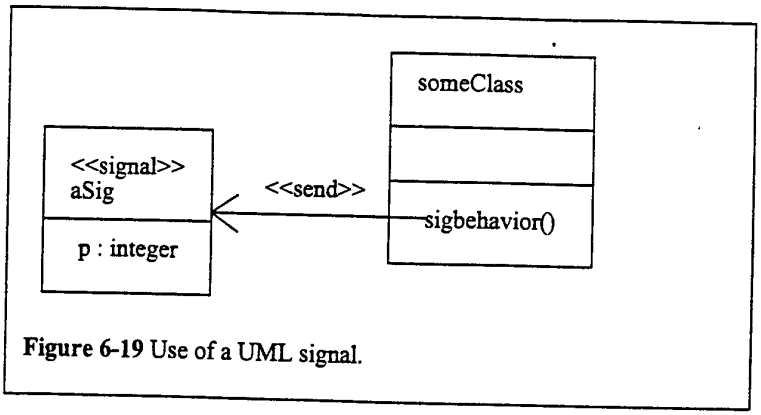


Figure 6-19 Use of a UML signal.

6.7.3 Shared Memory Communication

Conceptually, semaphores are the mechanism we use to make shared memory safe. POSIX supports semaphores but it also supports a direct shared memory mechanism.

POSIX supports counting semaphores in the _POSIX_SEMAPHORES option. A counting semaphore allows more than one process access to a resource at a time. If the semaphore allows up to N resources, then it will not block until N processes have simultaneously passed the semaphore; at that point, the blocked process can resume only after one of the processes has given up its semaphore.The simplest way to think about counting semaphores is that they count down to 0—when the semaphore value is 0, the process must wait until another process gives up the semaphore and increments the count.

Since there may be many semaphores in the system, each one is given a name. Names are similar to file names except that they are not arbitrary paths—they should always start with "/" and should have no other "/". Here is how you create a new semaphore called /sem1, then close it:

```
int i, oflags;
sem_t *my_semaphore; /* descriptor for the semaphore */

my_semaphore = sem_open("/sem1",oflags);
/* do useful work here */
i = sem_close(my_semaphore);
```

The POSIX names for *P* and *V* are sem_wait() and *sem_post()* respectively. POSIX also provides a sem_trywait() function that tests the semaphore but does not block. Here are examples of their use:

```
int i;
i = sem_wait(my_semaphore); /* P */
/* do useful work */
i = sem_post(my_semaphore); /* V */
/* sem_trywait tests without blocking */
i = sem_trywait(my_semaphore);
```

POSIX shared memory is supported under the _POSIX_SHARED_MEMORY_OBJECTS option. The shared memory functions create blocks of memory that can be used by several processes.

The shm_open() function opens a shared memory object:

```
objdesc = shm_open("/memobj1",O_RDWR);
```

This code creates a shared memory object called /memobj1 with read/write access; the O_RDONLY mode allows reading only. It returns an integer which we can use as a descriptor for the shared memory object. The ftruncate() function allows a process to set the size of the shared memory object:

```
if (ftruncate(objdesc,1000) < 0) { /* error */ }
```

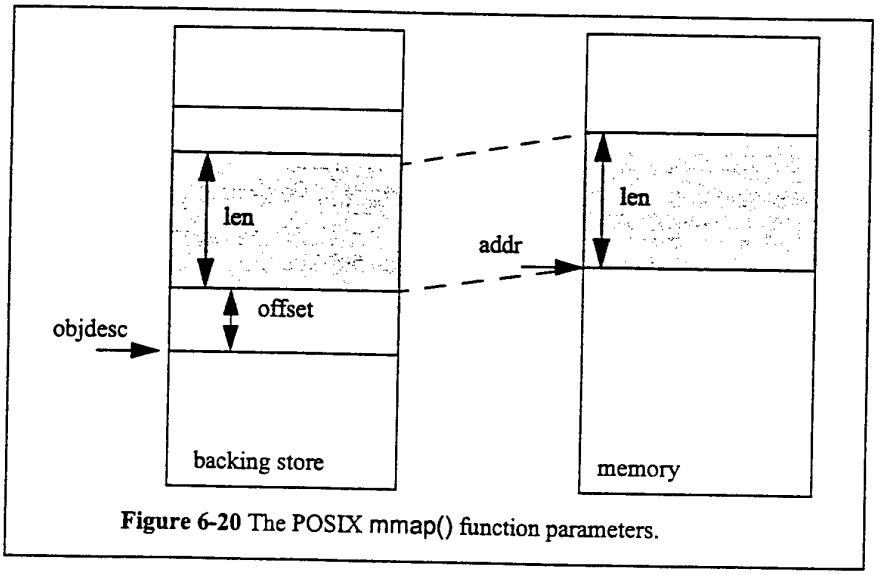


Figure 6-20 The POSIX mmap() function parameters.

Before using the shared memory object, we must map it into the process memory space using the mmap() function. POSIX assumes that shared memory objects fundamentally reside in a backing store such as a disk and are then mapped into the address space of the process. Figure 6-20 shows the definitions of the basic mmap() parameters. The value returned by shm_open(), objdesc, is the origin of the shared memory in the backing store. mmap allows the process to map in a subset of that space starting at the offset. The length of the mapped space is len. The start of the block in the process's memory space is addr. mmap() also requires you to set the protection mode for the mapped memory (O_RDWR, etc.). Here is a sample call to mmap():

```
if (mmap(addr,len,O_RDWR,MAP_SHARED,objdesc,0)  == NULL) {
    /* error*/
}
```

The MAPS_SHARED parameter tells mmap to propagate all writes to all processes that share this memory block. You use the munmap() function to unmap the memory when the process is done with it:

```
if (munmap(startadrs,len) < 0) { /* error */ }
```

This function unmaps shared memory from startadrs to startadrs+len. Finally, the close() function is used to dispose of the shared memory block:

```
close(objdesc);
```

Only one process calls shm_open() to create the shared memory object and close() to destroy it; every process (including the one that created the object) must use mmap() and mummap() to map it into their address space.

6.7.4 Message-Based Communication

The **pipe** is very familiar to Unix users from its shell syntax:

```
% foo file1 | baz > file2
```

In this command, the output of foo is sent directly to the baz program's standard input by the operating system. The vertical bar (|) is the shell's notation for a pipe; programs use the pipe() function to create pipes.

A parent process uses the pipe() function to create a pipe to talk to a child. It must do so before the child itself is created or it won't have any way to pass a pointer to the pipe to the child. Each end of a pipe appears to the programs as a file—the process at the head of the pipe writes to one file descriptor while the tail process reads from another file descriptor. The pipe() function returns an array of file descriptors, the first for the write end and the second for the read end.

Here is an example:

```
if (pipe(pipe_ends) < 0) { /* create the pipe, check for errors */
    perror("pipe");
    break;
}
/* create the process */
childid = fork();
if (childid == 0) { /* the child reads from pipe_ends[1]*/
    childargs[0] = pipe_ends[1];
    /* pass the read end descriptor to the new incarnation of child */
    execv("mychild",childargs);
    perror("execv");
    exit(1);
}
else { /* the parent writes to pipe_ends[0] */
    ...
}
```

POSIX also supports message queues under the _POSIX_MESSAGE_PASSING facility. The advantage of a queue over a pipe is that, since queues have names, we don't have to create the pipe descriptor before creating the other process using it, as with pipes.

The name of a queue follows the same rules as for semaphores and shared memory: it starts with a "/" and contains no other "/" characters. In this code, the O_CREAT flag to mq_open() causes it to creates the named queue if it doesn't yet exist and just opens the queue for the process if it does already exist:

```
struct mq_attr mq_attr; /* attributes of the queue */
mqd_t myq; /* the queue descriptor */

mq_attr.mq_maxmsg = 50; /* maximum number of messages */
mq_attr.mq_msgsize = 64; /* maximum size of a message */
mq_attr.mq_flags = 0; /* flags */
myq = mq_open("/q1",O_CREAT | RDWR,S_IRWXU,&mq_attr);
```

We use the queue descriptor myq to enqueue and dequeue messages:

```
char data[MAXLEN], rcvbuf[MAXLEN];

if (mq_send(myq,data,len,priority) < 0) { /* error */ }
nbytes = mq_receive(myq,rcvbuf,MAXLEN,&prio);
```

Messages can be prioritized, with a priority value between 0 and MQ_PRIO_MAX (there are at least 32 priorities available). Messages are inserted into the queue such that they are after all existing messages of equal or higher priority and before all lower-priority messages.

When a process is done with a queue, it calls mq_close():

```
i = mq_close(myq);
```

6.8 Evaluating Operating System Performance

The scheduling policy does not tell us all that we would like to know about the performance of a real system running processes. Our analysis of scheduling policies makes several simplifying assumptions:

- We have assumed that context switches require zero time. Although it is often reasonable to neglect context switch time when it is much smaller than the process execution time, context switching can add significant delay in some cases.
- We have assumed that we know the execution time of the processes. In fact, we learned from Section 5.6 that program time is not a single number, but can be bounded by worst-case and best-case execution times.
- We probably determined worst-case or best-case times for the processes in isolation. But, in fact, they interact with each other in the cache. Cache conflicts between processes can drastically degrade process execution time.

RTOS simulators

Some RTOSs provide simulators or other tools that allow you to view the operation of the processes in the system. These tools will show not only abstract events like process activations, but also context switching time, interrupt response time, and other overheads. This sort of view can be helpful in both functional and performance debugging.

caching effects

Many real-time systems have been designed assuming there is no cache present, even though one actually exists. This grossly conservative assumption is made because the system architects lack tools which let them analyze the effect of caching. Since they do not know where caching will cause problems, they are forced to retreat to the simplifying assumption that there is no cache. The result is extremely over-designed hardware, which has much more computational power than is necessary. However, just as experience tells us that a well-designed cache provides significant performance benefits for a single program, a properly-sized cache can allow a microprocessor to run a set of processes much more quickly. By analyzing the effects of the cache, we can make much better use of the available hardware.

cache modeling

Li and Wolf [Li00] developed a model for estimating the performance of multiple processes sharing a cache. In the model, some processes can be given reservations in the cache, such that only that process can inhabit that section of the cache; other processes are left to share the cache. We generally want to use cache partitions only for performance-critical processes since cache reservations are wasteful of limited cache space. Performance is estimated by constructing a schedule, taking into account not just execution time of the processes but also the state of the cache. Each process in the shared section of the cache are modeled by a binary variable: 1 if present in the cache and zero if not. Each process is also characterized by three total execution times: assuming no caching; with typical caching; and with all code always resident in the cache. The always-resident time is unrealistically optimistic but it can be used to find a lower-bound on the required schedule time. During construction of the schedule, we can look at the current cache state to see whether the no-cache or typical-caching execution time should be used at this point in the schedule. We can also update the cache state if the cache is needed for another process. Although this model is simple, it provides much more realistic performance estimates than either assuming the cache is non-existent or is perfect. Example 6-10 shows how cache management can improve CPU utilization.

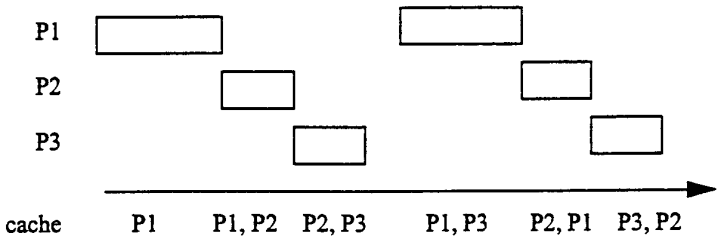
Example 6-10 Effects of scheduling on the cache

Consider a system which has three processes:

process	worst-case CPU time	average-case CPU time
P1	8	6
P2	4	3
P3	4	3

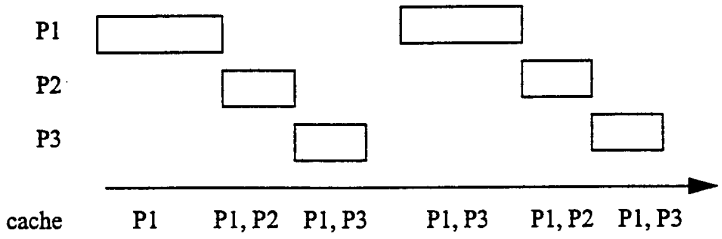
Each process takes up half the cache, so only two processes can be in the cache at the same time.

Here is a first schedule which uses a least-recently-used cache replacement policy on a process-by-process basis:



In the first iteration, we must fill up the cache, but even in subsequent iterations, competition between all three processes ensures that a process is never in the cache when it starts to execute. As a result, we must always use the worst-case execution time.

Here is another schedule in which we have reserved half the cache for P1. This leaves P2 and P3 to fight over the other half of the cache:



In this case, P2 and P3 still compete, but P1 is always ready. After the first iteration, we can use the average-case execution time for P1, which gives us some spare CPU time that could be used for additional operations.

6.9 Power Optimizations for Processes

We learned in Section 3.7 about the features that CPUs provide to manage power consumption. The RTOS and system architecture can use static and dynamic power management mechanisms to help manage the system's power consumption. A **power management policy** [Ben00] is a strategy for determining when to perform certain power management operations. A power management policy in general looks at the state of the system to determine when to take actions; however, the overall strategy embodied in the policy should be designed based upon the characteristics of the static and dynamic power management mechanisms.

cost of power management modes

Going into a low-power mode takes time; generally, the more that is shut off, the longer the delay incurred during reactivation. Because power-down and power-up are not free, modes should be changed carefully. Determining

when to switch into and out of a power-up mode requires an analysis of the overall system activity:

- Avoiding a power-down mode can cost unnecessary power.
- Powering down too soon can cause severe performance penalties.

Re-entering run mode typically costs a considerable amount of time.

power-up strategies

A straightforward method is to power-up the system when a request is received. This works so long as the delay in handling the request is acceptable.

predictive shutdown

A more sophisticated technique is **predictive shutdown**. The goal is to predict when the next request will be made and to start the system just before that time, saving the requestor the startup time. Clearly, it is possible to exactly predict request times only in very limited circumstances, but some such systems do exist, as described in Application Note 6-2.

Application Note 6-2 Predictive shutdown in pagers

A pager can use a predictive shutdown very effectively because the pager network is synchronous. The network assigns a time slot to each pager on the system. The pager uses a timer to determine when the next message should arrive. Between messages, the bulk of the pager remains powered down, while the counter remains on. At the proper time, the timer wakes up the rest of the system, which then listens for its message. Because a typical pager system runs at only 1200 baud, the timer can be built to sufficient accuracy even in a low-power, low-cost system. As a result, the wakeup times are totally predictable.

---

probabilistic techniques

In general, predictive shutdown techniques are probabilistic—they make guesses about activity patterns based on a probabilistic model of expected behavior. Because they rely on statistics, they may not always correctly guess the time of the next activity. This can cause two sorts of problems:

- The requestor may have to wait for an activity period. In the worst case, the requestor may not make a deadline due to the delay incurred by system startup.
- The system may restart itself when no activity is imminent. As a result, the system will waste power

Clearly, the choice of a good probabilistic model of service requests is important. The policy mechanism should also not be too complex, since the power it consumes to make decisions is part of the total system power budget.

Several predictive techniques are possible. A very simple technique is to use fixed times: if the system does not receive any inputs in an interval of length T_{on}, it shuts down; a powered-down system waits for a period T_{off} before returning to the power-on mode. The choice of T_{off} and T_{on} must be determined by experimentation. Srivastava et al [Sri94] found one useful rule for

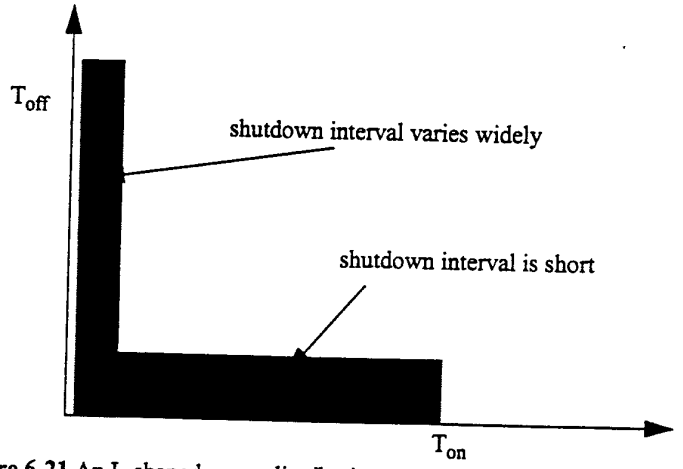


Figure 6-21 An L-shaped usage distribution.

graphics terminals. They plotted the observed idle time (T_{off}) of a graphics terminal vs. the immediately preceding active time (T_{on}). The result was an L-shaped distribution as illustrated in Figure 6-21. In this distribution, the idle period after a long active period is usually very short and the length of the idle period after a short active period is uniformly distributed. Based on this distribution, they proposed a shutdown threshold that depended upon the length of the last active period: they shut down when the active period length was below a threshold, putting the system in the vertical portion of the L distribution.

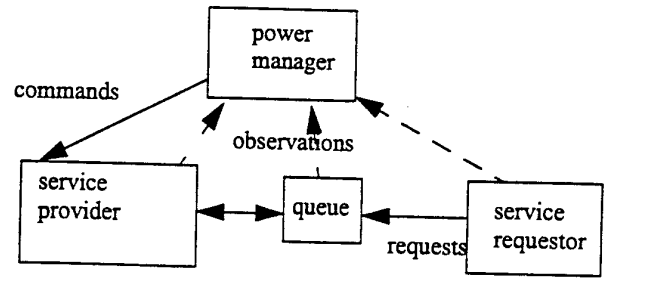


Figure 6-22 Architecture of a power managed system [Ben00].

A more advanced technique is based upon a more thorough analysis of the system state [Pal00]. As shown in Figure 6-22, we need to consider several elements of the total managed system: the service provider is the machine whose power is being managed; the service requestor is the machine or person making requests of that power-managed system; a queue is used to hold pending requests (for example, while waiting for the service provider is powering up); and the power manager is responsible for sending power management commands to the provider. The power manager can observe the

behavior of the requestor, provider, and queue. Each of these elements can be modeled by a Markov model, which is a probabilistic state machine. Paleologo et al showed that a power management policy that maximizes system performance while meeting some power consumption limitation can be solved in polynomial time.

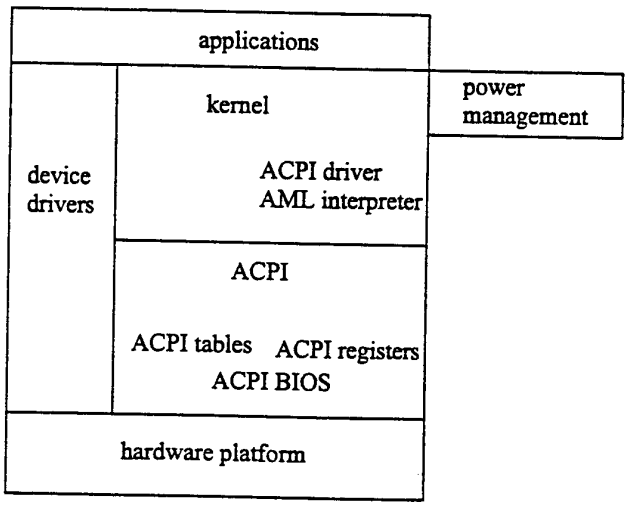


Figure 6-23 The Advanced Configuration and Power Interface and its relationship to a complete system.

ACPI power management system

The **Advanced Configuration and Power Interface (ACPI)** is an open industry standard for power management services. It is designed to be compatible with a wide variety of operating systems. It was targeted initially to PCs. The role of ACPI in the system is illustrated in Figure 6-23: ACPI provides some basic power management facilities and abstracts the hardware layer; the operating system has its own power management module that determines the policy; the operating system then uses ACPI to send the required controls to the hardware and to observe the hardware's state as input to the power manager.

ACPI supports five basic global power states:

- G3. the mechanical off state, in which the system consumes no power.
- G2, the soft off state, which requires a full operating system reboot to restore the machine to working condition. This state has four substates:
  - S1: a low wake-up latency state with no loss of system context;
  - S2: a low wake-up latency state with a loss of CPU and system cache state;
  - S3: a low wake-up latency state in which all system state except for main memory is lost;
  - S4: the lowest-power sleeping state, in which all devices are turned off.

- G1, the sleeping state, in which the system appears to be off and the time required to return to working condition is inversely proportional to the power consumption.
- G0, the working state, in which the system is fully usable.
- The legacy state, in which the system does not comply with ACPI.

The power manager typically includes an observer that receives messages through the ACPI interface that describe the system behavior. It also includes a decision module that determines power management actions based upon those observations.

6.10 Example: Telephone Answering Machine

In this section we will design a digital telephone answering machine. The system will store messages in digital form rather than on an analog tape. To make life more interesting, we will use a simple algorithm to compress the voice data so that we can make more efficient use of the limited amount of available memory.

6.10.1 Theory of Operation and Requirements

In addition to studying the compression algorithm, we also need to learn a little about the operation of telephone systems.

ADPCM

The compression scheme we will use is known as **adaptive differential pulse code modulation** (**ADPCM**). Despite the long name, the technique is relatively simple but can yield 2x compression ratios on voice data.

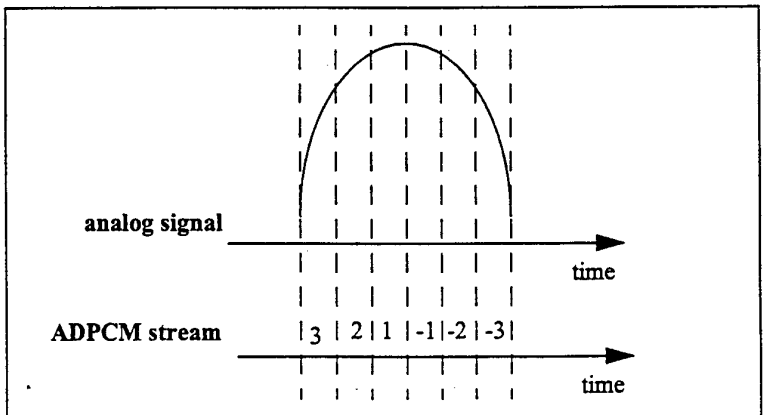


Figure 6-24 The ADPCM coding scheme.

The ADPCM coding scheme is illustrated in Figure 6-24. Unlike traditional sampling, in which each sample shows the magnitude of the signal at a particular time, ADPCM encodes changes in the signal. The samples are

expressed in a **coding alphabet** whose values are in a relatively small range which spans both negative and positive values: in this case, {-3,-2,-1,1,2,3}. Each sample is used to predict the value of the signal at the current instant from the previous value. At each point in time, the sample is chosen such that the error between the predicted value and the actual signal value is minimized.

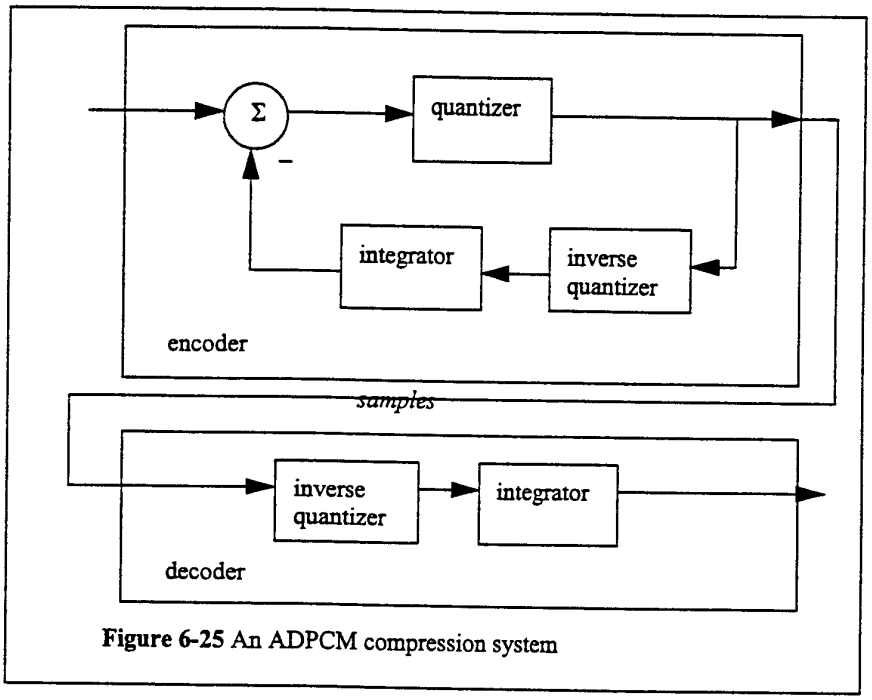


Figure 6-25 An ADPCM compression system

An ADPCM compression system, including an encoder and decoder, is shown in Figure 6-25. The encoder is more complex, but both the encoder and decoder use an integrator to reconstruct the waveform from the samples. The integrator simply computes a running sum of the history of the samples; since the samples are differential, integration reconstructs the original signal. The encoder compares the incoming waveform to the predicted waveform (the waveform that will be generated in the decoder). The quantizer encodes this difference as the best predictor of the next waveform value. The inverse quantizer allows us to map bit-level symbols onto real numerical values; for example, the 8 possible codes in a 3-bit code may be mapped onto floating-point numbers. The decoder simply uses an inverse quantizer and integrator to turn the differential samples into the waveform.

telephone system basics

Our answering machine will ultimately be connected to a telephone **subscriber line** (though for testing purposes we will construct a simulated line). At the other end of the subscriber line is the **central office**. All information is carried on the phone line in analog form over a pair of wires. In addition to analog/digital and digital/analog converters to send and receive voice data, we need to sense two other characteristics of the line:

- **ringing**: The central office sends a ringing signal to the telephone when a call is waiting. The ringing signal is in fact a 90 V RMS sinusoid, but we can use analog circuitry to produce a 0 for no ringing and 1 for ringing.
- **off-hook**: The telephone industry term for answering a call is going **off-hook**; the technical term for hanging up is going **on-hook**. (This creates some initial confusion since *off-hook* means the telephone is active and *on-hooks* means it is not in use, but the terminology starts to make sense after a few uses.) Our interface will send a digital signal to take the phone line off-hook, which will cause analog circuitry to make the necessary connection so that voice data can be sent and received during the call.

requirements

We can now write the requirements for the answering machine. We will assume that the interface is not to the actual phone line but to some circuitry that provides voice samples, off-hook command, etc. That will let us test our system with a telephone line simulator, then build the analog circuitry necessary to connect to a real phone line. We'll use the term **outgoing message (OGM)** to refer to the message recorded by the owner of the machine and played at the start of every phone call.

name:	Digital telephone answering machine.
purpose:	Telephone answering machine with digital memory, using speech compression.
inputs:	*Telephone:* voice samples, ring indicator. *User interface:* microphone, play messages button, record OGM button.
outputs:	*Telephone:* voice samples, on-hook/off-hook command. *User interface:* speaker, # messages indicator, message light.

functions:	*Default mode:* When machine receives ring indicator, it signals off-hook, plays the OGM, then records the incoming message. Maximum recording length for incoming message is 30 seconds, at which time the machine hangs up. If the machine runs out of memory, the OGM is played and the machine then hangs up without recording. *Playback mode:* When the play button is depressed, the machine plays all messages. If the play button is depressed again within five seconds, the messages are played again. Messages are erased after playback. *OGM editing mode:* When the user hits the record OGM button, the machine records an outgoing message of up to 10 seconds. When the user holds down the record OGM button and hits the play button, the OGM is played back.
performance:	Should be able to record about 30 minutes of total voice, including incoming and outgoing messages. Voice data is sampled at the standard telephone rate of 8 kHz.
manufacturing cost:	Consumer product range: approx. $50.
power:	Powered by AC through a standard power supply.
physical size and weight:	Comparable in size and weight to a desk telephone.

We have made a few arbitrary decisions about the user interface in these requirements. The amount of voice data that can be saved by the machine should in fact be determined by two factors: the price per unit of DRAM at the time at which the device goes into manufacturing (since the cost will almost certainly drop from the start of design to manufacture) and the projected retail price at which the machine must sell. The protocol when the memory is full is also arbitrary; it would make at least as much sense to throw out old messages and replace them with new ones and ideally the user could select which protocol to use. Extra features like an indicator showing the number of messages or a save messages feature would also be nice to have in a real consumer product.

6.10.2 Specification

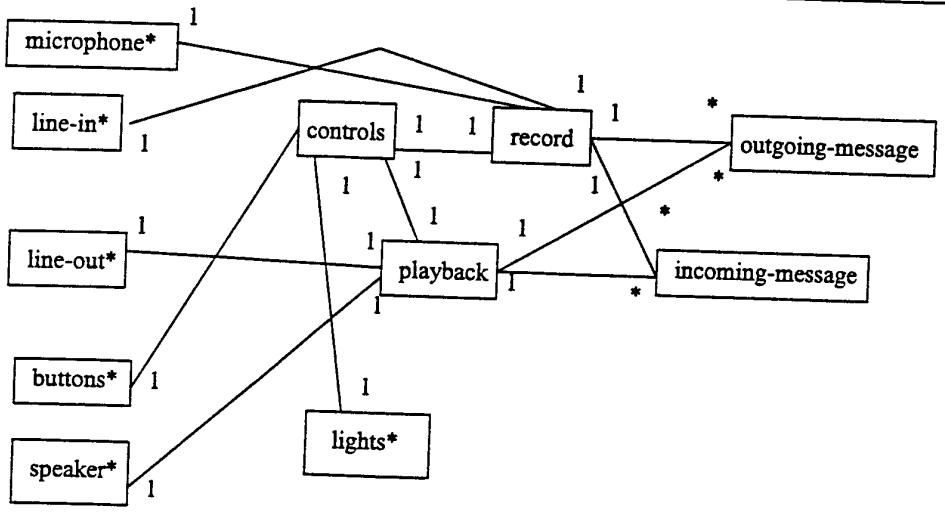


Figure 6-26 Class diagram for the answering machine.

Figure 6-26 shows the class diagram for the answering machine. In addition

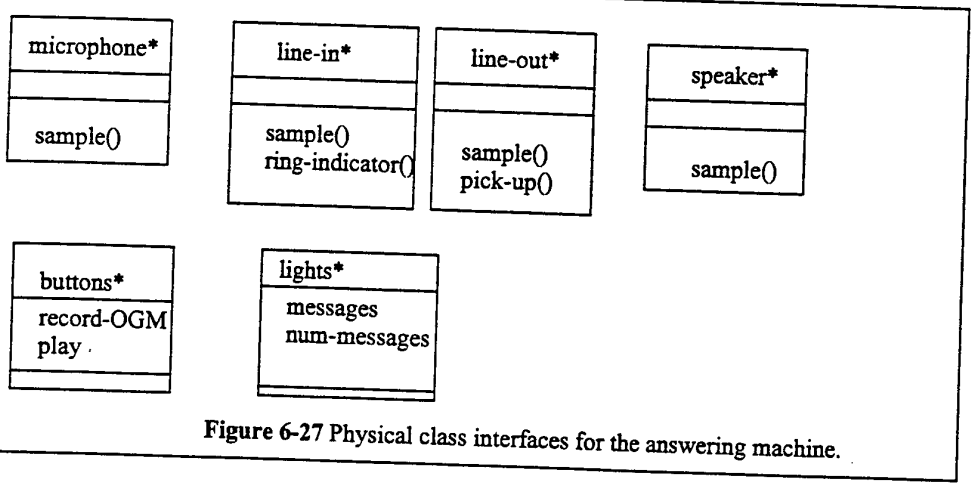


Figure 6-27 Physical class interfaces for the answering machine.

to the classes that perform the major functions, we also use classes to describe the incoming and outgoing messages. As we will see shortly, these classes are related.

The definitions of the physical interface classes are shown in Figure 6-27. The buttons and lights simply provide attributes for their input and output values. The phone line, microphone, and speaker are given behaviors that let us sample their current values.

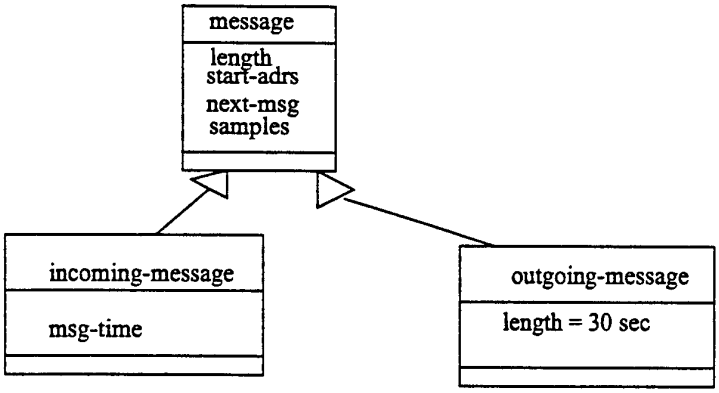


Figure 6-28 The message classes for the answering machine.

The message classes are defined in Figure 6-28. Since incoming and outgo-

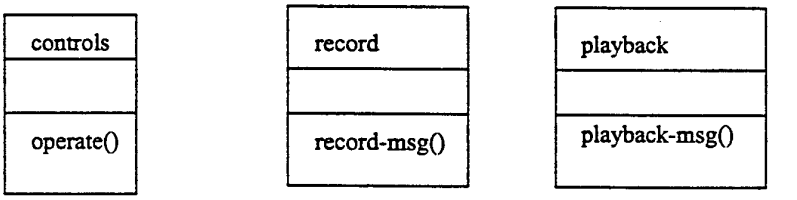


Figure 6-29 Operational classes for the answering machine.

ing message types share many characteristics, we derive them both from a more fundamental message type.

The major operational classes—*controls*, *record*, and *playback*—are defined in Figure 6-29. The controls class provides an operate() behavior that oversees the user-level operations. The record and playback classes provide behaviors that handle writing and reading sample sequences.

The state diagram for the controls operate behavior is shown in Figure 6-30. Most of the user activities are relatively straightforward. The most complex is answering an incoming call. As with the answering machine of Section 5.10, we want to be sure that a single depression of a button causes the required action to be taken exactly once; this requires edge detection on the button signal.

State diagrams for *record-msg* and *playback-msg* are shown in Figure 6-31. We have parameterized the specification for record-msg so that it can be used either from the phone line or from the microphone. This requires parameterizing the source itself and the termination condition.

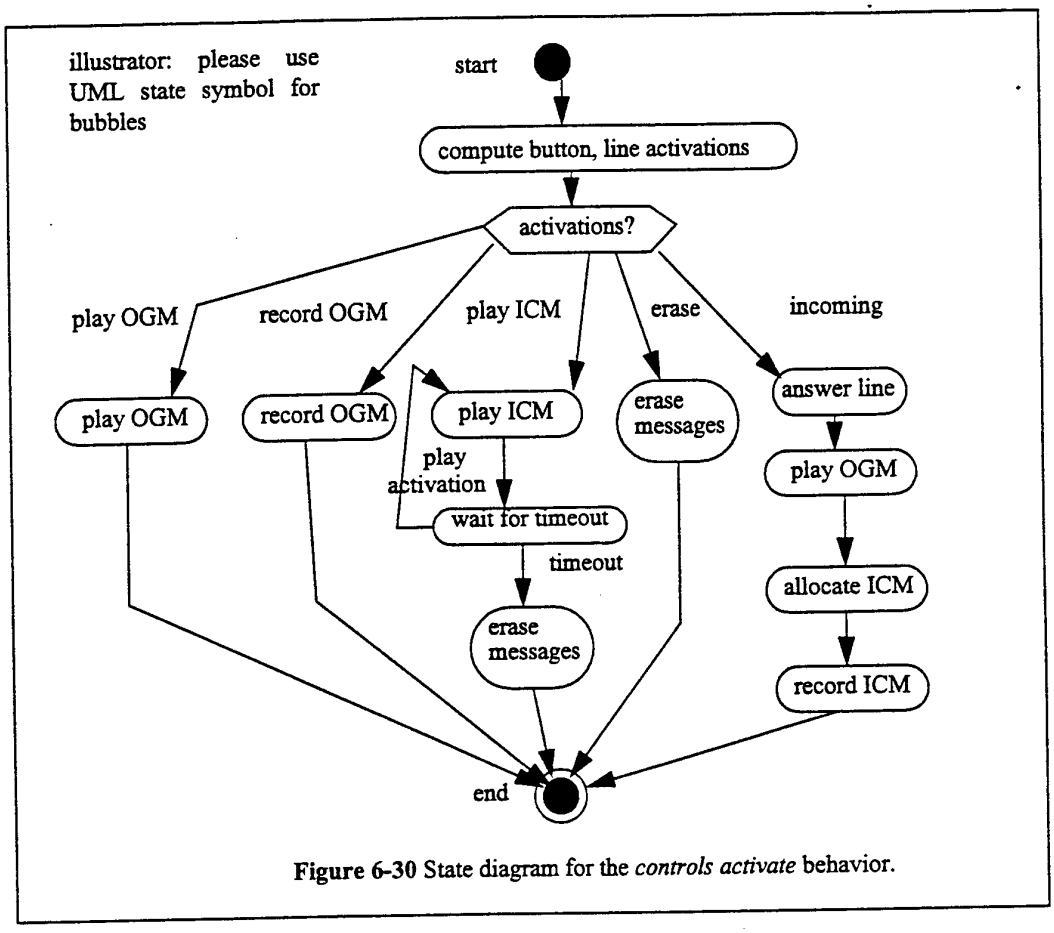


Figure 6-30 State diagram for the *controls activate* behavior.

6.10.3 System Architecture

The machine consists of two major subsystems from the user's point of view: the user interface and the telephone interface. The user and telephone interfaces both appear internally as I/O devices on the CPU bus with the main memory serving as the storage for the messages.

The software splits into seven major pieces:

- the **front panel module** handles the buttons and lights;
- the **speaker module** handles sending data to the user's speaker;
- the **telephone line module** handles off-hook detection and on-hook commands;
- the **telephone input and output modules** handle receiving samples from and sending samples to the telephone line.
- the **compression module** compresses data and stores it in memory;
- the **decompression module** uncompresses data and sends it to the speaker module.

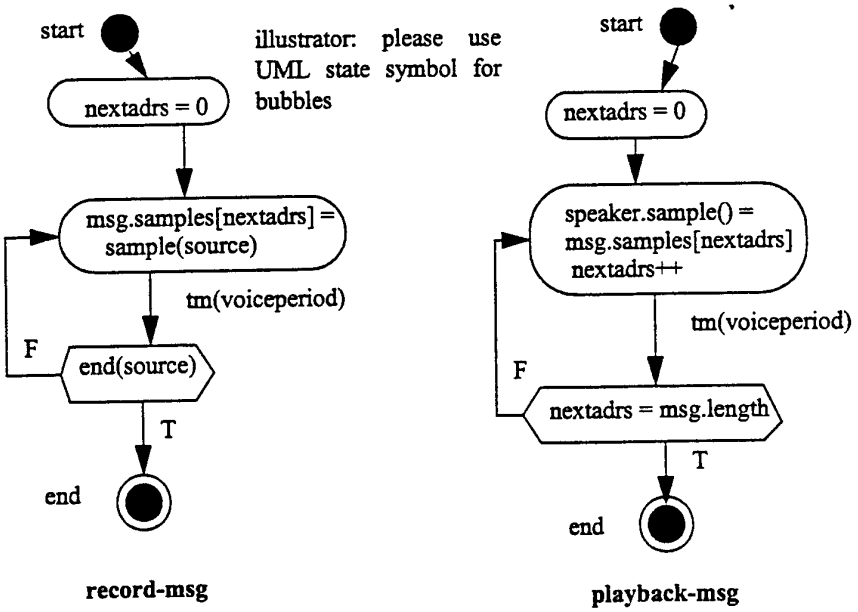


Figure 6-31 State diagrams for the *record-msg* and *playback-msg* behaviors.

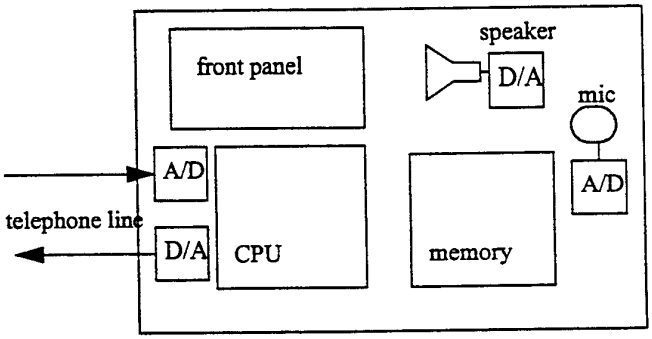


Figure 6-32 The hardware structure of the answering machine.

We can figure out the execution model for these modules based on the rates at which these modules must work and the ways in which they communicate.

- The front panel and telephone line modules must regularly test the buttons and phone line but this can be done at a fairly low rate. They can therefore run as polled processes in the software's main loop:

```
while (TRUE) {
    check_phone_line();
    run_front_panel();
}
```

- The speaker and phone input and output modules must run at higher, regular rates and are natural candidates for interrupt processing. These modules don't run all the time and so can be disabled by the front panel and telephone line modules when they are not needed.
- The compression and decompression modules run at the same rate as the speaker and telephone I/O modules but they are not directly connected to devices. We will therefore call them as subroutines to the interrupt modules.

One subtlety is that we must construct a very simple file system for messages, since we have a variable number messages each with variable length. Since messages vary in length, we must record the length of each one. In this simple specification we always play back the messages in the order in which they were recorded, so we don't have to keep a full-fledged directory. If we allowed users to selectively delete messages and save others, we would have to build some sort of directory structure for the messages.

The hardware architecture is straightforward and illustrated in Figure 6-32. The speaker and telephone I/O devices appear as standard A/D and D/A converters. The telephone line appears as a one-bit input device (ring detect) and a one-bit output device (off-hook/on-hook). The compressed data is kept in main memory.

6.10.4 Component Design and Testing

Performance analysis is important in this case because we want to make sure that we don't spend so much time compressing that we miss voice samples. In a real consumer product, we would carefully design the code so that we could use the slowest, cheapest possible CPU that would still perform the required processing in the available time between samples. In this case, we will choose the microprocessor in advance for simplicity and simply ensure that all the deadlines are met.

An important class of problems that should be adequately tested is memory overflow. The system can run out of memory at any time, not just between messages. The modules should be tested to ensure that they do reasonable things when all the available memory is used up.

6.10.5 System Integration and Testing

We can test partial integrations of the software on our host platform. Final testing with real voice data must wait until the application is moved to the target platform.

Testing your system by connecting it directly to the phone line is not a very good idea. In the United States, the Federal Communications Commission regulates equipment connected to phone lines. Beyond legal problems, a bad circuit can damage the phone line and incur the wrath of your service provider. The required analog circuitry also requires some amount of tuning, and you need a second telephone line to generate phone calls for tests. We

can build a telephone line simulator to test the hardware independent of a real telephone line. The phone line simulator consists of A/D and D/A converters plus speaker and microphone for voice data, an LED for off-hook/on-hook indication, and a button for ring generation. The telephone line interface can easily be adapted to connect to these components and for purposes of testing the answering machine the simulator behaves identically to the real phone line.

