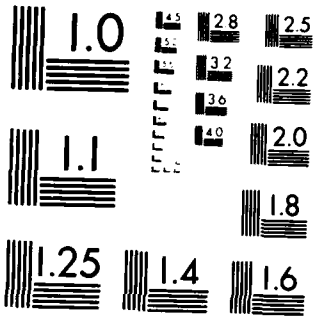
AD-A185 561 A TAXONOMY OF ANALYTICAL COMPUTER PERFORMANCE MODELS FOR COMPUTER DESIGN(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH M W STREVELL 1986 1/2

UNCLASSIFIED AFIT/CI/NR-87-97T F/G 12/8 NL



UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFIT/CI/NR 87-97T
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): A Taxonomy Of Analytical Computer Performance Models For Computer Design
5. TYPE OF REPORT & PERIOD COVERED: THESIS/DISSERTATION
6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s): Michael William Strevell
8. CONTRACT OR GRANT NUMBER(s):

9. PERFORMING ORGANIZATION NAME AND ADDRESS: AFIT STUDENT AT: The University of Texas
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:

11. CONTROLLING OFFICE NAME AND ADDRESS: AFIT/NR, WPAFB OH 45433-6583
12. REPORT DATE: 1986
13. NUMBER OF PAGES: 109

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):
15. SECURITY CLASS. (of this report): UNCLASSIFIED
15a. DECLASSIFICATION DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report): APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report):

18. SUPPLEMENTARY NOTES: APPROVED FOR PUBLIC RELEASE: IAW AFR 190-1

LYNN E. WOLAVER 27 Sept 87
Dean for Research and Professional Development
AFIT/NR

19. KEY WORDS (Continue on reverse side if necessary and identify by block number):

20. ABSTRACT (Continue on reverse side if necessary and identify by block number): ATTACHED

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

A TAXONOMY OF ANALYTICAL COMPUTER PERFORMANCE MODELS FOR COMPUTER DESIGN

BY

MICHAEL WILLIAM STREVELL, B.S., M.S.

THESIS

Presented to the Faculty of the Graduate School of

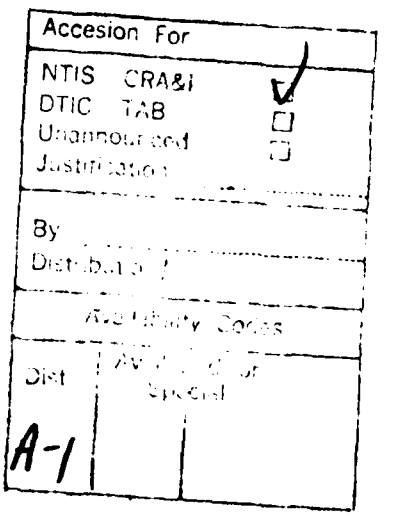
The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING



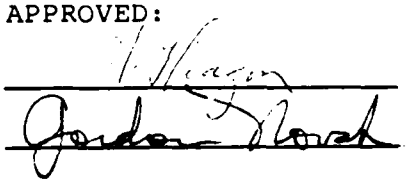
THE UNIVERSITY OF TEXAS AT AUSTIN

December 1986

A TAXONOMY OF ANALYTICAL COMPUTER

PERFORMANCE MODELS FOR COMPUTER DESIGN

APPROVED:



ACKNOWLEDGEMENTS

I would like to thank Professor Harvey Cragon for his support and for allowing me to a codify a portion of his extensive knowledge of computer design.

I am grateful to the United States Air Force Academy for sponsoring me for this degree.

Finally, I would like to acknowledge Intellicorp for providing the copy of KEE which made this research possible.

November, 1986

ABSTRACT

The computer architect has no easy way of making performance tradeoffs while designing a new computer. This thesis provides a taxonomy of over 75 weighted average computer performance models which aid the architect in comparing alternative designs. These models have been implemented in the KEE expert system building tool. The user can interact with the models using a mouse and visual images to change values and monitor results.

Key Words and Phrases: computer performance models, analytical models, computer design, expert sytems, KEE.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

". . . the virtuosity of a model builder is reflected in the simplicity of the model yielding information of the desired accuracy." [Kimbleton]

PROBLEM: NO PERFORMANCE FEEDBACK

1.1 Overview

The computer architect has no easy way of making performance tradeoffs while designing a new computer. The next section explains the importance of providing immediate performance feedback to the designer. Section three and four review the various types of performance models and their uses. The final section of this chapter briefly reviews the current state of the art in design tools.

1.2 Purpose of Performance Models

A designer is primarily interested in two things: function and performance. This is true whether he is designing computers or tractors or spaceships. First, it must be able to accomplish its purpose. And second, it is desirable that it accomplish its purpose efficiently. This thesis will concentrate on performance.

Efficiency is generally measured in cost and time. A designer needs feedback on the results of his

design in order to determine its efficiency. In some situations, feedback is not available until the design has been actually built and tested (Figure 1-1). This was tragically true of the space shuttle's booster rocket joints, and is often true of computer systems. The performance of a computer is frequently not known until many years after the computer has been produced and extensively benchmarked [Cragon 86B].

It is preferable to provide immediate feedback on a design prior to actually building it (Figure 1-2). This shortens the duration and cost of the design loop. The importance of these early design decisions is shown in Figure 1-3. In the early stages of a design, small changes can have a significant effect on the final cost, and performance, of the product. But as the design progresses, it becomes more difficult to improve the design. This is because each decision along the way constrains future decisions. In the end, it takes herculean efforts to improve the performance to any significant extent. This shows the importance of effective design tools early in the design lifecycle. The next section examines some of the different types of performance models that are used to provide this immediate feedback.

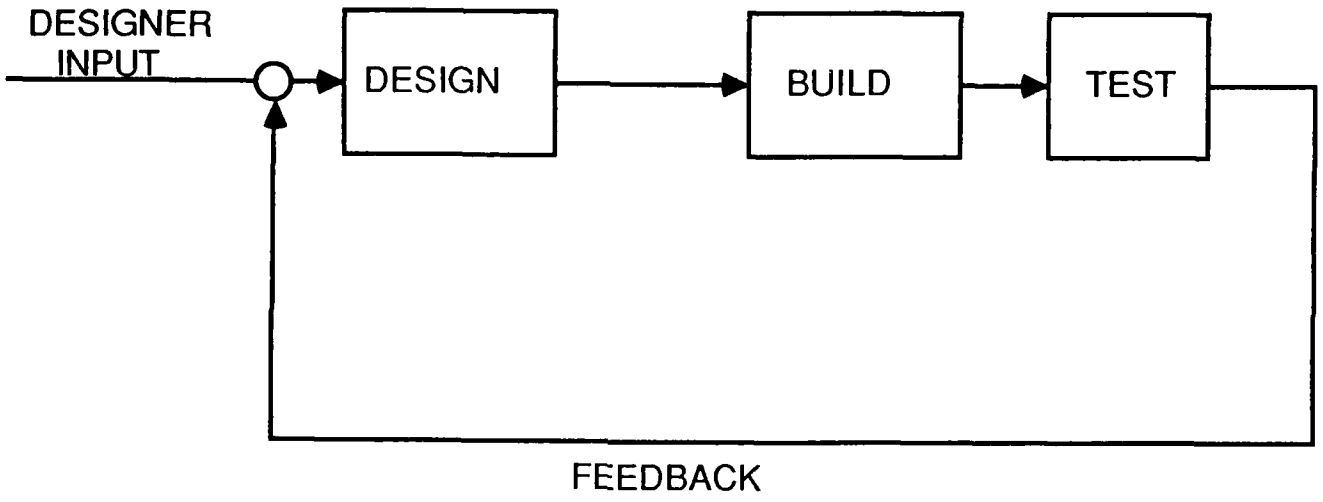


Figure 1-1 Feedback after build and test.

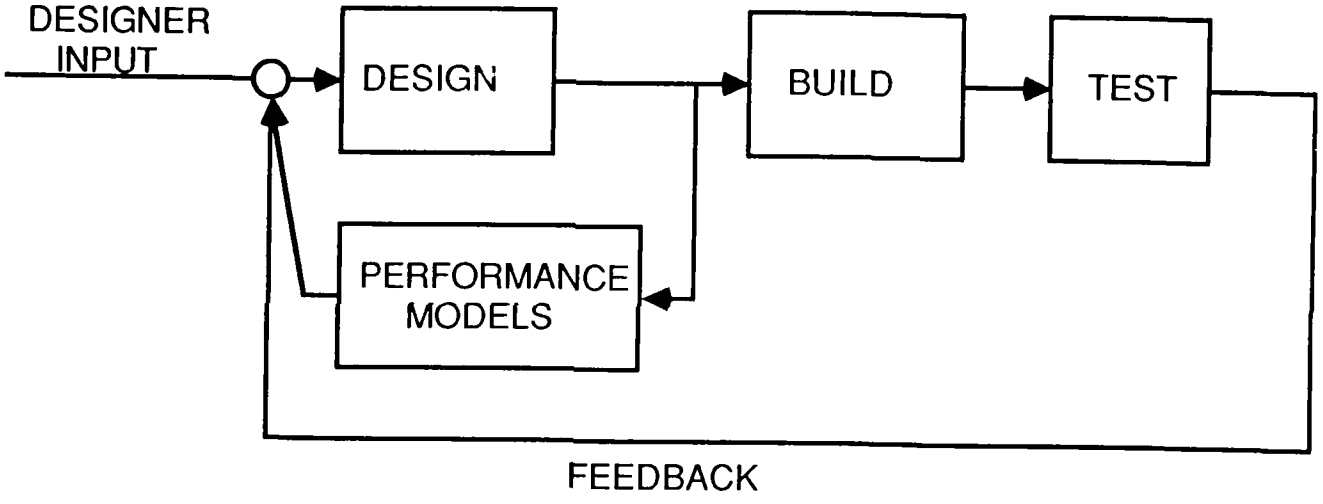


Figure 1-2 Feedback prior to build and test.

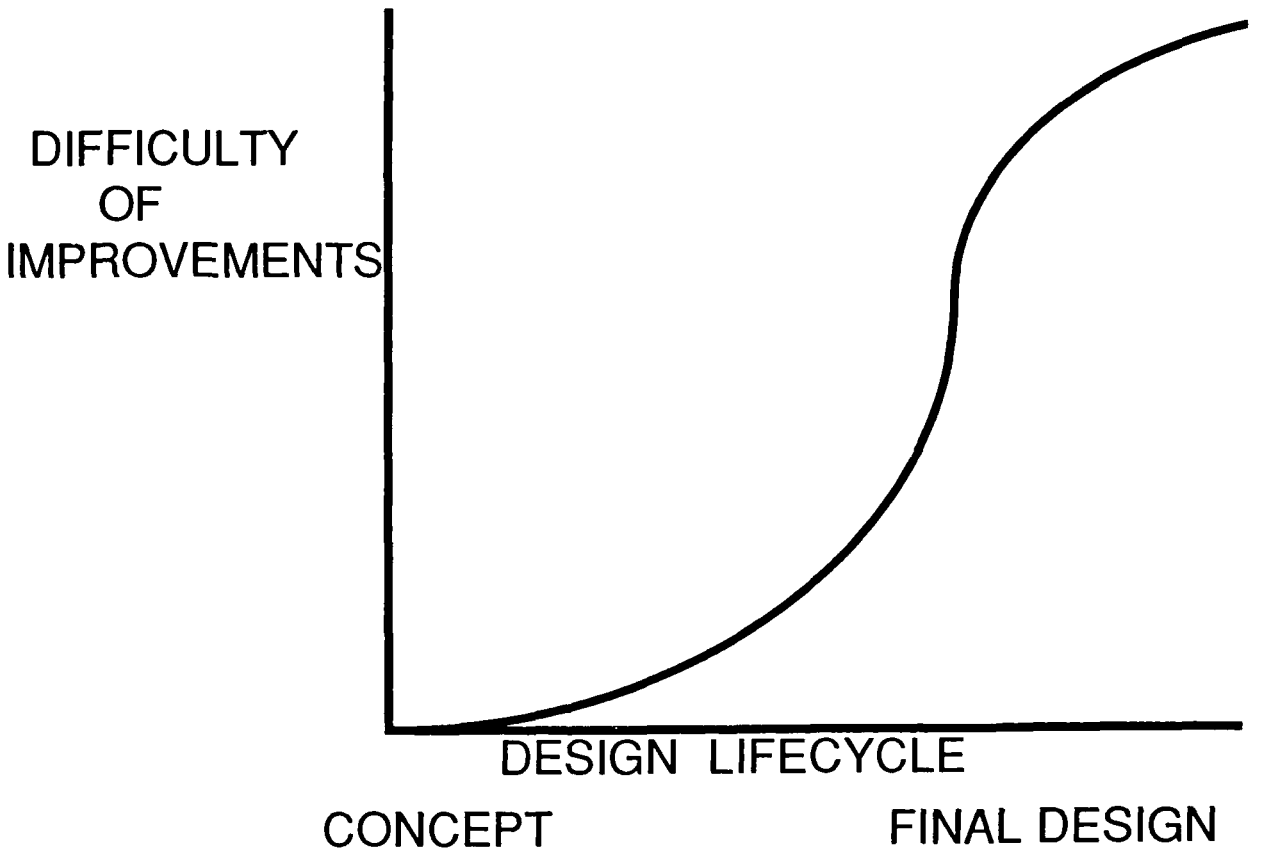


Figure 1-3 Final cost and performance determined early in lifecycle.

1.3 Types of Performance Models

A broad spectrum of performance model types are shown in Figure 1-4 [Allen]. Model types range from those that are low in both cost and accuracy, to those that are high in both cost and accuracy. The dividing lines between the classes of models is often not clear-cut.

The first class of models at the low end of the scale are rules of thumb. These are inexpensive and easy to apply, but are general in nature. An example rule of thumb is Grosch's Law which states that the power of a computer system varies with the square of its cost.

The second class of models is linear projections. These are also known as empirical models. These are constructed by fitting simple equations, usually linear, to measured performance data [Bard].

The third class, analytic models, consists of an equation or sets of equations which approximate the relevant aspects of a system's behavior [Bard]. Analytical models can be further broken down into weighted averages and queueing models. Weighted averages are not random processes. They are completely deterministic. Queueing models, on the other

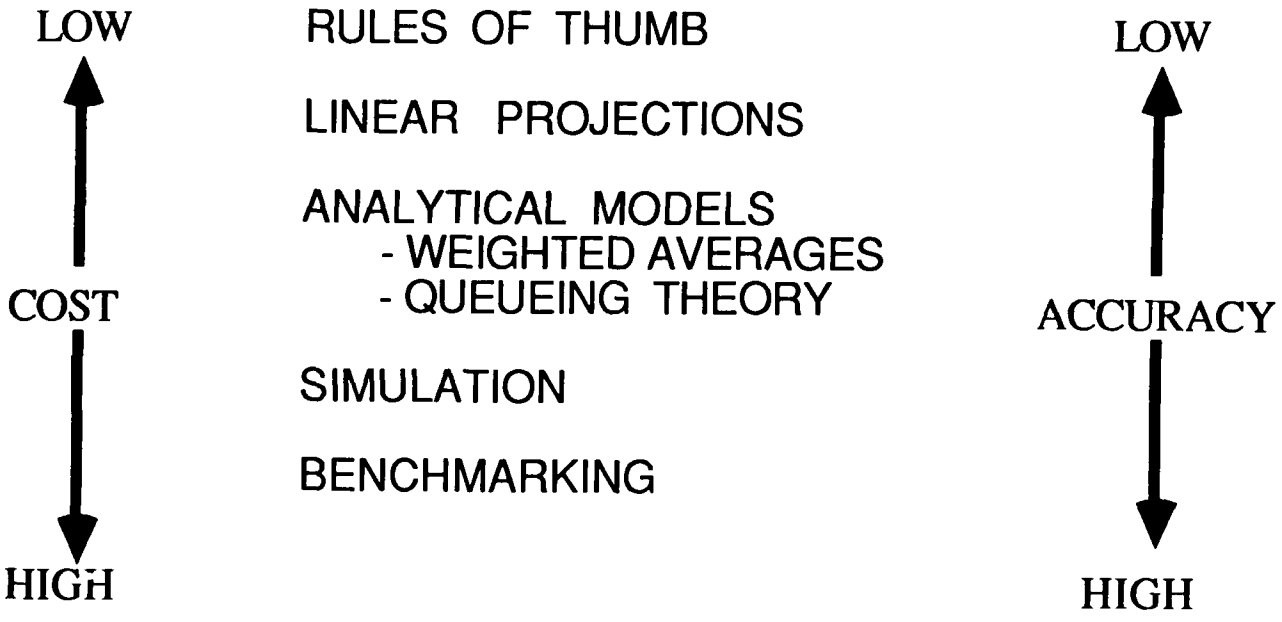


Figure 1-4 Types of Performance Models.

hand, use probability theory to generate the time between events and other data.

A simulation model is a computer program which imitates the behavior of the real system [Bard]. A simulator is capable of as much accuracy as the designer cares to build into it. The limitations are cost and time.

The final model is benchmarking. Actually, this isn't really a model; it's the real thing.

> The technique sounds disarmingly simple. You simply collect a representative sample of your workload, run it on the proposed machine, and measure the performance. Unfortunately, all three of these tasks are very difficult in practice [Allen].

1.4 Uses of Performance Models

Performance modeling can be applied in four situations: design, fine-tuning, capacity planning, and selection [Bucci].

The design level requires general models for comparing alternative architectures. Fine-tuning is aimed at optimizing system performance for a given system configuration and workload. Capacity planning provides estimates of hardware, software, and human resources needed to cope with the expected demand. Selection, or purchase, compares the cost and

performance of available systems to user requirements. This thesis deals with performance models that are useful to the high level architect in designing the computer. These models generally fall into the categories of rules of thumb, linear projections, and non-stochastic analytical models. Queueing models are not included because they are more suited for fine-tuning than for the initial high level design.

1.5 State of the Art

Although a number of automated design tools are available at the logic and implementation levels, virtually no automated high level architecture design tools are available. However, many performance models are available in the literature which are useful at the architectural level. These described in more detail in Chapter 2. At the logic level, tools such as LOGSIM are available on microcomputers for simulating a proposed logic design. At the implementation level, tools such as CAESER and SPICE aid in the design and simulation of VLSI layouts. But the high level computer architecture has virtually no automated design tools for conducting tradeoff studies. Tradeoffs, if done at all, are generally done with a few simple calculations on scratch

pads. One of the primary limitations is the experience of the architects on the project. This is illustrated by the design approach used on the Hewlett-Packard Spectrum:

> For the first five weeks of the project, every team member was asked to present all the computer architectures and ideas he or she was familiar with. [Electronics]

This is a classic example of a knowledge bottleneck which makes this area a likely candidate for an expert system.

CHAPTER 2

SOLUTION: TAXONOMY OF PERFORMANCE MODELS

2.1 Workload, Performance, and Cost

To design an efficient new computer, an architect must be familiar with three general issues. First, he should know what the computer will be asked to do. This is called the workload model. Second, he must have cost models so that the final cost/performance ratio will be competitive. And finally, he needs to be able to compare the efficiency of alternative architectures for performing that work. These are called performance models.

Workload models contain factors such as scalar to vector ratio; vector length; frequency of tag processing, garbage collection, and procedure calls; and depth of procedure call nesting and recursion.

Cost models include the cost of design, production, and possibly maintenance. These costs are often divided into recurring and non-recurring costs.

Performance models provide a quantitative comparison between architectures. Some architectures

can handle specific tasks better than other architectures. Analytic performance models range from high level models comparing the cost/performance ratios of entire computer systems, to very detailed models of specific subsystems such as disk latency or pipeline throughput.

2.2 Taxonomy of Performance Models

This thesis provides a taxonomy of over 75 analytic computer performance models which aid the high level computer architect in comparing alternative designs.

The ultimate goal is a generalized computer performance model. The immediate goal of this thesis is to identify existing performance models which are useful to the high level architect, as well as to identify voids where models are needed.

The structure of this taxonomy is hierarchical and modular. It starts with models at the highest level (computer), and branches out to provide models for the various components (memory, CPU, etc.). The original concept for this taxonomy is shown in Figure 2-1. This taxonomy is breadth, rather depth, oriented. There are

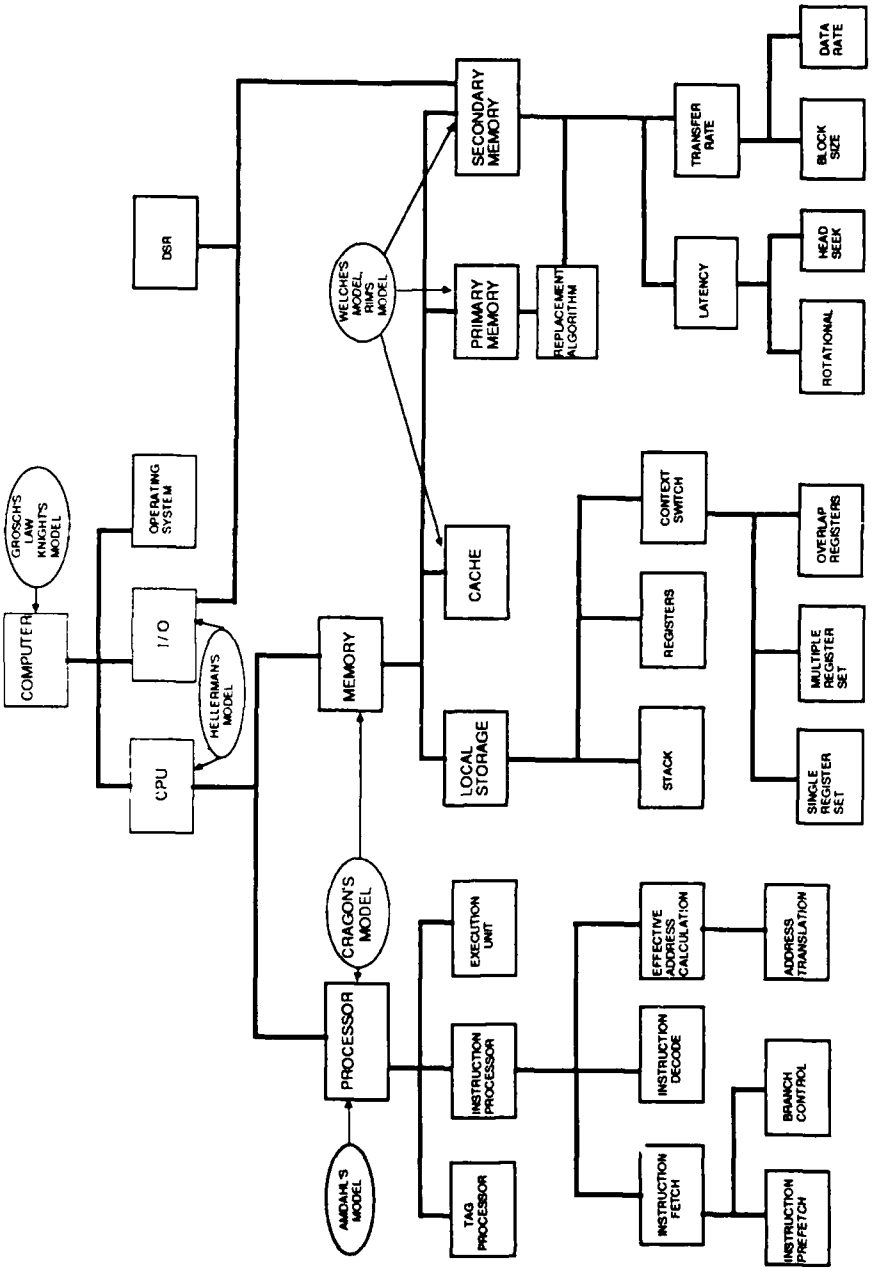


Figure 2-1 Original Taxonomy.

a large number of detailed models in the literature, and new models are constantly being developed. The modular and hierarchical design allows the addition of new models and other projects.

The scope of this taxonomy is limited to single processors and does not include models for operating systems or software.

Although a large body of literature is devoted to performance modeling and performance evaluation, most of it is targeted at the computer user rather than the computer designer. This is understandable since the users greatly outnumber the designers. This thesis, however, examines models suitable for the computer designer, or high level architect.

Performance models for the architect tend to be simpler than the models used to evaluate the performance of actual computers. Complex performance models for actual computers can be readily justified by the expected return on investment (ROI) to both the producer and user. The architect, on the other hand, has difficulty justifying a great deal of analytical effort on some aspect of the design of a hypothetical computer which may never make it to the market, or even to the laboratory. In addition, simple models are necessary to

prune the exponentially large search tree. Simple models are necessary to cope with the vast number of decisions and tradeoffs to be made. But the simplicity of a model doesn't reflect negatively on its merit. Indeed, "the virtuosity of a model builder is reflected in the simplicity of the model yielding information of the desired accuracy." [Kimbleton] As the design of a computer progresses, certain critical areas may require more complex models.

2.3 Periodic Table of Performance Models

During the initial stages of research on this project, I considered the potential for a "Periodic Table of Computer Performance Models." This was suggested by Harvey G. Cragon. The concept is analogous to the periodic table of elements discovered by Mendeleyev. We saw two major benefits of such a table. First, it would provide a useful structure for illustrating and studying the relationships between various performance models. The periodic table of elements provides such a logical structure for the study of the elements. The second benefit would be the ability to identify blanks in the table corresponding to models which had not yet been developed. The table of

elements provides this type of predictive ability. Although the hierarchical structure chosen to represent performance models differs from the row and column structure of the elements, some parallels do exist. This section briefly reviews the history and structure of the periodic table of elements, and examines the parallels between the two structures.

Over 52 years passed between the first attempts (1817) to establish relationships between the elements, and Mendeleyev's presentation of the periodic law (1869). The first periodic table was incomplete and many of the elements were out of order (Figure 2-2). But its key value was in providing a logical structure for the study of the elements. Another 53 years passed before the theoretical basis for the periodic law was explained by Niels Bohr (1922) and others in terms of the electron structure of the atom. In hindsight, we now know that the periodicity of the properties of the elements is caused by similarities in their electron shells. Each successive row of the periodic table represents an additional shell which can hold multiple electrons. And each successive column of the table represents an additional electron in the outer shell.

Group 0	I		II		III		IV		V		VI		VII		VIII
	a	b	a	b	a	b	a	b	a	b	a	b	a	b	
	H 1														
He 2	Li 3		Be 4		B 5		C 6		N 7		O 8		F 9		
Ne 10	Na 11		Mg 12		Al 13		Si 14		P 15		S 16		Cl 17		
Ar 18	K 19	Cu 29	Ca 20	Zn 30	Sc 21	Ga 31	Ti 22	Ge 32	V 23	As 33	Cr 24	Se 34	Mn 25	Br 35	Fe 26, Co 27, Ni 28
Kr 36	Rb 37	Ag 47	Sr 38	Cd 48	Y 39	In 49	Zr 40	Sn 50	Nb 41	Sb 51	Mo 42	Te 52	Tc 43	I 53	Ru 44, Rh 45, Pd 46
Xe 54	Cs 55	Au 79	Ba 56	Hg 80	57–71*	Tl 81	Hf 72	Pb 82	Ta 73	Bi 83	W 74	Po 84	Re 75	At 85	Os 76, Ir 77, Pt 78
Rn 86	Fr 87		Ra 88		Ac 89		Th 90		Pa 91		U 92		Np 93		Pu 94, Am 95, Cm 96

*Lanthanum and the lanthanons

Figure 2-2 Short-period form of periodic system of elements.