6.11 Summary

The process abstraction is forced upon us by the need to satisfy complex timing requirements, particularly for multi-rate systems. It is too difficult to write a single program which simultaneously satisfies deadlines at multiple rates because the control structure of the program becomes unintelligible. The process encapsulates the state of a computation, allowing us to easily switch between different computations.

The operating system encapsulates the complex control which coordinates process. The scheme used to determine the transfer of control between processes is known as a scheduling policy. A good scheduling policy is useful across many different applications while providing good utilization of the available CPU cycles.

It is difficult, however, to achieve 100% utilization of the CPU for complex applications. Variations in data arrivals and in computation times makes it necessary to reserve some cycles to meet worst-case conditions. Some scheduling policies achieve higher utilizations than others, but often at the cost of unpredictability—they may not guarantee that all deadlines are met. Knowledge of the characteristics of an application can be used to increase CPU utilization while complying with deadlines.

What we learned:

A process is a single thread of execution.

Preemption is the act of changing the CPU's execution from one process to another.

A scheduling policy is a set of rules that determines what process should run.

Rate monotonic scheduling (RMS) is a simple yet powerful scheduling policy.

Several interprocess communication mechanisms exist, including signals, shared memory, and pipes.

Scheduling analysis often ignores some real-world effects; cache interactions between processes are the most important effects to consider when designing a system.

Further Reading

Gallmeister [Gal95] gives a thorough and very readable introduction to POSIX in general and its real-time aspects in particular. Liu and Layland [Liu73] introduced rate-monotonic scheduling. Their paper became the foundation for real-time systems analysis and design. Benini et al [Ben00] provide a good survey of system-level power management techniques. Falik and Intrater [Fal92] describe a custom chip designed to perform answering machine operations.

Questions

Q6-1. Identify activities that operate at different rates in:

a. a PDA;

b. a laser printer;

c. an airplane.

Q6-2. Name an embedded system that requires both periodic and aperiodic computation.

Q6-3. An audio system processes samples at a rate of 44.1 kHz. At what rate could we sample the system's front panel to both simplify analysis of the system schedule and provide adequate response to the user's front panel requests?

Q6-4. Which should have lower overhead, a preemptive or co-operative context switch mechanism?

Q6-5. What information is needed in the activation record of a heavyweight process but not a lightweight process?

Q6-6. Which of these code fragments would be acceptable in a reentrant program?

a.

```
static int xxx = 5;
...
y = xxx * z;
```

b.

```
static int ptr1 = NULL;
...
ptr1 = init_list();
```

c.

```
teststring = "";
...
int i = 0;
while (i < argc && teststring != NULL) {
    teststring = argv[i];
    ...
```

Q6-7. Draw a UML class diagram for a process in an operating system. The process class should include the necessary attributes and behaviors required of a typical process.

Q6-8. Draw a UML sequence diagram showing a cooperative context switch.

Q6-9. Draw a UML sequence diagram showing a preemptive context switch.

Q6-10. What factors provide a lower bound on the period at which the system timer interrupts for preemptive context switching?

Q6-11. What factors provide an upper bound on the period at which the system timer interrupts for preemptive context switching?

Q6-12. What state would be useful to keep in a process that represents a hotel guest?

Q6-13. What is the distinction between the ready and waiting states of process scheduling?

Q6-14. Give an example of:

a. blocking interprocess communication;

b. non-blocking interprocess communication.

Q6-15. Assuming that you have a routine called swap(int *a,int *b) that atomically swaps the values of the memory locations pointed to b a and b, write C code for:

a. P();

b. V().

Q6-16. Draw UML sequence diagrams of two versions of P(): one that incorrectly uses a non-atomic operation to test and set the semaphore location, and another that does use an atomic test-and-set.

Q6-17. For these processes, what is the shortest interval we must examine to see all combinations of deadlines:

a.

process	deadline
P1	2
P2	5
P3	10

b.

process	deadline
P1	2
P2	4
P3	5
P4	10

c.

process	deadline
P1	3
P2	4
P3	5
P4	6
P5	10

Q6-18. Consider this system of processes executing on a single CPU:

process	execution time	deadline
P1	4	200
P2	1	10
P3	2	40
P4	6	50

Can we add another instance of P1 to the system and still meet all the deadlines using RMS?

Q6-19. Given this set of processes running on a single CPU, what is the maximum execution time P5 for which all the processes will be schedulable using RMS?

process	execution time	deadline
P1	1	10
P2	18	100
P3	2	20
P4	5	50
P5	x	25

Q6-20. A set of processes is scheduled using RMS. For the process execution times and periods shown below, show the state of the processes at the critical instant for each of these processes:

a. P1;

b. P2;

c. P3.

process	time	deadline
P1	1	4
P2	1	5
P3	1	10

Q6-21. For the given process execution times and periods, show how much CPU time of higher-priority processes will be required during one period of each of these processes:

a. P1;

b. P2;

c. P3;

d. P4.

process	time	deadline
P1	1	5
P2	2	10
P3	2	25
P5	5	50

Q6-22. For the processes shown below:

a. Schedule the processes using an RMS policy.

b. Schedule the processes using an EDF policy.

In each case, compute the schedule for an interval equal to the least-common multiple of the periods of the processes. Time starts at t=0.

process	time	deadline
P1	1	3
P2	1	4
P3	1	12

Q6-23. For the processes shown below:

a. Schedule the processes using an RMS policy.

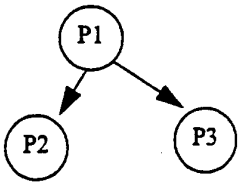
b. Schedule the processes using an EDF policy.

In each case, compute the schedule for an interval equal to the least-common multiple of the periods of the processes. Time starts at t=0.

process	time	deadline
P1	1	3

process	time	deadline
P2	1	4
P3	2	6

Q6-24. For the given set of processes, all of which share the same deadline of 12:



process	execution time
P1	2
P2	1
P3	2

a. Schedule the processes for the given arrival times using standard rate-monotonic scheduling (no data dependencies).

b. Schedule the processes taking advantage of the data dependencies. By how much is the CPU utilization reduced?

Q6-25. For the processes given below, find a valid schedule:

a. using standard RMS;

b. adding one unit of overhead for each context switch.

process	time	deadline
P1	2	30
P2	5	40
P3	7	120
P4	5	60
P5	1	15

Q6-26. For the processes and deadlines given below:

a. schedule the processes using RMS;

b. schedule using EDF and compare the number of context switches required for EDF and RMS.

process	time	deadline
P1	1	5
P2	1	10
P3	2	20
P4	10	50
P5	7	100

Q6-27. Draw a UML class diagram for two classes: a front-panel class that reads panel activity and an action class that performs actions based on the front panel requests. The front-panel class uses a Unix signal to communicate with action.

Q6-28. In each of these circumstances, would shared memory or message-passing communication be better? Explain.

a. A cascaded set of digital filters.

b. A digital video decoder and a process that overlays user menus on the display.

c. A software modem process and a printing process in a fax machine.

Q6-29. If you wanted to reduce the cache conflicts between the most computationally-intensive parts of two processes, what are two ways that you could control the locations of the processes' cache footprints?

Q6-30. Draw a state diagram for the preditive shutdown mechanism of a pager. The pager wakes itself up once every five minutes for 0.01 second to listen for its address. It goes back to sleep if it does not hear its address or after it has received its message.

Q6-31. How would you use the ADPCM method to encode an unvarying (DC) signal with the coding alphabet {-3,-2,-1,1,2,3}?

Lab Exercises

L6-1. Using your favorite operating system, write code to spawn a process that writes "Hello, world" to the screen.

L6-2. Build a small serial-port device that lights LEDs based on the last character written to the serial port. Create a process that will light LEDs based upon keyboard input.

L6-3. Write a driver for an I/O device.

L6-4. Write context-switch code for your favorite CPU.

L6-5. Measure context switching overhead on an operating system.

L6-6. Using a CPU that runs an operating system that uses RMS, try to get the CPU utilization up to 100%. Vary the data arrival times to test the robustness of the system.

L6-7. Using a CPU that runs an operating system that uses EDF, try to get the CPU utilization as close to 100% as possible without failing. Try a variety of data arrival times to see how sensitive your process set is to environmental variations.

System design techniques

- Private branch exchange (PBX).
- Ink-jet printer.
- PDAs.
- Set-top boxes.
- Systems-on-silicon.

Digital telephone switches

- High-end switches are highly reliable:
  - 30 seconds downtime per year.
- Companies, homes install private branch exchanges (PBXs):
  - intercom features;
  - management of long distance charges.

Telephone switching systems

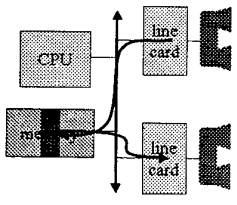
- Establish telephone calls:
  - within switch, find other phone line;
  - outside switch, find route to other line.
- Route voice samples between phones.
- Measure call time for billing.
- Provide maintenance access to switch.

Telephone terminology

- Line: distinct telephone connection.
- Off-hook: active telephone.
- On-hook: inactive telephone.
- Trunk line: group of phone lines between switches.
- POTS: plain old telephone service (analog, no fancy services).

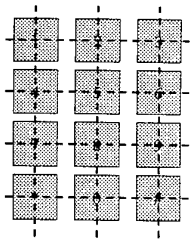
Computer-controlled telephone switching

- Voice data rates:
  - 8 bits sample (μ or A law);
  - 125 μ s period (8 kHz).
- Telephones are I/O devices.
- Computer bus is switch.

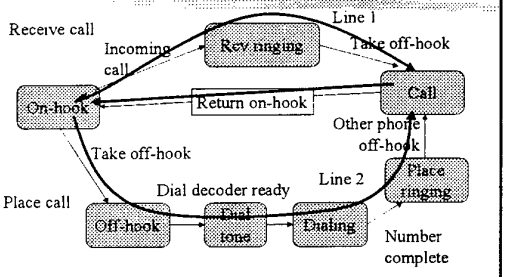


Dialing

- Dual-tone multi-frequency (DTMF): tones define row, column of key.
- Must be held for 0.1 sec.

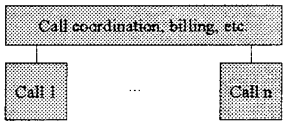


Call states



PBX model

- Calls are parallel processes:

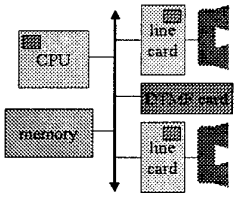


Tigerswitch system architecture

- PC platform.
  - Switch calls over ISA bus.
- Line cards are custom devices.

DTMF sensing

- Software process on CPU.
  - Uses lots of CPU time.
- Analog filter bank on line cards.
  - Expensive in volume.
- DSP on separate card.
  - Requires new design.



Switching

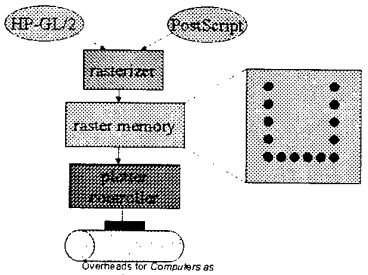
- Process-per-call is bad implementation.
  - Context switch per call per 125 μs.
- Unroll calls into one loop.
  - Each loop iteration is one call for one sample.

```
while (TRUE) {
    for (i=0; i<n_calls; i++) {
        /* from 1 to 2 */
        data = read(call[i].line1);
        write(call[i].line2,data);
        /* from 2 to 1 */
        data = read(call[i].line2);
        write(call[i].line1,data);
    }
}
```


HP DesignJet drafting plotter

- Plots up to 36 inches wide at 300 DPI.
- Combines a variety of tasks:
  - host communication;
  - graphics language interpretation;
  - rasterization;
  - device control.

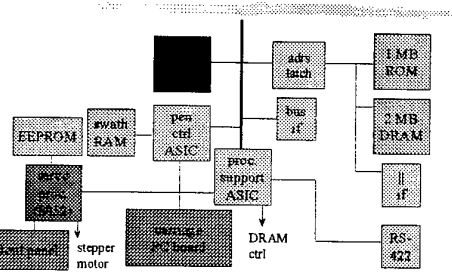
The plotting process



Design considerations

- Memory utilization is important.
  - 36 inches X large X 300 DPI X n bits/pixel is a lot of memory.
  - Requires clever algorithms to minimize raster memory requirements.
- Requires real-time control.
- Requires concurrency: read new data, rasterize, control print head.

HP DesignJet hardware architecture



Early architectural decisions

- Chose Intel 80960KA as main processor.
  - Handled parsing, rasterization control, print engine control.
  - Multiplexed bus reduced pin count.
  - Could be upgraded to floating-point if necessary.
- Used modular I/O to host system.
- Did not use disk for local storage.

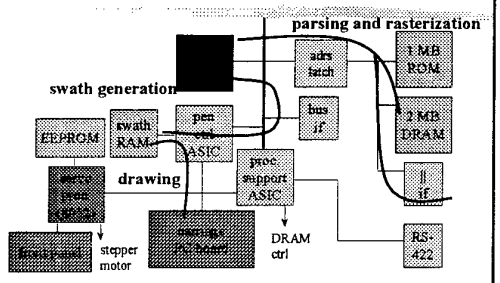
System components

- 2 MB RAM (SIMM sockets for more).
- Three ASICs:
  - pen interface;
  - processor support;
  - carriage.
- Servo processing performed by 8052 microcontroller.

Rasterization

- Plot is generated in swaths.
  - Separate swath memory.
- Pixels are generated in row order by main processor.
- Pixels are fed to pens in column order.
- Pen interface ASIC transforms row order to column order.

Data flows



Operations

- Servo processor controls stepper motor.
- Carriage processor must write, read pen alignment marks.
- Processor support ASIC provides muotiple functions: interrupt and mailbox communication.
- Motion controller decodes position of print carriage and paper; watchdogs servo.

Pen interface ASIC

- Interfaces to i960 bus, swath memory, carriage ASIC.
- Pen interface reads pixels from swath in predetermined pattern using pixel address generator.
- Must support bidirectional printing since head prints both ways.

Carriage ASIC

- Interrfaces to carriage processor bus, pen interface ASIC, servo controller.
- Reads timing control registers using the CPU bus.
- Delay registers add correction for pen alignment.

Development process

- Pixel shuffling algorithm for pen interface/carriage ASICs was prototyped in C.
- Built emulators for ASICs to allow parallel development of i960 software and hardware.

Software development environment

- Plotter software could be run on Unix workstation or target platform.
  - Differed in I/O and print engine subsystems.
  - Print engine was emulated on host with X window interface showing swath state.
- Used in-house RTOS.
- HP-GL/2 parser was legacy code.

Software development environment, cont'd.

- Rewrote vector/raster converter from assembly language to C to port to i960.
- Used gdb960 as monitor debugger on target system, communicating with host.
- Front panel developed on PC, tested by user interface designers, marketing.
- Paper loading designed by mechanical engineers.

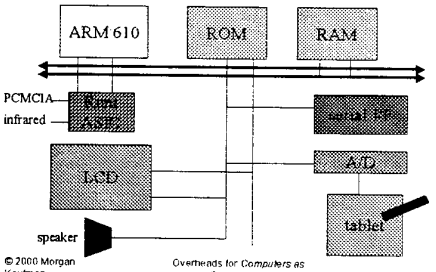
Personal digital assistant

- PDA: portable, specialized information device.
- Characteristics:
  - low cost for consumer market;
  - physically small;
  - battery-powered;
  - software-rich.

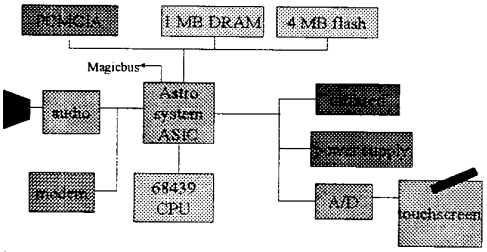
Apple Newton

- First modern PDA.
- Original used ARM 610; later version used StrongARM.
- Support operations in Runt ASIC: DMA, real-time clock, video interface, audio, PCMCIA.
- Software written in NewtonScript language.

Newton hardware architecture



Motorola Envoy hardware architecture



Feature creep

- Designers tend to add features to system during design.
  - Increases power consuption.
  - Changes mechanical design.
  - Makes software design more complex.
- Software thrashing can reduce battery life.

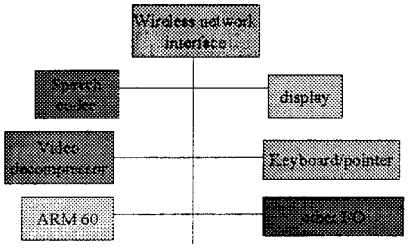
PDA power supply

- System must be designed to gracefully handle low battery power.
  - Abrupt power loss can destroy lots of data in RAM.
- Smart Battery System puts electronics in battery to measure battery performance.

InfoPad

- Brodersen et al: advanced networked multimedia information appliance.
- System performed many functions on remote systems to increase battery life.
- Made use of specialized hardware units to reduce power consumption over software implementation.

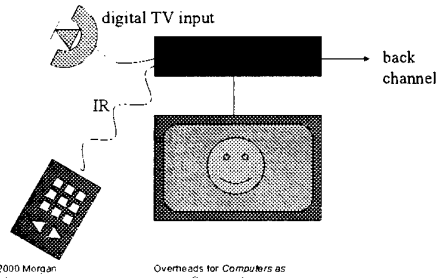
InfoPad hardware architecture



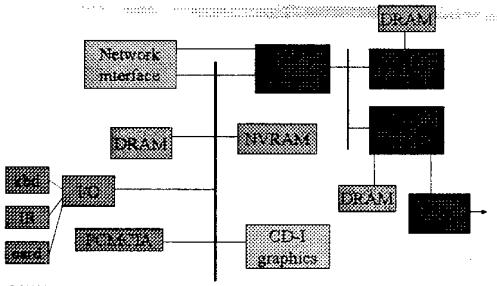
Set-top boxes

- Interface between cable/satellite and TV:
  - digital television;
  - user interface;
  - may include back channel for purchases, etc.
- Very cost-sensitive market.

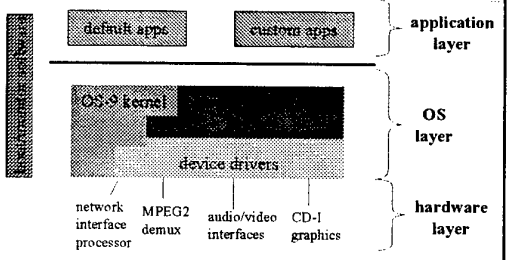
Set-top box in system



Fiber-to-curb box hardware



Fiber-to-curb box software

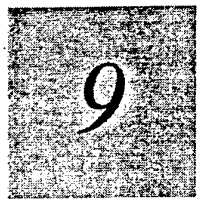


Systems-on-silicon

- Can build significant embedded systems on single chip:
  - one or more high-performance CPUs;
  - I/O devices;
  - memory.
- Advantages:
  - higher performance and lower power;
  - lower cost.

Design challenges

- Verification:
  - long turnaround time;
  - can't probe interior directly.
- Limited die size:
  - use custom architectures;
  - add custom components as accelerators.



System Design Techniques

9.1 Introduction

In this chapter we will consider the techniques required to create complex embedded systems. So far, our design examples have been small so that important concepts can be conveyed relatively simply. However, most real embedded system designs are inherently complex: their functional specifications are rich and they must obey multiple other requirements on cost, performance, etc. Here we will both look at some real embedded system designs and study some techniques that help us get a handle on the design process.

In the next section we will look at design methodologies in more detail. Section 9.3 studies requirements analysis, which captures informal descriptions of what a system must do, while Section 9.4 considers techniques for more formally specifying system functionality. Section 9.5 looks in more detail at system analysis methodologies. Section 9.6 looks at the topic of quality assurance, which must be considered throughout the design process to ensure a high-quality design. The next three sections describe more complex design examples: Section 9.7 considers a telephone switching system or PBX; Section 9.8 looks at the design of a sophisticated ink-jet printer; and Section 9.9 studies the design of personal digital assistants. Section 9.10

Draft: July 3, 2000 10:11 pm

describes the design of television set-top boxes. The next several sections study several different real-world design examples. We will wrap up with a brief consideration of **systems-on-silicon**—single-chip embedded computing systems.

In this chapter:

A deeper look into design methodologies, requirements, specification, and system analysis.

Quality assurance.

A number of real-world design examples: telephone system, ink-jet printer, PDA, set-top boxes.

Systems-on-silicon.

9.2 Design Methodologies

9.2.1 Why Design Methodologies?

This section considers the complete **design methodology**—a **design process**—for embedded computing systems. Why is process important? Because without it, we can't reliably deliver the products we want to create. Thinking about the sequence of steps necessary to build something may seem superfluous. But the fact is that everyone has their own design process, even if they don't articulate it. If you are designing embedded systems in your basement, by yourself, having your own work habits is fine. But when several people work together on a project, they need to agree on who will do things and how they will get done. Being explicit about process is important when people work together. And since many embedded computing systems are too complex to be designed and built by one person, we have to think about design processes.

product metrics

The obvious goal of a design process is to create a product that does something useful. Typical specifications for a product will include functionality (e.g., personal digital assistant), manufacturing cost (must have a retail price below $200), performance (must power up within 3 seconds), power consumption (must run for 12 hours on two AA batteries), or other properties. But a design process has several important goals beyond function, performance, and power:

- **Time-to-market**. Customers always want new features. The product that comes out first can win the market, even setting customer preferences for future generations of the product. The profitable market life for some products is 3-6 months—if you are three months late, you will never make money. In some categories, the competition is against the calendar,

not just competitors. Calculators, for example, are disproportionately sold just before school starts in the fall; miss your market window and you have to wait a year for another sales season.

- **Design cost**. Many consumer products are very cost sensitive. Industrial purchasers are also increasingly concerned about cost. The costs of designing the system are distinct from manufacturing cost—the cost of engineers' salaries, computers used in design, etc. must be spread across the units sold. In some cases, only one or a few copies of an embedded system may be built, so design costs can dominate manufacturing costs. Design costs can also be important for high-volume consumer devices when time-to-market pressures cause teams to swell in size.
- **Quality**. Customers not only want their products fast and cheap, they also want them to be right. A design methodology that cranks out shoddy products will soon be forced out of the marketplace. Correctness, reliability, and usability must be explicitly addressed from the beginning of the design job to get a high-quality product at the end.

Processes evolve over time. They change due to external and internal forces. Customers may change, requirements change, products change, the available components change. Internally, people learn how to do things better, old people move on to other projects and new people come in, companies are bought and sold to merge and shape corporate cultures.

Software engineers have spent a great deal of time thinking about software design processes. Much of this thinking has been motivated by mainframe software such as databases. But embedded applications have also inspired some important thinking about software design processes.

A good methodology is critical to building systems that work properly. Delivering buggy systems to customers always causes dissatisfaction. But in some applications, such as medical and automotive systems, bugs create serious safety problems that can endanger the lives of users. We will discuss quality in more detail in Section 9.6. As an introduction, Application Note 9-1 describes problems with that led to the loss of an unmanned Martian space probe.

Application Note 9-1Loss of the Mars Climate Orbiter

In September 1999, the Mars Climate Observer, an unmanned U. S. spacecraft designed to study Mars was lost—it most likely exploded as it heated up in the atmosphere of Mars after approaching the planet too closely. The spacecraft came too close to Mars because of a series of problems, according to an analysis by *IEEE Spectrum* and contributing editor James Oberg [Obe99]. From an embedded systems perspective, the first problem is best classified as a requirements problem. The contractors who built the spacecraft at Lockheed Martin calculated some values for flight controllers at the Jet Propulsion Laboratory (JPL). JPL did not specify the physical units to be used, but they expected them to be in newtons. The Lockheed Martin engineers returned values in units of pound force. This resulted in trajectory adjustments being 4.45 times larger than they should have been. The error

was not caught by a software configuration process; it was also not caught by manual inspections. There were concerns about the spacecraft's trajectory, however, but errors in the calculation of the spacecraft's position were not caught in time.

9.2.2 Design Flows

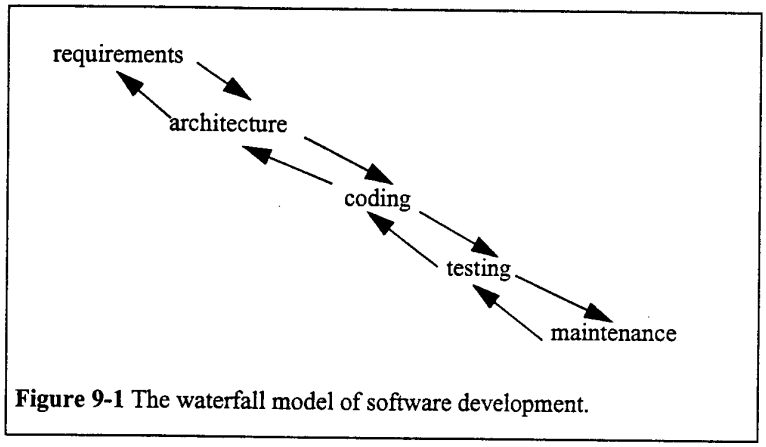


Figure 9-1 The waterfall model of software development.

A **design flow** is a sequence of steps to be followed during a design. Some of the steps may be performed by tools, such as compilers or CAD systems; other steps may be performed by hand. In this section we'll look at the basic characteristics of design flows.

waterfall model

Figure 9-1 shows the **waterfall model** introduced by Royce [Dav90], the first model proposed for the software development process. The waterfall development model has five major phases: *requirements analysis* determines the basic characteristics of the system; *architecture design* decomposes the functionality into major components; *coding* implements the pieces and integrates them; *testing* uncovers bugs; *maintenance* entails deployment in the field, bug fixes, and upgrades. The waterfall model gets its name from the largely one-way flow of work and information from higher levels of abstraction to more detailed design steps (with some limited amount of feedback to the next-higher level of abstraction). Although top-down design is ideal since it implies good foreknowledge of the implementation during early design phases, most designs are clearly not quite so top-down. Most design projects entail experimentation and changes which require bottom-up feedback. As a result, the waterfall model is today cited as a non-realistic design process. However, it is important to know what the waterfall model is to be able to understand how others are reacting against it.

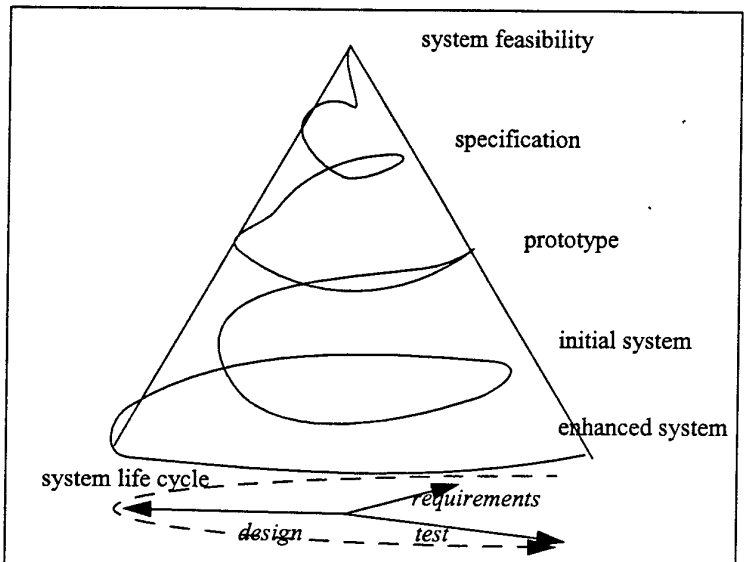


Figure 9-2 The spiral model of software design.

alternatives to the waterfall model

Figure 9-2 illustrates an alternative model of software development, the **spiral model** [Boe87]. While the waterfall model assumes that the system is built once in its entirety, the spiral model assumes that several versions of the system will be built. Early systems will be simple mock-ups constructed to aid the intuition and build experience with the system. As design progresses, more complex systems will be constructed. At each level of design, the designers go through requirements, construction, and testing phases. At later stages when more complete versions of the system are constructed, each phase requires more work, widening the design spiral. This

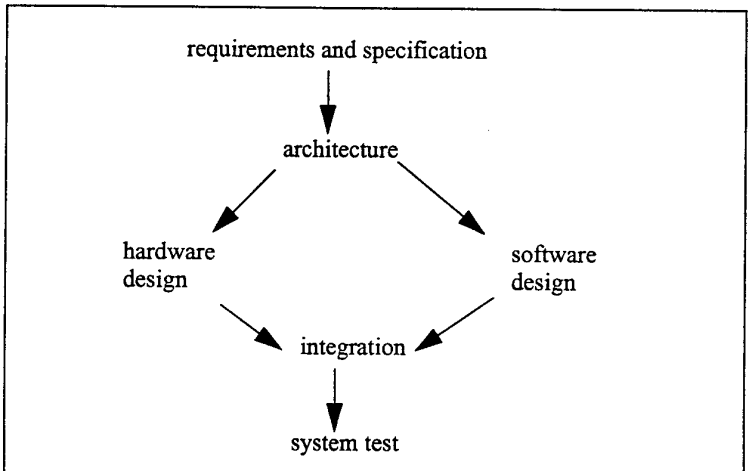


Figure 9-3 A simple hardware/software design methodology.

successive-refinement approach helps the designers understand the system

they are designing through a series of design cycles: the first cycles at the top of the spiral are very small and short, the final cycles at the spiral's bottom add detail learned from the earlier cycles of the spiral. The spiral model is more realistic than the waterfall model because we often have to take multiple iterations to add enough detail to have a completed design. However, a spiral methodology with too many spirals may take too long when design time is a major requirement.

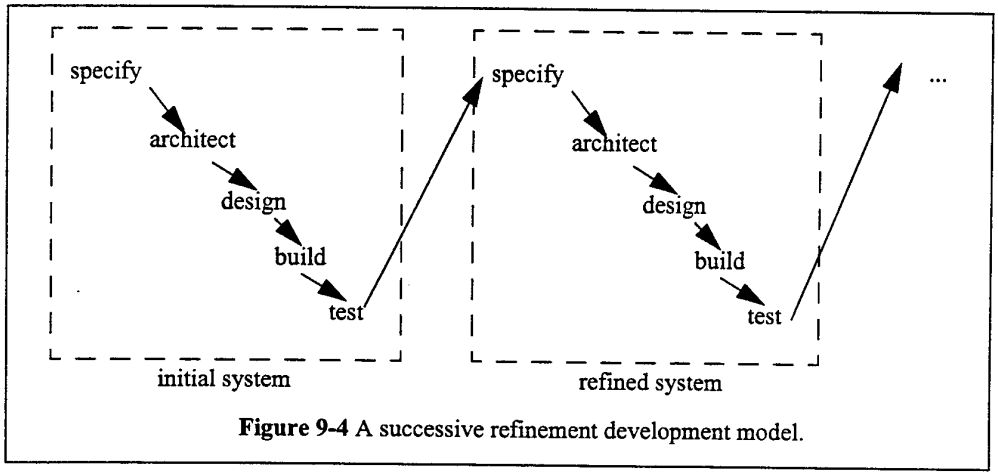


Figure 9-4 A successive refinement development model.

successive refinement

Figure 9-4 shows a **successive refinement** design methodology. In this approach, the system is built several times. A first system is used as a rough prototype; successive models of the system are further refined. This methodology makes sense when you are relatively unfamiliar with the application domain for which you are building the system—refining the system by building several increasingly complex systems allows you to test out architecture and design techniques. The various iterations may also be only partially completed; for example, continuing an initial system only through the detailed design phase may teach you enough to help you avoid many mistakes in a second design iteration that is carried through to completion.

Embedded computing systems often design their own hardware as well as software. Even if you aren't designing a board, you may be selecting boards and plugging together multiple hardware components as well as writing code. Figure 9-3 shows a design methodology for a combined hardware/software project. Front-end activities like specification and architecture simultaneously look at hardware and software aspects. Similarly, back-end integration and testing consider the whole system. In the middle, however, development of hardware and software components can go on relatively independently—while some testing of one will require stubs of the other, most of the hardware and software work can go on relatively independently.

hierarchical design flows

In fact, many complex embedded systems are themselves built of smaller designs: the complete system may require the design of significant software components, application-specific integrated circuits (ASICs), etc.; these may in turn may be built from smaller components that need to be designed. The design flow follows the levels of abstraction in the system, from complete

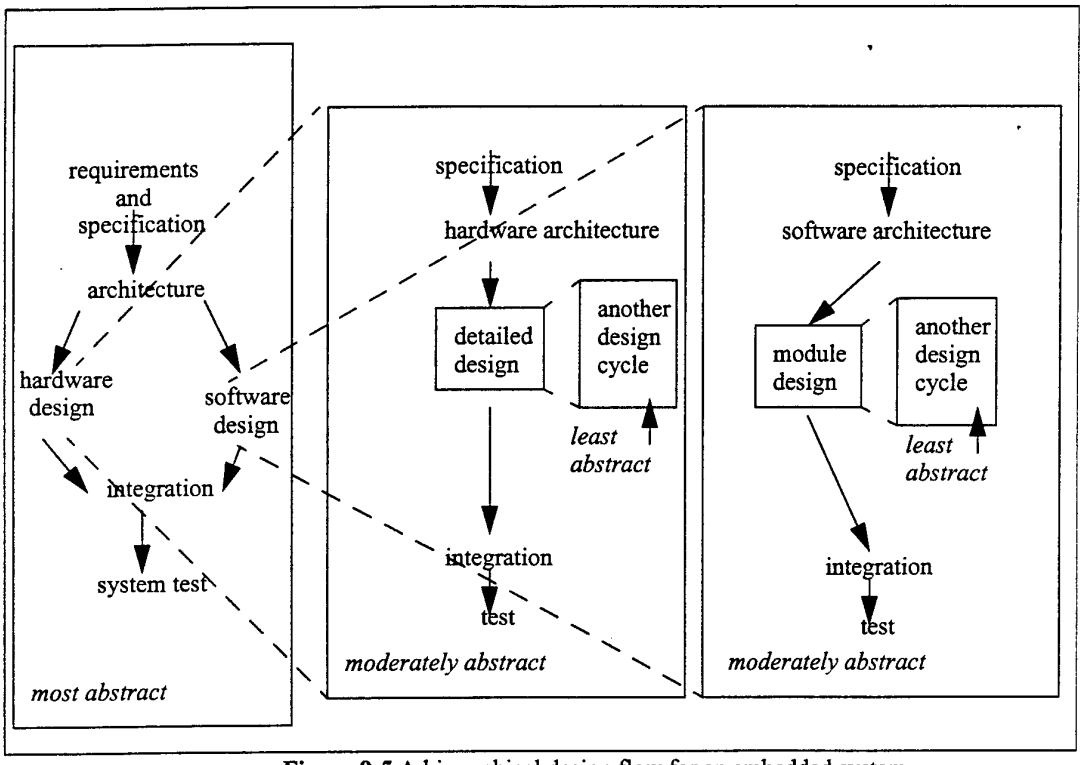


Figure 9-5 A hierarchical design flow for an embedded system.

system design flows at the most abstract to design flows for individual components. The design flow for these complex systems looks like the flow shown in Figure 9-5: the implementation phase of a flow is itself a complete flow from specification through testing. In such a large project, each flow will probably be handled by separate people or teams. The teams must rely on each other's results: the component teams take their requirements from the team handling the next higher level of abstraction; that higher-level team relies on the quality of design and testing performed by the component team. Good communication is vital in such large projects.

concurrent engineering

When designing a large system with many people, it is easy to lose track of the complete design flow and have each designer take a narrow view of his or her role in the design flow. **Concurrent engineering** tries to take a broader approach and optimize the total flow. Reduced design time is one important goal for concurrent engineering, but it can help with any aspect of the design that cuts across the design flow: reliability, performance, power consumption, etc. It tries to eliminate "over-the-wall" design steps, in which one designer performs an isolated task, then throws the result over the wall to the next designer, with little interaction between the two. In particular, reaping the most benefits from concurrent engineering usually requires eliminating the wall between design and manufacturing. Concurrent engineering efforts are made up of several elements:

- **Cross-functional teams** include members from various disciplines involved in the process, including manufacturing, hardware and software design, marketing, etc.
- **Concurrent product realization** process activities are at the heart of concurrent engineering. Doing several things at once, such as designing various subsystems simultaneously, is critical to reducing design time.
- **Incremental information sharing** and use helps minimize the chance that concurrent product realization will lead to surprises. As soon as new information becomes available, it is shared and integrated into the design. Cross-functional teams are important to the effective sharing of information in a timely fashion.
- **Integrated project management** ensures that someone is responsible for the entire project, and that responsibility is not abdicated once one aspect of the work is done.
- **Early and continual supplier involvement** helps make the best use of suppliers' capabilities.
- **Early and continual customer focus** helps ensure that the product best meets customers' needs.

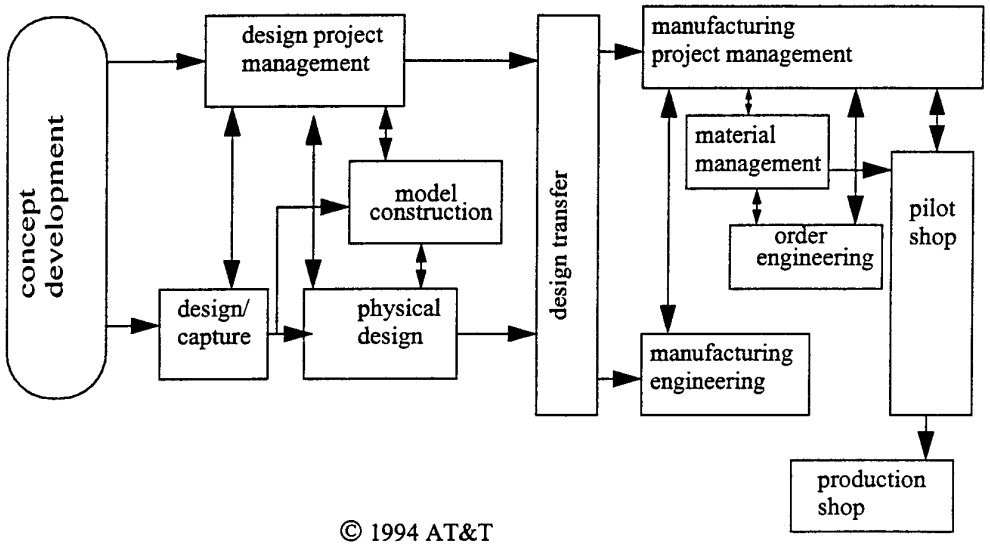
Example 9-1 describes the experiences of a telephone system design organization with concurrent engineering.

Example 9-1 Concurrent engineering applied to telephone systems

A group at AT&T applied concurrent engineering to the design of PBXs (telephone switching systems) [Gat94]. They had a large existing organization and methodology for designing PBXs; their goal was to re-engineer their process to reduce design time and make other improvements to the end product. They followed a seven-step process:

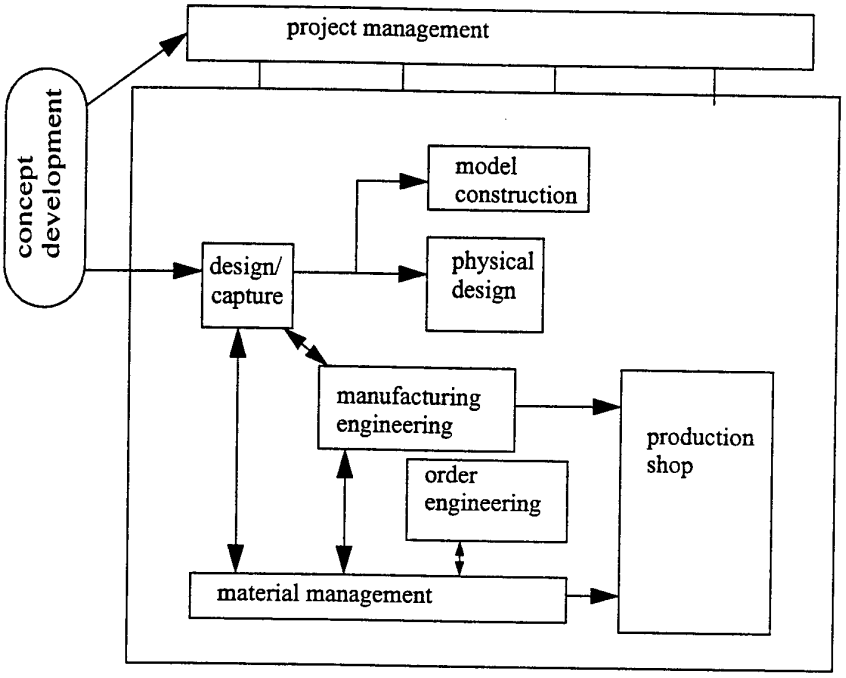
1. **Benchmarking** They compared themselves to competitors and found that it took them 30% longer to introduce a new product than it took for their best competitors. Based on this study, they decided to shoot for a 40% reduction in design time.
2. **Breakthrough improvement** Next, they identified the factors that would influence their effort. They identified three major factors: they would need increased partnership between design and manufacturing; they could not, however, change the basic organization of design labs and manufacturing; and the effort would require the support of managers at least two levels above the working level. As a result, they instituted three groups to help manage the effort. A *steering committee* was formed by mid-level managers to give feedback on the project. A *project office* was formed by an engineering manager and an operations analyst from the AT&T internal consulting organization. Finally, a *core team* of engineers and analysts was formed to make things happen.

3. Characterization of the current process The core team built flow charts and used other techniques to understand the current product development process. The existing design and manufacturing process looked like this:



They came up with several root causes of delays that had to be fixed. First, too many design and manufacturing tasks were performed sequentially. Second, groups tended to focus on intermediate milestones related to their narrow job descriptions, rather than trying to take into account the effects of their decisions on other aspects of the development process. Third, too much time was spent waiting in queues—jobs were handed off from one person to another very frequently; in many cases, the recipient of a set of jobs didn't know how to best prioritize the incoming tasks. Fixing this problem was deemed to be fundamentally a managerial problem, not a technical one. Finally, they found that too many groups had their own design databases, creating redundant data that had to be maintained and synchronized.

4. **Create the target process** Based on their studies, the core team created a model for their new development process:



5. **Verify the new process** The team undertook a pilot product development project to test out the new process. The process was found to be basically sound. Some challenges were found: for example, in the sequential project the design of circuit boards took longer than that of the mechanical enclosures, while in the new process the enclosures ended up taking longer, pointing out the need to start designing them earlier.
6. **Implement across the product line** After the pilot project, the new methodology was rolled out across the product lines. This took training of personnel, documentation of the new standards and procedures, and improvements to their information systems.
7. **Measure results and improve** The performance of the new design flow was measured. They found that product development time had been reduced from 18-30 months to 11 months.