Each element differs from the ones surrounding it by one electron or one shell. The modern version of the periodic table is shown in Figure 2-3.

Perhaps the lesson to be learned from the table of elements is that the structure chosen to represent the models should parallel the underlying structure of the computer itself. From this standpoint, a hierarchical representation seems to be a logical choice to represent computer performance models. The computer is composed of major components, each of which can be broken down into smaller components. This hierarchical structure at least partially provides the two major benefits initially cited. It provide a useful structure for illustrating and studying the relationships between performance models. And to the extent that the general structure of a computer is known, it can identify where models are needed. Its predictive abilities are limited in that it can't predict new computer architectures. But once developed, they can be added to the hierarchy. Actually, this is not so much different from Mendeleyev's addition of a new group (column) to the periodic table after the discovery of the noble gases (helium, neon, argon, etc.) by Rayleigh and Ramsay.

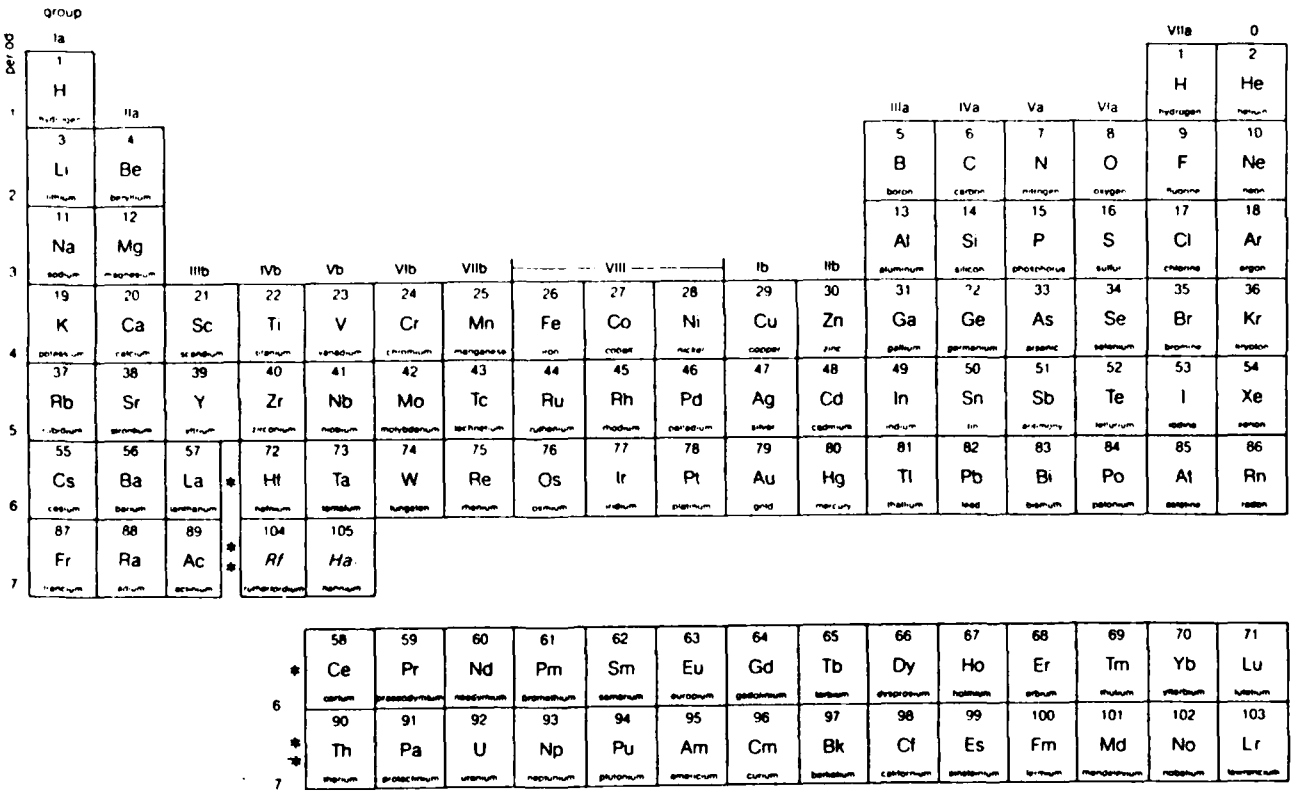


Figure 2-3 A modern version of the periodic table of elements.

Note: The hierarchical structure for the performance models was chosen before examining the development of the periodic table. Some parallels can be drawn between the two structures. However, it is not clear whether these parallels are of any practical value.

CHAPTER 3

SELECTION OF IMPLEMENTATION TOOLS

3.1 Advantages of Artificial Intelligence Techniques

One of the thrusts of this thesis, and related architecture design projects, is an examination of the benefits of artificial intelligence (AI) techniques in the area of computer design. Most of the implementation is being done on LISP machines, also called symbolic processors. One of the techniques which has emerged from the artificial intelligence community is object-oriented programming. The basic concept behind this programming paradigm, or technique, is that all the information related to a certain object is stored with that object. Object-oriented programming is discussed further in sections 3.3, 4.4, and in Chapter 6 (Lotus 1-2-3).

3.2 Use Existing Expert System Building Tool

Over the last three years, specialized programs have been developed for LISP machines which aid in the development of expert systems. These are called expert system building tools. Expert systems are programs

which contain a great deal of knowledge about a narrow area, and are capable of answering specific questions about that area. Expert system building tools provide a preformatted structure for that knowledge and a mechanism for asking questions. The purpose of this project was not to build a new AI tool. Instead, I decided to use an existing expert system building tool.

3.3 KEE Selected

I used the KEE expert system building tool because it has a number of useful features and was available. KEE stands for Knowledge Engineering Environment and is sold by Intellicorp.

KEE has a number of features which appeared to be well-suited for representing the taxonomy. KEE provides a hierarchical database (or knowledgebase) using frames (called units in KEE). This hierarchical structure allows the user to make a large model from a collection of small models. The small models are contained within the large model. This provides direct support for grouping knowledge in modules organized by the part of the computer. This approach of grouping knowledge in modules by part was used by Ford Aerospace in developing STARPLAN, an expert system for diagnosing

problems with satellites [Fikes]. KEE automatically generates a graph of the hierarchical database. Object-oriented programming is another important aspect of the KEE system.

> Object-oriented programming allows descriptive and procedural attributes of an object to be associated directly with that object in a frame. Thus descriptions and programs can be attached to objects. This makes a very important contribution to the needs for modularity and rapid prototyping. In KEE this object-orientation is a unifying factor because representation, rule-based reasoning, access to Lisp, graphics and active values, are all object-oriented. [KEE]

This makes it possible to send messages to units, and from one unit to another. The user can also interact with a visual image of his model.

A final reason for the selection of KEE was its availability. Intellicorp provided two copies for our Explorer LISP machines at no charge. In addition, students in the Computer Science department have used KEE.

CHAPTER 4

IMPLEMENTATION METHODOLOGY

4.1 KEE Knowledge Structures

This chapter presents only the information necessary to illustrate the major concepts and features of KEE used in this thesis. The details necessary to actually create and use a knowledgebase are not discussed. The interested reader should consult the KEE Users Manual [KEE].

This section provides an overview of the knowledge representation structures and relationships which are supported by KEE. These structures and relationships are important because they constrain the manner in which the desired knowledge can be represented. If the desired knowledge doesn't naturally fit the structure provided, it will be more difficult to codify and use the knowledge.

Starting at the top, the entire "program" is called a knowledge base (kb). The basic building blocks of a knowledge base are units. A schematic view of the structure of a unit appears in Figure 4-1. Examples of units are COMPUTER, MICROCOMPUTER, IBM.PC AND MACINTOSH. Note that periods are used as characters in names for

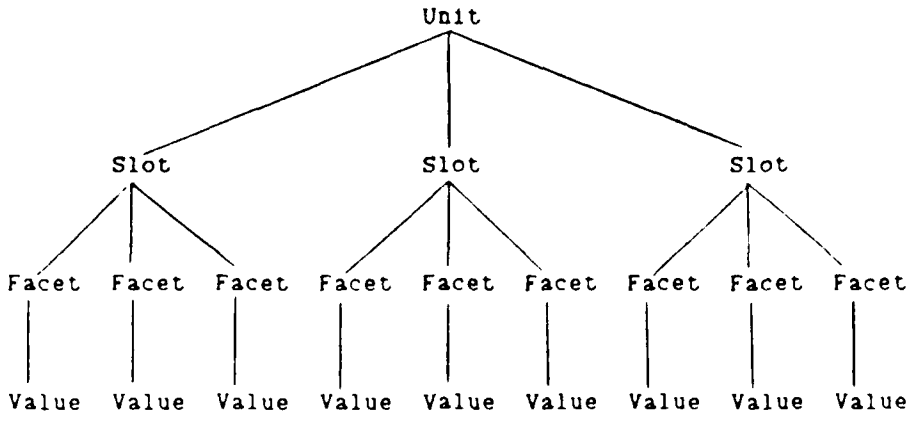


Figure 4-1 A schematic view of the structure of a Unit.

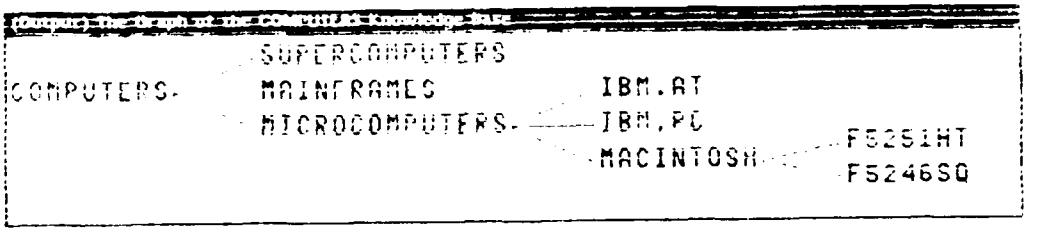


Figure 4-2 Subclass and Membership Relationships.

readability. Each unit can have a number of slots which describe characteristics, or attributes, of that unit. Examples of slots for a COMPUTER would be COST, MEMORY.SIZE, and WORD.SIZE. The values a particular slot can take are constrained by facets. Each slot has the following six facets predefined:

> Comment -- An optional textual description.
>
> Inheritance role -- This specifies how the slot will inherit attributes from units above it in the hierarchy.
>
> Value Class -- This facet specifies what kinds of values will appear in the value facet.
>
> Cardinality max -- This specifies the maximum number of values that can appear in the value facet.
>
> Cardinality min -- This specifies the minimum number of values that can appear in the value facet.
>
> Value -- This is where the value or values of the slot are stored. [KEE]

For example, the Value Class facet of the MEMORY.SIZE slot on the MACINTOSH would be the range 128 to 1,000 (k bytes). If the user tried to enter a value outside this specified range, KEE would generate an error message. The facet we are usually most interested in is the Value facet. Note that each of the facets has a value, but only one of the facets is the Value facet. Now that we

understand the structure of units we can look at the relationships between units.

Relationships between units are important in building a knowledge base. The two relationships supported by KEE are subclass and membership. A subclass is a specialized category, or class, of some object whereas a member is a particular object. Figure 4-2 illustrates this. The unit COMPUTERS is broken down into three subclasses: SUPERCOMPUTERS, MAINFRAMES, and MICROCOMPUTERS. MICROCOMPUTERS is further broken down into three subclasses: IBM.AT, IBM.PC, and MACINTOSH. Subclass links are represented by solid lines. Membership is illustrated by the unit F5251HT and F5246SQ. These are actual MacIntosh computers. Membership links are represented by dotted lines.

One of the primary benefits of these relationships is inheritance. Information can be passed automatically to subclasses. For example, the unit COMPUTERS is given slots for COST, MEMORY.SIZE, and WORD.SIZE. Everything beneath COMPUTERS in the hierarchy automatically inherits those slots. This makes it possible to quickly create detailed data structures.

4.2 Subclass/Member Taxonomy Representation

This section will first present the relationships which are of interest in the performance model taxonomy. It will then examine the advantages and disadvantages of the subclass and member relationships (or links) in representing the performance model taxonomy.

In the performance model taxonomy I am primarily interested in two relationships: the parts of a computer, and performance models for each part. The structure for the taxonomy of performance models is based on the structure of the computer itself. This allows the models to be organized in a logical fashion. Each model is a unit, or hierarchy of units, which is linked to the computer part which it models.

In KEE, the subclass relationship is intended for units which are specialized versions of their parent. For example, MICROCOMPUTERS are a specialized type of computer. It is convenient for MICROCOMPUTERS to inherit attributes from COMPUTERS.

In the performance model taxonomy, I have used the subclass relationship (solid line) to represent the parts of a computer. The disadvantage of this representation is that the subclasses, i.e. CPU and IO,

are not specialized versions of their superclass, i.e. COMPUTER, but rather parts of the superclass. CPU and IO don't inherit any attributes from COMPUTER.

Inheritance is of little value in this taxonomy because component parts usually don't inherit slots from the unit above them.

In KEE, the member relationship indicates that the unit is an actual instance, or instantiation, of its parent class. For example, F5251HT is an actual instance of the MACINTOSH subclass of COMPUTERS.

In the taxonomy I have used the member relationship to link models to the part they model. The disadvantage of this representation is that a model is not an instantiation of the part it models. For example, CRAGONS.MODEL of CPU's is not an actual instance of a CPU. On the other hand, if we view the CPU unit as being a class of CPU models, as opposed to actual CPU's, then the relationship is more appropriate.

I have mentioned some of the disadvantages of using the subclass and member links for the taxonomy. There are, however, some definite advantages. First, the use of these links is well-supported by the KEE user interface. It is very easy to create, add, and delete these links. Second, a graph of the hierarchy of links

is automatically generated by the KEE system. The user can simply look at the database to see the relationships between parts and models. New units (both parts and models) can be added by simply mousing on the appropriate portion of the graph. Finally, just because the inheritance feature of the relationships is not used doesn't mean the relationships can't be used. Although this version of the taxonomy doesn't use inheritance very much, future versions might use it. One possible way of doing this would be to reverse the tree. Start with the detailed models at the left, or root, of the tree, and allow their computed characteristics to be inherited up to the computer. This approach might be useful after the taxonomy is developed, but it is easier to develop the taxonomy by starting at the top and working down.