9.3 Requirements Analysis

Before designing our system, we need to know what we are designing. The terms requirements and specifications are used in a variety of ways: some

people use them as synonyms, while others use them as distinct phases. We will use them to mean related but distinct steps in the design process: **requirements** are informal descriptions of what the customer wants, while **specifications** are more detailed, precise, and consistent descriptions of the system that can be used to create the architecture. Both requirements and specifications are, however, directed to the outward behavior of the system, not its internal structure.

The overall goal of creating a requirements document is to effectively communicate between the customers and the designers. The designers should know what they are expected to design for the customers; the customers, whether they are known in advance or represented by marketing, should understand what they will get.

functional and non-functional requirements

We have two types of requirements: **functional** and **non-functional**. A functional requirement says what the system must do, such as compute an FFT. A non-functional requirement can be any number of other attributes: physical size, cost, power consumption, design time, reliability, etc.

A good set of requirements should meet several tests [Dav90]:

- **correctness**: The requirements should not mistakenly describe what the customer wants. Part of correctness is avoiding over-requiring—the requirements should not add conditions that are not really necessary.
- **unambiguousness**: The requirements document should be clear and have only one plain-language interpretation.
- **completeness**: All the requirements should be included.
- **verifiability**: There should be a cost-effective way to ensure that each requirement is satisfied in the final product. For example, a requirement that the system package be "attractive" would be hard to verify without some agreed-upon definition of attractiveness.
- **consistency**: One part of the requirements should not contradict another requirement.
- **modifiability**: The requirements document should be structured so that it can be modified to meet changing requirements without losing consistency, verifiability, etc.
- **traceability**: Each requirement should be traceable in several ways:

  - We should be able to trace backward from the requirements to know why each requirement exists.

  - We should also be able to trace forward from documents created before the requirements (marketing memos, for example) to understand how they relate to the final requirements.

  - We should be able to trace forward to understand how each requirement is satisfied in the implementation.

  - We should also be able to trace backward from the implementation to know which requirements they were intended to satisfy.

How does one determine requirements? If the product is a continuation of a series, then many of the requirements are well-understood. But even in the most modest upgrade, it is valuable to talk to the customer. In a large com-

pany, marketing or sales departments may do most of the work of asking customers what they want, but a surprising number of companies have designers talk directly with customers. Direct customer contact gives the designer an unfiltered sample of what the customer says; it also helps build empathy with the customer, which often pays off in cleaner, easier-to-use customer interfaces. Talking to the customer may also include making surveys, focus groups, or asking selected customers to test out a mock-up or prototype.

9.4 Specifications

In this section we will take a look at some advanced techniques for specification and how they can be used.

9.4.1 Control-Oriented Specification Languages

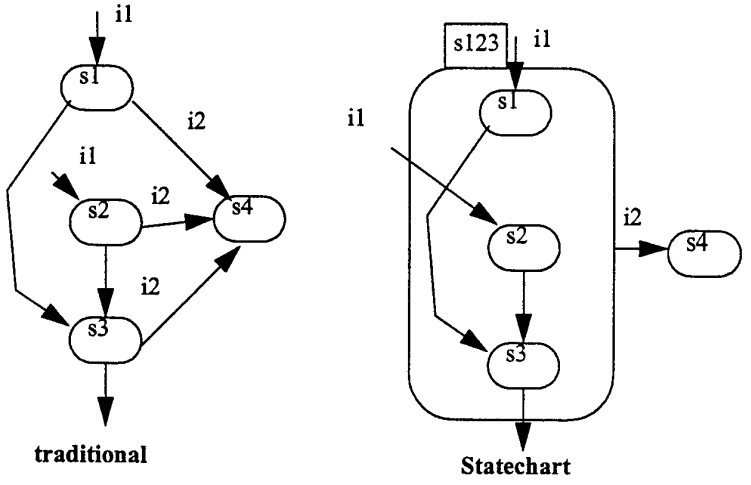


Figure 9-6 An OR state in Statecharts.

SDL

We have already seen how to use state machines to specify control in UML. An example of a widely-used state machine specification language is the SDL language [Roc82], which was developed by the communications industry for specifying communication protocols, telephone systems, etc. As illustrated in Figure 9-7, SDL specifications include states, actions, and both conditional and unconditional transitions between states. SDL is an event-oriented state machine model since transitions between states are caused by internal and external events.

Statecharts

There are other techniques that can be used to eliminate clutter and clarify the important structure of a state-based specification. The Statechart [Har87] is one well-known technique for state-based specification that introduced some important concepts. The Statechart notation uses an event-driven

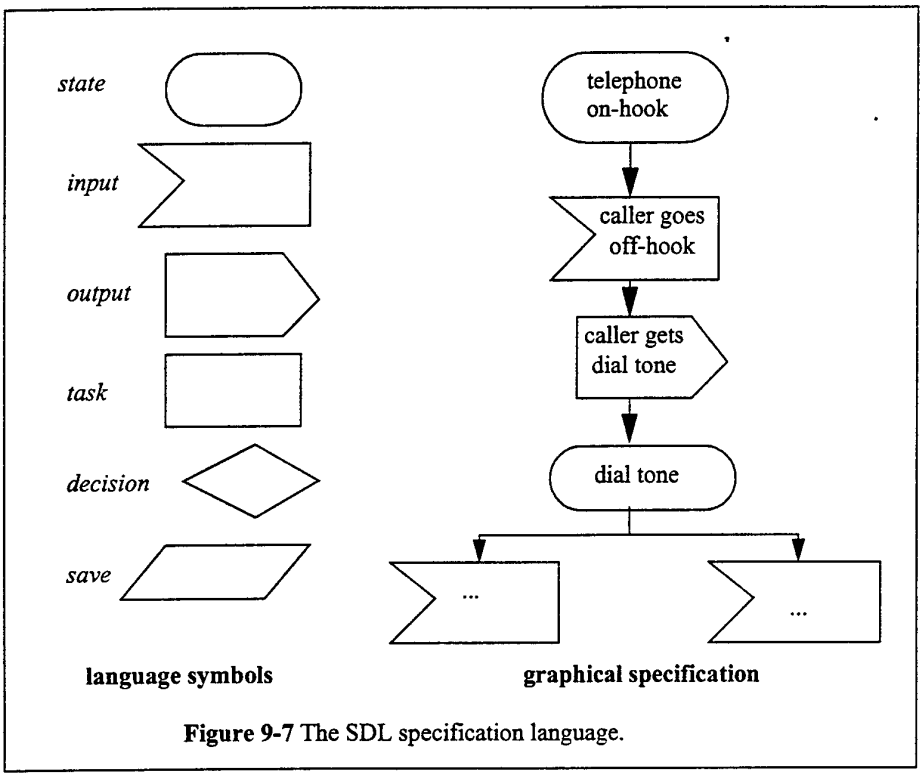


Figure 9-7 The SDL specification language.

model. Statecharts allow states to be grouped together to show common functionality. There are two basic groupings: **OR** and **AND**. Figure 9-6 shows an example of an OR state by comparing a traditional state transition diagram with a Statechart described with an OR state. The state machine specifies that the machine goes to state s4 from any of s1, s2, or s3 when they receive the input i2. The Statechart denotes this commonality by drawing an OR state around s1, s2, and s3 (the name of the OR state is given in the small box at the top of the state). A single transition out of the OR state s123 specifies that the machine goes into state s4 when it receives the i2 input while in any state included in s123. The OR state still allows interesting transition between its member states: there can be multiple ways to get into s123 (via s1 or s2), there can be transitions between states within the OR state (such as from s1 to s3 or s2 to s3). The OR state is simply a tool for specifying some of the transitions relating to these states.

Figure 9-8 shows an example of an AND state specified in Statechart notation as compared to the equivalent in the traditional FSM model. In the traditional model, there are a number of transitions between the states; there is also one entry point into this cluster of states and one exit transition out of the cluster. In the Statechart, the AND state sab is decomposed into two components, sa and sb. When the machine enters the AND state, it simultaneously inhabits the state s1 of component sa and the state s3 of component sb. We can think of the system's state as multidimensional: when it enters sab, knowing the complete state of the machine requires examining both sa

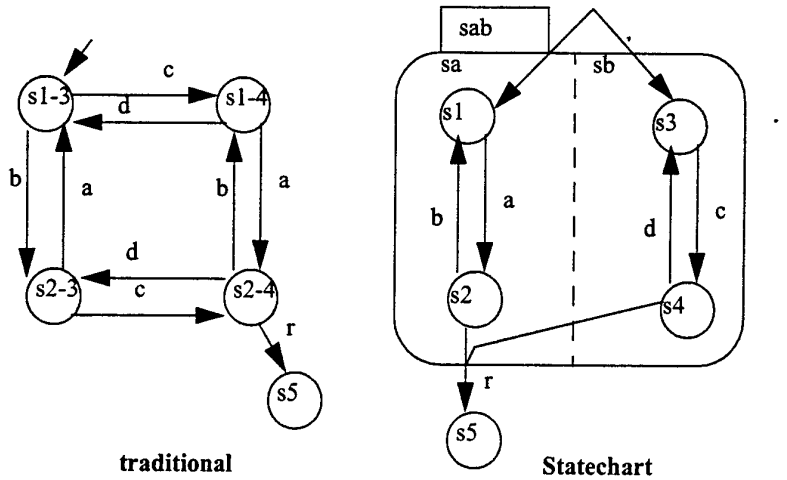


Figure 9-8 An AND state in Statecharts.

and sb. The names of the states in the traditional FSM give their relationship to the AND state components: state s1-3 corresponds to the Statechart machine having its sa component in s1 and its sb component in s3, et cetera. We can exit this cluster of states to go to state s5 only when, in the traditional specification, we are in state s2-4 and receive input r; in the AND state, this corresponds to sa being in state s2 while sb being in state s4 and the machine receiving the r input while in this composite state. Although the traditional and Statechart models describe the same behavior, each component has only two states and the relationships between these states is much simpler to see.

cond1 or (cond2 and !cond3)

expression

OR

AND	**cond1**	**T**	**-**
	cond2	**-**	**T**
	cond3	**-**	**F**

AND/OR table

Figure 9-9 An AND/OR table.

AND/OR tables

Leveson et al [Lev94] used a different format, the AND/OR table, to describe similar relationships between states. An example AND/OR table and the Boolean expression it describes are shown in Figure 9-9. The rows in the AND/OR table are labeled with the basic variables in the equation. Each column corresponds to an AND term in the expression. For example, the AND term (cond2 and not cond3) is represented in the second column with a T for cond2, an F for cond3, and a - (don't-care) for cond1; this corresponds to the fact that cond2 must be T and cond3 F for the AND term to be true. We use the table to evaluate whether a given condition holds in the system: we compare the current states of the variables to the table elements; a column evaluates to true if all the current variable values correspond to the requirements given in the column. If any one of the columns evaluates to true, then the table's expression evaluates to true, as we would expect for an AND/OR expression. The most important difference between this notation and Statecharts is that don't-cares are explicitly represented in the table; this was found to be of great help in identifying problems in a specification table.

9.4.2 Advanced Specifications

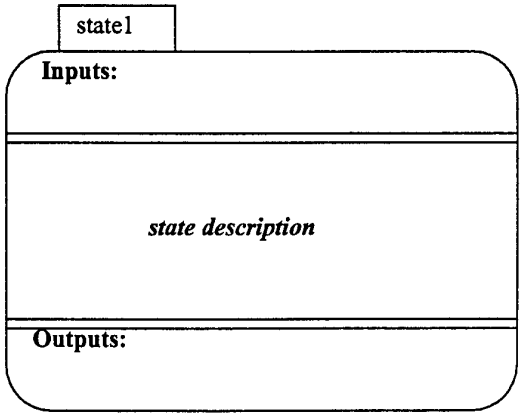
This section is devoted to a single example of a sophisticated system. Application Note 9-2 describes the specification of a real-world, safety-critical system used in aircraft. The specification techniques developed to ensure the correctness and safety of this system can also be used in many applications, particularly in systems where much of the complexity goes into the control structure.

Application Note 9-2 The TCAS II Specification

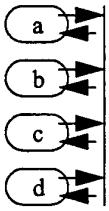
TCAS II (Traffic Alert and Collision Avoidance System) is a collision-avoidance system for aircraft. Based on a variety of information, a TCAS unit in an aircraft keeps track of the position of other nearby aircraft. If TCAS decides that a mid-air collision may be likely, it uses audio commands to suggest evasive action—for example, a pre-recorded voice may warn "DESCEND! DESCEND!" if it believes that an aircraft above poses a threat and that there is room to maneuver below. TCAS makes sophisticated decisions in real-time and is clearly safety-critical. On the one hand, it must detect as many potential collision events as possible (within the limits of its sensors, etc.). On the other hand, it must generate as few false alarms as possible, since the extreme maneuvers it recommends are themselves potentially dangerous.

Leveson et al [Lev94] developed a specification for the TCAS II system. We won't cover the entire specification here, just enough to give a flavor of it. The TCAS II specification was written in their RSML language. They use a

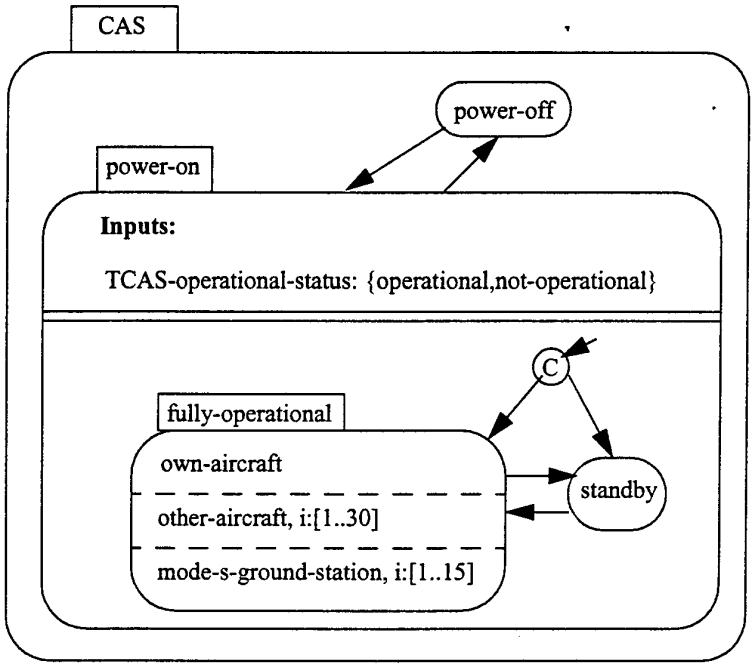
modified version of Statechart notation for specifying states, in which the inputs to and outputs of the state are made explicit:



They also use a transition bus to show sets of states in which there are transitions between all (or almost all) states. In this example, there are transitions from any of a, b, c, or d to any of the other states:

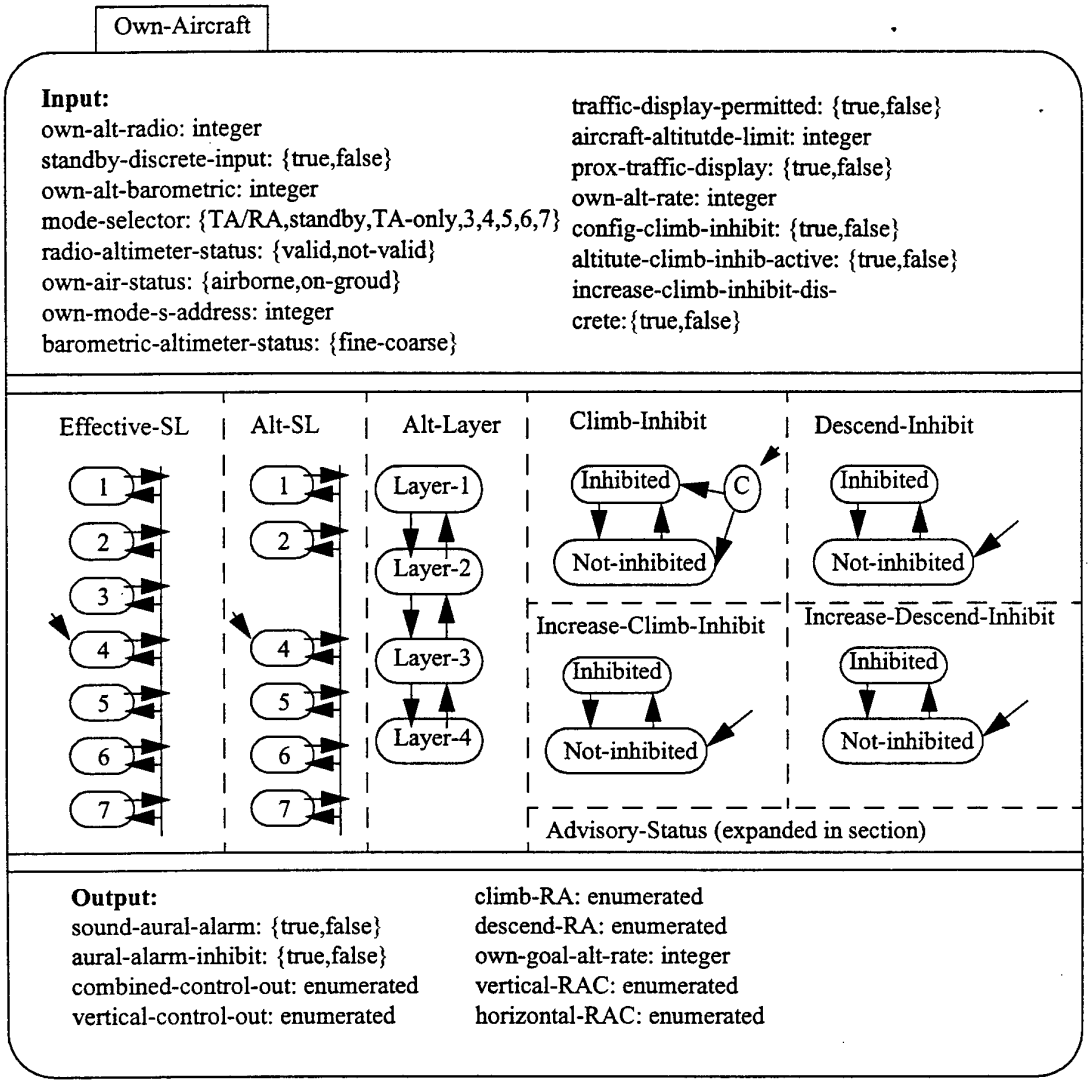


The top-level description of the collision avoidance system (CAS) is:



This specifies that the system has power-off and power-on states. In the power-on state, the system may be in standby mode or fully operational. In the fully-operational mode, three components are operating in parallel, as specified by the AND state: the own-aircraft subsystem; a subsystem to keep track of up to 30 other aircraft; and a subsystem to keep track of up to 15 Mode S ground stations, which provide radar information.

Here is a specification of the own-aircraft AND state:



Once again, the behavior of Own-Aircraft is an AND composition of several subbehaviors. The Effective-SL and Alt-SL states are two ways to control the sensitivity level (SL) of the system, with each state representing a different sensitivity level; differing sensitivities are required depending on distance from the ground and other factors. The Alt-Layer state divides the vertical airspace into layers, with this state keeping track of the current layer. Climb-Inhibit and Descent-Inhibit states are used to selectively inhibit climbs (which may be difficult at high altitudes) or descents (clearly dangerous near the ground), respectively. Similarly, the Increase-Climb-Inhibit and

Increase-Descend-Inhibit states can inhibit high-rate climbs and descents. The Advisory-Status state is rather complicated, so its details are not shown here.

9.5 System Analysis and Architecture Design

In this section we will consider how to turn a specification into an architecture design. We already have a number of techniques for making specific decisions; in this section we will look at how to get a handle on the overall system architecture. The CRC card methodology for system analysis is one very useful method for understanding the overall structure of a complex system.

9.5.1 CRC Cards

The **CRC card** methodology is a well-known and useful way to help analyze a system's structure. It is particularly well-suited to object-oriented design since it encourages the encapsulation of data and functions.

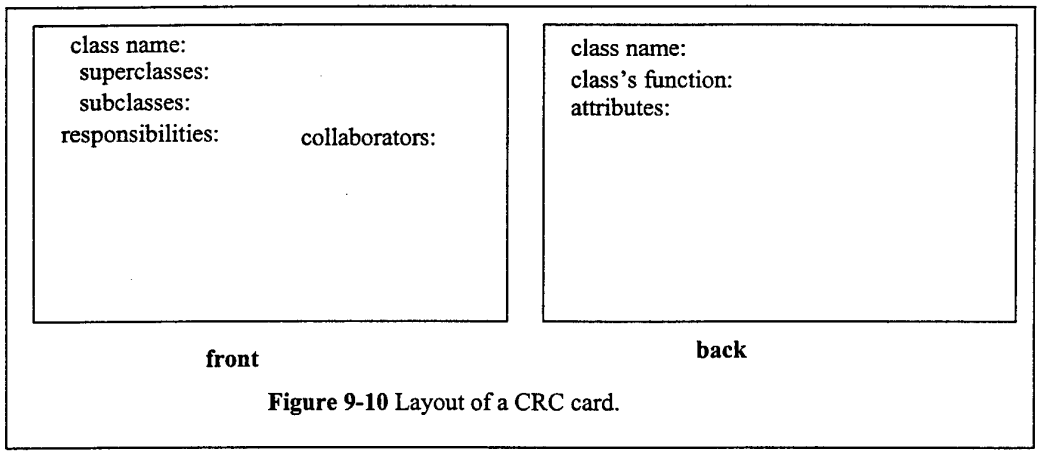


Figure 9-10 Layout of a CRC card.

The acronym *CRC* stands for the three major items that the methodology tries to identify:

- **classes** define the logical groupings of data and functionality;
- **responsibilities** describe what the classes do;
- **collaborators** are the other classes with which a given class works.

The name *CRC card* comes from the fact that the methodology is practiced by having people write on index cards. (In the U. S., the standard size for index cards is 3" X 5", so these cards are often called 3x5 cards.) An example card is shown in Figure 9-10; it has room to write down the class name, its responsibilities and collaborators, and other information. The essence of

the CRC card methodology is to have people write on these cards, talk about them, and update the cards until they are satisfied with their result. This technique may seem like a primitive way to design computer systems. However, it has several important advantages. First, it is easy to get non-computer people to create CRC cards. Getting the advice of domain experts (automobile designers for automotive electronics or human factors experts for PDA design, for example) is very important in system design. The CRC card methodology is informal enough that it will not intimidate non-computer specialists and allow you to capture their input. Second, it aids even computer specialists by encouraging them to work in a group and analyze scenarios. The walkthrough process used with CRC cards is very useful in scoping out a design and determining what about the system is poorly understood. This informal technique is a valuable technique to tool-based design and coding. If you still feel a need to use tools to help you practice the CRC methodology, there are software engineering tools that automate the creation of CRC cards.

Before going through the methodology, let's review the CRC concepts in a little more detail. We are familiar with classes—they encapsulate functionality. A class may represent a real-world object or it may describe an object that has been created solely to help architect the system. A class has both internal state and a functional interface; the functional interface describes the class's capabilities. The responsibility set is an informal way of describing that functional interface. The responsibilities give the class's interface, not its internal implementation. Unlike describing a class in a programming language, however, the responsibilities may be described informally in English (or your favorite language). The collaborators of a class are simply the classes that it talks to: classes that use its capabilities or that it calls upon to help it do its work.

The class terminology is a little misleading when an object-oriented programmer looks at CRC cards. In the methodology, a class is actually used more like an object in an OO programming language—the CRC card class is used to represent a real actor in the system. However, the CRC card class is easily transformable into a class definition in an object-oriented design.

CRC card analysis is performed by a team of people. It is possible to use it by yourself, but a lot of the benefit of the method comes from talking about the developing classes with others. Before starting the process itself, you should create a good number of CRC cards using the basic format shown in Figure 9-10. As you are working in your group, you will be writing on these cards; you will probably discard many of them and rewrite them as the system evolves. The CRC card methodology is informal, but there are several steps you should go through when using it to analyze a system:

1. **Develop an initial list of classes.** Write down the class name and perhaps a few words on what it does. A class may represent a real-world object or an architectural object; identifying which category the class falls into (perhaps by putting a star next to the name of a real-world object) is helpful. Each person can be responsible for handling a part of the system, but team members should talk during this process to be sure that no classes are missed and that duplicate classes aren't created.

2. **Write an initial list of responsibilities and collaborators.** The responsibilities list helps describe in a little more detail what the class does. The collaborators list should be built from obvious relationships between classes. Both the responsibilities and collaborators will be refined in the later stages.
3. **Create some usage scenarios.** These scenarios describe what the system does. Scenarios probably begin with some sort of outside stimulus, which is one important reason for identifying the relevant real-world objects.
4. **Walk through the scenarios.** This is the heart of the methodology. During the walkthrough, each person on the team represents one or more classes. The scenario should be simulated by acting: people can call out what their class is doing, ask other classes to perform operations, etc. Moving around, for example to show the transfer of data, may help you visualize the system's operation. During the walkthrough, all of the information created so far is targeted for updating and refinement: the classes, their responsibilities and collaborators, and the usage scenarios. Classes may be created, destroyed, or modified during this process. You will also probably find many holes in the scenario itself.
5. **Refine the classes, responsibilities, and collaborators.** Some of this will be done during the course of the walkthrough, but making a second pass after the scenarios is a good idea. The longer perspective will help you make more global changes to the CRC cards.
6. **Add class relationships.** Once the CRC cards have been refined, subclass and superclass relationships should become clearer and can be added to the cards.

Once you have the CRC cards, you need to somehow use them to help drive the implementation. In some cases, it may work best to use the CRC cards as direct source material for the implementors; this is particularly true if you can get the designers involved in the CRC card process. In other cases, you may want to write a more formal description, in UML or another language, of the information that was captured during the CRC card analysis, then use that formal description as the design document for the system implementors.

Example 9-2 illustrates the use of the CRC card methodology.

Example 9-2 CRC card analysis

Let's perform a CRC card analysis of the elevator system of Section 8.6. First, we need a basic set of classes:

- **real-world classes**: elevator car, passenger, floor control, car control, car sensor.
- **architectural classes**: car state, floor control reader, car control reader, car control sender, scheduler.

For each class, we need an initial set of responsibilities and collaborators (we use an asterisk to remind ourselves which classes represent real-world objects):

class	responsibilities	collaborators
elevator car*	**moves up and down**	**car control, car sensor, car control sender**
passenger*	**pushes floor control and car control buttons**	**floor control, car control**
floor control*	**transmits floor requests**	**passenger, floor control reader**
car control*	**transmits car requests**	**passenger, car control reader**
car sensor*	**senses car position**	**scheduler**
car state	**records current position of car**	**scheduler, car sensor**
floor control reader	**interface between floor control and rest of system**	**floor control, scheduler**
car control reader	**interface between car control and rest of system**	**car control, scheduler**
car control sender	**interface between scheduler and car**	**scheduler, elevator car**
scheduler	**sends commands to cars based upon requests**	**floor control reader, car control reader, car control sender, car state**

Several usage scenarios define the basic operation of the elevator system as well as some unusual scenarios:

1. One passenger requests a car on a floor, gets in the car when it arrives, requests another floor, gets out when the car reaches that floor.
2. One passenger requests a car on a floor, gets in the car when it arrives, requests the floor that the car is currently on.
3. A second passenger requests a car while another passenger is riding in the elevator.
4. Two people push floor buttons on different floors at the same time.
5. Two people push car control buttons in different cars at the same time.

At this point, we need to walk through the scenarios and make sure they are reasonable. Find a set of people and walk through these scenarios. Do the classes, responsibilities, collaborators, and scenarios make sense? How would you modify them to improve the system specification?

9.6 Quality Assurance

The quality of a product or service can be judged by how well it satisfies its intended function. A product can be of low quality for several reasons: it was

shoddily manufactured; its components were improperly designed; its architecture was poorly conceived; the product's requirements were poorly understood. Quality must be designed in. You can't test out enough bugs to deliver a high-quality product. The **quality assurance** (**QA**) process is vital for the delivery of a satisfactory system. In this section we will concentrate on portions of the methodology particularly aimed at improving the quality of the resulting system.

The software testing techniques we described earlier in the book were one component of quality assurance, but the pursuit of quality extends throughout the design flow. For example, settling on the proper requirements and specification cannot be overlooked as an important determinant of quality. If the system is too hard to design, it will probably be difficult to keep it working properly. Customers may desire features that sound nice but in fact don't add much to the overall usefulness of the system; in many cases, having too many features only makes the design more complicated and the final device more prone to breakage.

In the next sections, we will review the quality assurance process in more detail. Section 9.6.1 introduces the quality assurance process; Section 9.6.2 talks about verifying requirements and specifications; Section 9.6.3 discusses design reviews; and Section 9.6.4 talks about measurement-driven quality assurance.

To help us understand the importance of quality assurance, Application Note 9-3 describes serious safety problems in one computer-controlled medical system. Medical equipment, like aviation electronics, is a safety-critical application; unfortunately, this medical equipment caused deaths before its design errors were properly understood. This example also allows us to use specification techniques to understand software design problems. In the rest of the section, we'll look at several ways of improving quality: design reviews; measurement-based quality assurance; and techniques for debugging large systems.

Application Note 9-3 The Therac-25 Medical Imaging System

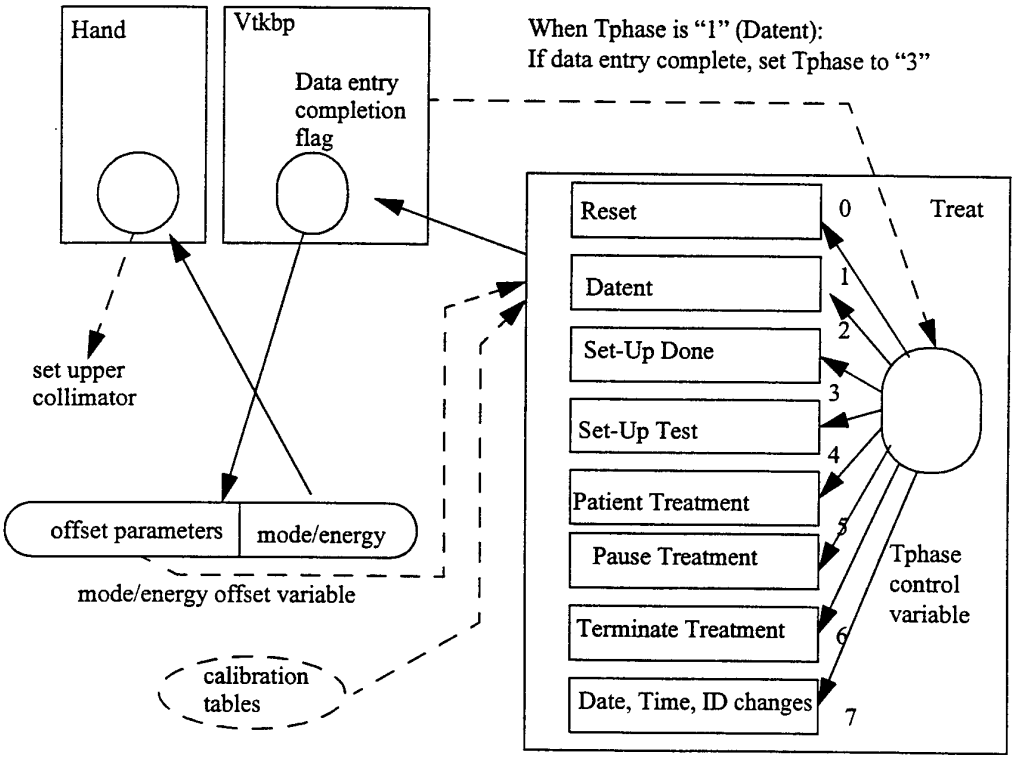
The Therac-25 medical imaging system caused what Leveson and Turner called "the most serious computer-related accidents to date (at least nonmilitary and admitted)" [Lev93]. In the course of six known accidents, these machines delivered massive radiation overdoses, causing deaths and serious injuries. Leveson and Turner analyzed the Therac-25 system and the causes for these accidents.

The Therac-25 was controlled by a PDP-11 minicomputer. The computer was responsible for controlling a radiation gun that delivered a dose of radiation to the patient. It also runs a terminal which presents the main user interface. The machine's software was developed by a single programmer in PDP-11 assembly language over several years. The software includes four major components: stored data, a scheduler, a set of tasks, and interrupt services. There are three major critical tasks in the system:

- A treatment monitor controls and monitors the setup and delivery of the treatment in eight phases.
- A servo task controls the radiation gun, machine motions, etc.
- A housekeeper task takes care of system-status interlocks and limit checks. (A limit check determines whether some system parameter has gone beyond pre-set limits.)

The code was relatively crude: the software allowed several processes access to shared memory, there is no synchronization mechanism aside from shared variables, and test and set for shared variables were not indivisible operations.

Let us examine the software problems responsible for one series of accidents. Leveson and Turner reverse-engineered a specification for the relevant software:



Treat is the treatment monitor task; it was divided into eight subroutines (Reset, Datent, etc.). Tphase is a variable which controls which of these subroutines is currently executing. Treat reschedules itself after the execution of each subroutine. The Datent subroutine communicates with the keyboard entry task via the data entry completion flag, which is a shared variable. Datent looks at this flag to determine when it should leave the Data Entry mode and go to the Set-Up Test mode. The mode/energy offset variable is a

shared variable: the top byte holds offset parameters that is used by the Datent subroutine; the low-order byte holds mode and energy offset used by the Hand task.

When the machine is run, the operator is forced to enter the mode and energy (there is one mode in which the energy is set to a default), but the operator can later edit the mode and energy separately. The software's behavior is timing-dependent: if the keyboard handler sets the completion variable before the operator changes the mode/energy data, the Datent task will not detect the change—once Treat leaves Datent, it will not enter that subroutine again during the treatment. However, the Hand task, which runs concurrently, will see the new mode/energy information. Apparently, the software included no checks to detect the incompatible data.

After the mode/energy data is set, the software sends parameters to a digital/analog converter, then calls the Magnet subroutine to set the bending magnets. Setting the magnets takes about eight seconds and a subroutine called Ptime is used to introduce a time delay. Due to the way that Datent, Magnet, and Ptime are written, it is possible that changes to the parameters made by the users can be shown on the screen but will not be sensed by Datent. One accident occurred when the operator initially entered mode/energy, went to the command line, changed mode/energy, and returned to the command line within 8 seconds. The error therefore depended on the typing speed of the operator; since operators become faster and more skillful with the machine over time, this error is more likely to occur with experienced operators.

Leveson and Turner emphasize that at the root of the particular bugs that led to these accident were poor design methodologies and flawed architectures:

- They performed a very limited safety analysis. For example, they assigned low probabilities to certain errors with no apparent justification.
- They did not use mechanical backups to check the operation of the machine (such as testing beam energy), even though they had done so in earlier models of the machine.
- They used overly complex programs based on unreliable coding styles.

They relied on system testing with insufficient module testing or formal analysis.

9.6.1 Quality Assurance Techniques

The International Standards Organization (ISO) has created a set of standards known as **ISO 9000** for quality. ISO 9000 was created to apply to a broad range of industries, including but not limited to embedded hardware and software. A standard developed for a particular product, such as wooden construction beams, could specify criteria particular to that product, such as the load that a beam must be able to carry. However, a wide-ranging standard like ISO 9000 cannot specify the detailed standards for every industry. As such, ISO 9000 concentrates on processes used to create the product or

service. The processes used to satisfy ISO 9000 affect the entire organization as well as the individual steps taken during design and manufacturing.

A detailed description of ISO 9000 is beyond the scope of this book; several books [Sch94, Jen95] describe ISO 9000's applicability to software development. We can, however, make several observations about quality management based on ISO 9000:

- **Process is crucial.** Haphazard development leads to haphazard products and low quality. Knowing what steps are to be followed to create a high-quality product is essential to ensuring that all the necessary steps are in fact followed.
- **Documentation is important.** Documentation performs several roles: the creation of the documents describing processes helps those involved understand the processes; documentation helps internal quality monitoring groups to ensure that the required processes are actually being followed; documentation also helps outside groups (customers, auditors, etc.) understand the processes and how they are being implemented.
- **Communication is important.** Quality ultimately relies on people. Good documentation is an aid for helping people understand the total quality process. The people in the organization should understand not only their specific tasks, but also how their jobs can affect the overall system quality.

design verification strategies

There are many types of techniques that can be used to verify system designs and ensure quality. Techniques can be either *manual* or *tool-based.* Manual techniques are surprisingly effective in practice. In Section 9.6.3 we will discuss design reviews, which are simply meetings at which the design is discussed and which are very successful in identifying bugs. Many of the software testing techniques described in Section 5.9 can be applied manually by tracing through the program to determine the required tests. Tool-based verification helps considerably in managing large quantities of information that may be generated in a complex design. Test generation programs can automate much of the drudgery of creating test sets for programs. Tracking tools can help ensure that various steps have been performed. Design flow tools automate the process of running design data through other tools.

metrics and quality

Metrics are important to the quality control process. To know whether we have achieved high levels of quality we must be able to measure aspects of the system and our design process. We may measure certain aspects of the system itself, such as the execution speed of programs or the coverage of test patterns. We may also measure aspects of the design process, such as the rate at which bugs are found. Section 9.6.4 describes ways in which measurements can be used in the quality assurance process.

Tool and manual techniques must fit into an overall process. The details of that process will be determined by several factors: the type of product being designed (video game? laser printer? air traffic control system?); the number of units to be manufactured and the time allowed for design; the existing practices in the company into which any new processes must be integrated; and many other factors. An important role of ISO 9000 is to help organiza-

tions look at their total process, not just particular segments that may appear to be important at one particular time.

One well-known way of measuring the quality of an organization's software development process is the **Capability Maturity Model (CMM)®**[1] developed by Carnegie Mellon University's Software Engineering Institute [SEI99]. The CMM provides a model for judging an organization. It defines five levels of maturity:

1. **Initial.** A poorly organized process, with very few well-defined process. Success of a project depends on the efforts of individuals, not the organization itself.
2. **Repeatable.** This level provides basic tracking mechanisms that allow management to understand cost, scheduling, and how well the systems under development meet their goals.
3. **Defined.** The management and engineering processes are documented and standardized. All projects make use of the documented and approved standard methods.
4. **Managed.** This phase makes detailed measurements of the development process and of product quality.
5. **Optimizing.** At the highest level, feedback from detailed measurements is used to continually improve the organization's processes.

The Software Engineering Institute has found very few organizations anywhere in the world that meet the highest level of continuous improvement and quite a few organizations that have only the chaotic processes of the first, initial level. However, the Capability Maturity Model provides a benchmark by which organizations can judge themselves and use that information for improvement.

1. Capability Maturity Model and CMM are registered trademarks of Carnegie Mellon University Software Engineering Institute.

9.6.2 Verifying the Specification

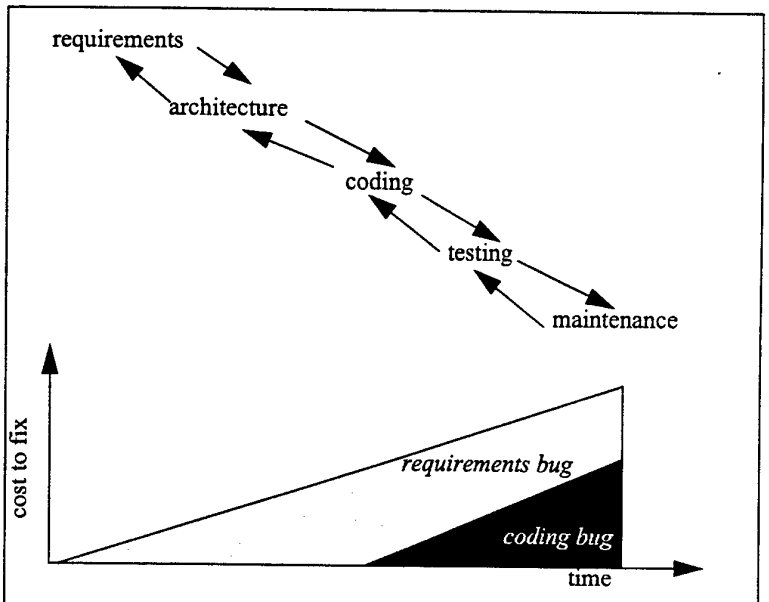


Figure 9-11 Long-lived bugs are more expensive to fix.

The requirements and specification are generated very early in the design process. Verifying the requirements and specification is very important for the simple reason that bugs in the requirements or specification can be extremely expensive to fix. Figure 9-11 shows how the cost of fixing bugs grows over the course of the design process (we use the waterfall model as a simple example, but the same holds for any design flow). The longer a bug survives in the system, the more expensive it will be to fix. A coding bug, if not found until after system deployment, will cost money to recall and reprogram existing systems, among other things. But a bug introduced earlier in the flow and not discovered until the same point will accrue all those costs and more costs as well. A bug introduced in the requirements or specification and left until maintenance could force an entire redesign of the product, not just the replacement of a ROM. Discovering bugs early is crucial: it prevents bugs from being released to customers; it minimizes design costs; it reduces design time. While some requirements and specification bugs will become apparent in the detailed design stages, for example as the consequences of certain requirements are better understood, it is possible and desirable to weed out many bugs even during the generation of the requirements and spec.

The goal of validating the requirements and specification is to ensure that they satisfy the criteria we originally applied in Section 9.3 to creating the specification: correctness, completeness, consistency, etc. Validation is in fact part of the effort of generating the requirements and specification. Some techniques can be applied while they are being created to help you under-

stand the requirements and specifications; others are applied on a draft, with results used to modify the specs.

requirements validation

Since requirements come from the customer and are inherently somewhat informal, it may seem like a challenge to validate them. However, there are many things that can be done to ensure that the customer and the person actually writing the requirements are communicating. **Prototypes** are a very useful tool when dealing with end users—rather than simply describe the system to them in broad, technical terms, a prototype can let them see, hear, and touch at least some of the important aspects of the system. Of course, the prototype will not be fully functional since the design work has not yet been done. However, users interfaces in particular are well-suited to prototyping and user testing. Canned or randomly generated data can be used to simulate the internal operation of the system. A prototype can help the end user critique a number of functional and non-functional requirements: data displays, speed of operation, size, weight, etc. Some programming languages, sometimes called **prototyping languages** or **specification languages**, are especially well-suited to prototyping: very high-level languages (such as Matlab in the signal processing domain) may be able to perform functional attributes, such as the mathematical function to be performed, but not non-functional attributes such as the speed of execution. **Pre-existing systems** can also be used to help the end user articulate his or her needs. Specifying what someone does or doesn't like about an existing machine is much easier than having them talk about the new system in the abstract. In some cases, it may be possible to construct a prototype of the new system from the pre-existing system.

specification validation

The techniques used to validate requirements are also useful in verifying that the specifications are correct. Building prototypes, specification languages, and comparisons to pre-existing systems are as useful to system analysis and designers as they are to end users. Auditing tools may be useful in verifying consistency, completeness, etc. Working through **usage scenarios** often helps designers fill out the details of a specification and ensure its completeness and correctness. In some cases, **formal techniques** (that is, design techniques that make use of mathematical proofs) may be useful. Proofs may be done either manually or automatically: in some cases, proving that a particular condition can or cannot occur according to the specification is important. Automated proofs are particularly useful in certain types of complex systems which can be specified succinctly but whose behavior over time is complex. For example, complex protocols have been successfully formally verified.