4.3 Nested Slot Taxonomy Representation

An alternative method of representing the performance model taxonomy is to nest each successive level of the taxonomy within the slot of the previous level. Release 2.1 of KEE now allows the values of slots to be units themselves.

> You are not restricted to expressing all unit-unit relations via class-subclass or class-member links. For example, whole-part relations can be expressed by giving units a PARTS slot which will contain the names of the units which are parts of the unit having the slot [KEE RN].

This appears to be the relationship we are looking for. Unfortunately, this approach has a number of problems. First, autocreation does not work when adding units to the PART slot. The Value Class facet of the PARTS slot is set to UNIT. Autocreation should create new units as soon as they are added as Values of the PARTS slot. But it doesn't. The units must be created as members of some dummy class first. Otherwise KEE doesn't recognize them as units. The name of that dummy class is put in the Value Class.

A second problem has to do with the SLOT.GRAPH function for graphing the resulting relations. I couldn't get it to work.

A third problem is that both the PARTS slot and the MODELS slot can't be displayed simultaneously. The MODELS can't be displayed associated with the appropriate part. This is the most serious problem with this approach.

4.4 Object-Oriented Programming and Messages

Each of the UNITS in a KEE knowledge base is also known as an object. Information related to a particular UNIT or object is stored with the object. "To take advantage of the KEE system's object-oriented capability it is important to be able to attach procedural knowledge [i.e. functions] to the objects in the knowledge base." [KEE] This is done by putting LISP code which will perform some desired function into a slot in the appropriate object.

Functions which are stored in slots are known as methods. For example, an object called CIRCLE could have two slots, RADIUS and AREA. The RADIUS slot simply holds a number corresponding to the radius of the CIRCLE. The AREA slot, on the other hand, holds a method or function which computes the area. The formula to compute the area of a circle is of course $(PI)(R)^2$, or in LISP, (* PI (expt R 2). You have to tell KEE what object and slot hold the value of RADIUS. So the formula would be:

```
(* PI (EXPT (GET.VALUE 'CIRCLE 'RADIUS) 2))
```

So now any other object which needs to know the area of CIRCLE can simply send a message to the AREA

slot and it will return the value of the area. The form of this message would be:

(UNITMSG 'CIRCLE 'AREA)

Using this arrangement, the area is computed everytime it is needed.

Now let's assume we want the value of the area to always be available (see section 4.6 on Active Images). The AREA slot can not hold both the method for computing the area, as well as the value of the area. So we need to set up a second slot for holding the value of the area which we will call AREA.VALUE. The AREA method could be rewritten to put the value in the AREA.VALUE slot as follows:

(PUT.VALUE 'CIRCLE 'AREA.VALUE (* PI (EXPT (GET.VALUE 'CIRCLE 'RADIUS) 2)))

The disadvantage of this is that the RADIUS could change, but the AREA.VALUE slot would still have the value of the old area. We need the capability to update the AREA.VALUE slot whenever the RADIUS changes. This is done using Active Values.

4.5 Active Values

The Active Value facility allows us to monitor a particular slot and take some action if it changes.

Using the example from the previous section, we can attach an Active Value unit to the RADIUS slot of CIRCLE and program it to update the AREA.VALUE whenever the RADIUS changes. The simplest way to do this is to program an Active Value unit, let's call it AV.AREA, to send a message to AREA anytime the RADIUS changes. All Active Value units have a slot call AVPUT which is triggered whenever there is a change in the slot to which it is attached. So we put the following code in the AVPUT slot:

```
(UNITMSG 'CIRCLE 'AREA)
```

So now anytime the value in RADIUS slot changes, the AV.AREA Active Value unit sends a message to AREA to recompute the area. In summary the Active Value facility allows us to keep related values updated. The final KEE capability we will discuss is visual monitoring of values.

4.6 ActiveImages User Interface

The ActiveImages package allows the KEE user to create graphic displays for both viewing and modifying KEE objects. "The ActiveImages package combines the object-oriented features of the KEE system and the power

of graphics to convey large amounts of information" [KEE AI]. Typical uses of images include:

- Monitor a value as it changes by watching an image
- Set a value by altering the image value with the mouse
- Represent an object [by] using an ICON image to stand for a unit
- Show relationships between different objects and values by using various images on an image panel to show how different things change relative to each other. [KEE AI]

ActiveImages can be attached to slots, units, and knowledge bases. Generally, single images are attached to slots and represent the value of that slot. A number of single images are often grouped together on an image panel which is attached to a unit or a knowledge base.

The six classes of images are numeric, state, method, general, position, and text. With many of these images, there is a choice between "read only" and "read/write." The read only image only displays the internal value, while the read/write image can be used to change the internal value by mousing on it. For example, the digimeter is read only, while the digiactuator is read/write. Many of the images allow the user to set an alarm value. "The image flashes

whenever the value of the slot is set to the predetermined alarm value." [KEE AI]

Numeric images are the most prevalent class of images used in this thesis. Types of numeric images include digital readouts, bargraphs, stripcharts, and histograms.

"State images are designed to be attached to slots that have only a small set (2 to 8) of possible values or 'states'." [KEE AI] For example, a small two-state image might allow the user to select between garbage collection "on" or "off". With the PUSH.BUTTON state image, more than one state can be selected. The remaining four classes of images are used very little or not at all in this thesis, hence their descriptions are brief.

A method image simply sends a message to that method when it is clicked on with the mouse.

General images display the value of a slot in a box.

Position images are used to plot x-y values on a graph.

The text image as its name implies, displays text.

CHAPTER 5

Format for Performance Models

5.1 Model Format

This section will explain the standard format which is used to summarize each of the performance models. The models themselves are listed in Appendix I. The standard format is shown in Figure 5-1.

PART NAME describes the computer part or function which is being modeled. At the upper levels of the hierarchy the PART NAME usually refers to a physical part such as CPU, MEMORY, PROCESSOR, etc. At the lower levels of the hierarchy the distinction between a *physical part and a specific function* becomes blurry. For example, CONTEX.SWITCH and ROTATIONAL.LATENCY.

MODEL NAME is the name, or brief description of, the model. For example, AMDAHLS.MODEL or PAGE.FAULT.TIME.

REFERENCE provides my source for the model. The complete reference is listed in the bibliography. Page numbers are given where appropriate. (The reference cited may not be the original source for the model).

DESCRIPTION verbally explains the nature and/or purpose of the model. The description is sometimes

PART NAME:
MODEL NAME:
REFERENCE:
DESCRIPTION:

MODEL:

VARIABLES:

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

Figure 5-1 Standard Format For Performance Models

omitted when the nature of the model is clear from its name.

MODEL is the actual mathematical equation for the model.

VARIABLES provides a verbal description of each of the variables in the equation. The description is sometimes omitted when the nature of the variable is clear from its name.

RELATIONSHIPS, if applicable, show how this model is related to other models. The two possible types of relationships are Superclass and Subclass. For example, the NUMBER.OF.PAGE.FAULTS model has Superclass = WORKLOAD.VECTOR and Subclass = P.MISS. This relationship information is identical to that which is displayed in the taxonomy. This information was used in building the taxonomy. It is easier to figure out the relationships just by looking at the taxonomy. The taxonomy is shown in Figure A-1 at the back of the Appendix. This makes it easy to refer to while reading the performance models in the Appendix.

5.2 Variable Names

The names of variables used in the performance models in Appendix I are usually the same as the name

used by the original author of the model. This makes it easier for the reader who wishes to refer to the source for more detail. However, variable names were changed in a number of cases to avoid duplication and maintain consistency. Duplication must be avoided because each unit in KEE must have a unique name. Many of the variables are units. For example, PM.BANDWIDTH, DISK.BANDWIDTH, and PROCESSOR.BANDWIDTH. Different units, however, can have identical slot names. This is often the case for similar units; i.e. P.MISS, C.MISS, and P.OVF.

The second reason for changing variable names was to maintain consistency between models, and between the KEE knowledgebase and this thesis. KEE doesn't support subscripts, superscripts, or lowercase letters (except lower case letters in comment fields). But KEE does support long names, which has an additional benefit of making variables self-documenting. So instead of T_t, I used TAG.PROCESSING.TIME. Periods are used for readability. However, long names make equations hard to understand. For this reason, and to make it easier to understand the large number of models, the following conventions were often used:

P.XXX - Probability of XXX, values 0 to 1.

T.XXX - time required for XXX.

N.XXX - number of XXX, usually positive integers.

CHAPTER 6

THE SYMBOLIC PROCESSOR MODEL

6.1 Background

The most highly integrated set of models in this taxonomy are those which fall under the SYMBOLIC.PROCESSOR.MODEL. This model has eight image panels associated with it which allow the user to view, and interact with, this model. The Symbolic Processor Model was originally implemented by Harvey G. Cragon under the Lotus 1-2-3 spreadsheet program on a Texas Instruments Business professional (IBM PC compatible). [Cragon 86B] All but one of the unit performance models in the Symbolic Processor Model were taken from the literature. The one exception is the "Probability of procedure call overflow" model which Cragon developed. One of the significant aspects of Cragon's work was that Lotus 1-2-3 was used as an object-oriented programming environment. Smaller models were tied together into larger models by using the inherent constraint propagation features of Lotus. This is similar to message passing between units. With this model, the computer designer can change various parameters of the

architecture and observe the effect on the execution time. The parameters fall under five major categories: memory hierarchy, instruction processor, tag processor, procedure calls, and garbage collection.

6.2 Image Panels

I implemented the Symbolic Processor Model under KEE, an expert system building tool, on the TI Explorer LISP machine. This implementation is very similar to a spreadsheet environment in that the user can change a value and watch the results ripple across the screen. Each of the five categories has an image panel associated with it (Figures 6-1 through 6-5). A sixth image panel summarizes the workload and the execution and disk times from the first five panels (Figure 6-6). These image panels are a solution of an instance of the model.

Two additional panels allow the designer to quickly create a plot showing the relationships between any variables in the model. The PLOTTER panel displays a normal-normal plot (Figure 6-7). The PLOT.LOG2.LOG10 panel plots the base 2 logarithm of the x variable, and the base 10 logarithm of the y variable (Figure 6-8). This log-log plot displays a straight line relationship

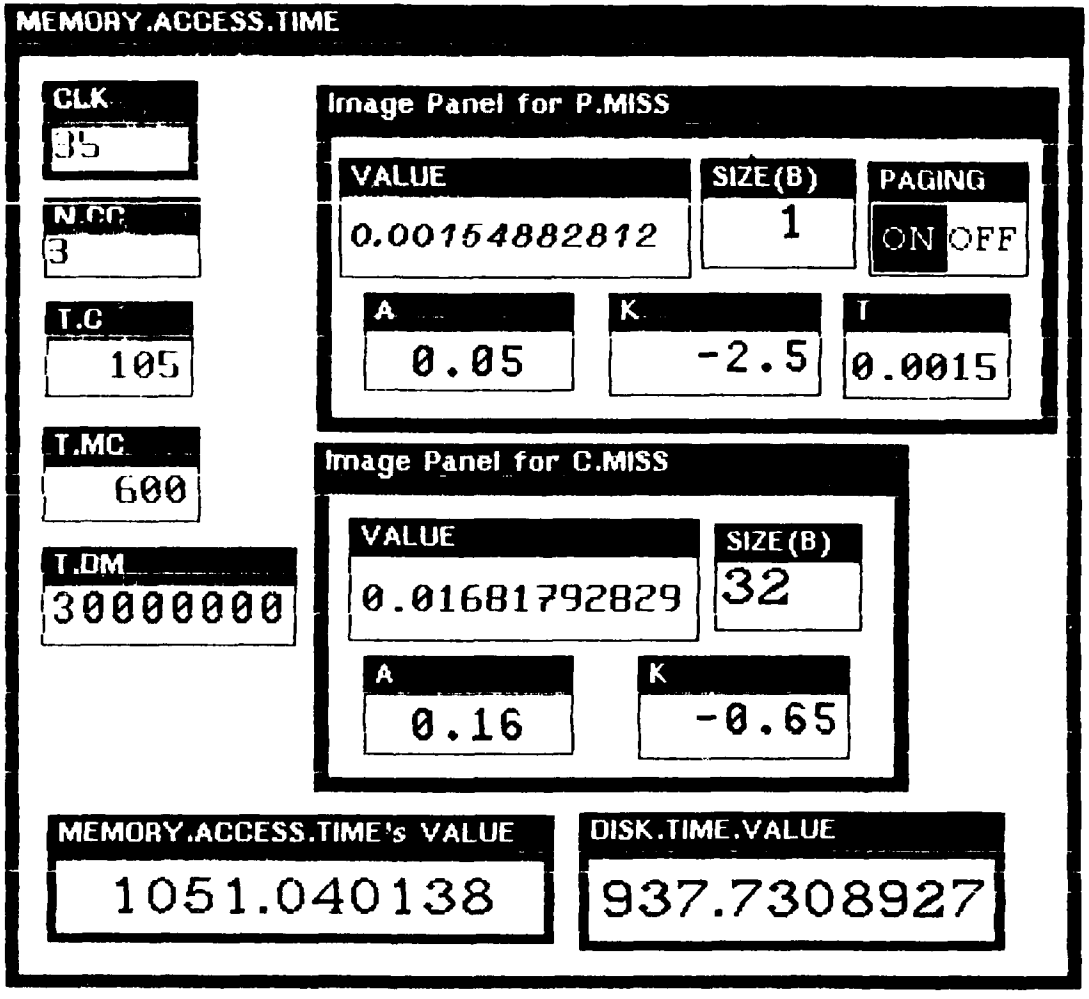


Figure 6-1 Image panel for the Memory hierarchy.