9.6.3 Design Reviews

The **design review** [Fag76] is a critical component of any quality assurance process. The design review is simple, low-cost way to catch bugs early in the design process. A design review is simply a meeting in which team members discuss a design, reviewing how a component of the system works. Some bugs are caught simply by preparing for the meeting, as the designer is forced to think through the design more thoroughly. Other bugs are caught by people attending the meeting, who will notice problems that may not be

caught by the unit's designer. By catching bugs early and not allowing them to propagate into the implementation, we reduce the time required to get a working system. We can also use the design process to improve the quality of the implementation and make future changes easier to implement.

design review format

A design review is held to review a particular component of the system. A design review team has several members:

- The **designers** of the component being reviewed are, of course, central to the design process. They will present their design to the rest of the team for review and analysis.
- The **review leader** coordinates the pre-meeting activities, the design review itself, and the post-meeting follow-up.
- The **review scribe** records the minutes of the meeting so that designers and others know what problems need to be fixed.
- The **review audience** studies the component. Some audience members will naturally include other members of the project for which this component is being designed. Audience members from other projects often add valuable perspective and may notice problems that team members have missed.

before the design review

The design review process begins before the meeting itself. The design team prepares a set of documents (code listings, flow charts, specifications, etc.) that will be used to describe the component. These documents are distributed to other members of the review team in advance of the meeting, so that all have time to familiarize themselves with the material. The review leader coordinates the meeting time, distribution of handouts, etc.

design review meeting

During the meeting, the leader is responsible for making sure the meeting runs smoothly, while the scribe takes notes about what happens. The designers are responsible for presenting their component design. A top-down presentation often works well: start with the requirements and interface description, then move to the overall structure of the component, then to the details, and finally the testing strategy. The audience should look for all sorts of problems at every level of detail:

- Is the design team's view of the component's specification consistent with the overall system specification, or have they misinterpreted something?
- Is the interface specification correct?
- Does the component's internal architecture work well?
- Are there coding errors within the component?
- Is the testing strategy adequate?

following up the design review

The notes taken by the scribe are used to follow-up the meeting. The design team should go back and correct any bugs and address any concerns raised at the meeting. While doing so, they should keep notes describing what they did. The design review leader coordinates with the design team, both to make sure that the changes are made and to distribute the change results to the audience. If the changes are straightforward, a written report of them is probably adequate. If the errors found during the review caused a major rework of the component, a new design review meeting for the new imple-

mentation, using as many of the original team members as possible, may be called for.

Example 9-3 describes the use of modern software tools for paperless software design reviews.

Example 9-3 Paperless code inspections

A paperless code inspection methodology was set up within AT&T Bell Laboratories for large software projects [Hsu94]. The inspection process used several off-the-shelf tools:

- The Procase SMARTsystem (SMARTsystem is a registered trademark of Procase Corporation) provided facilities for analysis, reverse engineering, and management of C code.
- FrameMaker (a registered trademark of Adobe Systems) was used for document preparation.
- nmake was an updated version of the traditional Unix make facility for controlling software builds.
- Sablime® was used to control versions of source code.
- The cimgr program was used to annotate documents.

Participants in the code reviews used tools at all points during the process. The moderator could use an on-line calendar to schedule the meeting. The designer used cimgr and the SMARTsystem to store the code, using nmake to ensure that the proper versions were saved. Cimgr also allowed the designer to link the source code to Frame documents to provide background documentation. Reviewers used cimgr to find the appropriate set of code and documents for the review and used the other tools to help them review the code; they could also use the annotation tools to provide notes before the meeting. During the meeting, all the participants used cimgr to manage documents and provide annotations, with the moderator eliminating redundant annotations. After the meeting, the designer uses the tools to make the changes. Cimgr can be used to collect statistics on the meeting, such as the amount of code reviewed, preparation time per inspector, etc.

9.6.4 Measurement-Driven Quality Assurance

The design review is an inspection-based system—we couldn't fully automate the design review because we don't have formal criteria for finding all types of bugs. The inspection process is important because there will always be creative ways to introduce bugs that require human analysis. However, we can make measurements of bug rates in our system and use those statistics to help us. Not only do bug statistics help us determine the proper amount of testing for a system, we can also use the knowledge gained in one project to predict where bugs are likely to occur in later projects.

functional design and misunderstandings	15%
logic design and misunderstandings	20%
coding problems	30%
documentation and others (not shown)	35%

percentage defects by source

	efficiency % against defects in function	logic	% coding	incorrect repairs
functional specification review	50	-	-	+1
logic specification review	40	50	-	+2
module specification review	60	70	-	+2
module code inspection	65	75	70	+3
unit test	10	10	25	+4
function test	20	25	55	+5
component test	15	20	65	+5
subsystem test	15	15	55	+7
system test	10	10	40	+10
cumulative efficiency	98	98	99	
net cumulative efficiency		98		

defect removal efficiency by source

Figure 9-12 Early statistics on software development [Jon78].

Measurements of the design process are important for the same reason that measurements help us in all sorts of other activities—they provide accurate information which leads into insight. Early studies of software development gave important insight into the software development process and those basic lessons are still relevant to today. One study of software projects at IBM [End75] found that 46% of the errors found in its sample set of software were rooted in misunderstanding of the problem, communication, knowledge of problem-solving techniques, etc., while only 38% were amenable to better tools and other technological fixes. Some of the results of another study at IBM [Jon78] are shown in Figure 9-12. These tables show both the original source of the errors and the steps in the development process where they were found (defect removal efficiency is simply the ratio of

defects found before release to defects found before and after release). These results show that many errors were due to high-level problems like poor functional design and bad documentation. They also show that defect removal procedures added a small number of new bugs by incorrectly fixing the problems found. On the positive side, they also show that design reviews caught many errors and that multiple levels of review caught new bugs.

The Hitachi Software Quality Evaluation (SQE) system [Cus91] is one early example of a successful measurement-driven quality assurance program. The SQE methodology kept track of customer-identified defects and, together with information on bugs found during in-house data, not only predicted the number of bugs to be expected in similar products and identified tests to find these bugs. The Hitachi effort measured relatively simple properties of programs: program size and structure, amount of code written in high-level languages, number of design and test work hours, as well as some other relevant parameters.

The purpose of measuring your design process is not to eliminate bugs totally—as Figure 9-12 illustrates, even bug removal processes can introduce new bugs. Rather, measurement-driven quality assurance is essential to continuous process improvement. While we cannot expect perfection, we can always improve on what we are doing; measuring design processes and using those results to improve processes allow us to continuously improve how we work and what we do.

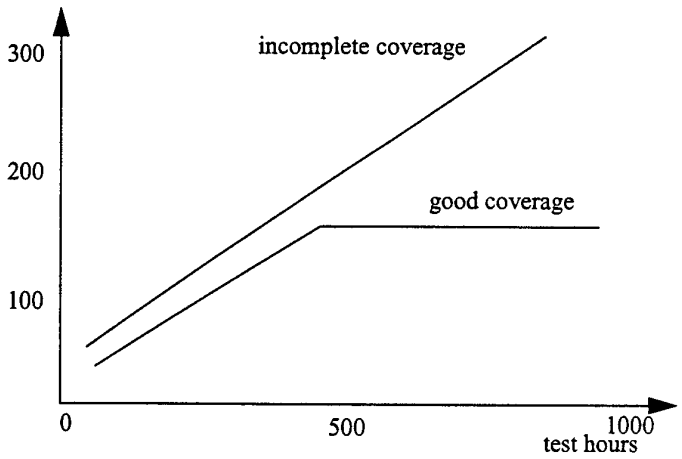
Application Note 9-4 describes the quality assurance process for microprocessor-controlled instruments.

Application Note 9-4 Software Quality Assurance for Instruments

Asada and Yan [Asa98] describe improvements they made to the software quality assurance process used at the Hewlett-Packard Kobe Instrument Division. The process they used through the early 1990's separated unit and integration test from system test: the R&D group was responsible for unit and integration testing; they handed over the complete product to a software quality assurance group for system test. The system test phase happened just before product shipment.

But as time-to-market pressures reduced the amount of development time available while simultaneously requiring products to be extended with new features, they noticed that more single-function defects made it past unit and integration testing into system testing phase. As a result, system testing failed to see stable defect reporting rates. Defects as a function of time

should stabilize for a well-tested product but will continue to climb when testing is incomplete:



They found that, for a product designed in 1990, 17.8% of the defects found in system test were single-function defects, but that number climbed to 48.0% for a product designed in 1996. Because so many single-function defects were making it into system testing, it became more difficult to uncover multiple-function defects. As a result, the cost-per-function of an enhancement rose from 61.3 hours/function in 1991 to 132 in 1995.

To find bugs earlier and reduce total testing costs, they changed their development process. First, they brought software Q&A into the unit and integration process, rather than having them wait until system testing. Bringing in Q&A early helped them integrate design and test to catch problems earlier. They captured the definition of instrument functions early in tables like this:

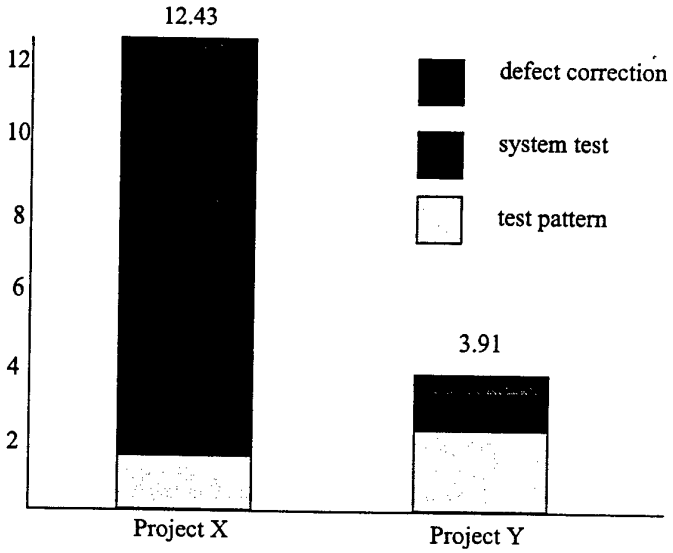
command	range	initial value	description
POIN <val>	**2 to 801 (int)**	**201 (NA,ZA), 801 (SA)**	**In SWPT != List, set MEAS point of sweep.**
SPAN <val> (Hz)	**0 to 510 M(Hz)**	**500 MHz**	**In SWPT != LIST, set SPAN value of sweep parameter.**

command	range	initial value	description
STAR <val> (Hz)	**0 to max (Hz)**	**0 Hz**	**In SWPT != LIST, set START value of sweep parameter. Min and max value of START, STOP, CENTER depend on RBW.**
Stop <val> (Hz)	**min to 510 M(Hz)**	**500 MHz**	**In SWPT != LIST, set STOP value of sweep parameter. Min and max value of START, STOP, CENTER depend on RBW.**

The function descriptions help in the design phase by allowing incomplete function definitions, conflicts between functions, and other problems to be caught early, before code is written. The descriptions help later by driving an automatic test generation process. Engineers manually write scripts, based on the function descriptions, to automatically generate unit and integration tests. They use equivalence partitioning and boundary value analysis to design test scripts for single-function defects. The function definitions guide the design of tests for multiple-function defects. The test scripts allow tests to be run immediately after new code for a function is written.

The new methodology helped reduce the costs of product design. The authors compared a Project Y to the Project X product of which it is an extension. Project X's original design required 1049 testing hours to find 309 defects. Project Y's tests naturally covered Project X code as well as changes made for Project Y; they found 88 defects in Project X code in 200 hours of testing for Product Y. These 88 defects had remained in Project X even though they had spent over 1000 hours testing the original product. Engineer-months per defect went from 0.0402 for Project X to 0.0273 in

Project Y. Testing costs went down significantly for Project Y, even though they spent more time on test pattern generation:

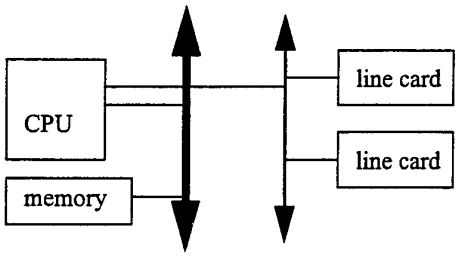


9.7 Example: Telephone PBX

A telephone **private branch exchange (PBX)** presents an interesting array of problems to the embedded system designer: high data rates, sophisticated functionality, low cost, and high reliability. We will use TigerSwitch, a PC-based PBX designed at Princeton University [Wol96] as the basis for our discussion; the basic principles of TigerSwitch apply to non-PC-based PBXs, such as the ROLM PBX of Application Note 9-5, and some commercial PBXs are in fact built on standard PCs. After describing the basic principles of operation of a PBX, we will describe the system architecture.

Application Note 9-5 The ROLM PBX

The ROLM PBX was based on a general-purpose computer. It uses two busses:



The CPU has one bus for data and instruction fetching. The telephone interfaces (called line cards) were attached to a separate bus. Using two busses kept the CPU operations from interfering with telephone data transfers.

The PBX used data transfers between line cards to make phone calls. At each sampling interval, the CPU would transfer voice data between the phone lines involved in the call. A table in memory told the CPU which phone lines were connected by calls.

9.7.1 Theory of Operation

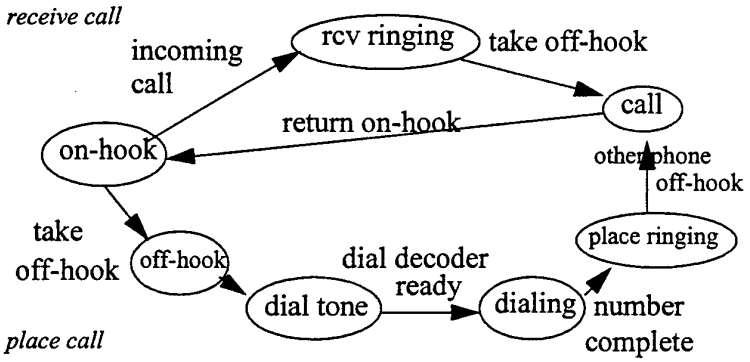


Figure 9-13 States of a telephone line for making and receiving calls.

telephone terminology

The first problem is to understand the procedure for completing calls. Each distinct telephone connection is called a **line**, so a call is completed by connecting two lines. We briefly described telephone calls when we designed the answering machine of Section 6.10—remember that a telephone can be on-hook or off-hook and that telephone voice data is sampled at an 8 kHz rate.

When making a call, actions are required on each of the two lines to be connected—one must initiate the call by dialing a number, while the other side must pick up the phone. We can describe the operations on each of the lines using a single state transition graph that covers placing and receiving calls. The state transition graph for a single phone line, showing both placing and receiving calls, is shown in Figure 9-13. To understand the state transition diagram, assume that the phone line starts on-hook. When a call is placed to this line, the phone starts to ring. If the phone is taken off-hook while the line is ringing, the call is completed. Once the phone is put back on-hook, the call is terminated and the line goes back to its initial state. To place a call, the user takes the phone off-hook. At this point, the user must wait until a dialing decoder is ready before a dial tone is sent over the line; we will see that the dialing decoder consumes significant resources and not all telephones can be dialed simultaneously. Once the dial tone is received, the dialing procedure can begin. Once the PBX recognizes that it has received a complete call, it goes into its own ringing state and waits for an answer. The rcv ringing and place ringing states are distinct even though they both denote to a line ringing: in the place ringing state, another line is ringing, while in the rcv ringing state this line is ringing.

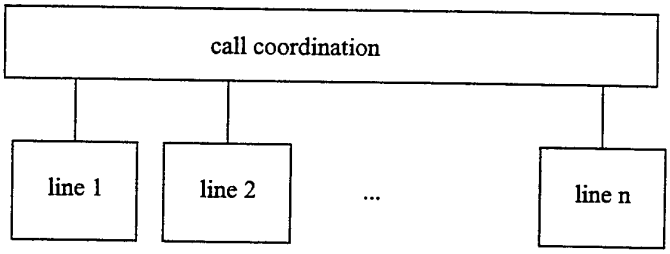


Figure 9-14 The PBX as a set of coordinated lines.

The behavior of the entire PBX can be understood as a set of loosely coupled state machines, one for each line, as illustrated in Figure 9-14. Each line is represented by a state machine. An additional state machine takes care of the coordination and interaction between lines—for example, when a receiving line goes off-hook, the placing line must be put in the call state. This specification style naturally captures the parallelism in a phone system—calls are independent and do not interact.

One of the important steps in placing a call is routing the call based on the telephone number. If the telephone call is to another line within the PBX, the call is sent there directly. If the call is outside the PBX, then it must be routed through an external line (called a **trunk line**) to another telephone switching system; this requires that the PBX set up a connection with the external switch.

We will use **dual-tone mutli-frequency (DTMF)** dialing rather than pulse dialing. As illustrated in Figure 9-15, each key on a DTMF dialing pad gen-

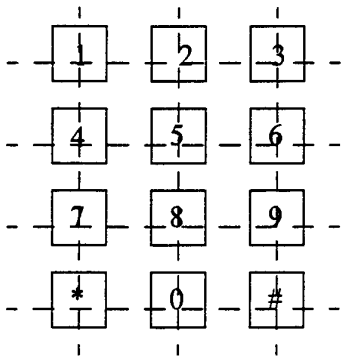


Figure 9-15 Frequencies assigned to DTMF digits.

erates two tones when depressed, one to designate its row and another for its column. The PBX senses digits dialed by looking for pairs of frequencies. The two tones must be simultaneously present for at least 0.1 sec for a digit to be registered.

The PBX must also collect billing information for all calls. This requires keeping track of the destination number and duration of each call. The length of a call must be determined accurately to create accurate billing statements; some telephone companies use their finer time measurement granularity in their marketing, but that finer-grained measurement of call times adds to the CPU load on the PBX.

Finally, the PBX must support some standard maintenance operations, such as adding new telephone lines and their associated phone numbers, updating call routing information, and testing the PBX.

9.7.2 System Architecture

The basic hardware architecture of PBX is shown in Figure 9-16. Each telephone line is an I/O device. The interface logic between the telephone and the PBX is known as a **line card**; it contains A/D and D/A converters, circuitry for the ringing signal, etc. The CPU copies data between the line cards to make telephone calls—at each sampling interval, the CPU reads the A/D converter from one line and sends it to the other line's D/A converter, then performs the copy in the opposite direction. (The standard PC timer cannot generate interrupts at an 8 kHz rate, so we built a special 8 kHz interrupt timer as an I/O card for TigerSwitch.)

Before considering other operations such as dialing, let's determine the architecture of the basic call mechanism. For each call that is placed, we must perform two reads (from each line's A/D) and two writes (to each line's D/A) at the 8 kHz sampling rate. The specification of Figure 9-14 would suggest that we implement each line as a separate process. However,

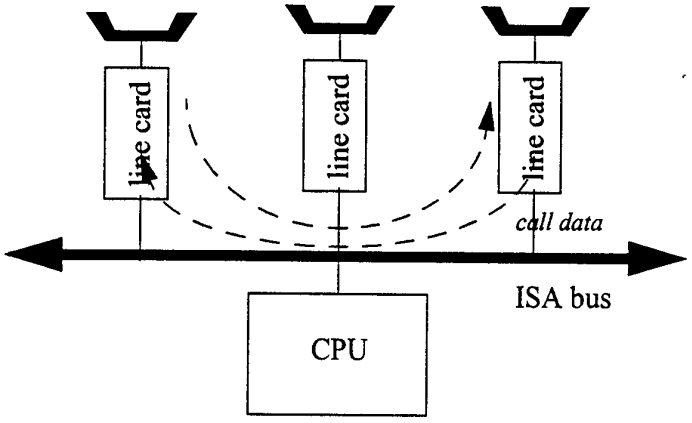


Figure 9-16 Basic hardware architecture of TigerSwitch.

that would lead to disastrous performance consequences during calls: each of the two lines involved in a call would have to be activated at the 8 kHz sampling rate. Even a small number of calls would introduce so much context switching overhead as to overload a typical PC. An alternative is to implement one call handling process which performs the data transfers for all active calls. We build a table which shows the connections between lines. At each sampling interval, we loop through the active calls and perform the necessary transfers:

```
for (i=0; i< n_calls; i++) {
    /* transfer data from line 1 to line 2 */
    data = read_line(call[i].line1);
    write_line(call[i].line2,data);
    /* transfer data from line 2 to line 1 */
    data = read_line(call[i].line2);
    write_line(call[i].line1,data);
}
```

This code is much more efficient than using separate processes for the lines. This loop structure can be understood as a very simple implementation of the processes. An operating system would keep a table of the current state of all processes; the call[] table in our loop corresponds to that process state table. However, since the loop deals with only a small amount of data, it need not save and restore the complete system state between iterations.

Call initiation and completion can be handled by a lower priority process that runs much slower than the sampling rate. For instance, when a line goes off-hook, this process can sense the off-hook state by polling the line cards, then initiate the next action based on the current state of the line.

Sensing the DTMF signals during dialing can be expensive. Since dialing takes a relatively small fraction of the total time a phone line is in use, most switching systems provide only a few DTMF detectors: when a line goes

off-hook, it will not get a dial tone until one of the detectors is available. The tones can be sensed using a bank of filters, one for each frequency. We could use an analog filter bank on each line card, but that would be expensive. (Although our PBX has a relatively small number of line cards, large PBXs have 10,000 line cards which make up the largest single cost component of the machine.) We can also use digital filters, which can be implemented either on the CPU or on a separate card with a DSP attached to the bus. While it may seem expensive to build a separate card with a DSP solely for DTMF detection, it must be remembered that the digital filters will compete with the call activity when placed on the main CPU. The call handling process is being activated at the 8 kHz rate, and the DTMF detection process would also be activated at that rate as well. Since DTMF detection takes up more CPU time than call handling, it can cause significant CPU loads. Our experiments indicated that putting DTMF detection on the main CPU would require us to use a Pentium processor, while off-loading DTMF detection to a separate low-end DSP would allow us to use an 80386 for the main CPU. At the time of our project, the difference in cost between the x386 and Pentium was over $500, while the DSP required cost only a few dollars.

Call routing and maintenance can both be implemented as low-priority processes. Call routing has a soft deadline, since we do not want to excessively delay the call setup. Maintenance for the most part does not require deadlines, though certain shutdown operations may require timely execution.

9.8 Example: Ink-Jet Printer

The HP DesignJet drafting plotter [Boe92,Sch92], is a large ink-jet plotter designed for creating engineering drawings. It can plot on paper up to 36 inches wide at a resolution of 300 dots per inch (dpi).

printer terminology

At a very high level, the plotting process is comprised of the steps shown in Figure 9-17. The plot is given to the plotter in a **page description language (PDL)**; the original DesignJet took *HP-GL/2*, an HP graphics language; later versions also took *PostScript*, a well-known page description language. The page description is interpreted and then converted into **raster format**, which describes the picture as a two-dimensional array of dots, very similar to a television picture. At each point in the raster, a quadruplet of intensities for red, green, blue, and black are stored to describe the picture. A plotter controller reads the rasterized image and sends commands to the **plotter carriage** to put ink on paper in accordance with the raster. The plotter carriage and related units are called the **print engine**.

There are several important considerations in the design of this system:

- **Memory space is important.** A complete 36 inch plot described at 300 dpi would require a large amount of memory. Memory is always a significant cost factor and at the time this system was designed, using a brute-force solution and requiring enough memory to rasterize a complete D-size plot would make the system too expensive for most of the target customers. A variety of techniques had to be developed to reduce the amount of raster memory.

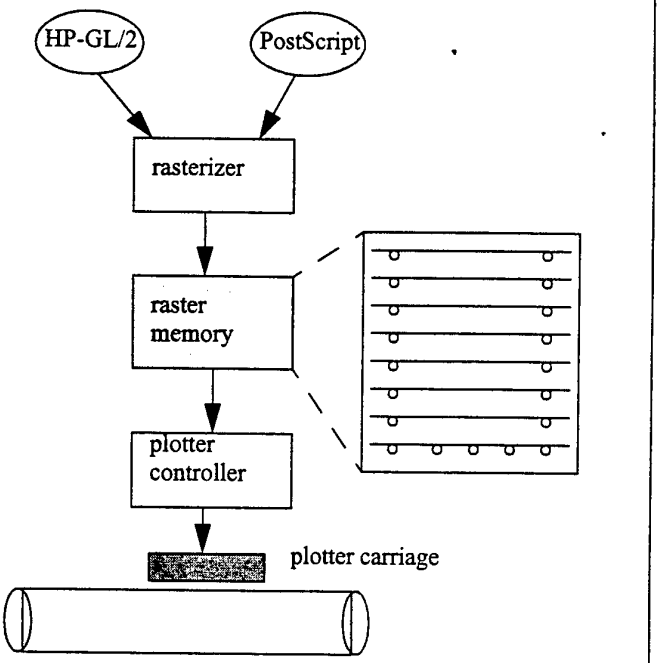


Figure 9-17 A high-level view of the plotting process.

- **Real-time control is required.** The inkjet requires some very high-speed control in order to transfer the ink from the cartridge to the paper. The intensity of each inkjet must be carefully controlled based upon the rasterized information. In addition, the inkjet head must be scanned across the page and the paper must be moved up and down with servos.
- **Multiple events are happening at once.** While the inkjets are spitting ink, the head is moving across the paper. The paper may move at the same time as the carriage is being positioned for the next pass.

The design team considered several different architectures early in the project. They first looked at specialized graphics processors, but found that they were much too slow for parsing the page description language. They then considered using a general-purpose processor for parsing and other tasks while using a graphics chip for raster conversion. This architecture was deemed too expensive since in general parsing and raster conversion would not be running simultaneously.

They finally chose a single RISC microprocessor, the Intel 80960KA to perform parsing, rasterization, and print engine control. One reason to chose the 960 was that its bus multiplexes address and data, which reduces the pin count for the application-specific ICs (ASICs) which would be used to speed up certain functions. Another reason was that another version, the 960KB, had a floating-point unit and was pin-compatible with the KA. If they determined in mid-project that faster floating-point arithmetic was required, the design could easily be upgraded.

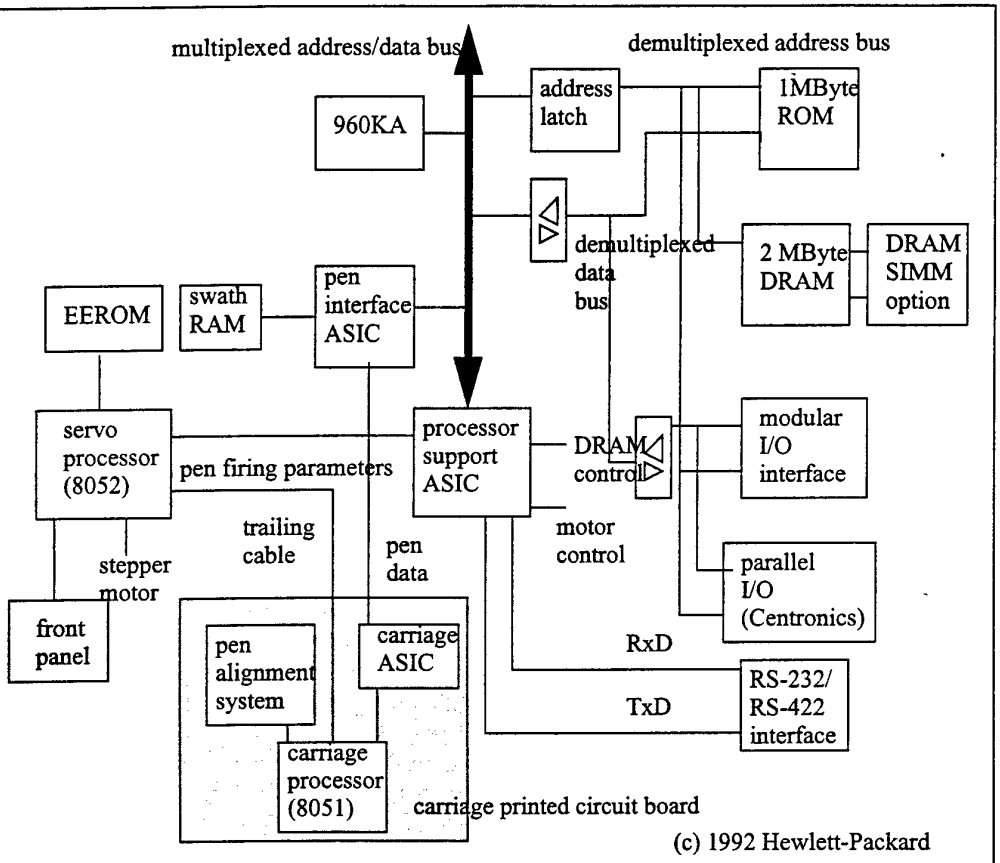


Figure 9-18 Block diagram of the electronics of the HP DesignJet plotter (after Mebane et al [Meb92]).

Several other decisions were made early:

- They decided to add a modular I/O (MIO) port to allow I/O ports to be added to the machine by the user. The RS-232/422 and parallel interfaces were kept as basic interfaces. MIO modules for both HP-IB (an instrumentation bus) and Ethernet were provided.
- They decided against using a disk to hold intermediate data during plotting. A disk would add too much to the cost of the plotter. However, since some users with more complex plots might need more memory, they chose to add sockets for single inline memory modules (SIMMs).

To understand the plotter, let's first look at the hardware architecture, then the software that runs on that hardware platform.

9.8.1 Hardware Design

The hardware architecture is shown in Figure 9-18. This system is rather complex, including a RISC microprocessor, two microcontroller, and three

application-specific ICs (ASICs). This architecture is complex because the system performs several different types of processing—parsing the page description language, rasterizing, controlling the print engine—and must do so at reasonable cost. Both ASICs and microcontrollers were added as separate processing elements to enhance performance over the basic main processor system.

Notice that the hardware architecture doesn't strictly fit the typical accelerated system architecture of accelerators attached to a single CPU bus. The servo processor is only loosely coupled to the i960. This makes sense since the servo commands are relatively simple on the i960 side but control of the servo itself takes fast responses.

The system includes 2 Mbytes of DRAM (with SIMM sockets to hold more) and 1 MByte of ROM. In addition to the main CPU (the i960KA), there are three ASICs: the pen interface ASIC, the processor support ASIC, and the carriage ASIC (which is part of the pen alignment system). There are also two microcontrollers. The servo processor controls a servo motor on the print carriage, controls the front panel display and scants its keyboard, and runs a service station for the inkjet pens.

We can make better sense of Figure 9-18 by identifying the major operations and the flow of data through the system. Figure 9-19 shows how data flows during three major operations. First, the page description comes through an I/O port (the parallel port in this instance) to the main processor and its main memory. Here, the page description is parsed and pixels are generated. Second, the generated pixels are stored in a **swath**—a section of the rasterized plot. The swath memory is kept separate from the main processor memory because it is more than a buffer for the generated pixels. The pixels are generated by the main processor in row order, but they must be fed to the inkjet pens in column order. The swath memory is used by the pen interface ASIC to accomplish this task. To actually plot, the pixels are read from the swath memory and transferred to the carriage control system in the appropriate order.

The servo processor, which is an 8052, operates as a slave to the main processor. The processor support ASIC implements a bidirectional port which can be used either to send commands or as a mailbox through which the servo processor can send data back to the main processor or generate several different types of interrupts on the 960. Using an external processing element was found to significantly off-load the main processor. The microcontroller was both faster to design and cheaper to manufacture than an ASIC.

The inkjet pens each have 50 nozzles and the pens may not be closely aligned with each other—tight alignment would make the pens hard for the user to install and would be hard to maintain in any case. As a result, the hardware includes units to measure the current alignment of the pens and to compensate for misalignments during printing. To align itself, the printer first prints a test pattern on a piece of paper, then scans the result to determine the misalignment of the pens. The carriage processor microcontroller, an 8051, controls the optical alignment system. The carriage processor also provides the serial communication link from the 960 to the servo processor.

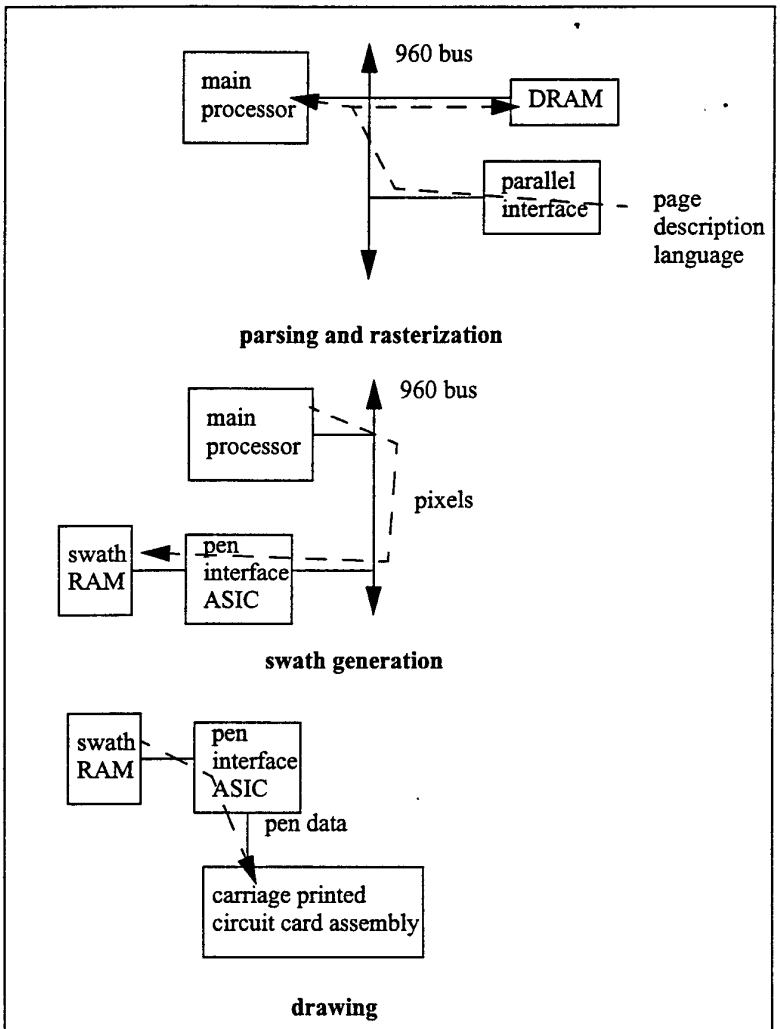


Figure 9-19 Major operations in the DesignJet plotter.

The processor support ASIC performs several different functions in the printer, all of which were collected in a single chip. It connects to both the 960 and the servo processor; it provides both interrupt-driven unidirectional transfers and polled, bidirectional mailboxes. The *main processor interface* allows it to talk to the 960 and to generate interrupts. The *servo processor interface* talks to the 8052. The *interprocessor communication* block intermediates the communication between the two processors.

The *motion controller* block decodes the position of the print carriage and paper axes, based upon encoded signals provided to it. It also senses the servo motor current and uses a watchdog timer to determine if the servo has run out of control; if the servo runs for too long, the controller shuts it off. The carriage axis and paper axis outputs are pulse-width modulated (as with

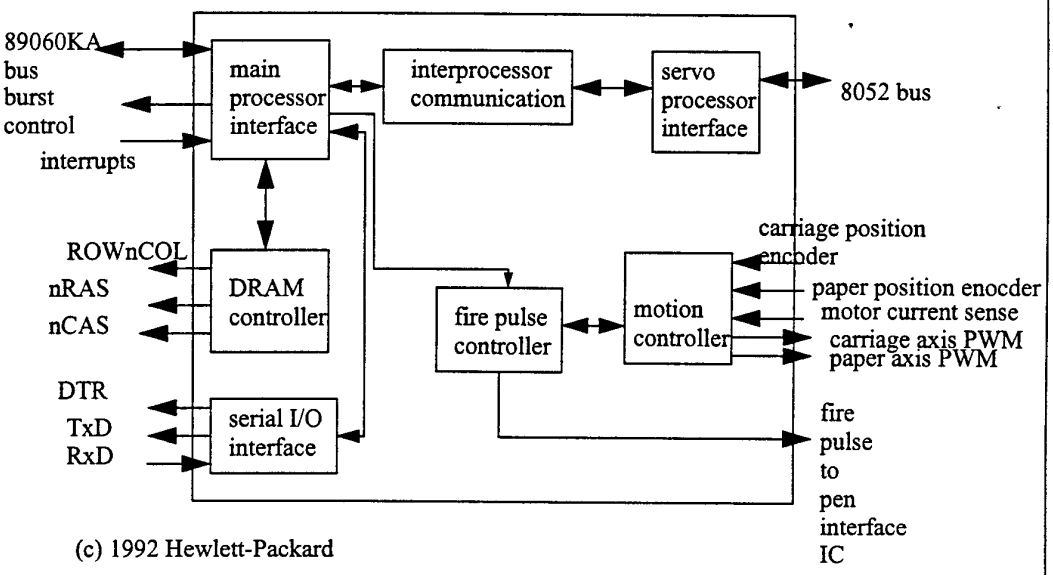


Figure 9-20 The processor support ASIC for the DesignJet printer [Meb92].

the model train controller). The *fire pulse controller* block generates a synchronizing output for each column of pen data.

The remaining blocks provide basic service functions. The *DRAM controller* block generates refresh signals for the DRAM. The *serial I/O interface* provides the basic RS-232/422 interface.

The pen interface ASIC architecture is shown in Figure 9-21. The pen interface is a key component of the shuffler subsystem, which maps the pixels generated by the main processor in a row orientation to a column orientation to feed the pens. This chip includes interfaces to the main processor bus, the swath memory, and the serial link to the carriage ASIC. The pen interface reads pixels from the swath in a predetermined order according to a predetermined pattern. Keeping the swath memory on a separate bus greatly reduces traffic on the main processor bus and also makes it easier to support different pen nozzle configurations. When data is copied from the main processor to memory, data flows from the main processor interface to the swath RAM; when printing, the data flows from the swath RAM to the serial interface. The pixel address generator includes an SRAM array for a programmable sequencer, a column counter, and an adder. The sequencer memory is loaded with pixel address offsets for each pen; since the printer head prints in both directions, it must be able to run through sequences in both directions. A pixel counter counts the number of pixels to be printed; this value can be used to calculate ink use.

The carriage ASIC block diagram is shown in Figure 9-22. This ASIC interfaces to the carriage processor bus, the pen interface ASIC, and the servo

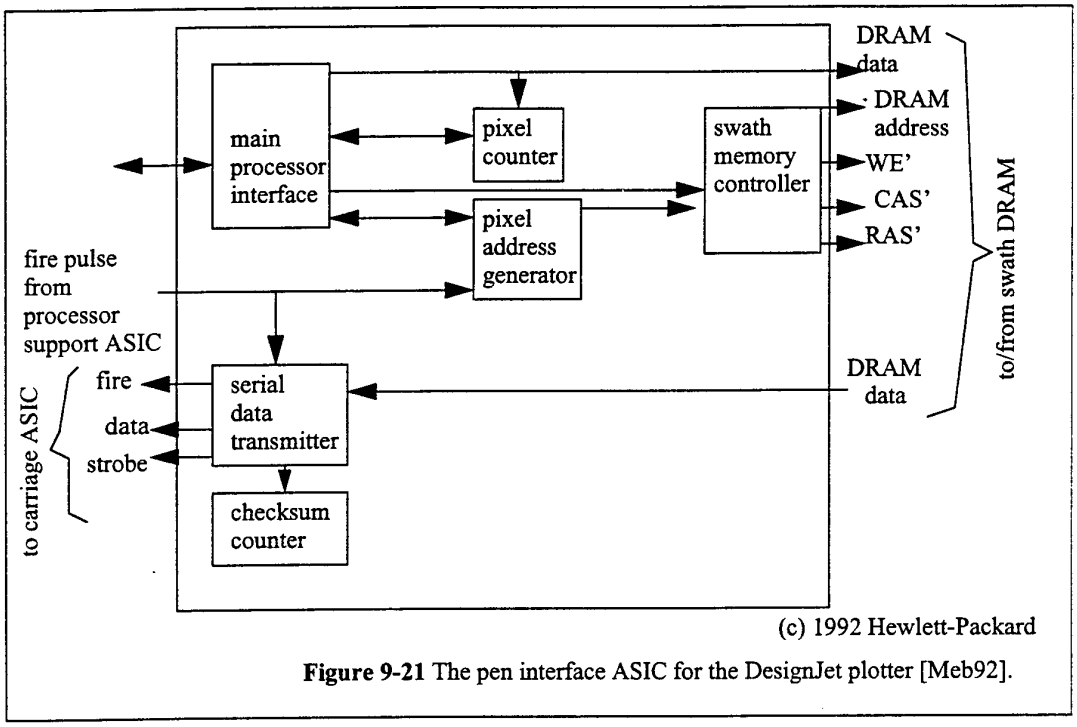


Figure 9-21 The pen interface ASIC for the DesignJet plotter [Meb92].

processor. The carriage ASIC reads timing control registers using the processor bus. The pen interface data comes in serial form; it is converted into parallel form on chip and used to drive four different pen channels. The delayed pipeline registers are programmed with delay values in order to control the pen alignment.

Since the shuffling algorithm implemented in the pen interface and carriage ASICs had not been built before, the algorithms were developed using a behavioral model written in C. Graphical stubs were written for both input and output to help visualize the results of processing. This allowed the shuffling algorithm to be debugged before designing the ASICs.

In order to be able to develop the printer software in parallel with the hardware, the design team built emulators for the ASICs. The processor support and pen notifies ASICs were built with a combination of fast PLDs, standard parts, and UARTs; the high-speed PLDs were required to meet the fast clock speeds for these subsystems. The carriage ASIC emulator was built from an FPGA since it did not have to run as fast.