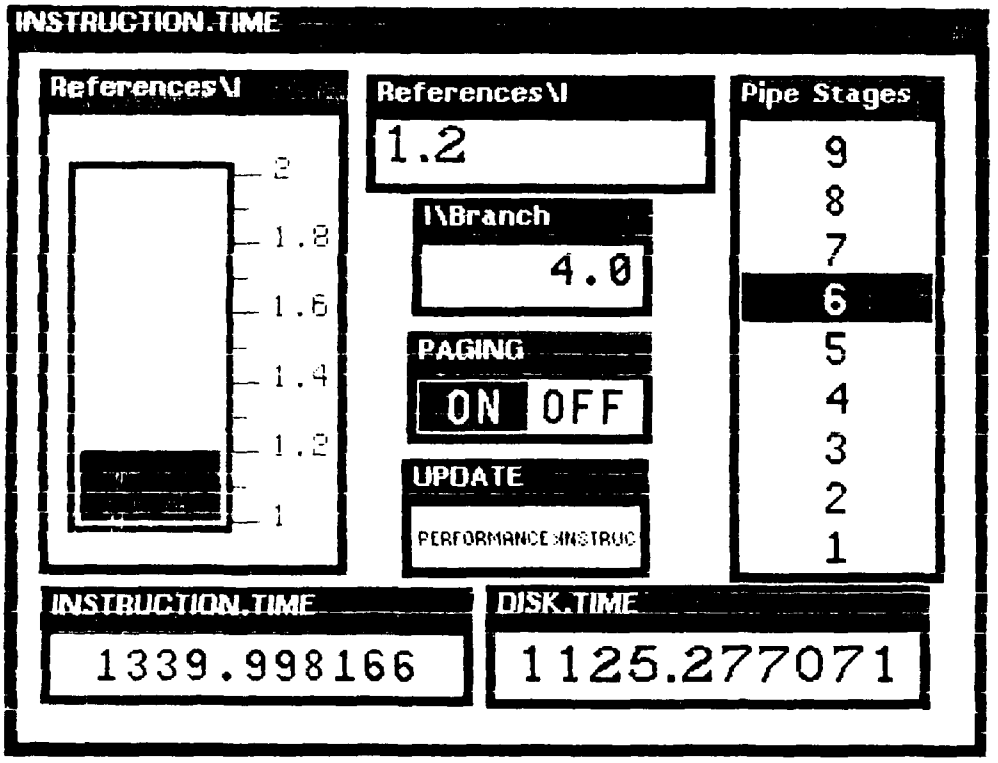


Figure 6-2 Image panel for the Instruction Processor.

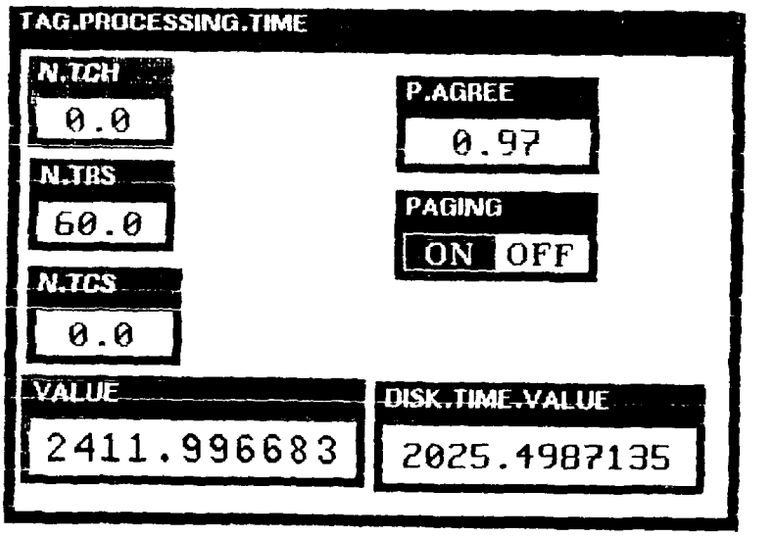


Figure 6-3 Image panel for the Tag Processor.

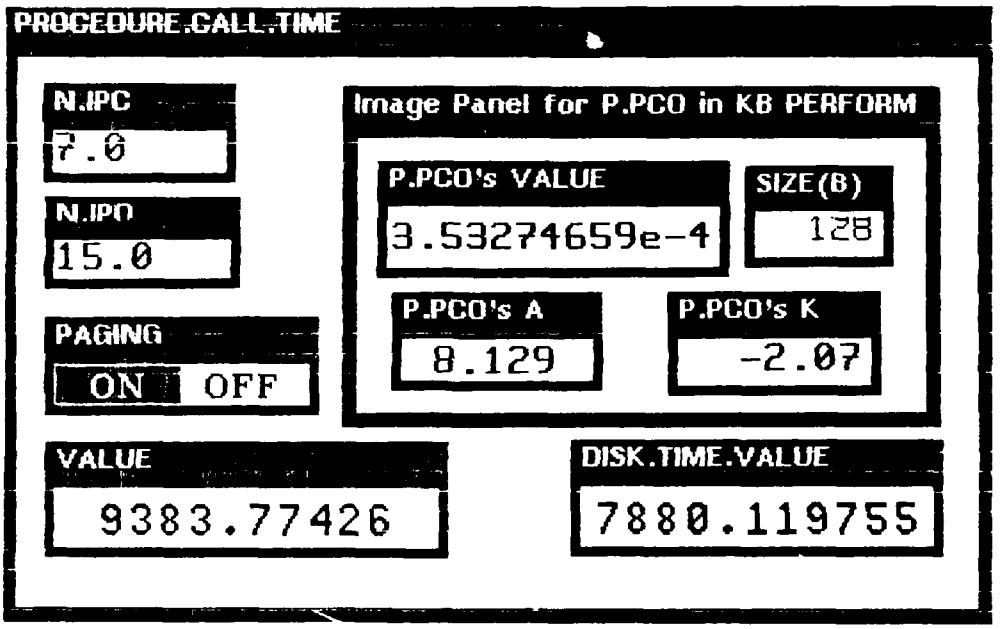


Figure 6-4 Image panel for Procedure Calls.

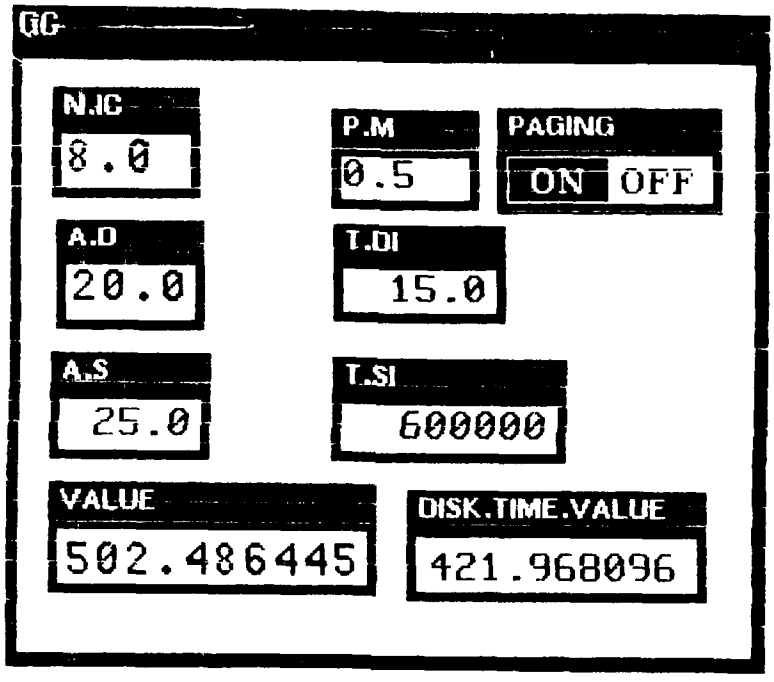


Figure 6-5 Image panel for Garbage Collection.

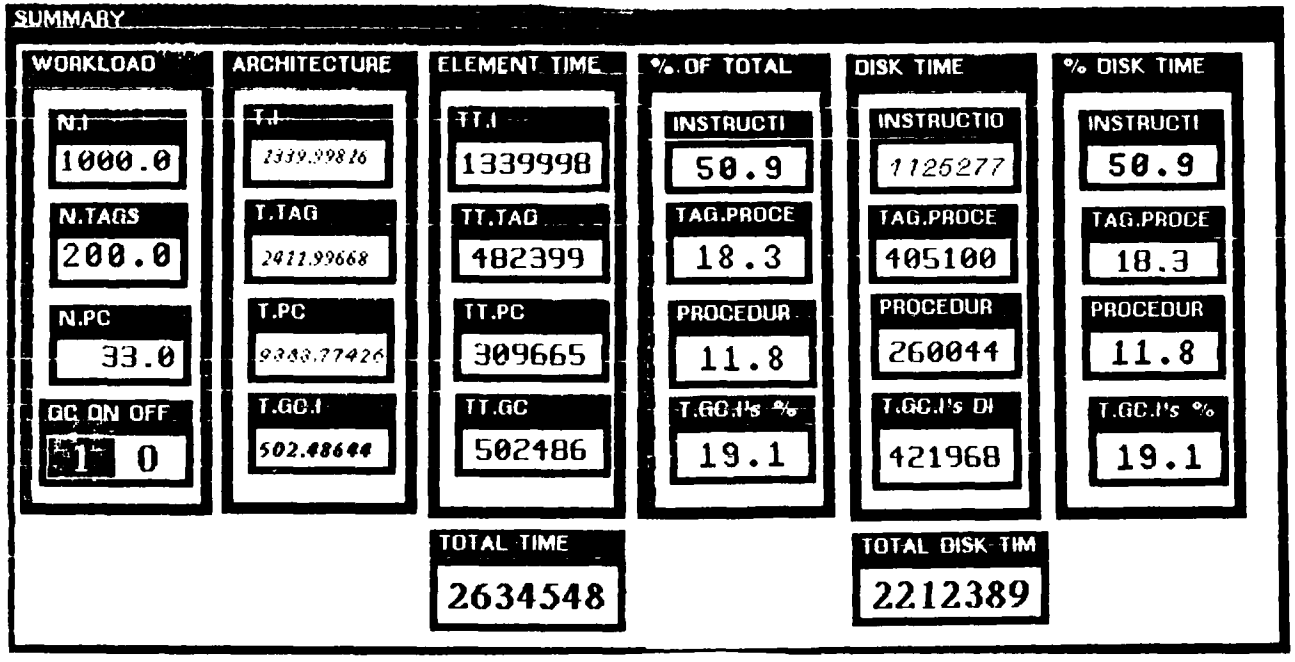


Figure 6-6 Summmary Image Panel.

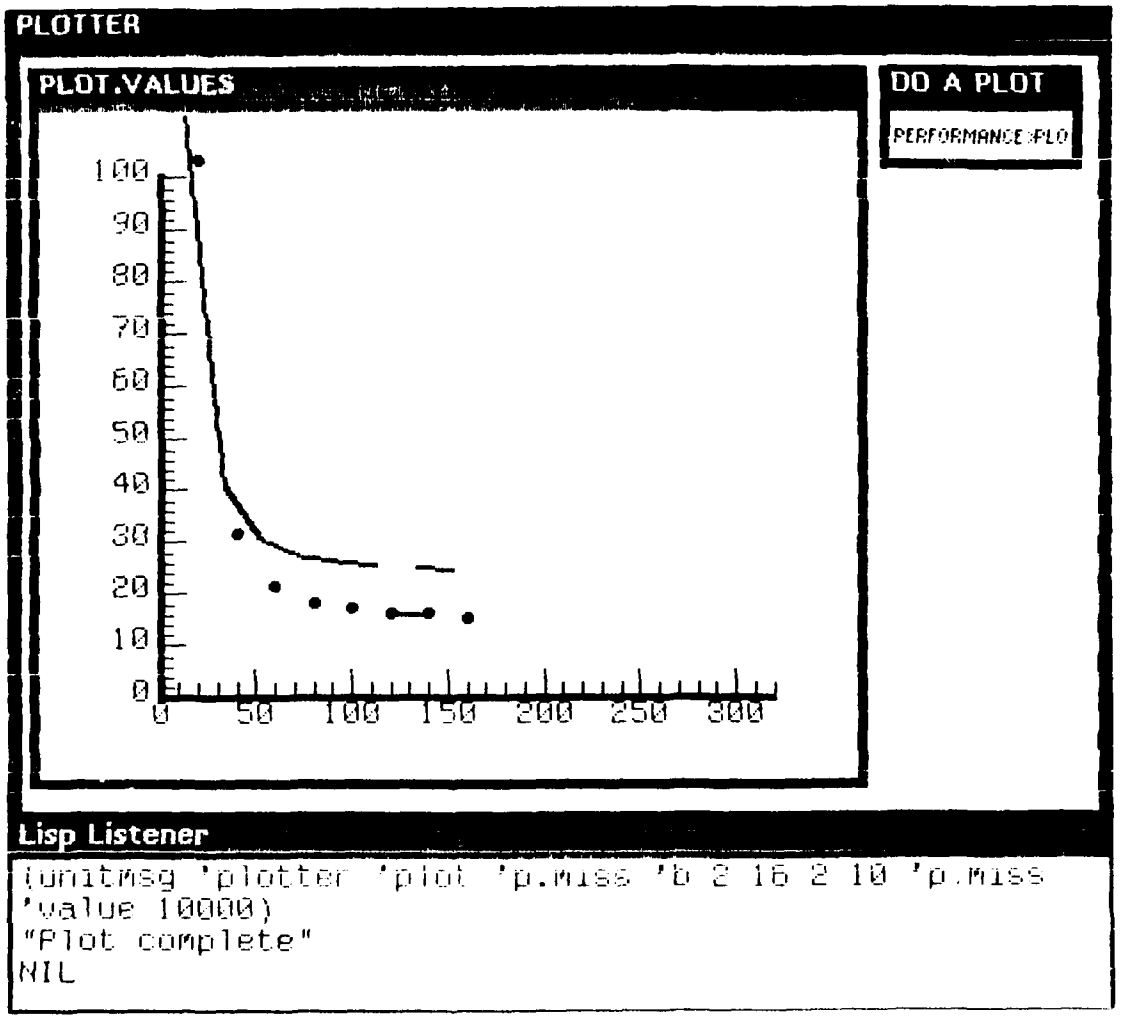


Figure 6-7 PLOTTER panel for normal-normal plot.