9.8.2 Software Design

The design team identified three major challenges for developing the plotter software. First, the hardware platform would not be available until several months after programming had begun. Second, the team was unfamiliar with the 960. Third, the design team was small. To solve all three problems, they

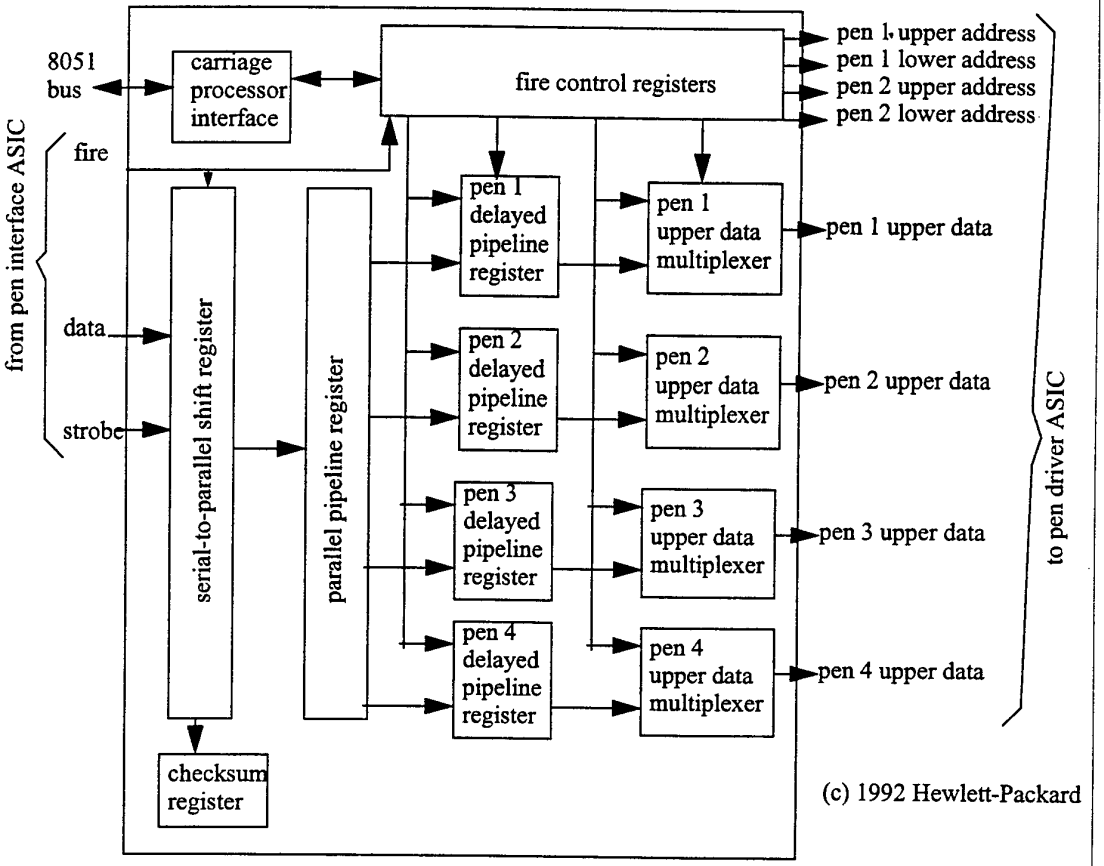


Figure 9-22 The carriage ASIC for the DesignJet plotter [Meb92].

created a software development environment which did not require the hardware platform and which maximized programmer productivity.

The software development environment created for the plotter design allowed software to be run either on a Unix workstation or on the target platform. The main differences between the two lay in the I/O and print engine subsystems. The I/O subsystem code was easily modified to accommodate file I/O when running on the host workstation. The print engine was emulated on the host using an X Windows interface that showed the current state of the swath.

The main processor software was developed around a real-time operating system developed in-house. An RTOS was chosen for two reasons: first, it provided standard mechanisms for interprocess communication; second, it eliminated many common timing problems by providing preemptive scheduling. The team reported very few timing problems in the design of the printer.

The software running on the main processor included both a parser for the page description language and a rasterizer. The DesignJet team used an HP-GL/2 language subsystem that had been used on several other printers. This reduced both development and debugging time. They rewrote an existing vector-to-raster converter from assembly language into C. This was clearly necessary since they were using a new processor.

To debug 960 code when running on the target platform, they used a remote debugger. The GNU debugger for the 960, gdb960, runs as a monitor program on the main processor and communicates with the workstation using a serial interface. The debugger provided C source-level debugging while running on the target.

The front panel interface was developed using a simulator running on a PC. Marketing and user interface experts were able to use the simulator to design their own front panels.

The process of loading paper into the printer evolved substantially over the course of the design. The basic algorithms were developed by mechanical engineers working on a breadboard. The mechanical engineers provided graphical descriptions of their work to software engineers who wrote programs to implement the necessary actions.

9.9 Example: Personal Digital Assistants

Personal digital assistants (PDAs) have become an important example of the information appliance—a device that can be used to receive, display, and transcribe information. While this product category is evolving rapidly, we can identify several important characteristics of these machines' architectures [Man99].

An ideal PDA has several important characteristics: it is cheap, physically small, lightweight, has a long battery life, and provides many features to the user. Some of these characteristics depend on mechanical design and the properties of important components, such as the display. However, many of these characteristics also greatly influence the design of the digital hardware platform and the software running on that platform.

The Apple Newton MessagePad is considered the first modern PDA. The hardware organization of the Newton is shown in Figure 9-23 [Fur96]. The first-generation Newton used the ARM 610 processor (a later version used the StrongARM). A variety of support operations were bundled into a single ASIC known as the Runt: DMA control, real-time clock, timers, bus interface and CPU control, digitizer and LCD interface, audio generation, and PCMCIA and infrared support. The bulk of the software was written in the NewtonScript language, which was an interpreted byte-code language supporting an object-oriented model. The software was organized around a real-time operating system which preemptively scheduled processes on the platform. All processes used a single address space, but the ARM MMU was used to provide memory protection via its subpage mechanism.

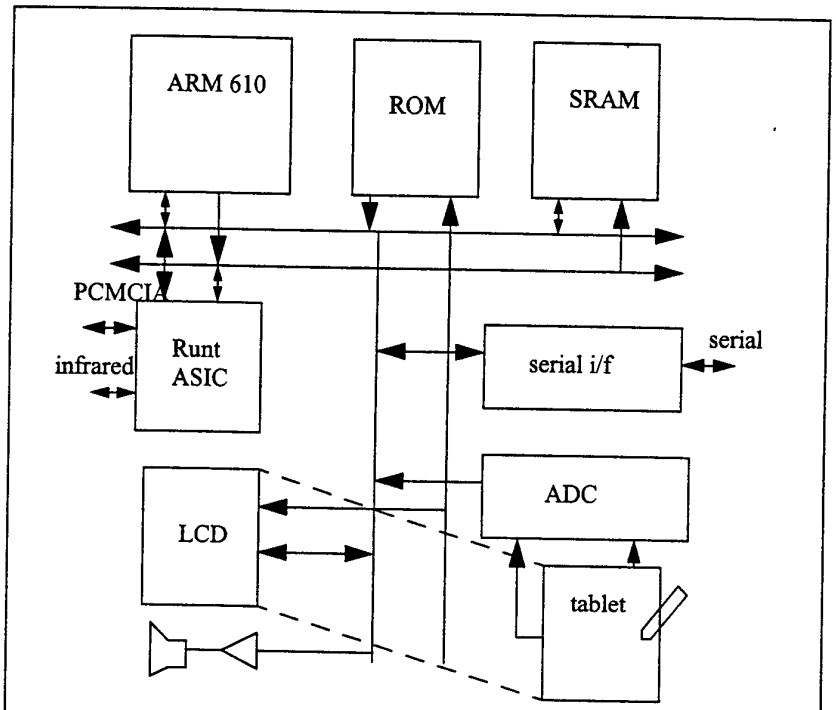


Figure 9-23 Hardware architecture of the Newton MessagePad [Fur96].

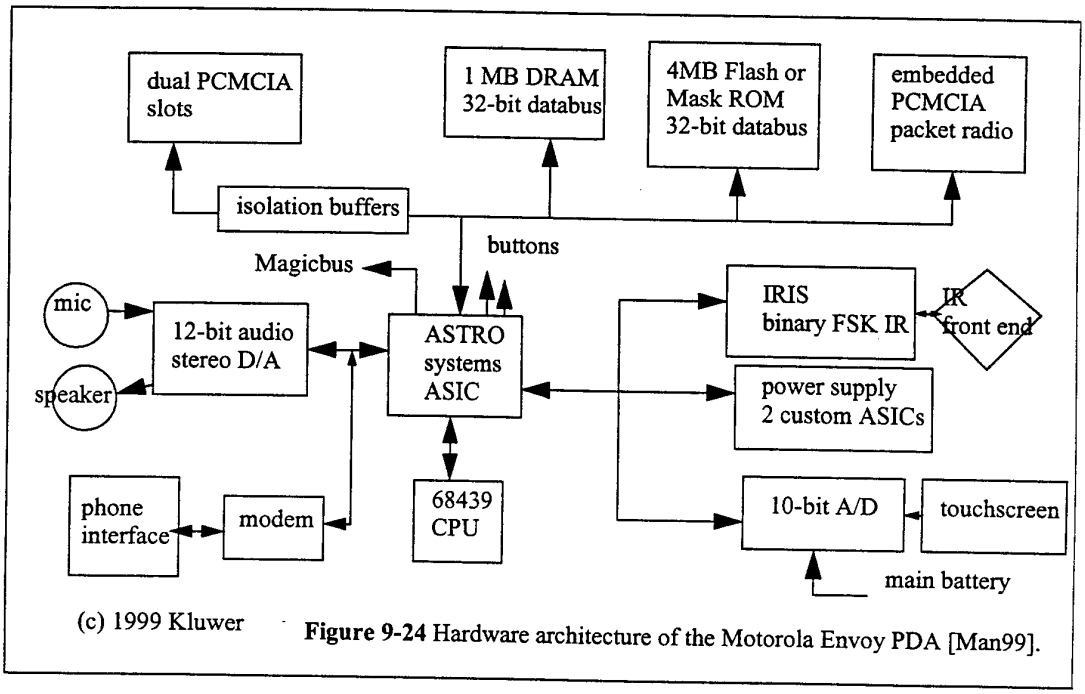


Figure 9-24 Hardware architecture of the Motorola Envoy PDA [Man99].

The Motorola Envoy PDA appeared in 1994; its hardware architecture is shown in Figure 9-24. The hardware architecture is in many ways similar to the Newton: in addition to a CPU (a Motorola 68349 in this case), an ASIC is used to encapsulate many system functions; other chips supply I/O functions for the touch screen, etc. The Envoy used the Magic Cap operating system, which made use of object-oriented software design.

One major design challenge in PDAs is *feature creep*—the tendency for the final system to have many more features than the architecture can adequately support. Features can be added in the form of additional interfaces for new types of devices; new interfaces not only increase power consumption, but also make the mechanical design of the system harder and may contribute to electromagnetic interference. Features can also enter in the form of software modules. Such features may tax the available RAM and ROM in the system. If the added software causes memory thrashing, the penalty is felt not only in slower execution time but also in decreased battery life due to the increased memory activity.

The design of the power supply is another important aspect of PDA design. Quite a bit of data is kept in RAM during execution; if the machine loses power suddenly, the PDA could be rendered useless. Many batteries have hard-to-predict lifetimes—voltage may go from an acceptable value to near zero very quickly. To combat this problem, Intel and Duracell developed the Smart Battery System. In their system, the battery is packaged with a microcontroller and a small amount of memory; the PDA or other device can talk to the battery's microcontroller over a simple serial bus. The battery's internal electronics can read the battery's charge status more accurately, allowing the system to read battery lifetime more accurately and provide for a safe shutdown before power fails.

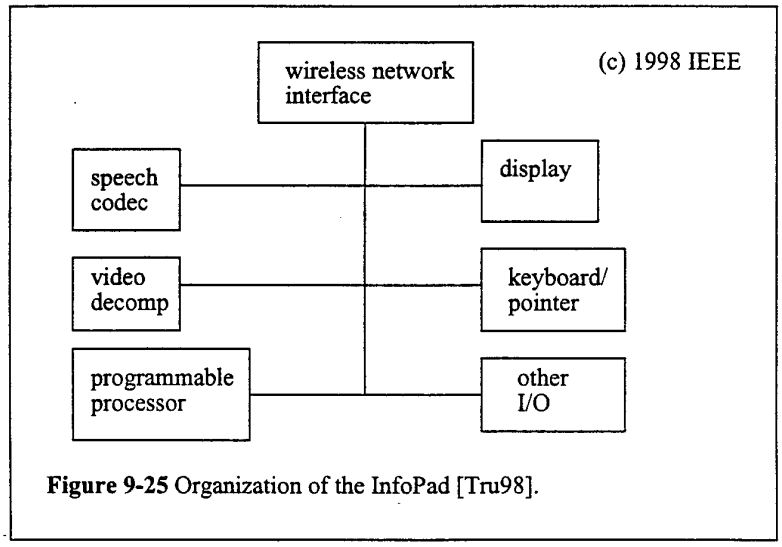


Figure 9-25 Organization of the InfoPad [Tru98].

An influential system for even more advanced handheld functions was the InfoPad [Tru98]. InfoPad not only provided a portable computer with a

screen—it also provided a wireless link and the ability to handle audio and video data. Because it had a wireless link, InfoPad was designed to act as a remote I/O interface. It used remote computers, accessed over the wireless network, to provide many computational functions; this helped with power, size, and weight. However, it did contain a significant amount of computational power itself: it used an ARM60 running at 10 MHz, 512 MB of RAM, and 128KB of ROM. As illustrated in Figure 9-25, the CPU was viewed as one element of the overall system. Some operations were implemented in special-purpose hardware in order to save power; other operations were implemented in the ARM60 in order to provide more flexibility. For this reason, the ARM60 was referred to as a Peripheral Processing Unit; it was viewed as a controller rather than as the central unit in the system.

9.10 Example: Set-Top Boxes

In this section we'll study two related designs [Aki96, Rat96] for the hardware and software designs of **set-top boxes**—devices that are used to interface a video delivery service, such as cable or satellite, with a consumer's television. Set-top boxes for cable TV were traditionally based on analog circuitry for the television signal with a small amount of user interface functionality provided by a microcontroller; we will concentrate here on set-top boxes for digital television, which typically provide more advanced user interfaces as well. Figure 9-26 shows the relationship of the set-top box to the complete television delivery system. The box decodes the digital TV signal and converts it into a form suitable for the consumer TV; it also provides user interface features, both by generating on-TV displays and by listening to a remote control through an infrared (IR) interface; some boxes also transmit information to the content provider through a **back channel**, often using a telephone line, for purchase of special programs or other options.

cost-sensitive

Set-top boxes are very cost-sensitive. These designs were targeted to a selling price of $250-$350 per box, or about equal to the cost of a mid-range television, when produced in volumes of tens of millions of units.

hardware architecture

Figure 9-27 shows the hardware architecture of a set-top box designed for a telephone company for a fiber-to-the-curb video delivery system. The network interface module receives and transmits data over the fiber-optic network. The video is delivered in the form of MPEG, an international standard for compressed video and audio. A collection of units turn the compressed bit stream into analog signals for the television set. The MPEG transport demultiplexer separates (demultiplexes) the MPEG data stream into audio and video components, which are then decoded by the MPEG audio and video decoders. A radio frequency (RF) modulator, an NTSC encoder, and analog D/A converters convert the digital signal into analog form for the television. The host microprocessor is a Motorola MC68341, which includes a 68020 CPU, two channels of DMA control, two serial channels, a timer, and a serial peripheral interface. A CD-i graphics processor can be used to generate graphics on the screen. An I/O microprocessor talks to the box's keyboard, the IR link that receives commands from the remote, and a card reader. A PCMCIA slot provides the ability to expand memory. Note that the

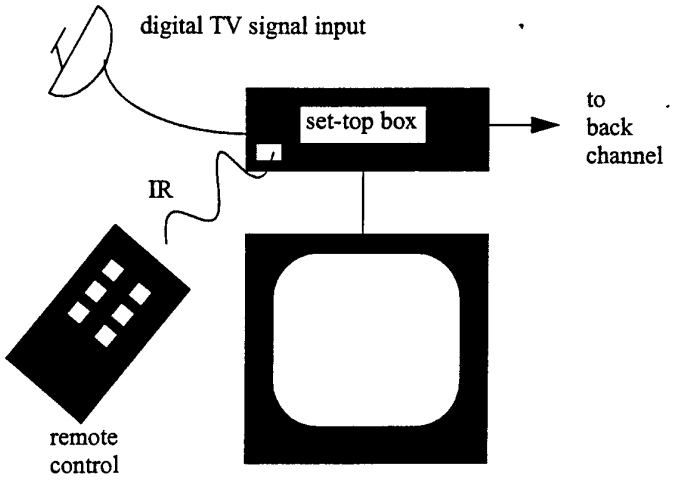


Figure 9-26 Role of a set-top box in digital TV system.

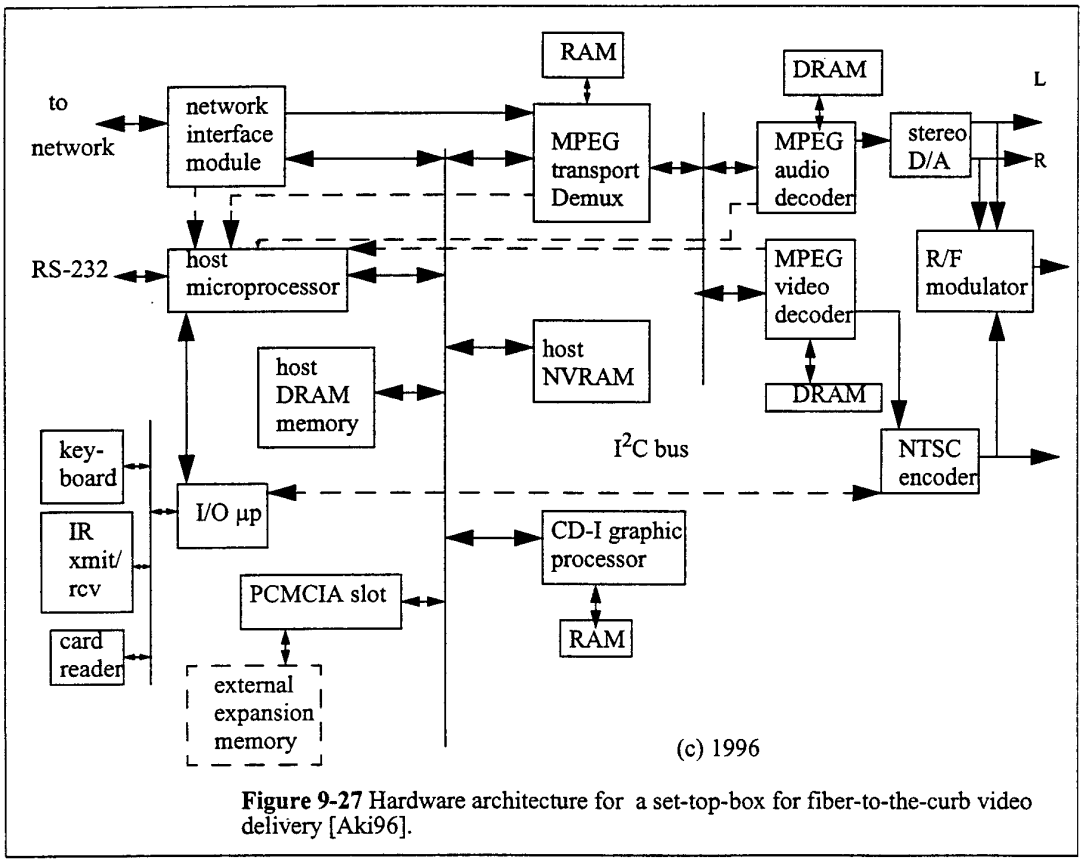


Figure 9-27 Hardware architecture for a set-top-box for fiber-to-the-curb video delivery [Aki96].

MPEG system, the host microprocessor, and the I/O microprocessor each have their own bus. Separating the MPEG system and the host processor is particularly important to ensure that video decoding receives enough bus capacity. Also note that many subsystems have their own memory—memory bandwidth would probably be too high if all memory were shared on a single bus.

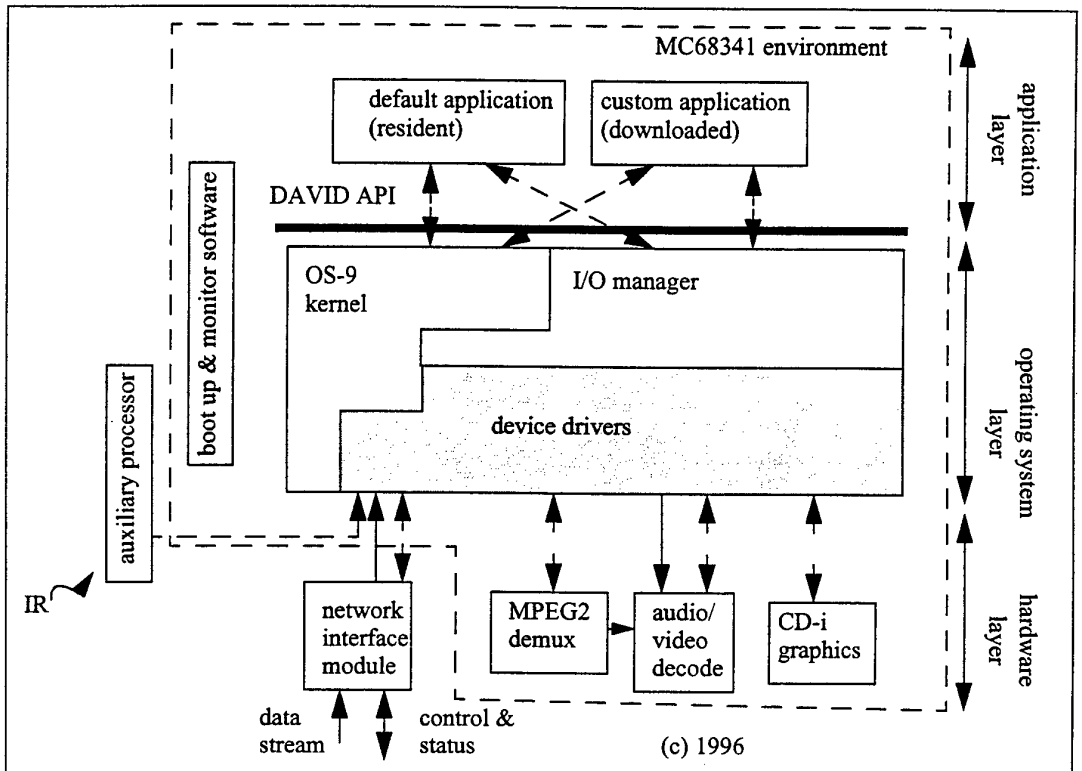


Figure 9-28 Software architecture of the set-top box [Aki96].

software architecture

The software architecture of the box is shown in Figure 9-28. The host processor runs the OS-9 operating system, whose application programming interface (API) is known as DAVID. Note that the MPEG2 decoder operates below the operating system. Because these functions are implemented in special-purpose hardware, they appear as very low-level operations that do not directly affect the scheduling of the application on the host processor.

Figure 9-29 shows the software architecture of a different set-top box [Rat96] in more detail than that shown in Figure 9-28. The hardware architecture of this box is slightly different but has the same flavor as that of Figure 9-27.

As with the other system, this box is built on top of the OS-9 operating system. The system level includes a number of basic processes. The software is structured so that all operations are performed on objects structured as files; for example, the MPEG stream appears as a file. RTNFM is the real-time

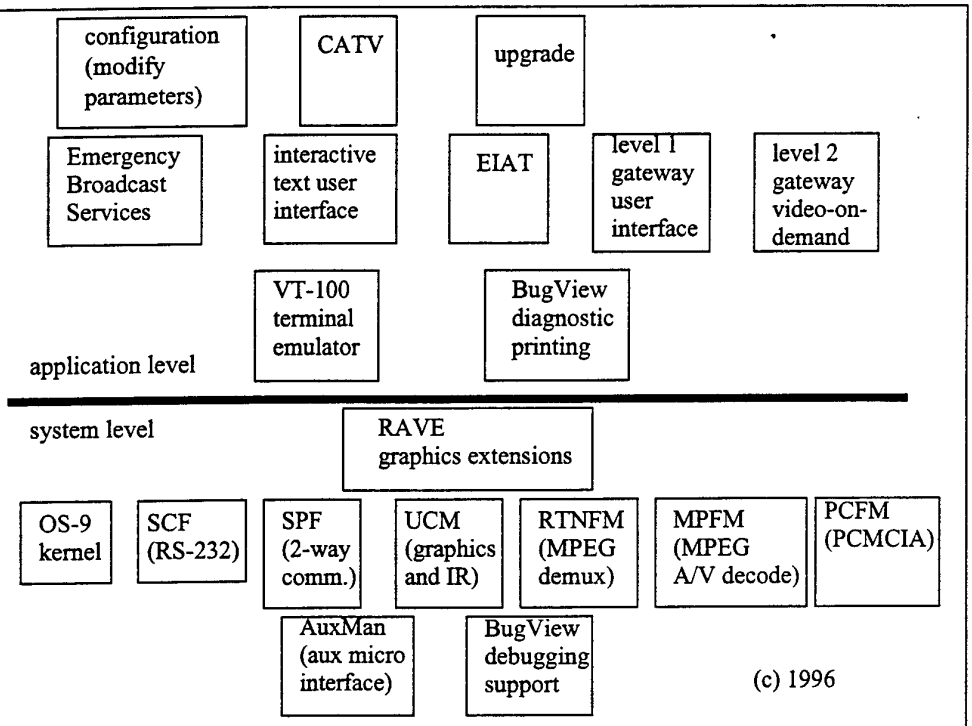


Figure 9-29 Software architecture of another set-top box [Rat96].

network file manager that handles all these file objects. SPF is the sequential packet file manager that handles the network interface. MPEG decoding is controlled by MPFM, the MPEG file manager. The CD-i graphics processor is managed by UCM, the user communication manager task. PCFM is the PCMCIA port manager.

At the application level, the navigator is the primary user interface. The navigator is in fact a distributed application that processes some commands locally and dispatches others to network elements. The level 1 and level 2 gateways provide means to get services from the network. The level 1 gateway is used to download new applications to the terminal; this could be used, for example, to supply video games. The level 2 gateway is used for the video stream of a video-on-demand service. The config application sets up the network address and other utilities. The display process provides a virtual terminal display on the TV set for messages from other applications. The Emergency Broadcast Services process provides notice of warnings about natural disasters and other emergencies that are transmitted over the network. The IAT application allows the user to send messages upstream to the provider, for example for interacting with game shows.

9.11 Systems-on-Silicon

While embedded systems have traditionally been built as multiple chips connected on a printed circuit board, an increasing number of embedded systems are single-chip systems known as **systems-on-silicon** or **systems-on-a-chip**. A system-on-silicon presents all the challenges of traditional embedded system design as well as many more problems.

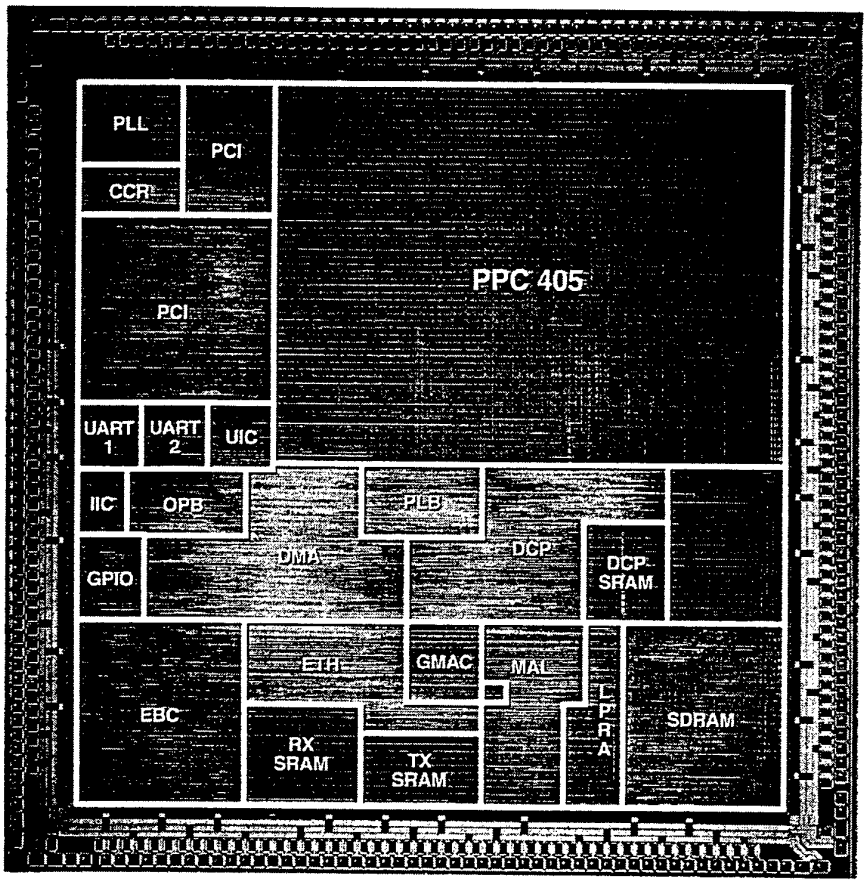


 Figure 9-30 An embedded processor for systems-on-silicon.

An example embedded processor for systems-on-silicon is shown in Figure 9-30. The IBM PowerPC 405GP includes a CPU core, 16K instruction and 8K data caches, an SDRAM controller, peripheral controllers, PCI and Ethernet interfaces. The design is available as a component that can be placed on a system-on-silicon, along with whatever other logic and memory is desired.

While a system-on-silicon does not inherently require an embedded CPU, most do for several reasons. Programming an embedded CPU provides a much more productive design technique than hardwired controllers. The embedded CPU also takes advantage of embedded RAM technology—since large amounts of RAM are available on-chip, it makes sense to choose a design methodology that takes advantage of it. Embedded software is also more easily upgradeable to new releases and versions of the product.

verification challenges

However, systems-on-silicon pose several unique challenges. Integrated circuit fabrication technology makes traditional embedded system debugging impractical. Fabricating the chip takes weeks or months and costs tens or hundreds of thousands of dollars. Although a printed circuit board design can be modified by fixing the printed circuit board (cutting traces, adding wires) or by replacing chips (for example, reprogramming the PROM), none of these techniques work on an integrated circuit. If the system can be tested only by changing the hardware, that change will incur the same costs in time and money of fabricating the original version. Furthermore, observing signals on the chip is much more difficult than attaching a logic analyzer to a printed circuit board.

As a result, systems-on-silicon must be designed under methodologies that aim for **first-time correctness**. Systems-on-silicon design makes more extensive use of computer-aided design:

- analysis tools help determine the characteristics of a proposed architecture;
- synthesis tools help configure the hardware and software architectures;
- co-simulation tools allow the system design to be exercised before building the complete system.

The cost of architectural mistakes in embedded systems has always been high, but it is extremely high in systems-on-silicon due to the time and expense of a design iteration.

other design challenges

An important advantage of systems-on-silicon is that we can customize components—we do not have to use stock microprocessors or other components. We can easily, for example, select the cache size and even cache organization to suit the needs of our application. It is even possible to modify the instruction set of the processor to suit our needs. An **application-specific instruction processor** (**ASIP**) is a CPU whose instruction set has been optimized for a particular set of applications.

9.12 Summary

System design takes a comprehensive view of the application and the system under design. To make sure that we design an acceptable system, we must understand the application and its requirements.A number of techniques, such as object-oriented design, can be used to create useful architectures from the system's original requirements. Along the way, by measuring our design processes, we can gain a clearer understanding of where bugs are

introduced, how to fix them, and how to avoid introducing them in the future.

What we learned:

Design methodologies and design flows can be organized in many different ways.

Building a system mock-up is one good way to help understand system requirements.

Statecharts are valuable in the specification of control and are part of UML.

CRC cards help us understand the system architecture in the initial phases of architecture design.

Further Reading

Pressman [Pre97] gives a thorough introduction to software engineering. Davis [Dav90] gives a good survey of software requirements. Beizer [Bei84] surveys system-level testing techniques. Leveson [Lev86] gives a good introduction to software safety. Schmauch [Sch94] and Jenner [Jen95] both describe ISO 9000 for software development. A tutorial edited by Chow [Cho85] includes a number of important early papers on software quality assurance. Cusumano [Cus91] gives a fascinating account of software factories in both the U.S. and Japan. Newman [New94] describes a small PBX design based on a microcontroller; a later article [New96] describes phone line interfaces.

Questions

Q9-1. Briefly describe the differences between the waterfall and the spiral development models.

Q9-2. What skills might be useful in a cross-functional team that is responsible for designing a set-top box?

Q9-3. Describe how you might use a spiral development model to design a PBX like that described in Section 9.7.

Q9-4. Describe how you might use successive refinement to design a printer like the one described in Section 9.8.

Q9-5. Describe how concurrent engineering practices might be applied to the design of a PDA like those described in Section 9.9.

Q9-6. Give realistic examples of how a requirements document may be:

a. ambiguous;

b. incorrect;

a. incomplete;

b. unverifiable.

Q9-7. Develop a state diagram that defines the basic actions of a telephone call: going off-hook, dialing, connecting, talking, hanging-up and tearing down the call. Use AND and OR states whenever possible.

Q9-8. Develop a state diagram that describes how a user of a set-top box can select a channel from an on-screen display using *up* and *down* buttons to select the desired channel. Use AND and OR states whenver possible.

Q9-9. Develop a set of CRC card descriptions for the PBX of Section 9.7.

Q9-10. Develop a set of CRC card descriptions for a PDA like the ones described in Section 9.9.

Q9-11. Develop a set of UML diagrams that describe a design pattern for an on-screen user interface; the design pattern should be applicable to both PDAs and set-top boxes.

Lab Exercises

L9-1. Develop UML diagrams to describe some of the systems in this chapter, either as specifications or as architectural diagrams.

L9-2. Draw a diagram showing the developmental steps of one of the projects you recently designed. What development model did you follow: waterfall, spiral, etc?

L9-3. Find a detailed description of a system of interest to you. Write your own description of what it does and how it works.

SECTION TWO

PRESENTER : PROFESSOR S-Y KUNG

Neural Networks

- *Overview*
- *Supervised learning: BP,MOE,DBNN, ...*
- *Unsupervised learning: VQ, EM, ...*
- *Channel Fusion*
- *Applications*
- *Implementation Issues*
- *PCA ,ICA...*

Overview

Modern information technology in the internet era should support interactive and intelligent processing that transforms and transfers information, for this NN will play a key role.

Human Neural System

10^{10} - 10^{11} Neurons
10^{14} Synapses

Switching time -- 100hz

Neuron Computation: 10^{10} x 10^{2}

Synaptic (Network) Computation: 10^{14} x 10^{2}

How to make the

Massive Computing Power

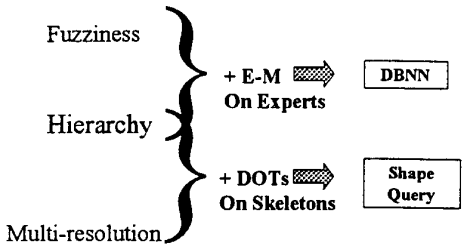
to work

Intelligently???

Synergistic Modeling and Applications of Hierarchical Fuzzy Neural Networks

By S.Y. Kung, et al., Proceedings of the IEEE, Special Issue on Computational Intelligence, Sept. 1999

Hierarchical Representation



Hierarchical Structure of NN

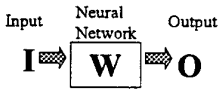
- ***0-level hierarchy:***
  BP
- ***1-level hierarchy:***
  MOE,DBNN
- ***2-level hierarchy:***
  PDBNN

Learning (Training) Phase

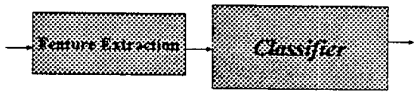
Supervised: (I,O)⇨(W)

Unsupervised: (I)⇨(W)

Retrieval (Search) Phase: (I,W)⇨(O)



Neural Network System



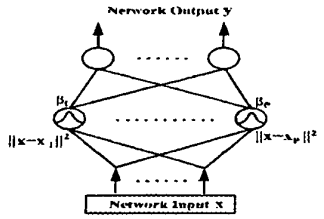
- ***Feature Extraction:***
  VQ,EM
  PCA
- ***Classifier:***
  BP
  DBNN

Application of Neural Networks

- given a set of candidate features, which ones contain most information and most helpful in classification?
  - PCA ...
- supervised learning: given training pairs of [feature vector, class], what is the classification rule?
  - BP,MOE,DBNN, ...
- unsupervised learning: how to classify samples and obtain representative?
  - VQ, EM, ...

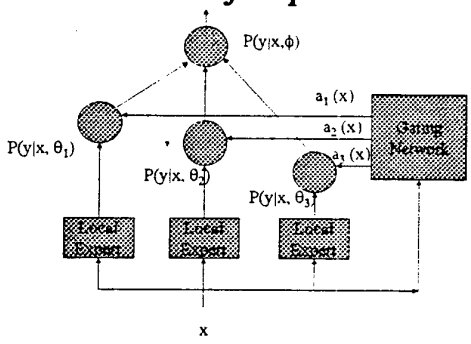
Supervised learning: BP,MOE,DBNN, ...

RBF BP Neural Network

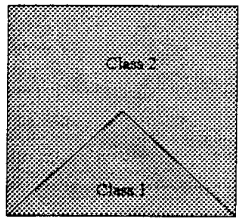


The centers and widths of the RBF Gaussian kernels are deterministic functions of the training data;

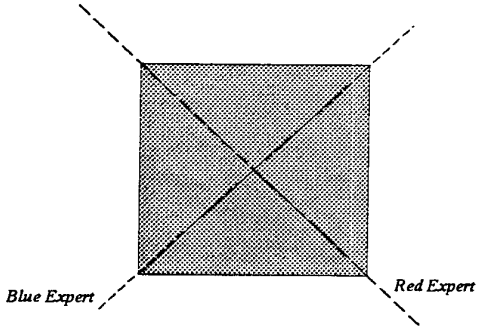
Mixture of Experts



Mixture of Experts



Mixture of Experts



Decision-Based NN (DBNN)

- *Discriminant function (Score function)*
- *Reinforced and Anti-reinforced Learning Rules*
- *Hierarchical and Modular Structures*

Discriminant function (Score function)

- *LBF Function (Mixture of)*
- *RBF Function (Mixture of)*
- *Prediction Error Function*
- *Likelihood Function : HMM*

Hierarchical and Modular DBNN

- ***Subcluster DBNN***
- ***Probabilistic DBNN***
- ***Local Experts via K-mean or EM***
- ***Reinforced and Anti-reinforced Learning***
- ***1-level hierarchy vs. 2-level hierarchy***

Reinforced and Anti-reinforced Learning

Reinforced Learning

$$\Delta w_i = + \eta \nabla_{w_i} \phi_i (x, w)$$

Anti-Reinforced Learning

$$\Delta w_j = - \eta \nabla_{w_j} \phi_j (x, w)$$

Reinforced and Anti-reinforced Learning for Linear Perceptron

For LBF Discriminant Function

$$\nabla \phi_j (x, w_j) = (x - w_j)$$

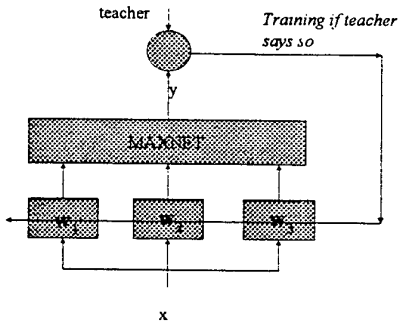
Reinforced Learning

$$\Delta w_i = + \eta (x - w_i)$$

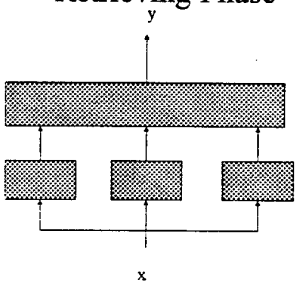
Anti-Reinforced Learning

$$\Delta w_j = - \eta (x - w_j)$$

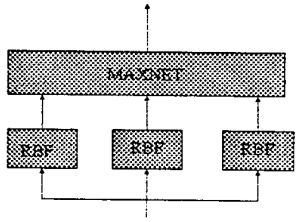
DBNN



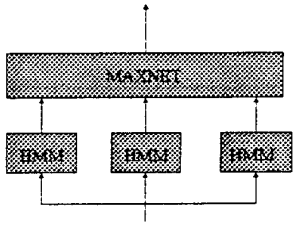
DBNN Structure for Retrieving Phase



Ex: Static DBNN (Retrieving)



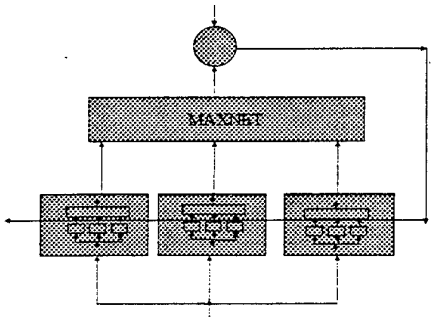
EX: Temporal DBNN (Retrieving)