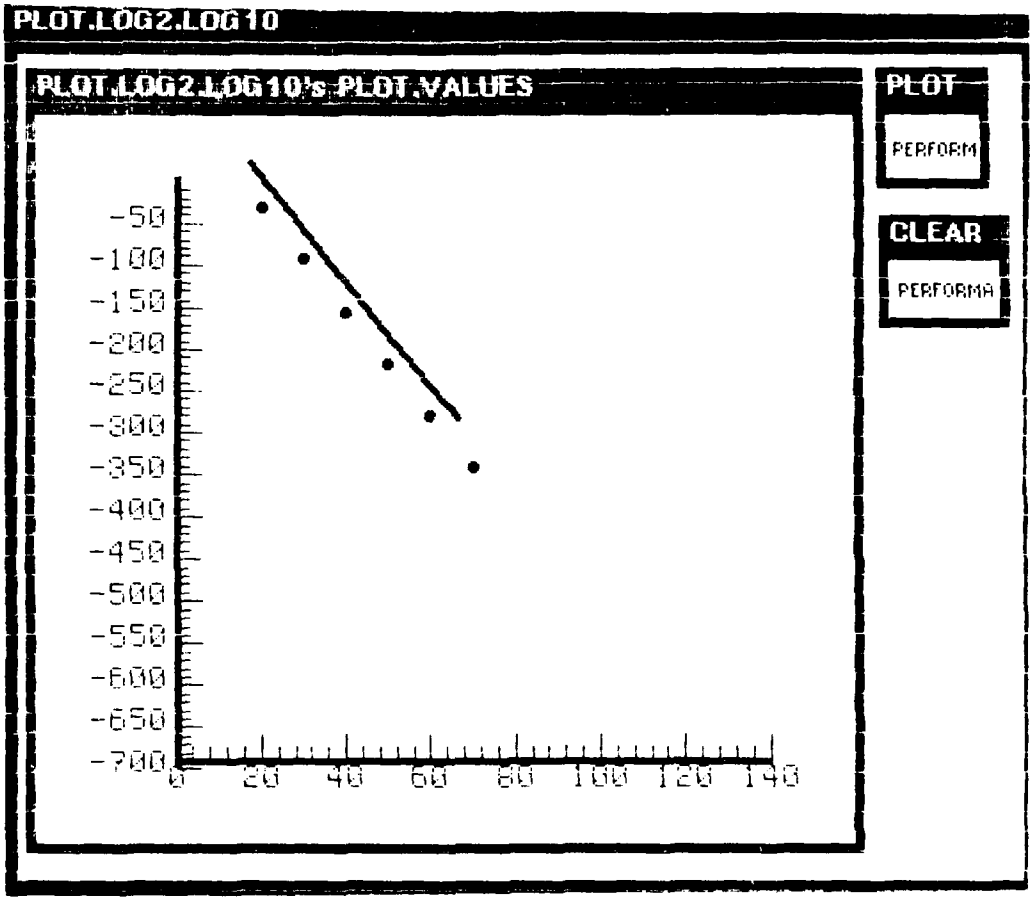


Figure 6-8 PLOT.LOG2.LOG10 panel for log-log plot.

of variables which are exponentially related such as memory size versus page fault probability. To create a plot, the user simply clicks the mouse on an actuator button attached to the panel, and he is then prompted for all required information: x and y variables, initial and final x values, x increment, and scaling factors for the x and y axis.

All the images and windows associated with this knowledgebase can't fit on the screen at the same time. Each image panel fills around 25% of the screen. In addition to the eight image panels, there are five other windows or images which the user will need at various times. As a result, only three image panels will generally be visible at any one time. The other panels can be reduced to small icons and expanded when they are needed. Figure 6-9 shows a representative screen configuration.

6.3 An Expert System?

Even though the Symbolic Processor Model was implemented on a LISP machine using a state-of-the-art expert system building tool, it does not fit the classic definitions of an expert system. This does not reduce the value of the model, it is simply an evaluation of

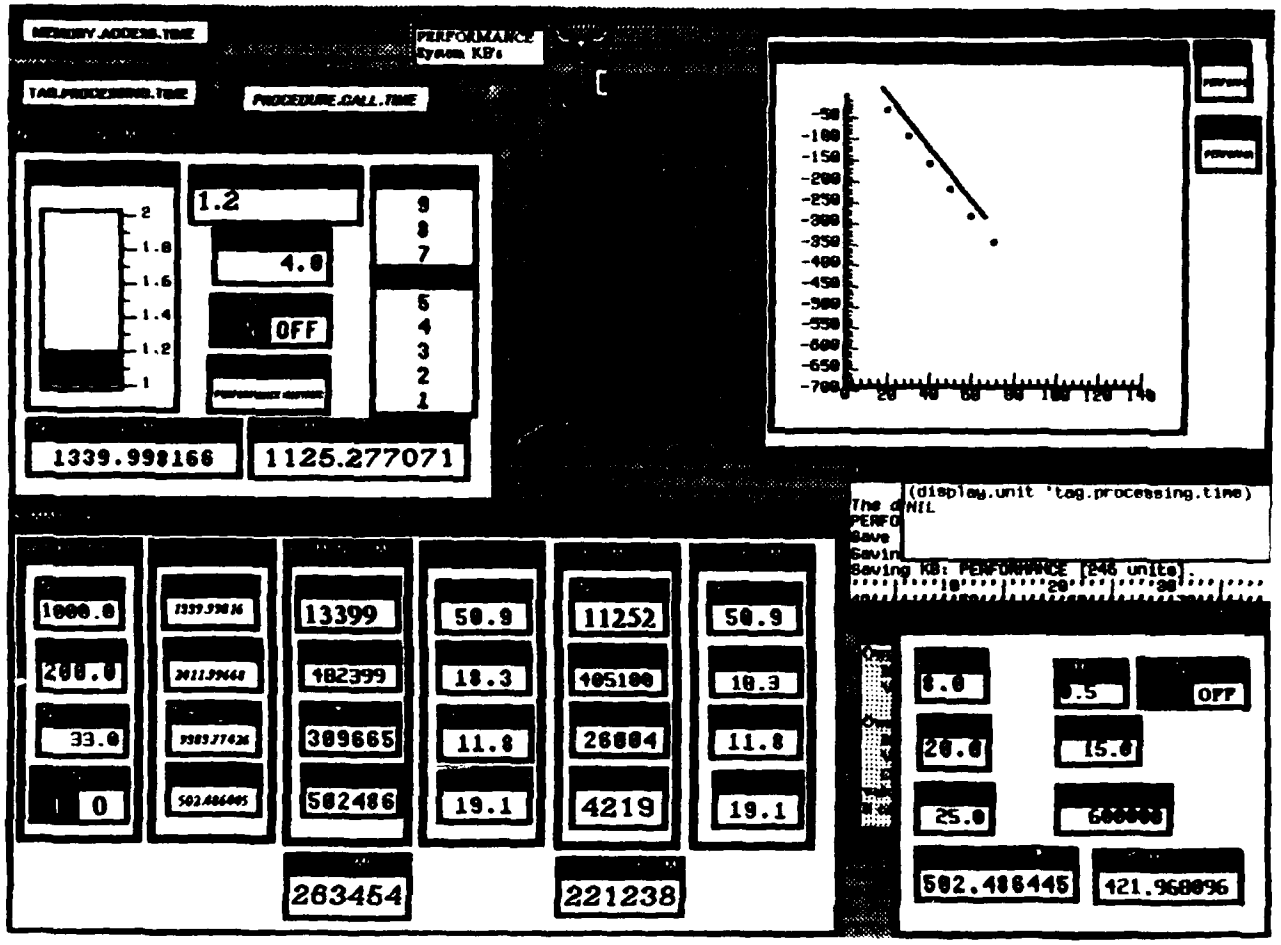


Figure 6-9 Representative screen configuration.

the value of the model; it is simply an evaluation of the object-oriented programming techniques used to build the model. Indeed, many classes of problems can be represented more efficiently with algorithmic solutions.

Three generally accepted features of expert systems include:

- lack of tractable algorithmic solutions for the task
- heuristically guided search through a large search space
- separation of knowledge and control [Hayes-Roth]

The Symbolic Processor Model has algorithmic solutions, does not need to search, and the knowledge and control are not completely separate.

The fact that a particular model or program does not fit the definition of an expert system in no way reduces its value. Actually, it is probably superior to an "expert system" because it indicates that the problem is better understood. Generally, the more a problem is understood, the less need there is for expert system programming techniques. Expert systems can be viewed as systems that find solutions to problems that are not well understood. Recent evidence also seems to indicate that as expert systems mature, they become more

deterministic. XCON/R1 has exemplified this trend toward deterministic solutions as it has matured.

Expert system techniques are not necessary for many problems. It is possible to represent "algorithmic" type problems using expert system techniques, albeit rather inefficiently. For example, there is no sense in searching all the rules for the next rule to fire if you always know what the next rule will be. Expert systems are useful for problems that have multiple, unpredictable paths through a large space. But they are not necessary for problems with algorithmic solutions.

Hayes-Roth, et al., point out that "... the most significant byproduct of expert systems work will be the codification of knowledge." These computer performance models clearly codify a great deal of knowledge on the relationships between variables in computer design.

6.4 KEE versus Lotus 1-2-3

An important factor to consider in evaluating any models is its cost effectiveness. Two of the costs considered in this section are development cost, and the cost of the hardware and software necessary to run the

model. Since the Symbolic Processor Model was originally implemented in Lotus 1-2-3 (Lotus), this section compares Lotus with KEE.

Although the Symbolic Processor model is not inherently a spreadsheet problem, Lotus was used because it automatically propagates changes throughout the model when a variable is changed.

The primary advantages of KEE over Lotus 1-2-3 for the Symbolic Processor Model are its graphic capabilities and expansion capabilities. The image panel for INSTRUCTION.TIME (Figure 6-2) has two examples of these capabilities. The bar graph actuator for references per instruction allows the user to quickly and intuitively see the current value in relation to the allowable range. The vertical traffic light for Pipe Stages accomplishes the same purpose, but in discrete steps.

KEE has a number of disadvantages with respect to Lotus. Lotus was specifically designed for working with numbers and equations. In Lotus, it is easy to specify the precision (numbers to the right of the decimal point) of the displayed values. Updating is done automatically when a variable changes. Two particular problems in KEE are the precision and

displaying of numbers, and the updating mechanism. To change the precision of a number in KEE, the formula for each value must be multiplied by 10^x, rounded, and divided by 10.0^x, where x is the precision desired. KEE would often not display the entire number in the window; i.e. 108543 instead of 1085430. When displaying many numbers with wide ranges of value, images lack precision and take up a lot of space on the screen. For the updating mechanism, I had to manually create the links between all related variables. It was quite difficult to set up the updating mechanism so that the new value of each variables was used. Another practical problem is price. The price of a PC-XT plus Lotus varies between $3,000-5,000. The single unit price of an Explorer plus KEE is around $120,000!

6.5 KEE - A Reasonable Choice

Even though Lotus is better than KEE at displaying numbers and updating them, KEE was a reasonable choice for the overall project. KEE was very useful for building the taxonomy structure. One of the reasons for using the Explorer and KEE was to become familiar with their capabilities, and this was accomplished. One lesson that can be learned, though,

is that inexpensive hardware and software can perform certain tasks more effectively than expensive equipment. The selection of hardware and software should be based on the task requirements.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 A Generalized Computer Performance Model

This chapter will examine the changes and extensions that would be useful in a commercial product to be used by high level computer architects. As with most new products, the dividing line between minimum required functionality and "nice to have" features is a moving target. As products mature, new features often become minimum requirements for future tools. The current implementation will be compared with an ideal generalized computer performance model.

As discussed in Section 2.2, the ultimate goal of this line of research is a generalized computer performance model. In other words, it can model the performance of any proposed architecture. Such a modeling tool could be viewed as consisting of three components:

- a knowledge base of performance models
- a mechanism for selecting and combining models
- Input and output windows for displaying the results

7.2 A Knowledge Base of Performance Models

The first component of this generalized performance model is a knowledge base of performance models. Ideally, all known performance models would be included in the knowledge base. Realistically, some large subset of the more useful performance models would be included initially. The tool should allow new models to be easily added to the knowledge base by the user. Existing models should be easy to modify or delete. This is important because different computer designs will require significantly different models. The architect who is designing a new computer will most likely be the person to develop the new model. He may be the first person to recognize the need for a particular model.

My work satisfied these aspects very well. The models were stored in KEE units, or frames. The organization of the models into the tahonomy was very intuitive and made it easy to find a particular model. Models were easy to add, delete, and modify. Models can be added and deleted by clicking on the taxonomy, and modified by editing the LISP code, or methods, in the slots.

The taxonomy could be improved by providing the whole-part relations discussed in Section 4.3. Both the PARTS slot and the MODELS slot should be displayed simultaneously. As also discussed in Section 4.3, reverse inheritance, or the ability to pass characteristics up the tree could be useful.

7.3 Selecting and Linking Models

The second component of the generalized performance model is a mechanism for selecting and linking models. The selection could be implemented as a series of menus from which the designer selects alternative models. For example, he might select between a *symbolic, pipeline, or non-pipeline* processor model. The mechanism would then link the selected models together.

Two capabilities would need to be developed for this mechanism: an automatic updating mechanism, and standard interfaces among alternative models. The updating mechanism was discussed briefly in Section 6.4. If a variable changes, all the functions that use that variable need to be recomputed. Lotus 1-2-3 takes care of updating automatically, whereas I had to manually create code to link every variable to every function

that used that variable. Another consideration for selecting appropriate models is an expert system. The user might enter information on the expected workload, and the expert system would recommend the appropriate models. In essence, the expert system would design the architecture.

7.4 Input and Output Windows

The third component of the generalized performance model consists of input and output windows. This is how the user interacts with the system. The input windows allow the designer to input characteristics of a proposed architecture. The output windows display the performance for the given input.

Due to the fact that the designer is primarily interested in comparing numbers, a spreadsheet format with rows and columns is simple and economical of screen space. However it is not particularly intuitive. Intuitively shaped icons could be used for some of the less abstract models such as disc drives and pipeline processors. Since all windows/models will not fit on the screen at the same time, it should be possible to open and close windows and access other models quickly.

My implementation had nice looking, readable windows. And other models could be accessed with just a few keystrokes. But it took a long time for the computer to actually close the window and get a new one displayed on the screen. Windows may be faster on other LISP machines, such as Interlisp environments.

It would be useful to look at the performance of two different architectures at the same time. For example, have two Summary Image Panels (Figure 6-6) side-by-side. Each panel would show the performance of a different architecture. In my implementation, only one architecture could be modeled at a time.

A final aspect of the input and output windows is that the designer should be able to save the current state of his model in a separate file. This would allow him to load different architectures into the performance model. In the current implementation, all the performance models must be saved with each architecture. This issue is discussed further in the next section.

7.5 Summary

The current state of the art is somewhere between Lotis 1-2-3 and the newer object-oriented simulators like SIMKIT from Intellicorp. The extensions

mentioned in this chapter would provide a powerful design tool for computer architects.

Future work should concentrate on fast windows, files to save the state of a model, and automatic updating and linking.

APPENDIX

Performance Models

Appendix

Summary of Performance Models

* Note: The original source may not be cited in the reference.

PART NAME: Computer
MODEL NAME: Hellerman's Model
REFERENCE: Baer p. 418-9, 426
DESCRIPTION: For computer organizations in which input and/or output run in parallel with the compute phase, a task can be either CPU bound or I/O bound. The total time to complete the task will be the greater of compute time or I/O time.

MODEL: Total time = Max [Compute Time, I/O Time]

VARIABLES:

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Computer
MODEL NAME: Levels of performance modelling
REFERENCE: Bucci p. 23
DESCRIPTION: Four levels of detail (he gives only three levels-skips design)
*1) design - lowest level of detail
2) selection (purchase) of a computer - cost/benefits
3) capacity planning - estimate of H/W and S/W required
4) fine tuning - optimize system performance
MODEL:
* I am only dealing with models at level one.

VARIABLES:

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Computer
MODEL NAME: Grosch's Law
REFERENCE: Knight 66,68
DESCRIPTION: In the late 1940's, Herb Grosch proposed that computing power increases proportional to the cost squared. This implies economies of scale.

MODEL: Computing Power = $(\text{Cost})^2$

VARIABLES:

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Computer
MODEL NAME: Knight's Model
REFERENCE: Knight 66, 68
DESCRIPTION: A general measure of performance used to compare the computing power of different computers. Grosch's Law is confirmed over the general range of commercial systems. However, gains become more expensive at the limits of the performance envelope. "As the bounds of technological knowledge are reached, additional computing power is purchased at a very high price."
MODEL:

$$\text{Computing power} = \frac{K \times \text{Memory size}}{\text{Compute time} + \text{I/O time}}$$

VARIABLES:
K: Constant:= 10^{12}

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Computer
MODEL NAME: Computer Intraconnection Networks - Busses
REFERENCE: JoDale Carothers
DESCRIPTION:
Design aid for bussing structures between modules.

MODEL: see thesis

VARIABLES:

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Computer
MODEL NAME: Bandwidth relationships
REFERENCE: Hwang p16
DESCRIPTION: Main memory has the highest bandwidth, since it must be updated by both the CPU and the I/O devices.

MODEL: $B_m \geq B_m^u \geq B_p \geq B_p^u > B_d$

VARIABLES:

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Computer
MODEL NAME: System bandwidth balance
REFERENCE: Hwang p16
DESCRIPTION: In the ideal case, memory bandwidth matches the bandwidth sum of the processor and I/O devices. The memory must be updated by both the processor and I/O devices.

MODEL: $B_m^u = B_p^u + B_d$

VARIABLES:

RELATIONSHIPS: SUPERCLASS:
SUBCLASS: B_p^u: utilized processor bandwidth
B_d: I/O bandwidth

PART NAME: Memory
MODEL NAME: Memory Hierarchy
REFERENCE: Welch 78, 79
DESCRIPTION: Minimizes memory system delay by trading off the speeds (and costs) between the different levels of the memory hierarchy. The percentage of system cost (unit cost/system cost = B_i/S) invested in level i should equal the percentage of system delay (weighted unit delay/total average delay = $t_i \times p_i/Tavg$) caused by level i.

MODEL: $$\frac{t_i \times P_i}{Tavg} = \frac{B_i}{S}$$

VARIABLES: Tavg: memory system delay,
P_i: probability of accessing level i,
C_i: cost of level i,
B_i: unit cost,
S: system cost,
t_i: access time for level i
RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Memory
MODEL NAME: Performance limit
REFERENCE: Welch 78, 79
DESCRIPTION: The maximum possible performance (minimum access time) achievable by adding an additional level of memory. Used to check if the performance improvement from an additional level of memory is worth the cost and complexity.

MODEL: $\min (Tavg) = (p_1^{1/3} c_1)^3$

VARIABLES: Tavg: average memory system delay
p1: probability of accessing level 1
c1: cost of level 1

RELATIONSHIPS: SUPERCLASS: Memory
SUBCLASS:

PART NAME: Memory hierarchy
MODEL NAME: Memory cost (B)
REFERENCE: Welch 78, 79
DESCRIPTION: Cost (B) of memory is inversely proportional to the access time (t). Although in reality the cost is a step function, it is modeled by a continuous curve. For fine-tuning, the steps must be considered.

MODEL: $B = a * t^{-k}$

VARIABLES: a: constant
k: constant
t: memory access time

RELATIONSHIPS: SUPERCLASS: Memory hierarchy
SUBCLASS:

PART NAME: Memory
MODEL NAME: Buffers
REFERENCE: Hwang p. 194
DESCRIPTION: Supplies a continuous stream of instructions (operands?), despite memory-access conflicts.

MODEL:

VARIABLES:

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Primary memory
MODEL NAME: Maximum memory bandwidth, B_m
REFERENCE: Hwang p. 14, p. 156
DESCRIPTION: Memory bandwidth is the average numer of words accessed per second. Ideally, it equals the processor demand rate.

MODEL:

$$Bm = \frac{W}{tm}$$

VARIABLES: W: words delivered (function of interleaving)
tm: memory cycle time

RELATIONSHIPS: SUPERCLASS: Processor memory demand rate
SUBCLASS: Memory configuration,
Utilized memory bandwidth

PART NAME: Primary memory
MODEL NAME: Utilized memory bandwidth ($B_m{}^u$)
REFERENCE: Hwang p. 14
DESCRIPTION: In practice, the usable memory bandwidth is less than the maximum (ideal) memory bandwidth. This is a result of memory access conflicts. As the number of interleaved modules increases, memory access conflicts also increases.

MODEL: $B_m{}^u = \frac{B_m}{M \cdot 5}$

VARIABLES: M: number of interleaved memory modules
B_m: Maximum memory bandwidth

RELATIONSHIPS: SUPERCLASS: Maximum memory bandwidth
SUBCLASS:

PART NAME: Primary Memory
MODEL NAME: Average memory bandwidth with access-conflict
REFERENCE: Hwang p. 163
DESCRIPTION: Approximation of average bandwidth considering access-conflict. Single processor, $1 \leq m \leq 45$.

MODEL: $B(1,m) = m^{0.56}$

VARIABLES:

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Primary memory
MODEL NAME: Memory configuration
REFERENCE: Hwang p. 156
DESCRIPTION: The primary factors affecting the bandwidth are the processor architecture, the memory configuration, and the memory module characteristics. The memory configuration is characterized by the number of memory modules and their addressing structure and bus width.

MODEL:

VARIABLES: number of memory modules,
addressing structure,
bus width

RELATIONSHIPS: SUPERCLASS: Memory bandwidth
SUBCLASS: Number of memory modules,
Module characteristics

PART NAME: Primary memory
MODEL NAME: Number of memory modules (N)
REFERENCE: Hwang p. 158
DESCRIPTION: The memory normally needs a number of processor clock periods to recover after being accessed. So alternate modules are necessary while the others are recovering.

MODEL:

$$N = \frac{\text{memory cycle time}}{\text{clock period}}$$

VARIABLES:

RELATIONSHIPS: SUPERCLASS: Memory configuration
SUBCLASS:

PART NAME: Primary memory
MODEL NAME: Memory module characteristics
REFERENCE: Hwang p. 156
DESCRIPTION: Module characteristics include size, access time, and cycle time (time between accesses).

MODEL:

VARIABLES: size,
access time,
cycle time

RELATIONSHIPS: SUPERCLASS: Memory configuration
SUBCLASS:

PART NAME: Primary memory
MODEL NAME: Memory bandwidth estimation
REFERENCE: Hwang p. 163-4, Rim
DESCRIPTION:

See Rim's thesis.

MODEL:

VARIABLES:

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Primary memory
MODEL NAME: Memory organization
REFERENCE: Hwang p. 158-162, Rim
DESCRIPTION:

See Rim's thesis.

MODEL:

VARIABLES:

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Primary memory
MODEL NAME: Optimum page size
REFERENCE: Hwang p. 80.
DESCRIPTION:

MODEL: $Z = (2CS)^{.5}$

VARIABLES: C: constant (related to page table size)
S: average segment size

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Primary memory
MODEL NAME: Replacement algorithm
REFERENCE: Hwang p. 91.
DESCRIPTION: List of alternative replacement algorithms for choosing which areas of memory are to be written over with new information.

MODEL: LRU, MIN, LFU, FIFO, FINUFO, LIFO, RANDOM

VARIABLES:

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Primary memory
MODEL NAME: Phase transition methods
REFERENCE: Hwang pp. 96-98
DESCRIPTION: Methods of removing the pages of an old program and prefetching the pages of a new program.

MODEL: swapping, one block lookahead (OBL)

VARIABLES:

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Primary memory
MODEL NAME: Paging curve
REFERENCE: Stroebel
DESCRIPTION: Number of disk I/Os per transaction

MODEL: $n = (n_0 + n_1 r^k) / (1 + n_{inf} r^k)$

VARIABLES: n_0: the number of disk I/Os at OCR = 0
n_1: the number of disk I/Os at OCR = 1
n_{inf}: the number of disk I/Os at OCR = infinity
k: application dependent
r: memory overcommitment ratio (OCR)

RELATIONSHIPS: SUPERCLASS:
SUBCLASS: OCR

PART NAME: Primary memory
MODEL NAME: OCR - Overcommitment ratio
REFERENCE: Stroebel
DESCRIPTION: The OCR is the sum of the active tasks' working set sizes divided by the amount of real memory available for their execution

MODEL:

VARIABLES:

RELATIONSHIPS: SUPERCLASS: Paging curve
SUBCLASS:

PART NAME: Primary memory
MODEL NAME: Primary memory update from cache
REFERENCE: Hwang p. 113-115
DESCRIPTION: Different methods of updating the primary memory from the cache

MODEL: WT, WTWA, WTNWA, SWB, FWB, FRWB

VARIABLES:

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Secondary memory
MODEL NAME: Disk bandwidth, B_d
REFERENCE: Hwang p. 14
DESCRIPTION: Disk bandwidth is inversely proportional to the cycle time.

MODEL: $B_d = \frac{1}{T_d}$

VARIABLES: t_d: disk cycle time

RELATIONSHIPS: SUPERCLASS:
SUBCLASS: Disk cycle time

PART NAME: Secondary Memory
MODEL NAME: Disk Cycle Time
REFERENCE: Baer p. 249
DESCRIPTION: Time to get data from a disk or drum

MODEL: T_d = latency + transfer rate

VARIABLES:

RELATIONSHIPS:SUPERCLASS:
SUBCLASS: Latency, Transfer rate

PART NAME: Secondary memory
MODEL NAME: Latency
REFERENCE: Baer p. 253
DESCRIPTION: Delays in accessing information on disks or drums.

MODEL: Latency = Rotational + Headseek
= ? MAX[Rotation + Headseek] ?

VARIABLES:

RELATIONSHIPS:SUPERCLASS: Disk Cycle Time
SUBCLASS: Rotational, Headseek

PART NAME: Secondary memory
MODEL NAME: Rotational
REFERENCE: Baer p. 263
DESCRIPTION: On the average, there is a delay of half a revolution to find the needed information on a disk or drum.

MODEL:

$$t_a = \frac{1}{(2)(\text{rotational speed})}$$

VARIABLES:

RELATIONSHIPS: SUPERCLASS: Latency
SUBCLASS:

PART NAME: Secondary memory
MODEL NAME: Headseek
REFERENCE: Baer p. 254
DESCRIPTION: Time to move disk read/write head to the correct cylinder. For fixed head disks, this time will be zero.

MODEL: Varies with the disk unit. Not quite linear.

VARIABLES:

RELATIONSHIPS: SUPERCLASS: Latency
SUBCLASS:

PART NAME: Secondary memory
MODEL NAME: Disk transfer time
REFERENCE: Baer p. 253.
DESCRIPTION: Time to transfer one sector

MODEL: $$t_t = \frac{1}{r\ s}$$

VARIABLES: r: rotational speed
s: sectors per track

RELATIONSHIPS: SUPERCLASS: Disk cycle time
SUBCLASS:

PART NAME: Secondary memory and IO
MODEL NAME: Disk delay
REFERENCE: Cho - PhD research
DESCRIPTION: Total disk delay is composed of wait time (waiting ina queue) plus the actual service time.