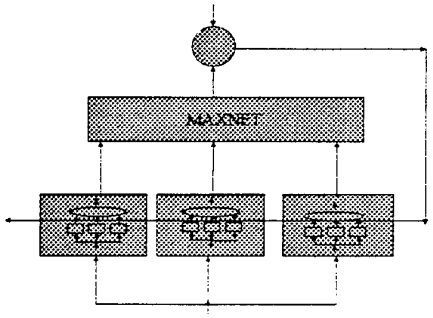
2-level Hierarchical DBNN

- ***Subcluster DBNN***
- ***Probabilistic DBNN***

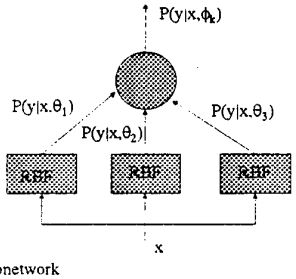
Subcluster DBNN



Probabilistic DBNN



Subnetwork of a Probabilistic DBNN is itself a collection of local experts



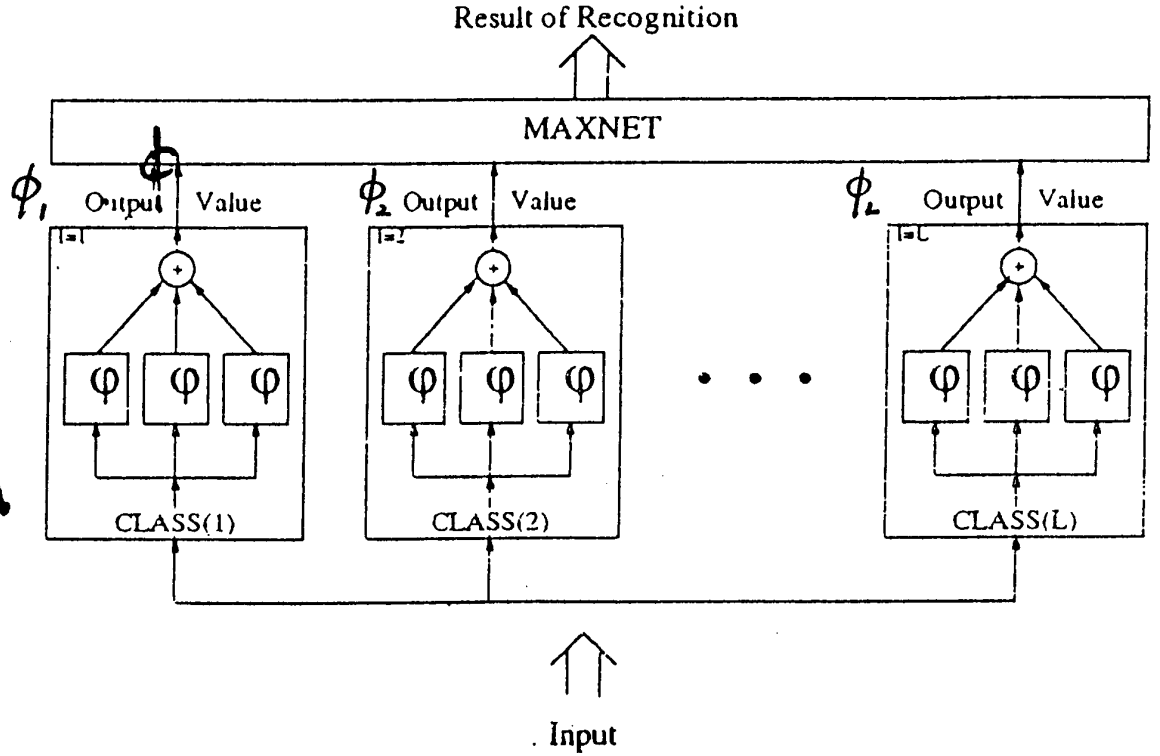
k-th subnetwork

Pictorial Presentation of Hierarchical DBNN

Probabilistic (Expert = Hidden-Node) DBNNs

$$\phi(\mathbf{x}, \mathbf{w}_l) = \sum_{k_l=1}^{K_l} c_{k_l} \psi_l(\mathbf{x}, \mathbf{w}_{k_l})$$

$$\Delta \mathbf{w}_l = \pm \eta \nabla \phi(\mathbf{x}, \mathbf{w}_l)$$



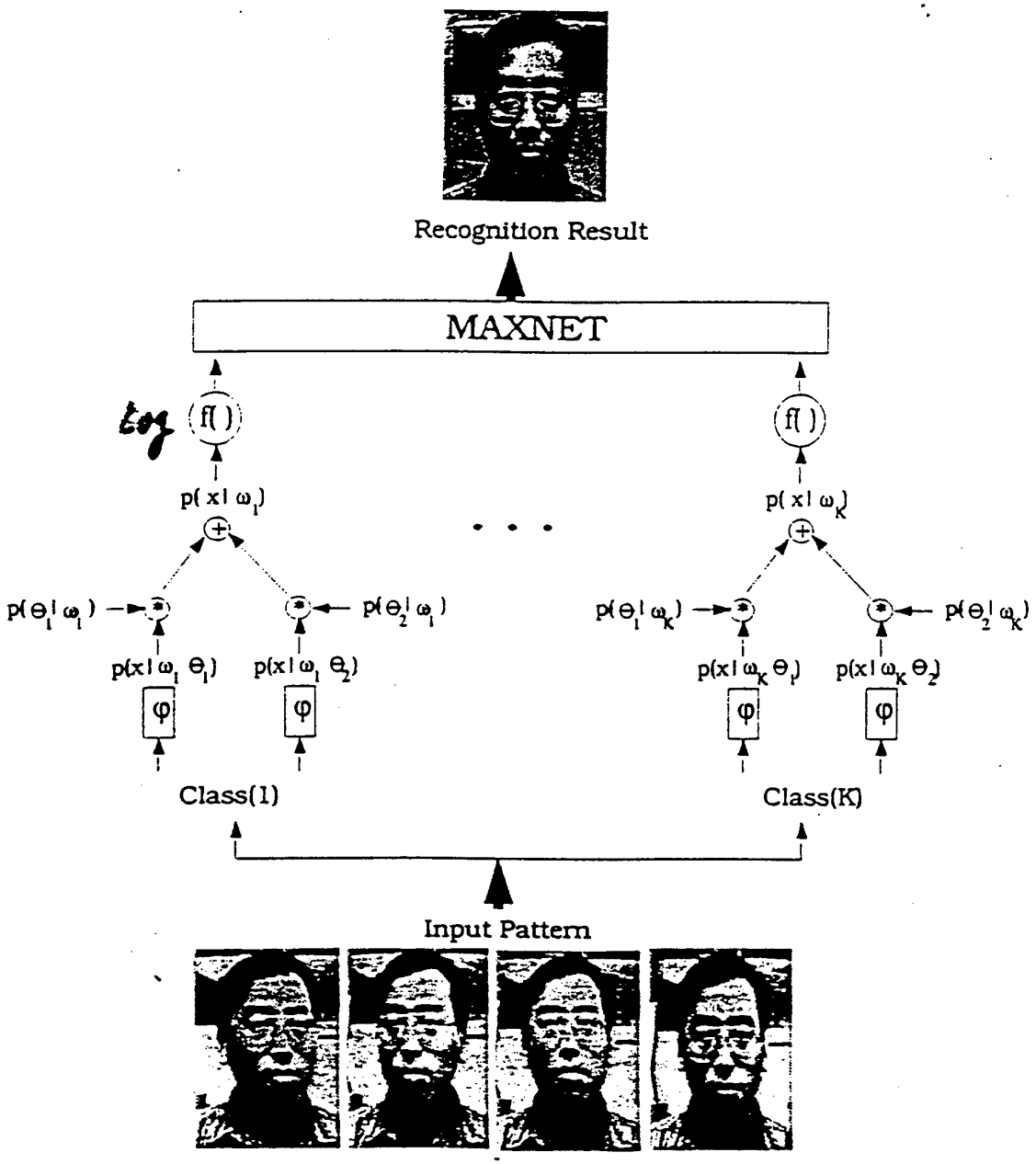
basis function

Figure 4: A schematic diagram of a hierarchical OCON structure: hidden-node structure

Probabilistic Model(I): Probabilistic DBNN

:aining goal: maximize class conditional likelihood density

$$p(x|\omega_i) = \Sigma_r P(\Theta_r|\omega_i)p(x|\Theta_r,\omega_i)$$ Θ_r: clustering



f( ) : posterior type - normalization operator
likelihood type - log operator

Training Scheme: PDBNN

- **Expectation Maximization(EM) Training**
  - Expectation Step(E-step):

$$Q_i(\Theta_r^{(j)}, \Theta) = E[\sum_t \sum_r z_r^{(t)} \log\Big(P(\Theta_r|\omega_i)p(x^{(t)}|\omega_i, \Theta_r)\Big)]$$
$$= \sum_t \sum_r h_r(t) \log\Big(P(\Theta_r|\omega_i)p(x^{(t)}|\omega_i, \Theta_r)\Big)$$

  - $$h_r^{(j)}(t) = \frac{P(\Theta_r|\omega_i)^{(j)} p(x^{(t)}|\omega_i, \Theta_r)}{\Sigma_\kappa P(\Theta_\kappa|\omega_i)p(x^{(t)}|\omega_i, \Theta_\kappa)}$$

  - Maximization Step(M-step):
    Maximize Q w.r.t. $P(\Theta_r|\omega_i)$ under probability normalization,

$$P(\Theta_r|\omega_i)^{(j+1)} = (1/N) \sum_{t=1}^{N} h_r^{(j)}(t)$$

– Maximization Step(M-step):

Maximize Q w.r.t. $P(\Theta_r|\omega_i)$ under probability normalization,

$$P(\Theta_r|\omega_i)^{(j+1)} = (1/N) \sum_{t=1}^{N} h_r^{(j)}(t)$$

Maximize Q w.r.t. mean and variance,

$$w_{rd}^{(j+1)} = (1/\sum_{t=1}^{N} h_r^{(j)}(t)) \sum_{t=1}^{N} h_r^{(j)}(t) x_d^{(t)}$$

$$1/\alpha_{rd}^{(j+1)} = (1/\sum_{t=1}^{N} h_r^{(j)}(t)) \sum_{t=1}^{N} h_r^{(j)}(t)(x_d^{(t)} - w_{rd}^{(j)})^2$$

EBF Approximation of Gaussian Distribution

Log-likelihood function of a D-dimensional Gaussian distribution with uncorrelated features:

$$\log p(x|\Theta_r, \omega_i) = -\frac{1}{2}\sum_{d=1}^{D}\frac{(x_d - w_{id})^2}{\sigma_{id}^2} - \sum_{d=1}^{D}\log \sigma_{id} - \frac{D}{2}\log(2\pi)$$

Elliptic Basis Function:

$$\psi(x, \omega_i, \Theta_r) = -\frac{1}{2}\sum_{d=1}^{D}\alpha_{id}(x_d - w_{id})^2 + \theta_i$$

Training Scheme of DBNN

- **Gradient Ascent Training** $p(x \mid \Theta_r, \omega_i)$

  Updating means and variances:

$$\pm \frac{\partial y_i(x)}{\partial w_{rd}} = \pm h_r(x) \cdot \alpha_{rd}^2 (x_d - w_{rd})$$

$$\pm \frac{\partial y_i(x)}{\partial \alpha_{rd}} = \pm h_r(x) \cdot \frac{1}{2}\left(\frac{1}{\alpha_{rd}} - (x_d - w_{rd})^2\right)$$

Here $\pm$ are for reinforced and anti-reinforced learning

for training (1) NEURAL WEIGHTS $\{w_{rd}, \alpha_{rd}\}$.

(2) ACCEPTANCE THRESHOLD $\{T_i\}$.

Supervised Training for GS learning

Reinforced Learning: $w^{(j+1)} = w^{(j)} + \eta \nabla \phi(x, w)$

Antireinforced Learning: $w^{(j+1)} = w^{(j)} - \eta \nabla \phi(x, w)$

$$\frac{\partial \phi(x(t), w)}{\partial w_{rd}}|_{w=w^{(j)}} = h_r^{(j)}(t) \cdot \beta_{rd}^{(j)}(x_d(t) - w_{rd}^{(j)})$$

$$\frac{\partial \phi(x(t), w)}{\partial \beta_{rd}}|_{w=w^{(j)}} = h_r^{(j)}(t) \cdot \frac{1}{2}\left(\frac{1}{\beta_{rd}^{(j)}} - (x_d(t) - w_{rd}^{(j)})^2\right)$$

$d(t) = \begin{cases} T - \phi(x(t), w), & \text{if } x \in w \\ \phi(x(t), w) - T, & \text{if } x \notin w \end{cases}$

$l(d) \approx \frac{1}{1 + \exp^{-d/\epsilon}}$

$$P^{(j+1)}(\Theta_r|\omega) = (1/N)\sum_{t=1}^{N} h_r^{(j)}(t)$$

$$T^{(j+1)} = T^{(j)} - \eta l'(d(t)) \quad \text{if } x(t) \in \omega \quad \text{(reinforced learning)} \qquad T \downarrow$$

$$T^{(j+1)} = T^{(j)} + \eta l'(d(t)) \quad \text{if } x(t) \notin \omega \quad \text{(antireinforced learning)} \qquad T \uparrow$$

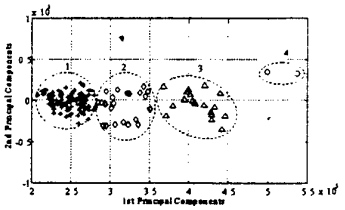
Extension to Multi-Class PDBNN

$$\phi(x(t), w_i) = \log p(x(t)|\omega_i) = \log[\sum_r P(\Theta_{r|i}|\omega_i) p(x(t)|\Theta_{r|i}, \omega_i)]$$

where $w_i \equiv \{\mu_{r|i}, \Sigma_{r|i}, P(\Theta_{r|i}|\omega_i), T_i\}$

Unsupervised learning: k-mean,VQ, EM, ...

Classify traffic patterns by K-mean, EM, etc.



Criterion of K-mean (VQ) Clustering Algorithm

Minimize:

$\Sigma_t \Sigma_r h_r(t)(\mathbf{x}_t - \mathbf{w}_r)^2$

Membership Function: $h_r(t) =$ 0 or 1

K-mean or VQ

Animated Iteration

Given 5 members, to be divided into 2 clusters.

3 iterations

Initial Iteration

After 1 E-M iteration

After 2 E-M iteration

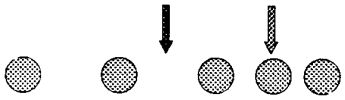
Converges after 3 E-M iteration

In Slow Motion

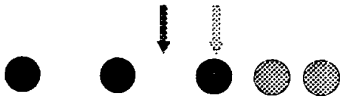
Initial Iteration

E-Step: To determine the initial centroids

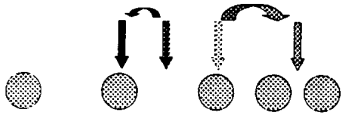
E-Step just completed: now ready for M-Step
i.e. the new memberships.



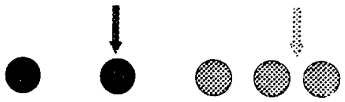
M-Step (of the 1st iteration) just completed:



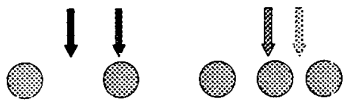
E-Step (of the 2nd iteration) just completed:



M-Step (of the 2nd iteration) just completed:



E-Step (of the 3rd iteration) just completed:



M-Step (of the 3rd iteration) just completed:

Convergence: Final Clustering

EM for NN

Criterion of EM Algorithm

- ***Minimize no longer a simple function*** *as:* $\Sigma_t \Sigma_r h_r(t)(\mathbf{x}_t - \mathbf{w}_r)^2$
- *Membership Function is continuous:* $h_r(t) \in [0, 1]$

Criterion of Expectation-Maximization

Maximize a log-likelihood function

$$\sum_t \log p(x_c)$$

$$\rightarrow \sum_t P\left((r)p(x_t|c_r)\right)$$

$$\rightarrow \sum_t \sum_r h_r(t) \log \left\{P((r)p(x_t|c_r)\right)$$

$$\rightarrow \sum_t \sum_r h_r(t) \log \left\{p(x_t|c_r)\right\}$$

$$+ \sum_t \sum_r h_r(t) \log \left\{P(c_r)\right\}$$

EM Iterations

Initial Iteration

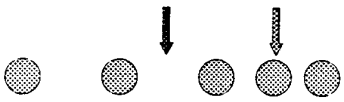
Initial Membership Weightings

.25	*.25*	*.25*	*0*	*.25*
0.0	0.0	0.0	1.0	0.0

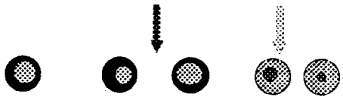
E-Step: To determine the initial centroids

.25	*.25*	*.25*	*0*	*.25*
0.0	0.0	0.0	1.0	0.0

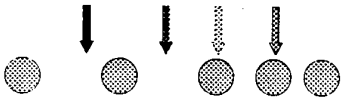
E-Step just completed: now ready for M-Step
i.e. the new memberships.



M-Step (of the 1st iteration) just completed:

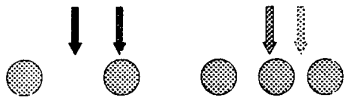


E-Step (of the 2nd iteration) just completed:



M-Step (of the 2nd iteration) just completed:

E-Step (of the 3rd iteration) just completed:

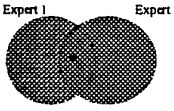


M-Step (of the 3rd iteration) just completed:

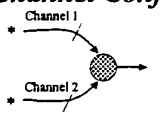
Final Convergence possible only
when T approaches 0

EM Applications

- ***Uncertain Clustering/ Model***



- ***Channel Confidence***

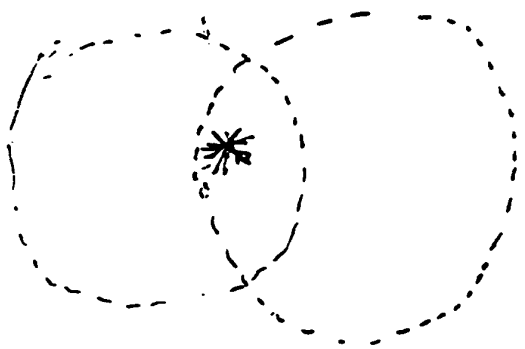


More Math. On EM

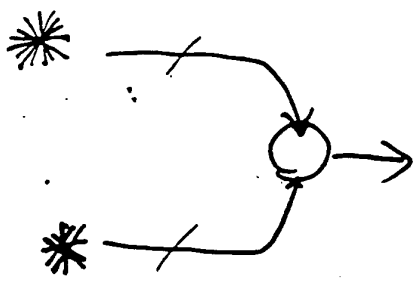
Model Based EM

Expectation - Maximization

UNCERTAIN CLUSTERING / MODEL, or



Channel Confidence



$$\sum_t \log p(x_t)$$

$$\rightarrow \sum_t \log \sum_r P(c_r)\, p(x_t \mid c_r)$$

$$\sum_{t,r} h_r(t) \log \left\{ P(c_r)\, p(x_t \mid c_r) \right\}$$

$$\rightarrow \sum_{t,r} h_r(t) \log p(x_t \mid c_r) + \sum_{t,r} h_r(t) \log P(c_r)$$

$h_r(t)$

RBF: $\| x_t - m_r \|^2$

m_r = CENTROID

*)

$$\text{Max} \sum_r \log\left(\sum_{t=1}^{N} h_r^{(j)}(t)\right) \cdot \log P_i(\theta_r | w)$$

$$P(\theta_r | w) = \frac{1}{N} \sum_{t=1}^{N} h_r^{(j)}(t)$$

Th: Solution for

Max $-\sum w_i \log \gamma_i$,

under constraint $\sum \gamma_i = 1$,

is $\gamma_j = \frac{w_j}{\sum w_j}$

*)

$$\text{Max} \; -\sum_{r=1} \sum_{t=1}^{N} h_r^{(j)}(t) \cdot \sum_{d=1}^{D} \frac{(X_d(t) - W_{rd})^2}{\sigma_{rd}^2}$$

equiv.

$$\text{Max} \; -\sum_r \sum_{t=1}^{N} h_r^{(j)}(t) \left(X_d(t) - W_{rd}\right)^2 =$$

equiv.

$$\text{Min} \; -2\sum_r \left(\sum_{t=1}^{N} h_r^{(j)}(t) X_d(t)\right) \cdot W_{rd} + \sum_r \left(\sum_{t=1}^{N} h_r^{(j)}(t)\right) \cdot W_{rd}^2$$

$$\Rightarrow W_{rd} = \frac{1}{\sum h_r^{(j)}(t)} \sum_{t=1}^{N} h_r^{(j)}(t) X_d(t)$$

Applications

Applications of NN to Intelligent Multimedia Processing

DBNN Applications

- *Automatic Music Note Recognition*
- *Mammogram Diagnosis*
- *Texture Segmentation*
- *Face Detection*
- *Money Recognition*
- *Multimedia Library*

Temproal DBNN Applications

- *Prediction Error: ECG application*
- *Likelihood Function : all HMM appplications*

RBF-BP NN for Dynamic Resource Allocation

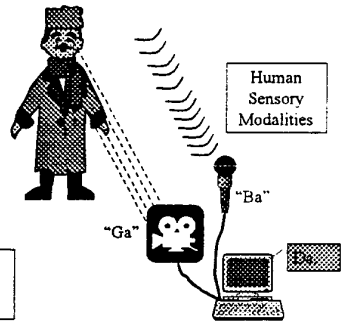
- *use content to determine renegotiation time*
- *use content/ST-traffic to estimate how much resource to request*

Neural network traffic predictor yields smaller prediction MSE and higher link utilization.

RBF-BP NN for CBIR

Channel Fusion

Sensor Fusion



Fusion Experimental Example

3D Toy Car Recognition

Implementation

Parallel Implementation of NN

- ***Matrix based operations***
- ***Rich Regular and Massive Parallelism can be exploited***

Dedicated hardware would be very cost-effective.

Regular Computations of NN

Learning Scheme	$\Delta w_j = \eta(\alpha x_i - \beta w_j)$
VQ = k-mean	$\Delta w_j = \eta(x_i - w_j)$
SOFM	$\Delta w_j = \eta(x_i - w_j)$
PCA	$\Delta w_j = \eta[(\mathbf{x}^T\mathbf{w})x_i - (\mathbf{x}^T\mathbf{w})^2 w_j)]$
ICA	$\Delta w_j = \eta(y^3 x_i - y^4 w_j)$

Intelligent Processing for Multimedia Internet and Wireless Communication

S.Y. Kung
Princeton University

Contents

Objectives

•Information Processing World is converging to (1) wireless, (2) internet, and (3) multimedia.

•Justification of intelligent multimedia technologies for the internet computing/communication era.

•Illustrate how intelligent integration of signal processing and neural net techniques could be a versatile tool to a broad spectrum of multimedia applications.

•Telecommunication Evolution: Broadband Wireless Access

•Demonstration of real systems of neural network applications.

• Demonstration of real systems in support of the MPEG-4 and MPEG-7 coding standards.

Trends of Information Processing

- *Wireless*
- *Internet*
- *Multimedia*

Ubiquitous Internet Data Access

Internet data access is rapidly becoming a vital part of everyone's daily life.

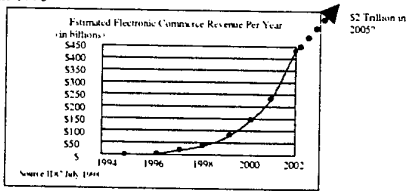
With the current growth rate, there will soon be more internet users than the telephone users (nearly one billion by some prediction).

Consider 40M users **X** 400M web-sites **X** usage of half hour per day. This will result in an enormous and rapid expanding demand on internet bandwidth, on both the backbone and access.

An estimate by Gardener suggests that wireless communication will possess 25\% of internet bandwidth in few years.

Worldwide Internet Commerce

- US E-commerce
  - 1998 is estimated to be $30B to $50B; business-to-business commerce
  - 57% of companies plan to implement E-commerce by end of 1999
- Worldwide revenue is expected to grow to $220B in 2001 from $2.6B in 1996

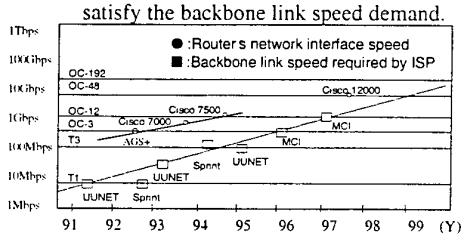


Internet Physical Infrastructure: Wired/Wireless Networking

Networking technologies are critical for multimedia database system to support interactive dynamic interfaces. They will involve both logical connection to support information sharing and physical connection via computer networks and data transfer.

Backbone Link Speed v.s. Router Interface Speed

Routers with Gigabit interfaces developed to satisfy the backbone link speed demand.



... Internet Enters the Mainstream

- In just 4 years, since 1995
  - Explosive growth
    - From a limited number of users to >70 million interconnected computers
    - Expanding to Intranet and Extranet
  - From a local, static, text-based medium to distributed, interactive, graphically rich multimedia
  - Opportunities for New technologies and rich applications
    - Push technologies, Internet telephony, Internet fax, Internet advertising, electronic commerce, WWW, e-mail, video on demand, interactive games and entertainment, etc.

Internet Information Infrastructure: Multimedia Processing

Multimedia Processing Systems must successfully combine digital video and audio, text animation, graphics, and knowledge about such information units and their inter-relationships in real-time.

Modern computing technology in the internet era should support interactive and intelligent processing that transforms and transfers information ubiquitously and in real-time speed.

Ubiquitous Multimedia

Distributed application means that

- Computing
- Content
- Consumer

will be separated by long distances.

Future internet computing must depend on constant connectivity (Communication) on a large network.

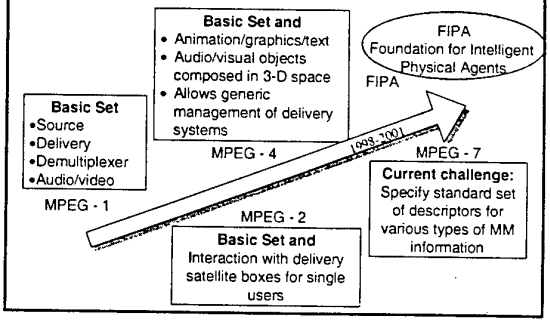
Convergence of 4 C's:

- Computer
- Communication
- Consumer
- Content

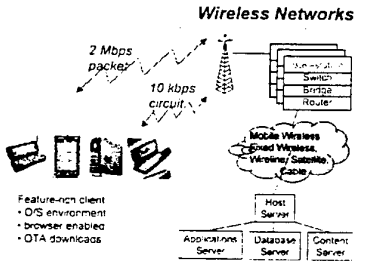
MPEG-7 Overview

"MPEG-7 will specify a standard set of descriptors that can be used to describe various types of multimedia information ... To allow for fast and efficient search for materials of a user's interest."

Multimedia Technology Standards



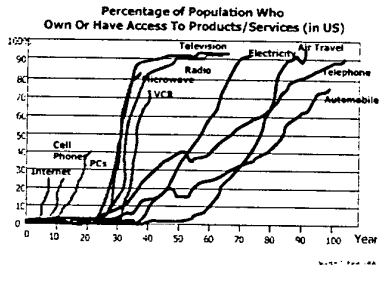
Wireless Communication



Wireless Market

- Wireless will catch up wireline in both telephony and internet
- All wireless predictions have fallen short of real progress
- Basestation market will reach (S65B/year by 2002)
- Wireless will have 30% + global telecommunication market by 2002
- data portion will grow from the current 2% to 25% by 2002

NTT DoCoMo has 4M users in one year
Erickson and Norkia 600M users by 2004
Sonera is joint telecom and ISP company with 35%/65% on value-added/voice



Wireless over Internet World

- Bandwidth,
- Portability,
- Security,
- Globalization

Technological Realization

- (1) Digital solution for Wireless LAN.
- (2) Digital solution for last mile BWA.
- (3) Digital radio processing, i.e. software radio

Two Central Problems on:

Artificial Intelligence

Hofstader: What is "Artificial"?

Knuth: What is "Intelligence"?

"Questing for the essence of Mind and Pattern", Hofstader, New York: Basic, 1995

Intelligent Machines

1842: A remark on the "analytical engine"

by Charles Babbage

"... the machine is not a thinking being, but a simply automaton..."

1997: Deep Blue defeated Garry Kasparov

Is it ...

•An advance of our understanding of intelligence?

Or merely

•the power of 200 Million possible chess evaluations per second?

AI Community's Consensus:

"intelligence requires knowledge"

Neural Network Community's Consensus:

"neural networks require training examples."

Design of intelligent machines hinges upon clever organization (representation and memorization) of given knowledge or training examples.

Learning (Training) Phase

Supervised: $(I,O) \Rightarrow (W)$

Unsupervised: $(I) \Rightarrow (W)$

Retrieval (Search) Phase:

$$(I,W) \Rightarrow (O)$$

Input → Neural Network → Output

$$I \Rightarrow \boxed{W} \Rightarrow O$$

Brain Activities vs. Performance

Group A: Low Activity

Group B: High Activity

Observations

Explanations?

Brain Activities vs. Performance

Group A: Low Activity

Group B: High Activity

Observations

Explanations?

Synergistic Modeling and Applications of Hierarchical Fuzzy Neural Networks

By S.Y. Kung, et al., Proceedings of the IEEE, Special Issue on Computational Intelligence, Sept. 1999

Hierarchical Representation

Fuzziness

Hierarchy

Multi-resolution

+ E-M ⇨ On Experts → **DBNN**

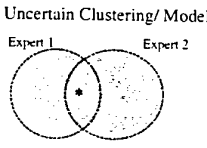
+ DOTs ⇨ On Skeletons → **Shape Query**

Application Examples of DBNN

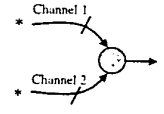
- Automatic Music Note Recognition
  - MMSP '99 - Demonstration
- Mammogram Diagnosis
- Texture Segmentation
- Face Detection
- Money Recognition
- Multimedia Library

Expectation-Maximization

Uncertain Clustering/ Model



or Channel Confidence



$$\sum_t \log p(x_t)$$

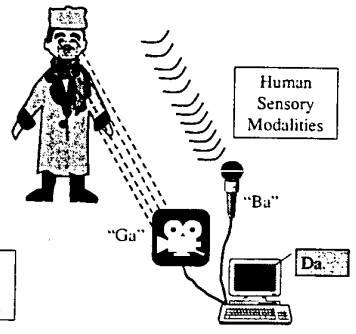
$$\rightarrow \sum_t P((r)p(x_t|c_r))$$

$$\rightarrow \sum_t \sum_r h_r(t) \log\{P((r)p(x_t|c_r))\}$$

$$\rightarrow \sum_t \sum_r h_r(t) \log\{p(x_t|c_r)\}$$

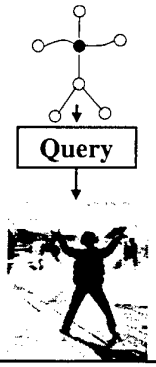
$$+ \sum_t \sum_r h_r(t) \log\{P(c_r)\}$$

Sensor Fusion



Video Query Problem Description

- For video warehousing in MPEG-7, a way for users to access the video data
  - Shape as a intuitive "handle" to query the database
- Our solution:
  - Derive "skeletons" from VOP, comparisons of "implicit structure" as query tool



Hierarchical Representation

Fuzziness
Hierarchy
Multi-resolution

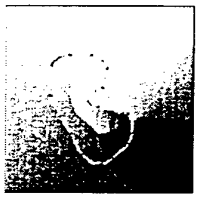
+ E-M On Experts ⇨ DBNN

+ DOTs On Skeletons ⇨ Shape Query

Definition of Voronoi Order

Treat a simple closed contour as a infinite string of Voronoi centers along that contour. define a ϕ function that maps a point to its corresponding Voronoi center.

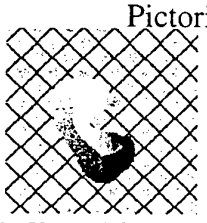
$$\phi(\vec{y}) = \min\left(\arg(\min(\|\vec{y} - \vec{x}(s)\|))\right) \quad C = \{\vec{x}(s) \mid 0 \le s < 1\}$$

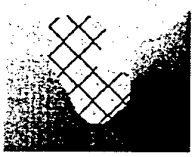


ϕ creates a warped space where one axis is the projection onto C and the other is the distance from C.

$$\vec{y} = (x, y) \xrightarrow{Warp} (\phi(\vec{y}), \|y - C\|)$$

Pictorial View





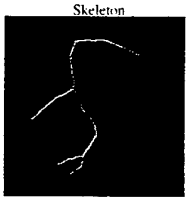
- The Voronoi Order of a Contour is the implication of its shape onto image space
- Voronoi Order has dual purpose:
  - Extract Shape Structure (MPEG-7) **or**
  - Express Structural Constraints (MPEG-4)

Voronoi Order Skeleton

- Circular Differential of the Voronoi Order
  - Find discontinuities in Voronoi Order

Voronoi Order

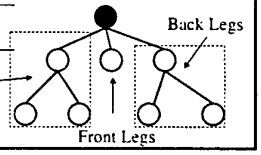
Voronoi Order Skeleton



- Weighted Version of MAT, Voronoi Skeleton

Ordered Tree Representation

VOS	→ Ordered Tree
Branching Points	Nodes
Segments	Edges
High to Low Weighting	Parent-Child
Max. Weighting	Root
Ordering	CCW Turning

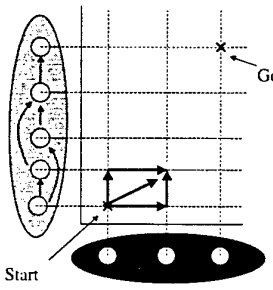


DAG

=

Directed Acyclic Graphs

Path Matching Between DAGs



Induction #2:

Find Optimal Score from Start to Goal through a series of Edge matches

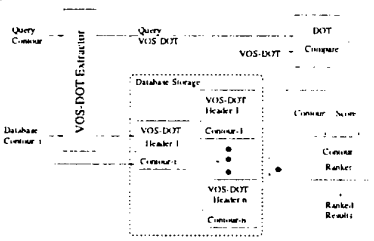
Dynamic Programming: Calculate next optimal score Based on previous optimal

= In Set of Optimal Scores

Ref: IEEE Trans. On Sig. Proc., Nov. 1997, v.45,#11,pp. 2701-8

Shape Query System

Simple Linear Query, w/ DOT-Compare as Similarity



On Two MPEG-7 databases :
Animals (~120)
Complete (~1400)

- Our Comparison algorithm allows complete flexibility at joints
- Each edge contains length, curvature, thickness, weight info.
- Multi-resolution comparison

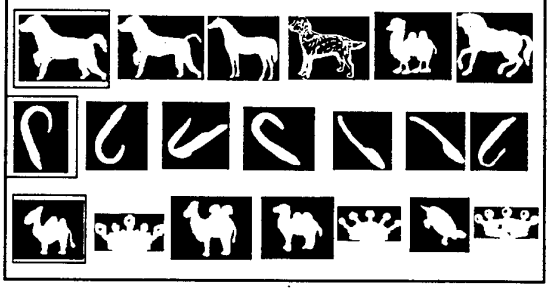
Results: Animal Subset

Shapes Database ⇒ Query ⇓	(A)	(B)	(C)	(D)	(E)	(F)	(G)	(H)
(α)	**1st**	6th	4th	**2nd**	5th	7th	**3rd**	8th
(β)	**2nd**	6th	7th	8th	5th	**3rd**	4th	**1st**
(γ)	5th	**2nd**	6th	**1st**	4th	**3rd**	8th	7th
(δ)	**2nd**	4th	**3rd**	**1st**	6th	8th	5th	7th

Figure 3: Results of preliminary shape query test on a small shape database using VOS and DOT as representation and the DOT-compare as similarity measure; note database and query shapes are all distinct.

Results: Large Database (~1400)

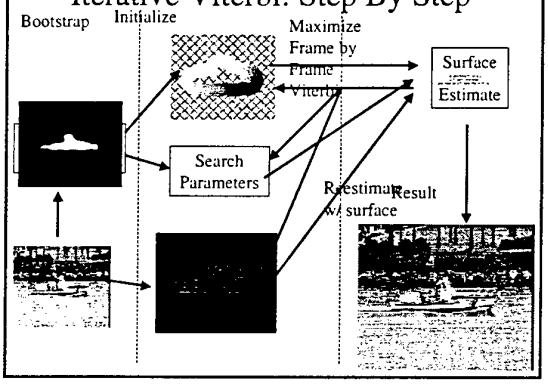
- Query in Red Box
- Top Ranking results from left to right



Summary: MPEG-7 Query System

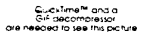
- Structural Comparison
  - Invariance to certain motions
  - Robustness to noise
  - Ability to do structure-wise warping
    - Bending at joints, etc.
  - Problems:
    - Aliasing: Focus is on structure, not necessarily on contour itself
    - Computationally expensive, compared to feature vector schemes

Iterative Viterbi: Step By Step



Results

The final results (in the best case) looks something like this a surface that moves toward edge and motion discontinuities, but maintains a structural balance



However, we will see that even in simple examples, our analysis fails to address all the difficulties of visual processing.

Multimedia System Implementation

A Deep Blue Lesson...

Multimedia Signal Processing

Multimedia Signal Processing (MSP), which represents a major part of the latter category, involves the joint processing of digital information in various representations. It covers a very broad spectrum of applications:

- Audio & speech processing: audio compression (G.711, G.722, G.728), Dolby surrounding;
- Image & video processing: resolution conversion, image enhancement, image restoration, image compression (JBIG, JPEG), video compression (MPEG).
- Content-based indexing & retrieval: feature extraction (fillet coordination, moment, histogram), pattern recognition, face detection/recognition, fusion of multi-modality.
- 2-D & 3-D graphics: volume rendering, modeling transformation, texture mapping, shading, shadowing, ray-tracing, computer-assisted animation, virtual reality, etc.

Implementation of Multimedia Processors

For multimedia processing, no algorithm is really useful unless there is an efficient implementation for it. On the device front, the VLSI technology will soon reach more than 100 million transistors in a chip, implying tremendous computation power for multimedia applications. Nevertheless, multimedia architectures have to offer highest integral performance to enable joint processing of multimedia tasks. As MSP applications crucially demand real-time processing, the required performance levels cannot be attained with conventional processor architectures. From algorithmic perspectives, important characteristics of the MSP algorithms can be summarized as following:

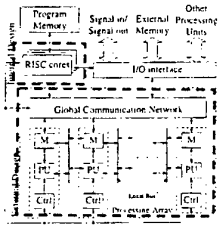
- Intensive computation for highly regular operations;
- Intensive I/O or memory accesses, data reusability, and data locality;
- High control complexity in less computational intensive tasks;
- Frequent encounters of operations smaller than the data-width;
- Adaptiveness via learning examples.

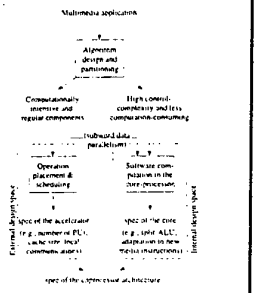
Multimedia signal processor design should also be driven by available VLSI technologies. There are two important features in VLSI technologies.

1. *External memory is slow:* There is a huge gap between memory speed and processor speed. Therefore, a high-speed on-chip data memory (register file, cache, RAM) is necessary to bridge the gap. For example, most of the announced programmable media processors (as listed in Table 1) use 16 KB to 64 KB on-chip data memory.

2. *Long-distance communication is slow:* Because the feature size of the processing technology is getting smaller and smaller, more and more of the signal delay is on the wire than the transistor [20]. Long-distance and global (one to many) communication takes longer and is more expensive than local communication. Hence, for a sound design, it is important to make use of local communication channels and it is necessary to support local communication efficiently.

Multimedia Architecture And Design Approach





- Computation-demanding parts efficiently conducted on the processing array
- Control-intensive part easily executed on the core-processor.

Critical Interfaces

- Machine to Machine: Internet
- Human to Machine: User Interface
- Human to Human: Multimedia

More on BWA!

Princeton University

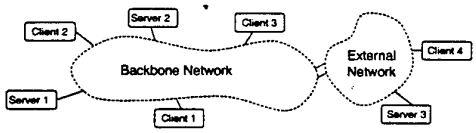
Video Traffic Modeling Using Content & Short-term Traffic Statistics

S.Y. Kung
Princeton University

This work is a joint research by Min Wu, Rob Joyce, S.Y.Kung of Princeton Univ., and Hau-San Wong, Ling Guan of Univ. of Sydney.

Motivation

- Shared network, diverse sources



- Non-constant bandwidth requirements
- Changing number of sources/receivers
- Need to treat problem dynamically, especially for high-bandwidth sources

Motivation (cont'd)

- Constant Bit Rate (CBR) vs. Variable Bit Rate (VBR)
- Dynamic resource allocation is desired

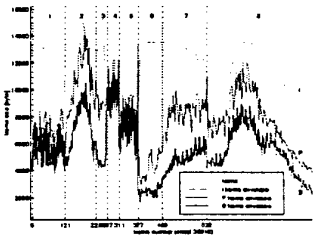
	bitrate	quality	coding efficiency	networking transmission
CBR video	constant	variable	low	easy
VBR video	variable	- constant	high	difficult

Background (Video)

- Constant Bit Rate (CBR) vs. Variable Bit Rate (VBR)
  - CBR
    - Simpler traffic modeling & allocation
    - Less efficient use of bandwidth per stream
    - MPEG-1/2
  - VBR
    - Consistent quality
    - Efficient use of bandwidth (per stream / single user)
    - Difficult for resource allocation (multiuser)
    - MPEG-1/2/4

VBR Video Example

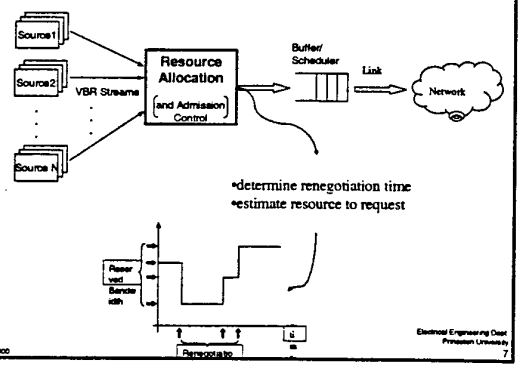
- "Forrest Gump" (Bocheck-Chang)



Outline

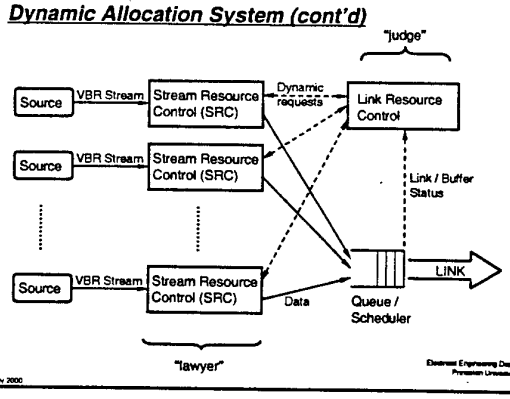
- Background and overview of dynamic resource allocation
- NN-based approach
  - use content to determine renegotiation time
  - use content/ST-traffic to estimate how much resource to request
- Systematic approach to select the best features for prediction
- Experimental results
  - prediction error
  - network utilization efficiency

Dynamic Allocation System



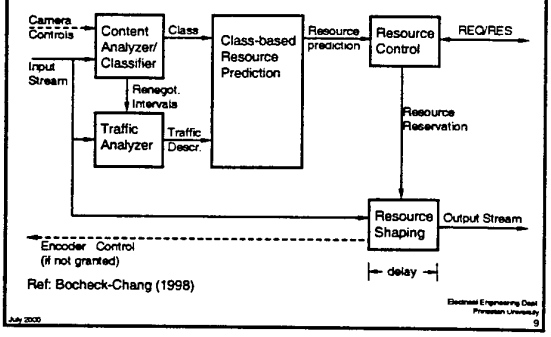
July 2000 — Electrical Engineering Dept, Princeton University — 7

Dynamic Allocation System (cont'd)



July 2000 — Electrical Engineering Dept, Princeton University — 8

SRC Module

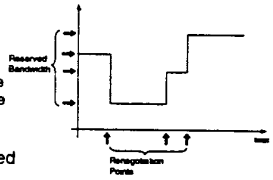


Ref: Bocheck-Chang (1998)

July 2000 — Electrical Engineering Dept, Princeton University — 9

Approaches

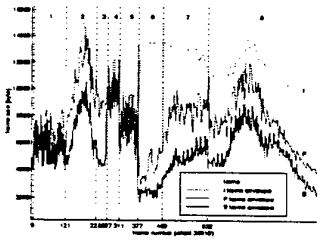
- Need to determine
  - determine renegotiation time
  - estimate how much resource to request
- When request is not granted
  - drop packets; transcoding; etc.
- Conventional traffic handling
  - bit stream level
  - adaptive or equally spaced renegotiation instants
  - Renegotiation as *reaction* to link load
  - Transcoding / traffic shaping according to bit budget



July 2000 — Electrical Engineering Dept, Princeton University — 10

Close Relationship Between Content & Traffic

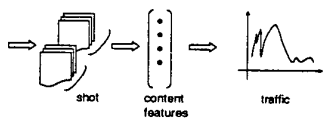
- "Forrest Gump" (Bocheck-Chang)



July 2000 — Electrical Engineering Dept, Princeton University — 11

Why Involving Content?

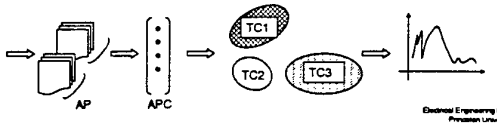
- Content-based approach
  - visual content useful for *predicting* traffic
  - "natural" renegotiation points: shot boundaries
- Prior work:
  - Bocheck-Chang (Columbia Univ.), etc.



July 2000 — Electrical Engineering Dept, Princeton University — 12

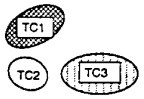
Why Involving Content?

- Strong correlation among
  - video content, network resource (b.w.), video quality
- How to obtain the relationship?
  - divide video into activity periods (AP) or scenes
  - represent each AP with content descriptor (APC)
  - classify traffic patterns of AP w.r.t. APC
  - associate AP w/ typical traffic param. of its traffic class



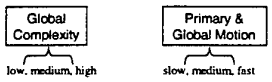
Traffic Classification: Basic Questions

- Determine traffic descriptor
- Determine content descriptor
- Relationship between traffic and content



Traffic Classification: Chang-Bocheck

- Candidate content features (empirical & intuitive)
  - finite levels for each feature
  - their combinations are the possible classes
  - every content class is assigned a typical traffic pattern
  - problem: too many classes (solution to follow)

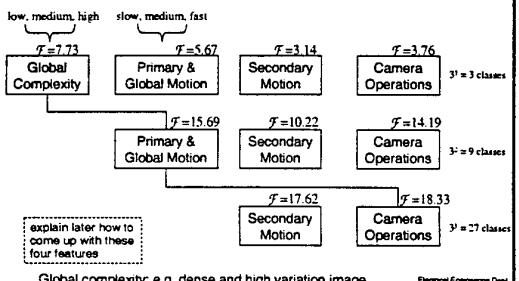


Feature Evaluation Rules

- Criterion
  - intra-class: as compact as possible (small distance)
  - inter-class: as separate as possible (large distance)
- "classification consistency": $\mathcal{F}$
  - total inter-distance / total intra-distance
  - the larger, the better
- Use what distance measure?
  - Euclidean distance between traffic descriptors

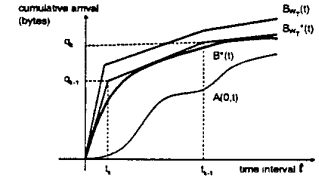
Feature Evaluation Rules (cont'd)

- Classification Tree



Global complexity: e.g. dense and high variation image

Traffic Descriptor: D-BIND (Zhang & Knightly)



D-BIND component for small (large) t reflects peak (average) rate

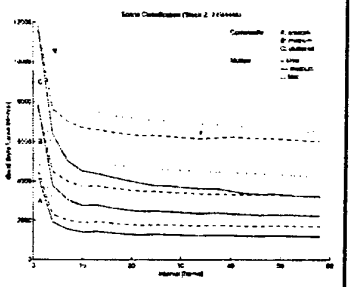
- Deterministic Bounding Interval Dependent model
  - A[τ, τ+t]: cumulative source arrivals during the interval
  - B*(t): empirical envelope of A[τ, τ+t]
    - $B^*(t) = \sup_{\tau \geq 0} A[\tau, \tau+t]$ (smallest upper bound)
  - $B_{W_T}(t)$: cont's piecewise linear func. bounding B*(t)
  - $W_T = \{ (q_k, t_k) \mid k = 1 \sim P \}$ (arrival bytes & time pairs)
  - $R_T = \{ (r_k, t_k) \mid k = 1 \sim P \}$, where $r_k = q_k / t_k$ (bounding rate)

Traffic Descriptor: D-BIND (cont'd)

- For deterministic service (worse-case guarantee)
  - somewhat conservative
- Advantages
  - Measure b.w. and burst over wide time-scale range
  - good tradeoff between complexity and efficiency

Resource Clustering & Mapping Result

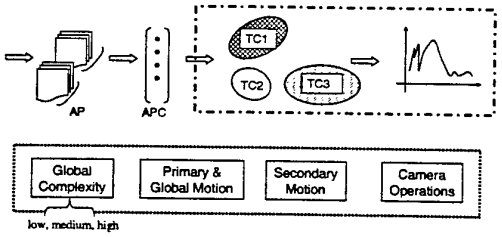
- Two features
  - $3^2 = 9$ classes
- Global complexity
  - smooth, medium, cluttered
- Primary & global motion
  - slow, medium, fast



D-BIND of 9 Classes (Bocheck &Chang)

Content Analysis

- determine renegotiation points
- extract content features for traffic prediction



Content Analysis

- Objectives
  - Temporal Segmentation (renegotiation points)
  - Spatial Segmentation (object hierarchy)
  - Ignore features irrelevant to traffic
- On-line ***Compressed-Domain Processing***
- Difference from traditional content analysis
  - Understanding the image/video is unnecessary
  - Some inaccuracies are tolerable; only need be "good enough"

Compressed Domain Processing

- Motivation
  - Speed (no IDCT) / storage
  - Duplication of effort
- Directly available:
  - Reduced resolution frame
    - low order transform coeffs.
  - "Complexity" / Texture
    - high order transform coeffs.
  - Camera/object motion estimates
    - Block prediction or skipping
    - Motion vectors
  - Objects (MPEG-4)

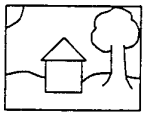
DC frame

Temporal Segmentation

- Activity Period / "Scene"
  - "video segment during which the visual content does not substantially change"
  - AP boundaries are renegotiation points
- Frame distance metrics
  - Pixel domain: $d_k^{(1)} = \frac{1}{N}\sum_i (f_k(i) - f_{k-1}(i))^2$
  - Histogram domain: $d_k^{(2)} = \frac{1}{M}\sum_j g(F_k(j) - F_{k-1}(j))$
  - Feature based: edges, other statistics
  - Frame-based classification (Chang/Bocheck)

Spatial Segmentation

- Hierarchical Division into objects



- Attributes:
  - Object size: $\sum_{k=0}^{N-1} 1_{\{b_k \in o_j\}}$
  - Spatial complexity: $\sum_{k=0}^{N-1} \left[1_{\{b_k \in o_j\}} \sum \hat{B}_k \right]$
  - Relative object motion mag.: (direction irrelevant) $\sum_{k=0}^{N-1} 1_{\{b_k \in o_j\}} \left\| \vec{m}_k \right\|$

N is # of blocks
b_k is block k
o_j is object j

Final Content Description

- Scene Type Descriptor: static, panning, zooming
  - From motion vector information
- Quantization ⇒ finite classes
  - Static
  - Learned classification
- Mapping from scene to APC:

$$s_i \rightarrow \{k, l, STD(s_i), VOD(o_1), VOD(o_2), \circledast\}$$

Application-1 Dynamic Resource allocation

- Recall VBR video
  - static bandwidth allocation: low network utilization
  - dynamic allocation: efficient but re-negotiation required
    ⇒ when to re-negotiate? how much resource to request?
- Content-based Approach
  - temporal segmentation to determine re-negotiation time
  - scene class to predict traffic requirement (real-time)
- Traditional Approach
  - renegotiated VBR
  - renegotiated CBR

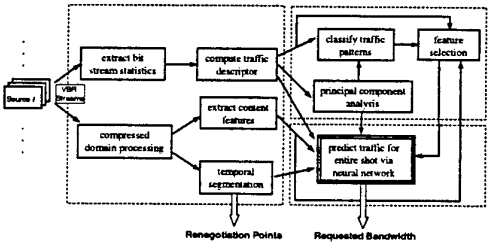
Application-2 Utility Function Estimation

- Utility Function
  - media quality vs. different available bandwidth
  - adapt media communication to varying network cond.
  - e.g., transcode high-rate media into low-rate on the fly

Comments on Bocheck & Chang Approach

- An interesting framework
  - associate visual content with bit stream characteristics
  - "smarter" network communication
- Largely an intuitive approach
  - specified candidate classes for traffic
  - simple classification scheme
  - optimality yet to be justified
- Possible improvement
  - via better traffic predictor
  - via different features

Our Approach



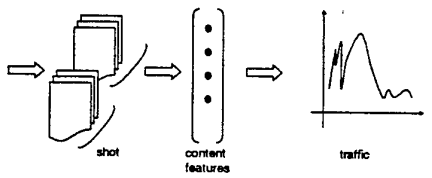
- Why Neural-Network Predictor?
  - the effect of content on traffic is not easy to model parametrically

Application of Neural Networks

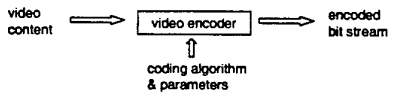
- given a set of candidate features, which ones contain most information and most helpful in classification?
  - PCA ...
- supervised learning: given training pairs of [feature vector, class], what is the classification rule?
  - BP ...
- unsupervised learning: how to classify samples and obtain representative?
  - VQ, EM, ...

Recall: Why Involving Content?

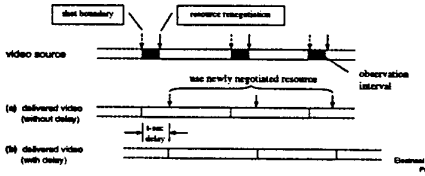
- Content-based approach
  - visual content useful for *predicting* traffic
  - "natural" renegotiation points: shot boundaries



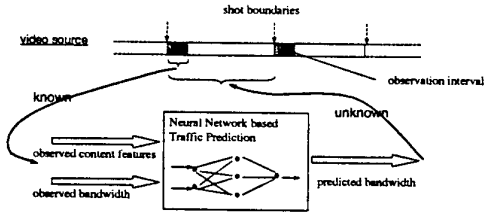
Content Alone is Inadequate!!



- Not all traffic info. can be inferred from content
- Exact short-term traffic is readily available



Our Approach (cont'd)



- Consequence of Prediction Error
  - above true traffic: low utilization
  - below true traffic: compromise service guarantee

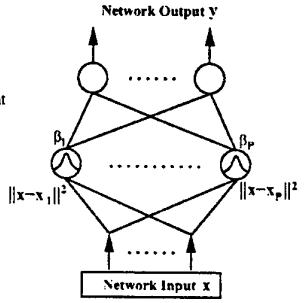
List of Features

CANDIDATE CONTENT AND TRAFFIC FEATURES TO USE IN PER-INTERVAL TRAFFIC PREDICTION

Feature	Description	Feature	Description
1	Frame complexity	10	Mean MV magnitude, lower-right
2	Mean MV magnitude	11	Var. of MV x components
3	Var. of MV directional histogram	12	Var. of MV y components
4	Fraction of intra-coded MBs	13	Cov. of MV x and y comp.
5	Mean magnitude of skewed vectors	14	Var. of MV magnitudes
6	Mean change in MV magnitudes	15	Short-term D-BIND τ_1
7	Mean MV magnitude, upper-left	16	Short-term D-BIND τ_2
8	Mean MV magnitude, upper-right	17	Short-term D-BIND τ_3
9	Mean MV magnitude, lower-left	18	Short-term D-BIND τ_4

SFS/GRNN-based Feature Selection

- The General Regression Neural Network (a special case of he RBF network) which we use to implement Sequential Forward Selection.
- The centers and widths of the Gaussian kernels are deterministic functions of the training data;
- Iterative training is no needed.



GRNN: An RBF Network

$$\mathbf{y} = \sum_{p=1}^{P} \alpha_p \mathbf{y}_p, \quad 0 \le \alpha_p \le 1, \sum_{p=1}^{P} \alpha_p = 1$$

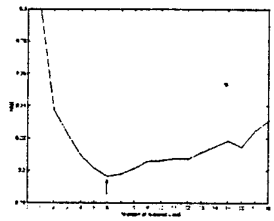
α_p are defined as follows

$$\alpha_p = \frac{\beta_p}{\sum_{p=1}^{P} \beta_p}, \quad p = 1, \ldots, P$$

RBF unit is given by

$$\beta_p = \exp\left[-\frac{(\mathbf{x}-\mathbf{x}_p)^T(\mathbf{x}-\mathbf{x}_p)}{2\sigma^2}\right]$$

Cumulative error for SFS/GRNN feature selection



Feat. Set	Feat.	Feat. Set	Feat.
F_1	15	F_{10}	$8 \cup F_9$
F_2	$18 \cup F_1$	F_{11}	$12 \cup F_{10}$
F_3	$17 \cup F_2$	F_{12}	$13 \cup F_{11}$
F_4	$16 \cup F_3$	F_{13}	$4 \cup F_{12}$
F_5	$1 \cup F_4$	F_{14}	$2 \cup F_{13}$
F_6	$6 \cup F_5$	F_{15}	$10 \cup F_{14}$
F_7	$3 \cup F_6$	F_{16}	$9 \cup F_{15}$
F_8	$5 \cup F_7$	F_{17}	$11 \cup F_{16}$
F_9	$7 \cup F_8$	F_{18}	$14 \cup F_{17}$

- The table shows which features are included after each SFS step. Minimum MSE is achieved after selecting six features,as indicated by an arrow. Our simulation study indicates that with feature selection order different from that listed in this figure, the dropping of MSE is less sharp. However,because of the increasing difficulties of characterizing mapping in high dimensional space, the feature order around and beyond the minimum may not reflect their actual importance, prompting the investigation of the alternative selection scheme for these features.

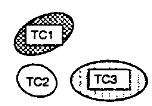
TABLE II

MSE TRAFFIC PREDICTION RESULTS USING CONTENT/TRAFFIC FEATURES SELECTED BY SFS/GRNN, TRAFFIC FEATURES ONLY, AND TWO RANDOM FEATURE SETS.

Feature Subset	no of hidden units	MSE (1st PCA)	MSE (1st and 2nd PCA)
F_6	10	0.0238	0.0277
D-BIND	10	0.0247	0.0281
Random Set 1	10	0.0559	0.0695
Random Set 2	10	0.0426	0.0545
F_4	20	0.0252	0.0298
D-BIND	20	0.0211	0.0279
Random Set 1	20	0.0579	0.0719
Random Set 2	20	0.0488	0.0617

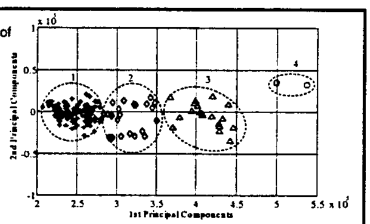
Consistency Based Feature Selection

- Classify traffic patterns
  - K-mean, EM, etc.
- Measure consistency of each feature
  - Criterion
    - intra-class: as compact as possible
    - inter-class: as separate as possible
  - "classification consistency":
    - $C = \frac{\text{mean inter-class distance}}{\text{mean intra-class distance}}$
    - the larger C the better

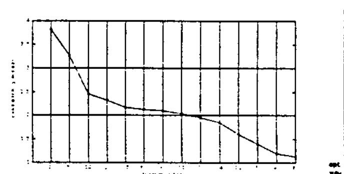


1st and 2nd PCs of shot D-BIND

Four traffic clusters obtained by k-mean

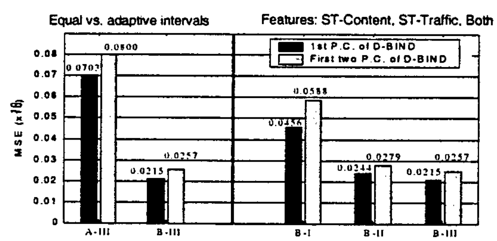


Consistency of content features: feature #1,#5,#2, and #13 are eventually adopted



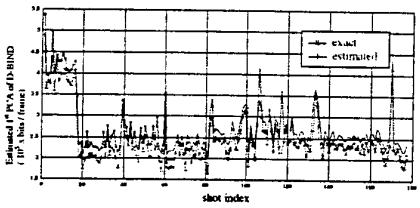
Experimental Result – Prediction MSE

Equal vs. adaptive intervals   Features: ST-Content, ST-Traffic, Both



Renegotiation points
(A) equal-length request intervals,
(B) shot boundaries from temporal segmentation.
Different neural network inputs:
(I) 4-dimension content feature of observed video,
(II) 4-dimension D-BIND of observed video,
(III) both of the above

Prediction MSE (cont'd)



- Approach can be applied to predict other traffic descriptors

Improvement of Network Link Utilization

- Static Peak-Rate
- Renegotiated-VBR Algorithm
- ST Content/Traffic Resource Allocation

Renegotiated-VBR Algorithm

- It is a heuristic dynamic renegotation algorithm using D-BIND descriptors.
- The average R-VBR renegotiation frequency is determined by α,β, K.
  - It raises the reserved bandwidth (described by D-BIND) by a factor α when the real bandwidth exceeds the reserved resource,
  - It lowers the reserved bandwidth by a factor β when the real bandwidth stays below the reserved resource for K frames.
- The R-VBR scheme is a pure bitstream level approach, without using any content information.

ST Content/Traffic Resource Allocation

- **It uses the shot boundaries, obtained from content-based temporal segmentation, as renegotiation points, and use NN traffic predictor to determine how much resource to ask for at each point.**
- **For the 177-shot identified in the video experiment, the D-BIND of each entire shot is computed from the two principal components which are the outputs of NN traffic predictor.**
- **The predicted D-BINDs are used for determining how much bandwidth to ask for in renegotiation. More exactly, 4 content features and 4-dim D-BIND of the observation period as input to a neural network to predict 2 D-BIND principal components, which in turn will determine requested bandwidth.**

Network Link utilization by trace-driven simulation

- Multiple video sources, based on the above mentioned sample video but with random starting points, are multiplexed into a T3 line (link speed c = 45 Mbps).
- A network buffer with maximum capacity Q and FCFS queuing policy are used to smoothen the bursty traffic. When a renegotiation request is received from the n-th source, the **(worst-case) buffer occupancy** is

$$Q_1 = \max\left\{0, \max_{1\le k\le p}\left\{t_k \cdot \left[\sum_i a_i \cdot r_{k(i)} - c\right]\right\}\right\}$$

  - i is source index
  - k is the index of D-BIND components
  - p is the dimension of D-BIND descriptor,
  - $r_{k(i)}$ is the k-th D-BIND components of the i-th source
  - a_i is 1 if the i-th source is admitted and 0 otherwise.

Simulation Results

- The requested resource is allowed only if the current available buffer size is equal to or greater than Q_1.
- Our scheme renegotiates once every 2.48 seconds on average.
- Given bound of rejection probability (e.g., 1%), link utilization is:

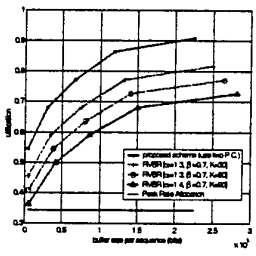
  u = Max number of admittedsources
  .....Number of admittable CBR sources with rate r_{avg}

→ where r_{avg} is the average rate of the entire sequence.

- Our scheme outperforms the R-VBR scheme
  - by 18%, with similar renegotiation frequency
  - by 9% with tripled renegotiation frequency.

Experimental Result – Network Utilization

- "Renegotiated-VBR"
  - bitrate based heuristic approach
  - scale bit-rate request by (1+a) when actual traffic exceed current reservation;
  - scale by (1-b) when well below
  - very conservative



Conclusion

- Systematic approach to select the best features for prediction
- Jointly use bitrate and content information to improve performance of network communication
- Neural network traffic predictor yields smaller prediction MSE and higher link utilization

Video Object Extraction and Representation: Theory and Applications

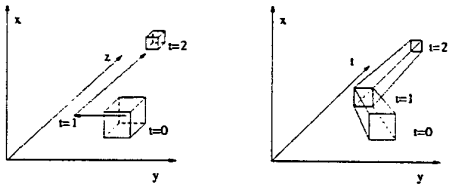
Thesis Defense

I-Jong Lin
Advisor: Prof. S.Y. Kung

Outline

1. Introduction
   - Definition of Video Object and their uses
   - Big picture goals of MPEG-4/7 Standards
2. Voronoi Ordered Space For MPEG-4/7
   - Video Object Extraction (MPEG-4)
   - Video Object Representation (MPEG-7)
3. Video Object Segmentation System (MPEG-4)
   - Surface Formulation/ Optimization
4. Video Object Representation System (MPEG-7)
   - DAG-Ordered Trees and their Application: Query by Shape
5. Conclusion

Physical vs. Video Object



A Video Object (VO) is a Visual Projection of a Physical Object into a Time Series of Images.

A Video Object Plane (VOP) is the Video Object in one Image

Why A Video Object?

Our research is focused toward a Media-Based vs. Vision-Based Goals

Media-Based Goal is for Communication

Tasks
- Create addressable packages of visual content
- Support efficient archival and query of content

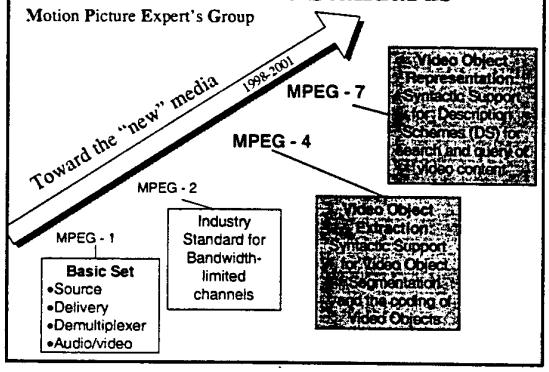
Vision-Based Goal is for Functionality

Tasks
- Reconstruct Physical Objects
- Imitate Human Visual Process

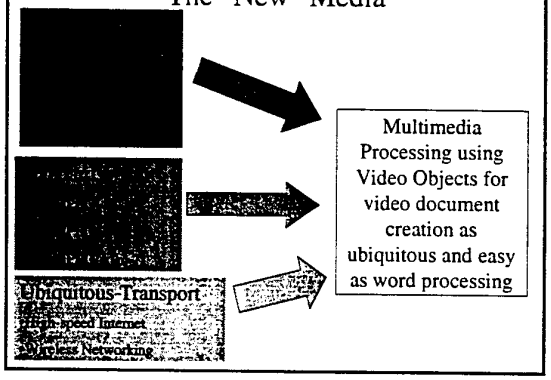
Video Objects are intuitive way of accessing video content supported by the MPEG Video Standards

MPEG Video Standards

Motion Picture Expert's Group



The "New" Media



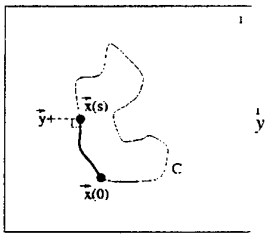
Outline

1. Introduction
   - Definition of Video Object / Media-Based Focus
   - MPEG-4/7 Standards
2. Voronoi Ordered Space
   - Video Object Extraction (MPEG-4)
   - Video Object Representation (MPEG-7)
3. Video Object Segmentation (MPEG-4)
   - Surface Formulation/ Optimization
4. DAG-Ordered Tree Representation
5. Video Object Representation (MPEG-7)
   - Query by Shape
6. Conclusion

Definition of Voronoi Ordered Space

Given a simple closed contour C, find the closest point on C and return its index. (like infinite string of Voronoi centers)

$$\phi(\vec{y}) = \arg\min_{s}(\|\vec{y} - \vec{x}(s)\|)$$

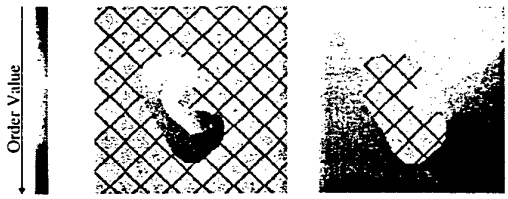


ϕ creates a warped space where one axis is the projection onto C and the other is the distance from C.

$$\vec{y} = (x, y) \xrightarrow{Warp} (\phi(\vec{y}), \|\vec{y} - C\|)$$

A mathematical way of expressing shape

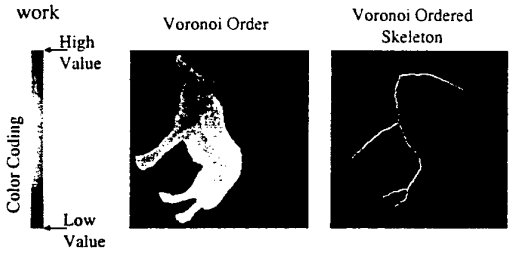
Pictorial View of Voronoi Order



- Voronoi Order has dual purpose:
  Extract Shape Structure (MPEG-7) **or**
  Express Shape Constraints (MPEG-4)

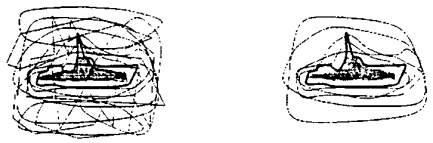
Shape Extraction via Voronoi Order

- Voronoi Order Skeleton: Jumps in Voronoi Order correspond to a physical skeleton
- Use the representation of a physical skeleton for MPEG-7 work



Voronoi Order for VOP Extraction

Voronoi Order restricts the search space of VOP boundaries for MPEG-4



A random set of possible contours

the subset of contours that are consistent with Voronoi Order

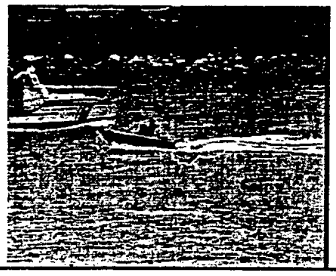
Outline

1. Introduction
2. Voronoi Ordered Space
   - Video Object Extraction (MPEG-4)
   - Video Object Representation (MPEG-7)
3. Video Object Segmentation (MPEG-4)
   - Problem Definition
   - Surface Representation of Video Object
   - Formulation of Problem as Surface Optimization
   - Iterative Viterbi Surface Optimization Algorithm
4. Video Object Representation (MPEG-7)
5. Conclusion

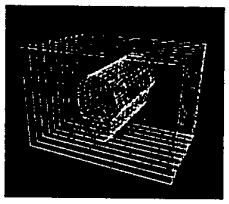
Problem Definition

MPEG-4 Problem: For a given video sequence, separate the video sequence into semantically relevant spatio-temporal regions (VOPs).

Easier Problem: Separate a video sequence into semantically relevant differently moving objects.



Surface Representation of Video Object



When object is in a series of frames, its set of boundaries form a 2-D surface.

Formulation of Video Object Segmentation Problem

For a single isolated object, we create an energy function for a surface S. To find the VOP, we merely calculate:

$$\arg\max_{S}(E(S,I,M))$$

Where I is the Image Sequence and M is object information

$$E(S,I,M) = E_{external}(S,I) - E_{internal}(S,M)$$

External energies fit visual artifacts of the sequence; internal energies fit the assumptions made about the objects.

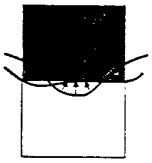
To find a good VO estimate, we balance these two energies.

External Energies

$$E_{visual}(S,I) = \int_{t}\int_{S_t} Edge(I_t(x,y))dsdt$$

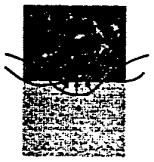
S_t is a contour at time t representing a VOP

Surface is attracted to Visual discontinuities (Edges)



$$E_{motion}(S,I) = \int_{S} |\nabla \vec{v}^{\Gamma}| ds$$

Surface is attracted to Motion discontinuities (Shearing, Occlusion)



Internal Energies

$$E_{smooth}(S) = \int_{S} \|\nabla^2 S\|^2 ds$$

$E_{internal}$ has Other Components:

- Locality
- Containment
- Shape via Voronoi Ordering.

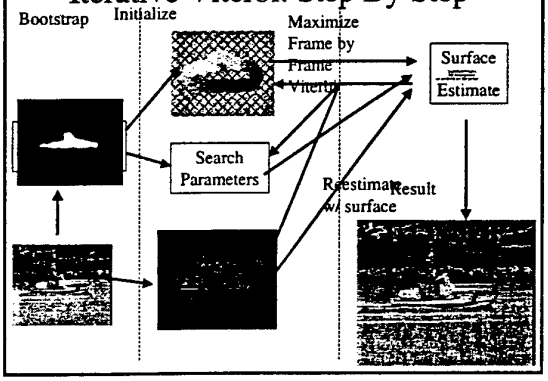
Surface Optimization: Iterative Viterbi

- Simplify our formulation, decomposing the surface to a series of contour optimizations (solvable by a modified Viterbi algorithm).

$$\max E(S) \approx \int_{t} \max\left(\iint_{S_t} E'(S_t^n, S^{(n-1)}, I, M)dxdy \right) dt$$

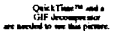
•As long as we estimate E' put our calculation into an iterative framework to simulate temporal connectivity

Iterative Viterbi: Step By Step



Single Object Results

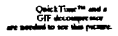
The final results (in the best case) looks something like this a surface that moves toward edge and motion discontinuities. but maintains a structural balance



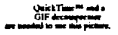
For multiple Objects, we start multiple isolated surface optimizations

Results from MPEG-4 Test Sequence Coastguard

Initialization from Principal Component Analysis of Motion Field



Video Object Extraction Results from our System

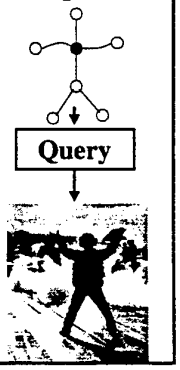


Outline

1. Introduction
   - Definition of Video Object / Media-Based Focus
   - MPEG-4/7 Standards
2. Voronoi Ordered Space
   - Segmentation
   - Representation
3. Video Object Segmentation (MPEG-4)
4. Video Object Representation (MPEG-7)
   - Shape Extraction, Ordered Trees and Problems
   - DAG-Ordered Tree
   - Application and Results: Query by Shape
5. Conclusion

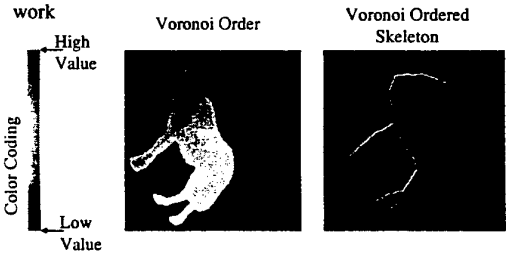
Video Query Problem Description

- Users wish to access specific video content from large video object archives
  - Shape is one way to query the database
- Our Description Scheme (DS):
  - Derive skeletons from VOP and calculate similarity using these skeletons
  - Skeletons are an intuitive and expressive way of describing shape



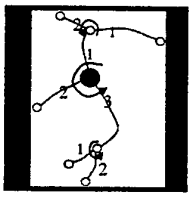
Shape Extraction via Voronoi Order

- Voronoi Order Skeleton: Jumps in Voronoi Order correspond to a physical skeleton
- Use the representation of a physical skeleton for MPEG-7 work

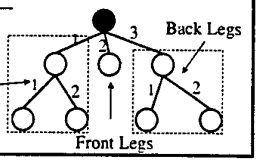


Ordered Tree Representation

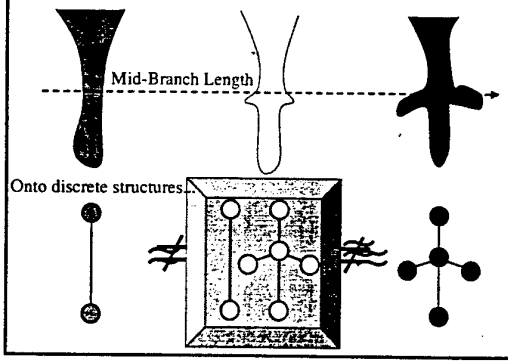
Voronoi Skeleton →	Tree
Branching Points	Nodes
Segments	Edges
High to Low Values	Parent-Child
Max. Value	Root



Ordering: To allow for efficient comparison, we order the sibling relations by their angle (circular, strict)

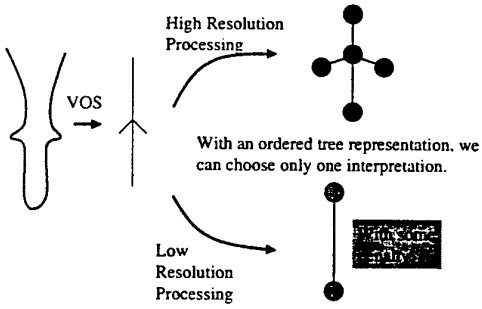


Sensitivity of Ordered Trees

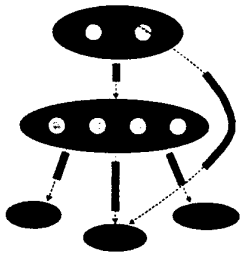


Sensitivity of Ordered Tree (2)

- In shape representation, we have a problem:



Ordered and DAG-Ordered Trees

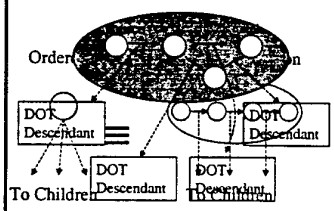


DOT is very similar ordered tree:

1. Descendant relations are ordered with a Directed Acyclic Graph (DAG)
2. Tree Hierarchy exists

DAG-Ordered Trees (DOTs)

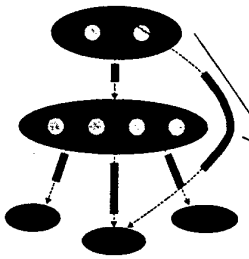
Example of DOT Node



Rules to constructing DOTs:

1. Inner Graph is a DAG
2. A DOT node can only refer to a DOT node of lower depth

Structure of DOTs

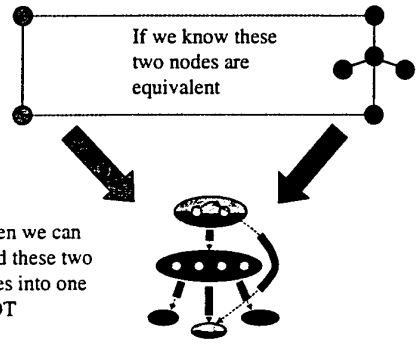


The structure of the DOT is still much like a ordered tree, but its order is topological order, instead of a linear order.

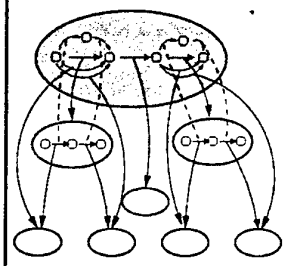
Ancestor-Descendant Relationship as a DAG

This loosening of order relationship as one DOAG

Merging Ordered Trees as a DOT



DOT as Compact Representation



(16 ordered trees)

The DOT codes multiple ordered trees to share hierarchy.

When ordered trees are uncertain or sensitive to noise, DOTs provide a robust solution that efficiently reuses structure with same algorithmic complexity.

Comparing DOTs

To compare DOTs as representation of multiple trees, we use the two critical properties of the DOT for a double induction for DOT-Compare Algorithm.

Induction #1:

Multi-resolution property of the DOT	→	Divide n' conquer by depth on the DOT

Induction #2:

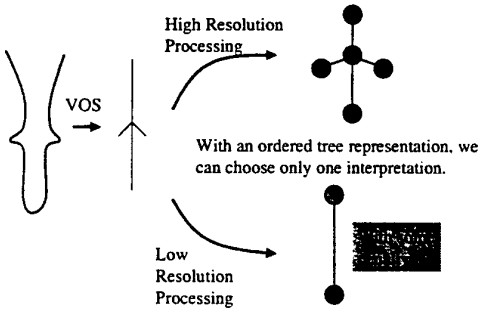
The order of the DOT descendants in the DAG	→	Dynamic programming solution.

DOT Applications

1. Cursive Handwriting Recognition
   - Working System
2. OCR Applications
   - Preliminary Research
3. Generalized Video Content DS
   - In Development for presentation at ICME
4. Shape Recognition
   - Working System, Tested on MPEG-7 Core Experiment Data

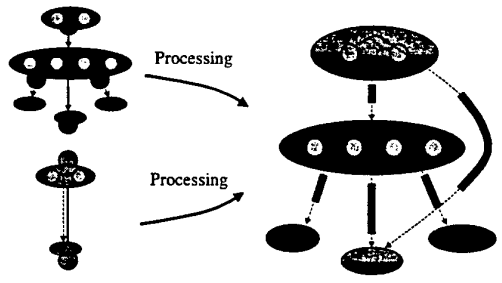
Sensitivity of Ordered Tree (2)

- In shape representation, we have a problem:



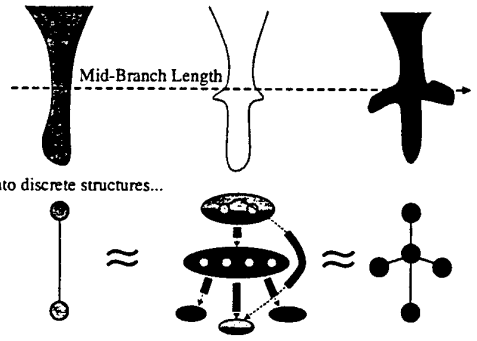
Robust Solution with DOTs

We can now code both interpretations into a single DOT.



Many other expressive transformations exist with DOTs.

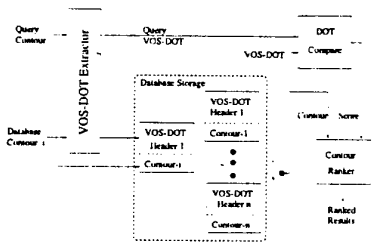
Continuity of DAG-Ordered Trees



Shape Query System

Simple Linear Query, w/ DOT-Compare as Similarity

On Two MPEG-7 databases :
Animals (~120)
Complete (~1400)



Comparison Settings:

- Allows complete flexibility at joints
- Each edge contains length, curvature, thickness, weight info.

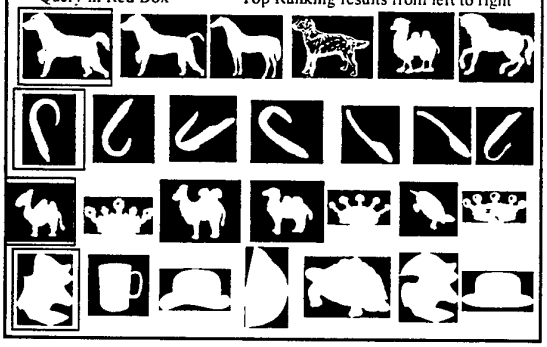
Results: Animal Subset

Shapes Database ⇒ Query ⇓	(A)	(B)	(C)	(D)	(E)	(F)	(G)	(H)
(α)	**0.77**	0.22	0.30	**0.53**	0.29	0.16	**0.31**	0.05
(β)	**0.28**	0.09	0.08	0.08	0.13	**0.27**	0.17	**0.47**
(γ)	0.19	**0.30**	0.15	**0.44**	0.22	**0.24**	0.04	0.12
(δ)	**0.42**	0.27	**0.39**	**0.53**	0.24	0.06	0.24	0.19

Figure 3: Results of preliminary shape query test on a small shape database using VOS and DOT as representation and the DOT-compare as similarity measure; note database and query shapes are all distinct.

Results: Large Database (~1400)

•Query in Red Box •Top Ranking results from left to right



Thesis Contributions (1):

Our MPEG-4 Video Object Segmentation System:

- novel approach integrates many information sources and can be easily expanded
- shows promise
- Caveat: Computationally intensive and may require parallel architectures and server-side driven computation

Thesis Contributions (2)

- DAG-Ordered Trees
  - Novel Extension to Ordered Tree
  - Robust Multi-Resolution Representation
- Our MPEG-7 Video Object Shape Query System
  - Another Application of DOT
  - Skeleton as a robust similarity alone
  - Skeleton used with **motion**, color and texture to compactly express a video object w/ temporal behavior

Future Work

Research-wise: Explore Synergy between MPEG-4 segmentation and MPEG-7 object information

Technology Development:

- Generalized Description Schemes
- Transcoding / Universal Multimedia Access
- Content-Based Network Management

Publication List

"Efficient Representation and Comparison of Multimedia Content Using DAG-Composition", I-Jong Lin, A. Divakaran, A. Vetro, and S-Y. Kung, Accepted to Int'l Conference Multimedia and Expo

"Shape Representation and Comparison Using Voronoi Order Skeletons and DAG-Ordered Trees," I-Jong Lin, A. Vetro, A. Divakaran and S-Y. Kung, Proc. Int'l Workshop on Very Low Bit-Rate Video Coding, Kyoto, Japan, Oct. 1999

"An Algorithmic Framework for Automatic Video Object Segmentation," I-Jong Lin and S.Y. Kung, 1999 IEEE Workshop on Multimedia Signal Processing

"Circular Viterbi Based Adaptive System for Automatic Video Object Segmentation," I-Jong Lin and S.Y. Kung, Proceedings of 1998 IEEE Workshop on Multimedia Signal Processing, Los Angeles, CA

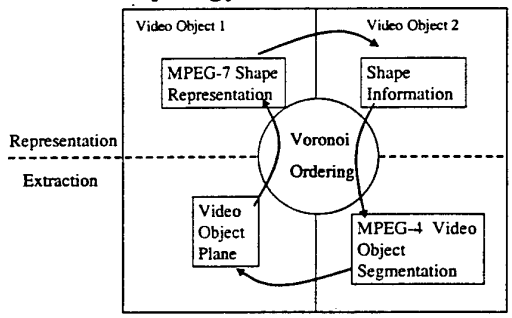
"Circular Viterbi: Boundary Detection With Dynamic Programming", I-Jong Lin, A. Vetro, H. Sun, S.Y. Kung, MPEG98/DOC3659, July 1998

"A Novel Learning Method By Structural Reduction Of DAGS For On-Line OCR applications," I-Jong Lin and S.Y. Kung, Proceedings of ICASSP98

"A Recursively Structured Solution for Handwriting and Speech Recognition," I-Jong Lin and S.Y. Kung, In Proceedings of 1997 IEEE Workshop on Multimedia Signal Processing, pp. 587-592

"Coding and Comparison of DAGs as a Novel Neural Structure with Applications to On-Line Handwriting Recognition," I-Jong Lin and S.Y. Kung, IEEE Trans. on Sig. Proc., Nov. 1997, v. 45, #11, pp. 2701-2708

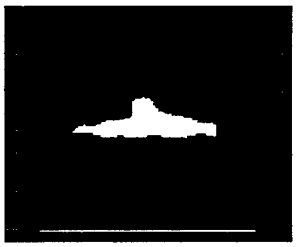
Synergy (Neural)



Extra MPEG-4 Slides

PC Motion Analysis

1. Multi-Resolution Optical Flow
2. PC Motion Paths
3. PC Surface Analysis

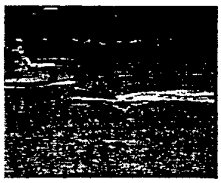


Reference: Y.T. Lin, "A Principle Component Clustering Approach to Object-Oriented Motion Segmentation and Estimation", J. of VLSI Sig. Proc., v. 17, #2/3,pp.163-187, Nov. 1997

PC Motion Analysis: Multiple Objects

Multiple object can be taken out of the flow by their different motions

- Separate the Energy Function by locality
- Optimize each surface independently



$$E(S_1, S_2, \mathrm{K}, S_n) \approx \sum_n E_i(S_i)$$

Compromise: We cannot handle occlusion.

Container

Harder: Occlusion due to pole in foreground

Homogeneous background, difficult to find motion discontinuities

Lower Right region is interesting. Conservation of mass rule is not described within the system.

QuickTime™ and a GIF decompressor are needed to see this picture.

QuickTime™ and a GIF decompressor are needed to see this picture.

Hall Monitor

Higher levels of processing are required to link the rigid objects together

Current System over-segments non-rigid objects into regions. However, we need extra analysis to take this into account.

QuickTime™ and a GIF decompressor are needed to see this picture.

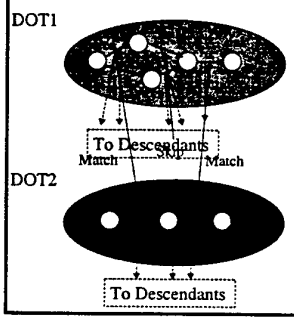
QuickTime™ and a GIF decompressor are needed to see this picture.

Summary: MPEG-4 Video Object Segmentation

- Formulation and Algorithmic Framework
  - Separates the problem: Extraction/ Surface Fitting
  - Allow extensions into the framework (color, texture, etc.)
  - Finds rigid elements which then may be combined into full objects
- Problems:
  - Occluding Objects (requires simultaneous surface optimization)
  - Over-segments (more high-level processing is needed)

Extra MPEG-7 Slides

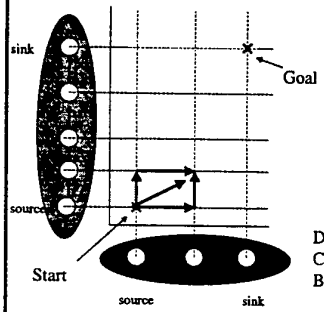
Comparing DOTs: Induction #1



Induction #1:
We assume all descendant-descendant comparisons are solved.

Simpler Problem:
Find best matching path between two DAGs

Comparing DOTs: Induction #2



= In Set of Optimal Scores

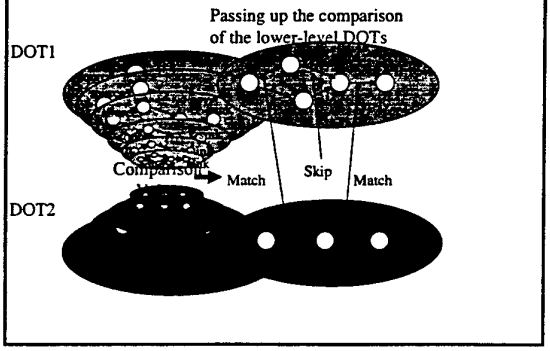
Path Matching Between DAGs :

Find Optimal Score from Start to Goal through a series of Edge matches

Dynamic Programming: Calculate next optimal score Based on previous optimal

Ref: IEEE Trans. On Sig. Proc., Nov. 1997, v.45,#11,pp. 2701-8

Recursing Back: Back to Induction #1



Quantitative Comparisons

MPEG-7 Core Experiment Tests for Shape

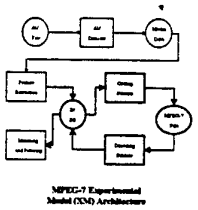
Technique	Test A	Test B	Test C	Average
VIL (319)	96	75	95	89
687	96	70	94	87
HHI (667)	93	68	93	85
ATL	85	60	83	71
517	84	71	80	80

Interactive Content-Based Image Retrieval

Ling Guan

Signal and Multimedia Processing Laboratory
Department of Electrical Engineering
The University of Sydney
Sydney, NSW 2006 Australia
ling@ee.usyd.edu.au

MPEG-7 Architecture



MPEG-7 Experimental Model (XM) Architecture

- MPEG-7 Working Group focuses on description interchange (the normative components -shaded blocks in the XM)
- The rest is left for open competition from industry and research organizations

What is CBIR?

- The objective: to find all the images in a DIVL similar to a query, based solely on content information
- The procedure:
  - choose a criterion (e.g. MSE) to measure the similarity
  - extract a set of features which best characterizes the query in a predefined sense
  - extract the same features from each of the images in the DIVL
  - use the criterion to measure the similarity between the query and each of the images
  - the images which are "more" similar to the query are presented to the user for further analysis

Problems with CBIR

- Gap between the high level concepts (semantic descriptions) and low level features (color, texture, shape, etc) affects query design
- The subjectivity of human perception cannot be effectively modeled by the linear criteria popuparly adopted in retrieval

Example: Compressed Domain

- Database in the compressed domain:
  - Many of the current image/video coding standards such as JPEG, MPEG-1,2, and H.261/3 are based on DCT.
  - The new image coding standard JPEG2000 adopted wavelets/VQ coding scheme.
- The gap between human visual perception and the information carried in DCT and/or wavelet coefficients could be significant (little direct correspondence).

Interactive CBIR

- In numerous applications, an attractive solution to the above problems is to adopt an interactive approach
- Main feature: an active role played by users in image retrieval to improve the retrieval accuracy
- Currently, some query design considerations have been introduced in interactive CBIR, but linear criteria still dominate similarity ranking.

Our Proposal

In the framework of relevance feedback, it offers the following benefits to CBIR:

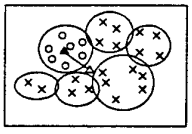
- Modeling: effectively mapping or linking a high level concept to low level features
- Matching:
  - capturing user's perceptual subjectivity of visual content to modify the query using a nonlinear measure
  - overcoming the difficulties faced by the traditional linear matching criteria

The Relevance Feedback Framework

- The goal is to measure feature relevance to improve performance of image matching in a retrieval process.
- It is a supervised learning procedure based on regression
  - For a given query z and a set of the retrieved items,
    $\mathbf{x}_n, n = 1,2,\ldots,N, \mathbf{x} \in \Re^P$
    classify $\mathbf{x}_n$ into two classes:
    the relevant class (visually similar to z): x_m, m = 1, 2, …, M, and
    the irrelevant class (not similar to z): x_q, q = 1, 2, …, Q
  - Use the information in x_m and x_q to structure a new query
  - The new query is applied in the next round of retrieval

Query Modification Model 1 (like K-mean)

- The new query: $z_1 = \{z_i\}_1^P = \frac{1}{M}\sum_{m=1}^{M} x'_m$

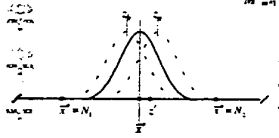


Query Modification Models 2&3
(like the anti-reinforced learning)

- The new query: $z_2 = \{z_i\}_1^P = \bar{x} - \alpha_N(\bar{x}' - z')$

$z_3 = z' + \alpha_R(\bar{x} - z') - \alpha_N(\bar{x}' - z')$

$\bar{x} = \frac{1}{M}\sum_{m=1}^{M} x'_m, \quad \bar{x}' = \frac{1}{Q}\sum_{q=1}^{Q} x'_q$



Example (1-D)

when $\bar{x} < z' \Rightarrow Z_2 = z_2 = \bar{x} + \alpha(z' - N_1)$

when $\bar{x} > z' \Rightarrow Z_2 = z_2 = \bar{x} + \alpha(z' - N_2)$

Where α_1, α_2 = small positive constants
$\bar{x}$ = Center of relevant items
$\bar{x}'$ = Center of non-relevant items
z' = query at previous iteration

RBF-Based Search
(Nonlinear Search Unit)

- RBF-Based Search (RBF-NSU)

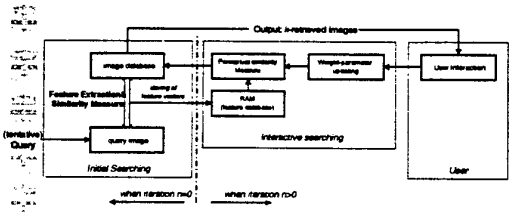
$$f(\mathbf{x}, \mathbf{z}) = \sum^{P} G(x - z) = \sum^{P} \exp\left[\frac{-(x - z)^2}{\sigma_i^2}\right]$$

- image feature vector:
- adjustable query vector:
- the tuning parameters (NSU widths):

$$\sigma_i = n \cdot \max_i |x_i - z_i|, \quad i = 1,\ldots,P$$

- Small σ_i : the feature is relevant (sensitive to change).
- Large σ_i : the feature is irrelevant (insensitive to change).

Architecture for Interactive CBIR



Test One:
DCT Domain Image Retrieval

DCT Coefficient Feature

- DC coefficient is the scaled average of a block.
- AC coefficients represent edges of different frequencies.
- Groupings of coefficients and masks used:

F1F:	DC	AC_{01}	AC_{02}	AC_{03}	
F2F:	AC_{10}	AC_{11}	AC_{12}	AC_{13}	
F3F:	AC_{20}	AC_{21}	AC_{22}	AC_{23}	
F4F:	AC_{30}	AC_{31}	AC_{32}	AC_{33}	

Masks	Groups used
F1	F1F
F2A, F2M	F1F + F2F
F2B	F2F
F3A, F3M	F1F + F2F + F3F
F3B	F2F + F3F
F4A, F4M	F1F + F2F + F3F + F4F
F4B	F2F + F3F + F4F

Test Database

- The experiment was performed on an image database consisting of nearly 4.700 real-life JPEG Images distributed by Media Graphic Inc. http://www.media-graphics.net [*"Photo Gallery 5,000 Vol. 1 CD-ROM"*]
- The images are maintained as a single unclassified image database.

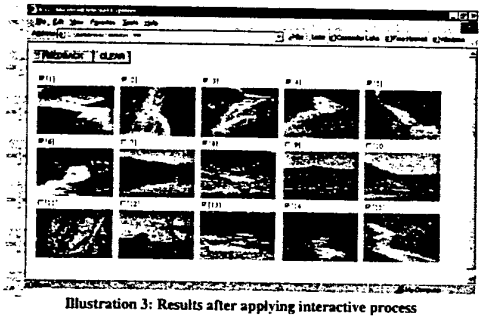


Illustration 3: Results after applying interactive process

Test Two:
Retrieval Using Gabor Wavelet Features

Gabor Wavelet Representation

- A two dimensional Gabor function $g(x,y)$ can be written as: *(a mother Gabor wavlet)*

$$g(x,y) = \left(\frac{1}{2\pi\sigma_x\sigma_y}\right)\exp\left[-\frac{1}{2}\left(\frac{x^2}{\sigma_x^2}+\frac{y^2}{\sigma_y^2}\right)+2\pi jWx\right]$$

- A set of self-similar filters is derived through the generating function :

$$g_{mn}(x,y) = a^{-m}g(x',y')$$

, and

- Given an image $I(x,y)$, its Gabor transform is then defined as:

- A Gabor Wavelet feature representation is constructed as:

$$f = [\mu_{11}\sigma_{11}\mu_{12}\ldots\mu_{ns}\sigma_{ns}]$$

, and

Gabor Wavelet Features

- In the experiments, we use 24 Gabor filters (four scales S = 4 and six orientations K = 6).
- The mean μ_{mn} and standard deviation σ_{mn} of the magnitude of the transform coefficients are used as feature components, resulting in a 48 dimensional feature vector

$$f = [\mu_{00}\sigma_{00}\mu_{01}\cdots\mu_{35}\sigma_{35}]$$

Performance Comparison with MARS

Methods Compared

- LSU2: linear search unit & query model 2
- NSU2: non-linear search unit & query model 2
- Interactive CBIR method in MARS: Multimedia Analysis and Retrieval System [developed at (UIUC)]

Test Database 1: Bordatz database

- The experiment was performed on a texture image database consisting of 1,856 patterns in 116 different classes, where each class contains 16 similar patterns. The database is provided by Mahjunath, at http://vivaldi.ece.ucsb.edu/users/wei/codes.html. ["*Department of Electrical and Computer Engineering, UCSB*"]
- The images are maintained as a single unclassified image database.

Retrieval Results for the Bordatz Database

[*116 different image classes*]

	Methods	Retrieval Rate (%)			
		l=0	l=1	l=2	l=3
Avg.	LSU2	73.7	83.0	85.1	85.9
	NSU2	73.7	84.9	88.2	89.2
	MARS	67.0	75.1	78.4	78.7

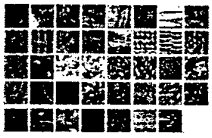
Table 2. Average retrieval rate (%) for 116 query images in the Bordatz database (1,856 patterns).

Note: The retrieval rate is defined as the average percentage of images belonging to the same class as the query in the top 16 matched.

Test Database 2: The MIT Database

- The MIT data set consists of 624 texture images classified into 39 classes. The original 39 512x512 texture images are obtained from MIT Media Lab at ftp://whitechapel.media.mit.edu/pub/VisTex/
- Each image is cut into 16 128×128 non-overlapping small images. The images in the same class are considered as relevant to one another.

Retrieval Results for the MIT Database

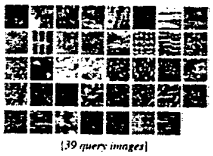


[*39 query images*]

Methods	l=0	l=1	l=2	l=3
LSU2	74.36	88.46	91.19	91.67
NSU2	74.36	89.10	92.15	93.27
MARS	64.26	77.72	79.97	80.13

Table 3. Average retrieval rate (%) for 39 query images in the MIT database

Retrieval Results for the MIT Database

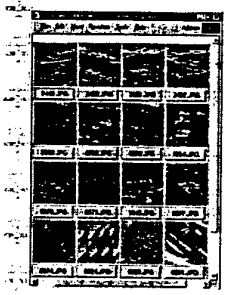


(39 query images)

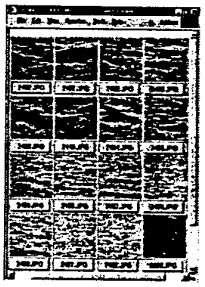
Methods	lv0	lv1	lv2	lv3
LSU2	74.36	88.46	91.19	91.67
NSU2	74.36	89.10	92.15	93.27
MARS	64.26	77.72	79.97	80.13

Table 3. Average retrieval rate (%) for 39 query images in the MIT database

Retrieval Example 1



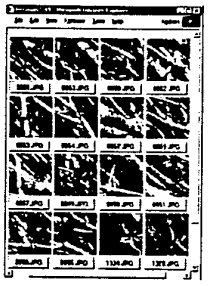
• Retrieval result by MARS



• Retrieval results by NSU2

Retrieval Example 2

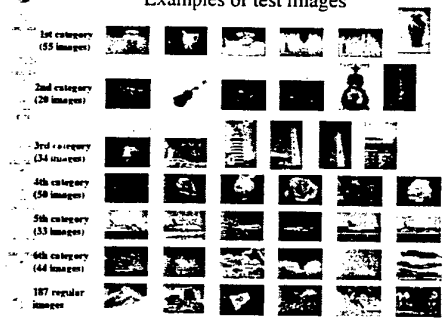
• Retrieval result by MARS



• Retrieval result by NSU2

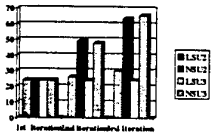
Test Three
Retrieval with Color/Shape Features

Examples of test images



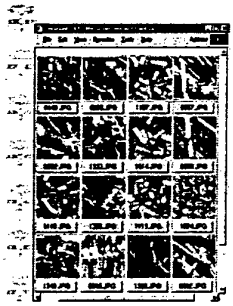
Experimental Results

Queries	Number of relevant image in the top 16 retrieved images											
	LSU2			LSU3			NSU2			NSU3		
	R(0)	R(1)	R(2)	R(0)	R(1)	R(2)	R(0)	R(1)	R(2)	R(0)	R(1)	R(2)
Q1: Blue sky with tower whose width is narrower at the top and wider at the bottom.	3	9	12	3	4	5	3	9	12	3	3	3
Q2: Plane flying in the sky not landing.	5	9	9	5	5	5	5	4	9	5	5	5
Q3: Blue sky with white cloud.	3	10	14	3	4	5	3	9	14	3	3	3
Q4: Yellow rose	6	9	11	6	6	6	6	10	11	6	6	6
Q5: Transparent white glass	3	3	9	3	3	4	3	3	9	3	3	3
Q6: Horse	3	7	8	3	3	4	3	7	8	3	3	3



A plot of the average retrieval rate (%) of the four retrieval models as a function of iterations

Retrieval Example 2



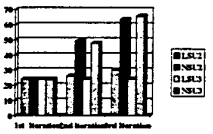
• Retrieval result by MARS

• Retrieval result by NSU2

Test Three
Retrieval with Color/Shape Features

Experimental Results

Queries	Number of relevant image in the top 16 retrieved images								
	NSU2			LSU2			NSU3			LSU3		
Q1: Blue sky with cirrus wisps within a narrow arc at the top and water at the bottom.	3	4	12	3	4	4	3	4	12	3	3	3
Q2: Plane flying in the sky not landing.	5	9	7	5	5	5	5	6	6	5	5	5
Q3: Blue sky with white cloud.	3	10	14	3	4	5	3	6	14	3	3	3
Q4: Yellow rose.	6	9	11	6	6	6	6	10	11	6	6	6
Q5: Tree/emerald white grass	3	3	6	3	3	4	3	3	6	3	3	3
Q6: Ocean	3	7	8	3	3	4	3	7	8	3	3	3



A plot of the average retrieval rate (%) of the four retrieval models as a function of iterations

Summary

- Interactive techniques:
  - provide a significant improvement of retrieval performance over the simple CBIR system
  - supports user queries satisfactorily
- It represents the state-of-the-art in interactive CBIR
- The query models are particularly effective when gaps exist between features and human perception
- The nonlinear search units are particularly effective when retrieval in color/spatial domain

Broadband Wireless Access for Internet Multimedia Networking

S.Y. Kung
Princeton University

Broadband wireless access and silicon solutions will be a major force in the so-called "Last Mile" broadband communication industry.

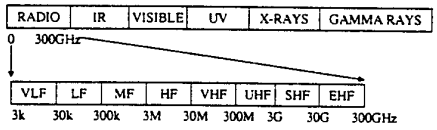
TIMELINE: PRE-WWII

- 1837: Morse demonstrates the telegraph.
- 1876: Bell patents of the telephone.
- 1887: Hertz demonstrates radio waves
- 1895-97: Marconi demonstrates wireless telegraph.
- 1900: Fessenden demonstrates wireless voice.
- 1906: DeForest invents the Audion triode
- 1912: Armstrong/Fessenden discover super-heterodyne principle
- 1920: First commercial radio station. (1927 - FCC formed)
- 1934: Armstrong invents FM
- 1930's: Development of television.

TIMELINE: POST-WWII

- WWII - Radar spurs improvements in radio components.
- 1947: invention of the transistor (Bell Labs)
- 1947: origination of cellular concept (Bell Labs)
- 1948: Shannon exposition of information theory (Bell Labs)
- 1958: Kilby/Noyce invent integrated circuits
- 1972: ARPANET demonstrated
- 1983: AT&T introduces AMPS (early cellular).
- 1992: GSM rolled out (European cellular).
- 1993: U. of Illinois develops Mosaic browser.
- 1995: Qualcomm introduces CDMA cellular.
- 1999: 3-COM introduces mobile Web browser.
- c. 2002: NTT to roll out 3G cellular.

FREQUENCY BAND DESIGNATIONS



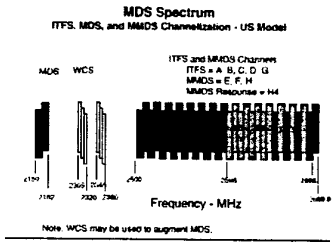
VLF: Very Low Frequency
LF: Low Frequency
MF: Medium Frequency
HF: High Frequency
VHF: Very High Frequency
UHF: Ultra High Frequency
SHF: Super High Frequency
EHF: Extremely High Frequency

Note: these designations were set byint'l conference in 1959.

SOME US FREQUENCY ALLOCATIONS

Submarine Comms: 30 kHz
Navigation (Loran C): 100 kHz
AM Radio: 540 – 1,600 kHz (medium wave)
Tactical Comms/Radio Amateur: 3 – 30 MHz (short wave)
Cordless Phones: 46 - 49 MHz (FM) or 902-928 MHz (Spread Spectrum)
FM Radio & Paging: 88 – 108 MHz
TV: 54 – 216 MHz (VHF) & 420 – 890 MHz (UHF) [not contiguous]
Cellular: 450MHz, 800 MHz, 900 MHz, 1.9 GHz (UHF)
Satellite Comms: SHF
Wireless LAN's: the upper ISM bands and IR (not regulated).

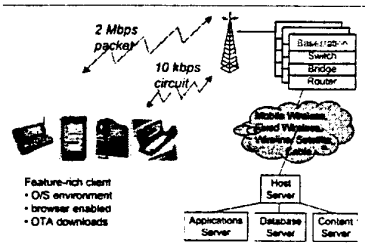
- See Homework Set #2 for a listing of websites showing detailed civilian and military uses of the radio spectrum.



BWA is Answer to Increasing Demand on Data Rates

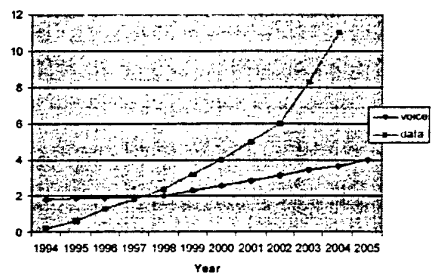
7/4/00

Wireless Networks



7/4/00

US Fixed Network Traffic (Billions of Gigabits /year)



Source BT Alex Brown, Nortel Lucent, Telcordia

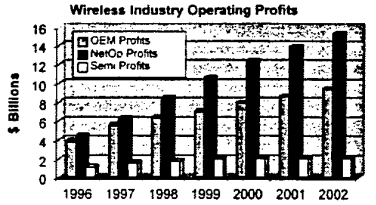
Wireless Market

- Wireless will catch up wireline in both telephony and internet
- All wireless predictions have fallen short of real progress
- Basestation market will reach ($65B/year by 2002)
- Wireless will have 30% + global telecommunication market by 2002
- data portion will grow from the current 2% to 25% by 2002

NTT DoCoMo has 4M users in one year
Erickson and Norkia 600M users by 2004
Sonera is joint telecom and ISP company with 35%/65% on value-added/voice

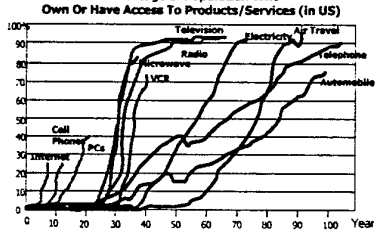
7/4/00

Wireless Industry Operating Profits



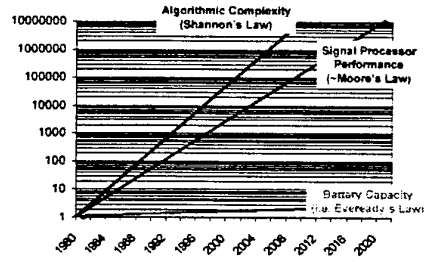
7/4/00

Percentage of Population Who Own Or Have Access To Products/Services (in US)



7/4/00

Wideband Communication Services



7/4/00

Standards & Services Proliferation

from NARROWB CIRCUIT VOICE to WIDEBAND PACKET DATA

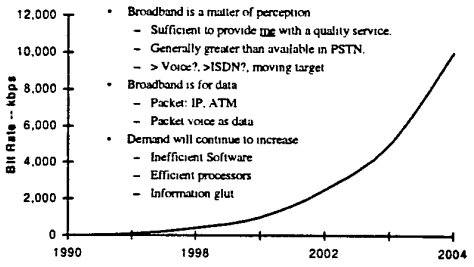
2G 9.6Kbps
- GSM
- DCS1800
- PCS1900
- IS-95
- IS-54B
- IS-136
- PDC

2.5G 64-384Kbps
- GPRS
- HCSD
- IS-95B
- IS-136+
- IS-136HS

3G 384-2000Kbps
- 3GPP WCDMA-FDD
- 3GPP WCDMA-TDD
- ARIB W-CDMA
- TIA cdma2000
- EDGE
- HDR
- W-TDMA (UWC)

7/4/00

Broadband Demand



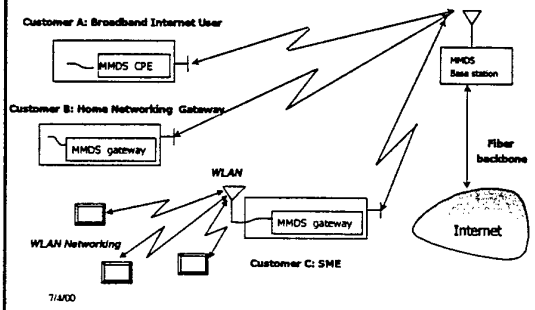
- Broadband is a matter of perception
  - Sufficient to provide me with a quality service.
  - Generally greater than available in PSTN.
  - > Voice?, >ISDN?, moving target
- Broadband is for data
  - Packet: IP, ATM
  - Packet voice as data
- Demand will continue to increase
  - Inefficient Software
  - Efficient processors
  - Information glut

7/4/00

MMDS BWA Standards

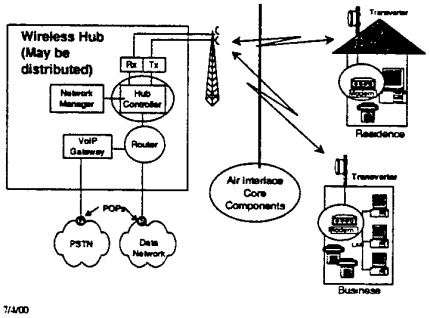
7/4/00

MMDS for Internet connection



7/4/00

Broadband Wireless Access(BWA)



7/4/00

MMDS (Multichannel Multipoint Distribution Service)

- *Frequency: 2.5 GHz -11GHz, Bandwidth: 200 MHz*
- *No LOS requirement - using spatial diversity*
- *More affordable (CMOS-RF) hardware components than LMDS*
- *Standards imminent (IEEE802-16, Cisco, Hyperaccess)*
- *MCI and Sprint bought the spectrum for broadband access*

7/4/00

Opportunities offered by BWA

- ***Create new services:*** High speed Internet access and combined data, voice and video services
- ***Bypass local loop:*** Offering competitive solution for the end users cover current service provider
- ***New revenue opportunities:*** Fast and flexible installation, providing access to under-served areas

7/4/00

Why Broadband Wireless?

- Key advantages over wire-line (e.g. high speed telephone line modem, ISDN, xDSL, cable modem)
  - High-speed > 100Mbps
  - Cost-effective infrastructure
  - Easy, flexible, and fast deployment
  - Granular investment
  - High data scalability
  - Low cost maintenance

7/4/00

Broadband Wireless Access

- *High bit rate per user depending on the standard IEEE802-16)*
- *Simple and low-cost Customer Premises Equipment (CPE)*
- *No Line-Of-Sight (LOS) requirement (MMDS)*
- *High capacity architecture, many users per area*
- *Plug and play customer installation*
- *Microcells*

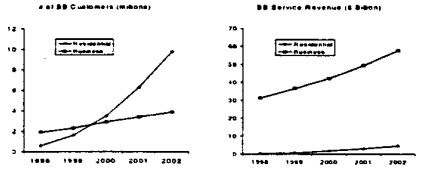
Although the bandwidth as proposed by the standard may very well be much higher, our scalable architectural design will effectively support 5-15Mb/s more imminently prevailing market domain.

MMDS End-User Market:

- *Residential Data Internet Access*
- *SME (Small Enterprise)*
- *SOHO (Small Office Home Office)*
- *Temporary high-speed data network, supporting major events.*

7/4/00

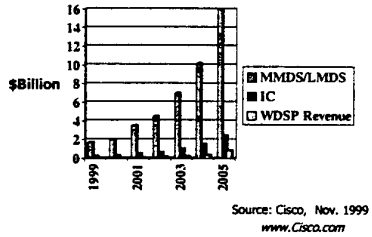
Broadband Market Projection -US



Quantity favors residential, Revenue favors business. MMDS positioned for both.

7/4/00

BWA Market Projection



Source: Cisco, Nov. 1999
www.Cisco.com

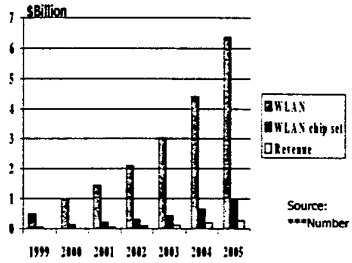
7/4/00

Wireless LAN(WLAN) End-User Market:

- A proven market: e.g. *Apple Airport* WLAN, Lucent's WaveLAN, Intersil WLAN set, *etc.*
- Campus, Company intra-net
- Residential home networking
- SME (Small Enterprise)
- SOHO (Small Office Home Office)
- Temporary high-speed data network

7/4/00

WLAN Market Projection



Source:
***Number to be confirmed

7/4/00

Market Issues

- Increasing Proliferation of Multiple Standards
- Increasing Network Complexity
- Increased Design Implementation Complexity
- Tightening Cost Pressures
- Shortened Product Cycles
- Time-to-Market Pressures
- Constrained Design Resources

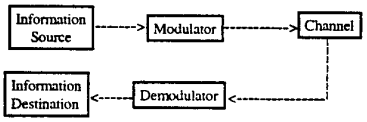
7/4/00

Critical BWA Technologies:

- *Spatio-Temporal Diversity*
- *OFDM*
- *PHY/ MAC/Gateway*

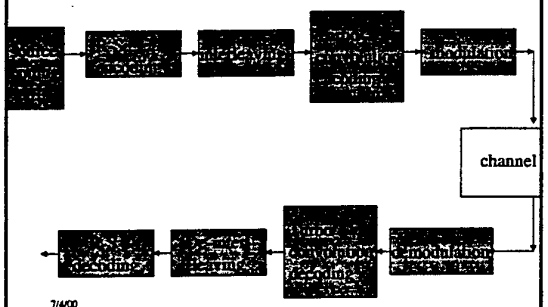
7/4/00

ABSTRACT COMMUNICATION MODEL

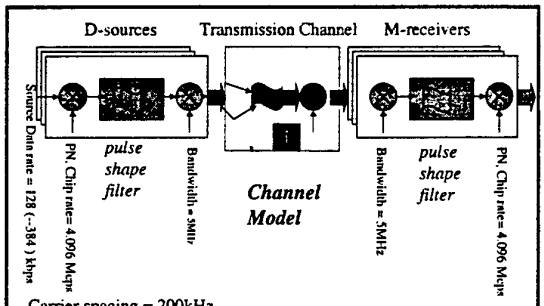


7/4/00 ELE 391: THE WIRELESS REVOLUTION

Typical Tranceiver Signal Flow System



7/4/00



Carrier spacing = 200kHz

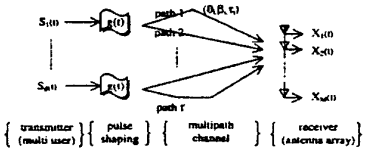
Spreading code = ovsf = orthogonal variable spreading factor

N=32 : Source Data rate=128 kbps and Chip rate=4.096Mcps

7/4/00

Spatio-Temporal Diversity for Wireless Multipath Channel

7/4/00



d : # of source signals θ_i : DOA of path i

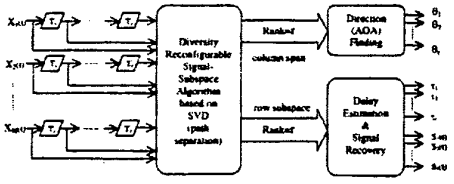
M : # of antenna elements β_i : fading factor of path i

r : # of multi paths τ_i : delay of path i

$g(t)$: temporal response L : length of g(t)

function (pulse shaping filter+channel spread)

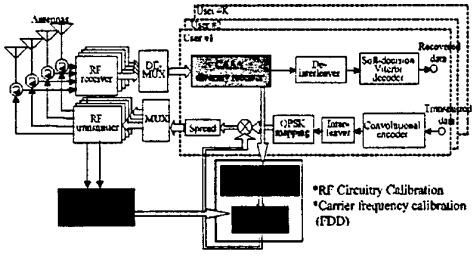
7/4/00



- *Dynamic Space-Time Diversity Combination & Subspace Analysis*
  - *Condition of Path Separation*
  - *Diversity Combination/Tradeoff*
  - *Matrix Presentation*
  - *Path Separation*

7/4/00

Adaptive Antenna Array Diversity Receiver

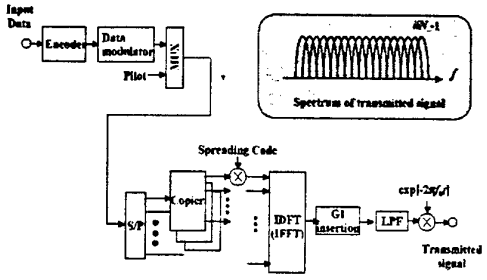


- *RF Circuitry Calibration
- *Carrier frequency calibration (FDD)

OFDM
for Wireless Communication

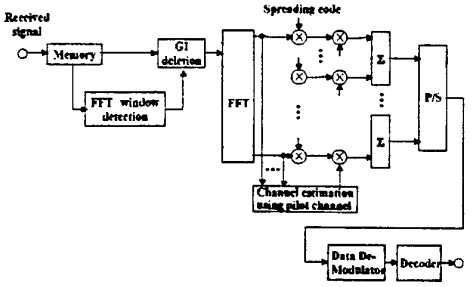
7/4/00

Multi-Carrier OFDM Transmitter



7/4/00

Multi-Carrier OFDM Receiver



7/4/00

Parameters

Bandwidth	80MHz
Information bit rate	2Mbps
Number of sub-carriers N_s	1,4,8 and 16 (MC/DS-CDMA) 512 (MC-CDMA)
Packet length	1024 symbols (N_s=960, N_p=64)
Roll-off factor α	0.25
Data modulation/Spreading	QPSK/QPSK
Channel coding/decoding	Convolutional coding (R=1/2,K=9)/Soft decision Viterbi decoding
Maximum Doppler frequency	$f_D T_s$=4.0 x 10-6

7/

OFDM: Overview

- Allows transmission of high data rate over hostile channels
- Adopted for European radio (DAB) and TV (DVB-T) standards
- 4G's data rate upto 40Mbps (compared with 3G's 2 Mbps) calls for multi-carrier OFDM.

7/4/00

OFDM

- OFDM advantage: robustness against multipath propagation
- OFDM disadvantages: sensitivities to synchronization and frequency offset
- Problems with wireless fading channel: coding and interleaving is used to exploit the channel diversity.

7/4/00

OFDM Approach

- The original stream of rate R is multiplexed into parallel stream with rate R/N
- Each of substream is modulated with a different frequency
- the resulting signals are transmitted together in the same band.

7/4/00

OFDM TASKS

- Propagation Channel Analysis
- QAM (or QPSK) Demodulator
- OFDM demodulator (FFT)
- Channel Estimation and Equalization
- Synchronization
- System Architecture:
  - Combining DSP with ASIC/FPGA
  - Balance power and flexibility.

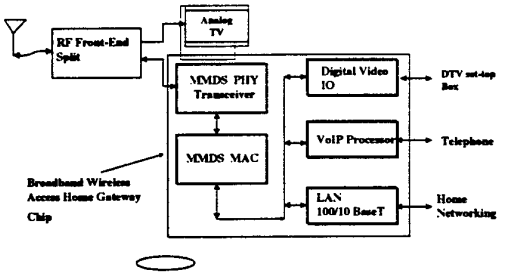
7/4/00

Wireless DSP Technology:

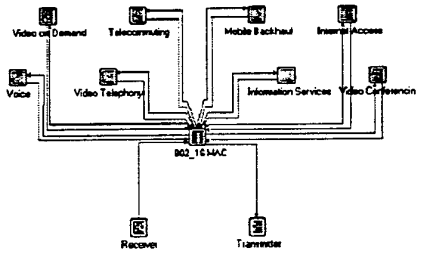
PHY/ MAC

7/4/00

Broadband Wireless Access (BWA) Home Gateway Function Blocks



7/4/00



OPNET Modeler Graphical User Interface - traffic models includes:
· Database · E-mail · FTP · HTTP · Voice
· Multi-tier applications · Remote login · Video Conferencing

7/4/00

Wireless DSP Technology

- Extensive expertise in DSP system development, wireless communication, gigabit Ethernet LAN and VLSI design
- Wireless LAN system design
- Fast algorithms, and its low-power and low-cost implementations
- MAC and higher layer network expertise

7/4/00

PHY-DSP Core Technologies

Technologies:

- **OFDM core: scalable architecture, fast, low power**
- **QAM core: timing synchronizer**
- **Equalizer core: blind equalizer, low complexity, low-power**
- **FIR/IIR filter: Polyphase filtering, channelizer, zero IF**
- **ECC core: low complexity, low power RS codec**
- **Viterbi core: deconvolution decoder**

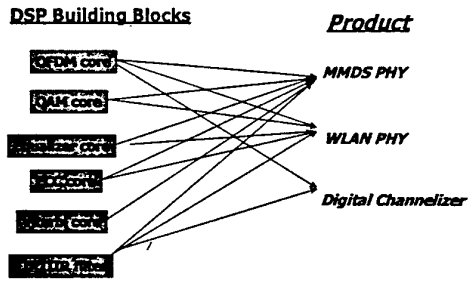
7/4/00

MAC/Gateway Core Technologies

- **BWA MAC: IEEE802.16 protocol, Scheduling, QoS**
- **VoIP DSP core, speech codec**
- **Home networking LAN**
- **MAC Simulator: Network Simulator, Real-time OS**
- **WLAN MAC: IEEE802.11 protocol, IEEE802.3 protocol**

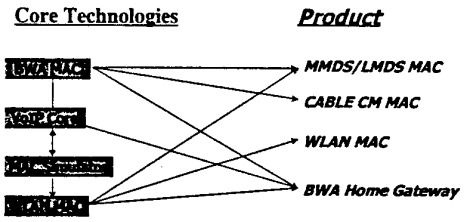
7/4/00

PHY-Layer Technology-Product Map



7/4/00

MAC-Layer Technology-Product Map



7/4/00

Promising Products

- *MMDS/LMDS Chip Set (PHY + MAC), BWA home gateway chip set* (the immediate focus)
- *Wireless LAN Chip Set* (MAC + PHY, IEEE802-11a complied)
- *Digital Channelizer* (a key component of Software Radio for mobile phone as well as cable data communication market)

7/4/00

MMDS/LMDS chip set Product

- *1*: A low cost MMDS MAC chip to target MMDS/LMDS and home network gateway markets
- *2*: PHY/MAC integrated chip for *MMDS CPE* and home gateway
- *3*: Chip set for BWA *home gateway* and MMDS *base station*

7/4/00

Wireless LAN Product

- *Target IEEE 802.11a 25 Mbps OFDM standard*
- *Mixed signal and highly integrated CMOS chip (possibly approaching 0.15 micron)*
- *Integrated MAC and PHY*
- *Use MMDS chip set as DSP building blocks*

7/4/00

Digital Channelizer Product

- *Target GSM/PCS cell phone base station, MMDS and cable CMTS market*
- *Reduces cost and simplify analog front-end*
- *Mixed signal, highly integrated 0.15 micron CMOS chip*
- *Potential Market size: one chip for each GSM, PCS, MMDS base station, and cable CMTS headend.*

7/4/00

Some Questions/answers:

Bit-rate for 802.16 is likely around 50Mbps for each user.
Cisco's Clarity reaches 54 Mbps via a vectorized OFDM.
Projected data-internet Market Domain 5Mbps to 15 Mbps

Smart antenna for soft handover

Short-code despreading:
For one basestation,i.e. one cell, it has 200 channels.
For each basestation, it has to process
long-code despread to cut the number of users down to
low-time: 2-3
day-time (office hour) average: 10 .
Maximum 60-70

7/4/00

Critical BWA Technologies:

VLSI Chips and SoC

7/4/00

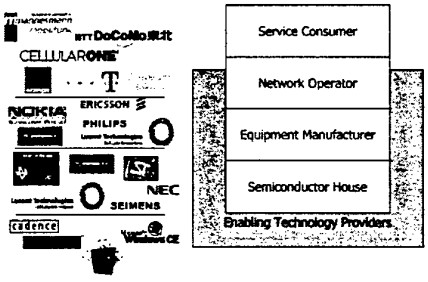
Role of Semiconductor House in Industrial Chain

- Service consumer
- Network operator ($20B/year) (warning: revenue/cost ratio is approaching 1.0)
- Equipment manufacturer (OEM) ($15B/year)
- Semiconductor house ($2B/year)

Enabling technologies: e.g. real-time OS

7/4/00

The Industry Chain



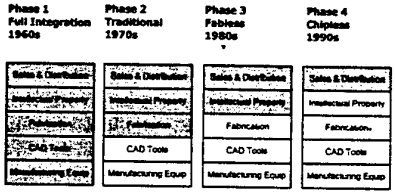
7/4/00

New Order

- Sales and Distribution
- Intellectual Property (Lucent, Norkia, ARM, RAMbus)
- Fabrication (TMSC)
- CAD (Candence, Synopsis)
- Manufacturing Equipment (KLA)

7/4/00

The Changing Face of The Semiconductor Industry



7/4/00

Challenge of OEM and Semiconductor House

- Being first to reach the market place
- System on chip
- semiconductor house is stretched: need 2 years to build products of life span for one year or less
- more silicon architecture in system house (revenue for semiconductor house vs. $12,000/wafer for subsystem architecture)

7/4/00

Semiconductor House Are Stretched!

Every new standard is getting more and more complex

AMPS->GSM 10x
GSM->IS-95CDMA 10x
IS-95CDMA->UTRA 10x

The time it takes to design the new standard-specific silicon platform is growing and getting expensive

The OEM <-> silicon tension eases

The wireless system becomes a "commoditized chip" faster

Equipment Manufacturers

Equipment manufacturers are facing an explosion in standards

New standards are driving new product architectures

Open interfaces are driving out proprietary interfaces in old products

Control of the product -- control of the product architecture --- silicon architects in system houses

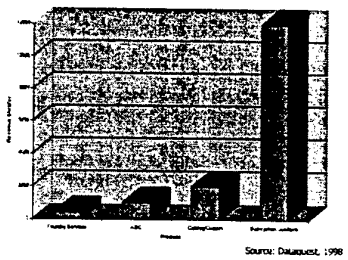
Equip manufacturers are financing more and more of their sales-moving to operating the network

By Morphics Technology, Inc.

7/4/00

What Types of Chips Bring High Margins?

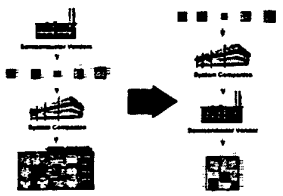
The Value-Added Chain in System ICs



Source: Dataquest, 1998

7/4/00

System-On-A-Chip Design



7/4/00

Semiconductor IP Industry

A new stage in the evolution of semiconductor industry
semiconductor house
fabless semiconductor house
fabless, chipless semiconductor house= SIP

SIP business model similar to that of software company

Digital SIP companies are establishing the business models

Analog SIP companies are starting up

7/4/00

Challenge of Fabless Company

- Increasing prolification of standards
- increasing networking complexity
- increasing implementation complexity
- tightening cost pressure
- tightening product cycle-time
- shorter time-to-market
- constrained design resource

Moore's Law on bandwidth : 10times/6years
Shannon Law on algorithmic complexity: even faster growth
Everday's Law on Power Supply: few percent per year

7/4/00