MODEL: $T_d = T_{sv} + T_{wt}$

VARIABLES: T_{sv}: service time
T_{wt}: wait time

RELATIONSHIPS: SUPERCLASS:
SUBCLASS: T_{sv}, T_{wt}

PART NAME: Secondary memory and IO
MODEL NAME: Disk delay due to waiting (T_{wt})
REFERENCE: Cho
DESCRIPTION: This delay is caused by various jobs competing for the use of the disk. T_{wt} is best modeled by queueing theory.

MODEL:

VARIABLES:

RELATIONSHIPS: SUPERCLASS: Disk delay
SUBCLASS:

PART NAME: Secondary memory and IO
MODEL NAME: Disk service time (T_{sv})
REFERENCE: Cho
DESCRIPTION: Time to find and transfer the information.

MODEL: $T_{sv} = T_{sk} + T_{sc} + T_{dt}$

VARIABLES: T_{sk} = seek time (headseek)
T_{sc} = search time (rotational)
T_{dt} = data transfer

RELATIONSHIPS: SUPERCLASS: Disk delay
SUBCLASS:

PART NAME: Cache
MODEL NAME: Cache access time
REFERENCE: Hwang p. 16
DESCRIPTION: The speed gap between the CPU and the main memory can be closed up by using a fast cache between them.

MODEL: $t_c = t_p$

VARIABLES: t_c = cache access time
t_p = processor cycle time

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Cache
MODEL NAME: Cache block replacement algorithms
REFERENCE: Hwang p. 115
DESCRIPTION: Replacement algorithms for choosing which areas of the cache are to be written over with new information.

MODEL: LRU, FIFO, RANDOM

VARIABLES:

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Registers
MODEL NAME: Context switch
REFERENCE: Cragon 86C, 22 April
DESCRIPTION: Computes the time required to switch the context, for example during a subroutine call or interrupt. Permits a comarison of different types of register sets, i.e. single register set (SRS), multiple register (MRS), and overlapped registers set (ORS).

MODEL: Total time = Basic.Instruction.Time + P.PR * T.PR

VARIABLES: T.PR: parameter reference time
P.PR: probability of parameter reference:=03

RELATIONSHIPS:SUPERCLASS:
SUBCLASS: T.PR, Basic Instruction Time

PART NAME: Registers
MODEL NAME: Cragon 86C, 22 April
REFERENCE: Basic instruction time
DESCRIPTION: Average time to execute an instruction. Does not include time to look up parameters (or operands) (see parameter reference time). IF (there is a procedure call) and (the register sets overflow) then (the instruction will take N.CPO cycles). Otherwise, the instruction only takes one cycle.

MODEL: Time = (1-P.PC)(1)+ P.PC(P.OVF*N.CPO+(1-P.OVF)*1)

VARIABLES:
P.PC: probability of a procedure call:=0.1
N.CPO: number of cycles per overflow (to store in memory)
(1-P.PC): probability of a single cycle instruction
P.OVF: probability of overflow

RELATIONSHIPS:SUPERCLASS: Context switch
SUBCLASS: P.OVF

PART NAME: Registers
MODEL NAME: Cragon 86C, 22 April
REFERENCE: Parameter reference time (T.PR)
DESCRIPTION: Average time required to retrieve operands.

MODEL: T.PR = P.MEM*T.MEM + P.REG*T.REG + P.OVL*T.OVL

VARIABLES:
P.XXX: probability of operand being in this memory
T.XXX: access time to this memory
MEM: main memory
REG: a different register set
OVL: operand in present (overlapped) register set
RELATIONSHIPS:SUPERCLASS: Context switch
SUBCLASS:

PART NAME: Registers
MODEL NAME: Probability of overflow (P.OVF)
REFERENCE: Cragon 86C 22 April
DESCRIPTION: As the number of register sets to be saved (D) increases, the probability of overflowing (P.OVF) the number of available register sets (R) increases.

MODEL: P.OVF = $0.4\ (R/2D)^{-1.3}$ for $(R/2D) \geq$
= 1 for $(R/2D) <$

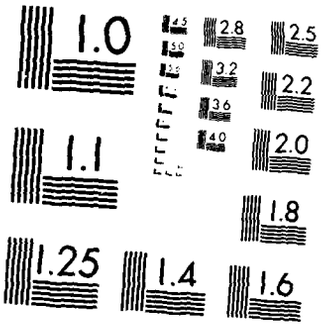
VARIABLES: R: number of register sets
D: number of register sets
(average depth

RELATIONSHIPS:SUPERCLASS:
SUBCLASS:

AD-A185 561 A TAXONOMY OF ANALYTICAL COMPUTER PERFORMANCE MODELS FOR COMPUTER DESIGN(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH M W STREVELL 1986 2/2

UNCLASSIFIED AFIT/CI/NR-87-97I F/G 12/8 NL

END
DATE FILMED
12 87



PART NAME: Processor
MODEL NAME: Cragon 79
REFERENCE: Cragon's Model
DESCRIPTION: Comparison of five architectures in terms of: 1) the number of bytes of code for an operation 2) number of available op codes and 3) performance in number of clocks per operation. The five architectures compared are stack, accumulator, register file, and two- and three-address memory-to-memory.

MODEL: Using the stack architecture as an example, the format for the models is: 1) $B_s = 2.2+1.2N$ 2)OpCodes= 252 3) $C_s = 17.4/W + 3.6J + 3.2$

VARIABLES: B: number of bytes of code,
N: size of address field
C: clocks
J: operand size/memory size

RELATIONSHIPS:SUPERCLASS:
SUBCLASS:

PART NAME: Processor
MODEL NAME: Processor bandwidth, B_p
REFERENCE: Hwang, p. 156.
DESCRIPTION: Words required from memory per second

Example: $\frac{\text{(operands+instructions+results)(\#pipelines)}}{\text{clock period}}$

MODEL: $B_p = \frac{\text{words required}}{\text{second}}$

VARIABLES:

RELATIONSHIPS:SUPERCLASS: Memory bandwidth
SUBCLASS:

PART NAME: Processor
MODEL NAME: Utilized processor bandwidth, B_p^u
REFERENCE: Hwang p. 14
DESCRIPTION: The actual bandwidth is simply the number of output results (in words) per second. (weak model)

MODEL:
$$B_p^u = \frac{R_w}{T_p}$$

VARIABLES: R_w: number of word results
T_p: the CPU time required to generate the R_w results

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Processor
MODEL NAME: Processor busses
REFERENCE: Hwang, p. 196
DESCRIPTION: Multiple data paths eliminate the time delay to store and retrieve intermediate results to and from the registers.

MODEL:

VARIABLES:

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Processor
MODEL NAME: Amdahl's Model
REFERENCE: Proceedings, Spring Joint Computer Conference, 1967, pp. 483-5
DESCRIPTION: Speedup factor achievable by running a given job on a parallel or vector processor rather than a scalar processor. The ideal speedup is k. The actual speedup is limited by the fact that some fraction of the code is sequential in nature and can't be vectorized or parallelized.

MODEL:

$$\text{speedup} = \frac{1}{(1-f)+f/k}$$

VARIABLES: f: fraction of code that can be vectorized
k: ratio of vector to scalar performance

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Symbolic Processor
MODEL NAME: Symbolic Processor Model
REFERENCE: Cragon 86B
DESCRIPTION: Total time is the summation of the time require to perform each function, times the number of those functions.

MODEL: $T = AV_j * WV_j$

VARIABLES: AV: architecture vector
WV: workload vector

RELATIONSHIPS: SUPERCLASS:
SUBCLASS: Architecture vector,
Workload vector

PART NAME: Symbolic processor
MODEL NAME: Workload vector
REFERENCE: Cragon 86B
DESCRIPTION: The workload can be modeled as a vector in which each element represents the number of those functions to be performed in a given session.

MODEL: $WV = (N_i, N_t, N_c, N_g, MEM)$

VARIABLES:
N_i: number of instructions executed in a session
N_t: number of tags processed in a session
N_c: number of procedure calls in a session
N_g: number of garbage collections in a sesion
MEM: memory system

RELATIONSHIPS: SUPERCLASS: Symbolic processor model
SUBCLASS: P.MISS, C.MISS

PART NAME: Symbolic processor
MODEL NAME: P.MISS
REFERENCE: Cragon 86
DESCRIPTION: Probability of a page fault in main memory; i.e. the value needed by the processor is not in main memory, hence must be retrieved from secondary memory. For PAGING=OFF, set P.MISS=0. This assumes all references are in memory.

MODEL: $P.MISS = a * B^k + t$

VARIABLES: a: page fault intercept:=0.05
k: page fault slope:=-2.5
B: buffer size in k bytes
t: transient page faults/working set:=0.0015

RELATIONSHIPS: SUPERCLASS: Memory system
SUBCLASS:

PART NAME: Symbolic processor
MODEL NAME: C.MISS (Cache miss ratio)
REFERENCE: Cragon 86B
DESCRIPTION: Probability of a cache fault; i.e. if the value needed by the processor is not in cache and must be retrieved from primary memory.

MODEL: $C.MISS = A.C. * B^{K.C}$

VARIABLES:
A.C. = cache fault intercept
B = cache (buffer) size
K.C. = cache fault slope

RELATIONSHIPS: SUPERCLASS: symbolic processor model
SUBCLASS:

PART NAME: Symbolic processor
MODEL NAME: Architecture vector (AV)
REFERENCE: Cragon 86
DESCRIPTION: The architecture can be moduled as a vector in which each element represents the time to perform a specified function.

MODEL: AV = (T.I, T.T, T.PC, T.G, T.AC)

VARIABLES: component times:
T.I: instruction
T.T: tag processing
T.PC: procedure call
T.G: garbage collection
T.AC: memory access

RELATIONSHIPS: SUPERCLASS: Symbolic processor model
SUBCLASS:

PART NAME: Symbolic processor
MODEL NAME: Instruction Time (T.I)
REFERENCE: Cragon 86B
DESCRIPTION: The time required to process one instruction. The second equation is the amount of the time that is due to the disk.

MODEL: $T.I = R.M * T.AC + T.C(P+N-1)/N$
$T.ID = R.M * T.ACD$

VARIABLES:
R.M: number of memory references per instruction
P: number of pipe stages
N: number of instructions between branches
T.AC: memory access time
T.C : cache access time

RELATIONSHIPS: SUPERCLASS: Architecture Vector
SUBCLASS:

PART NAME: Symbolic processor
MODEL NAME: Tag processing time (T.T)
REFERENCE: Cragon 86B
DESCRIPTION: The time to process a tag depends on whether the tags agree, and if the tag processing is done by hardware or software. At most one of the variables (N.TCH, N.TCS) will be nonzero. If overlapped hardware checking, both will be zero. The second equation is the amount of the time that is due to the disk.

MODEL: $T.T. = P.AGR * (N.TCH * T.CLK + N.TCS * T.I) + (1-P.AGR) * N.TRS * T.I$
$T.TD = T.ID * N.TCS + (1-P.AGR) * T.ID * N.TRS$

VARIABLES: P.AGR: Probability of tag agreement
N.TCH: number of clocks to check tags (if in hardware)
N.TCS: number of instructions to check tags (if in software)
N.TRS: number of instructions to resolve a tag fault

RELATIONSHIPS: SUPERCLASS: Architecture Vector
SUBCLASS:

PART NAME: Symbolic Processor
MODEL NAME: Memory access time (memory model)
REFERENCE: Cragon 86B
DESCRIPTION: Average access time for the memory hierarchy: cache, primary memory, disk. The second equation is the amount of the time that is due to the disk.

MODEL: T.AC = T.C. + C.MISS[T.MC+T.DM * P.MISS(1+N.W.)]
T.ACD = C.MISS * P.MISS * T.DM (1 + N.W)

VARIABLES: C.MISS: cache miss ratio
P.MISS: main memory miss ratio
T.C : cache.access.time
T.DM : disk to main memory transport (access) time (SM.ACCESS.TIME)
T.MC : main memory to cache access time
N.W : number of disk writes per read (0 to 1)

RELATIONSHIPS:SUPERCLASS: Architectural vector
SUBCLASS: P.MISS, C.MISS

PART NAME: Symbolic processor
MODEL NAME: Garbage collection time, T.G
REFERENCE: Cragon 86B
DESCRIPTION: Garbage collection time per instruction. The second equation is the amount of the time that is due to the disk.

MODEL: $T.G = P.M * N.IC * (T.I)^2 * 10^{-9} * (2^{A.S}/T.SI + 2^{A.D}/T.DI)$
$T.GD = T.G * (T.ID/T.I)$

VARIABLES: P.M : the marked cell ratio
N.IC: number of instructions to copy a cell
A.S : static address space (bit width)
A.D : dynamic address space (bit width)
T.SI: time interval between static GCs (seconds)
T.DI: time interval between dynamic GCs (seconds)

RELATIONSHIPS:SUPERCLASS: Architecture Vector
SUBCLASS:

PART NAME: Symbolic processor
MODEL NAME: Procedure call time (T.PC)
REFERENCE: Cragon 86B
DESCRIPTION: The time to process a procedure call depends upon whether the number of in-process registers required exceed the number of registers available (overflow). The second equation is the amount of the time that is due to the disk.

MODEL: $T.PC = T.I\ (P.PCO * N.IPO + (1-P.PCO) * N.IPC)$
$T.PCD = T.ID\ (P.PCO * N.IPO + (1-P.PCO) * N.IPC)$

VARIABLES: T.I : instruction time
P.PCO: probability of procedure call overflow
N.IPO: number of instructions for a procedure call with overflow
N.IPC: number of instructions for a procedure without overflow

RELATIONSHIPS: SUPERCLASS: Architectural vector
SUBCLASS: P.PCO

PART NAME: Symbolic processor
MODEL NAME: Probability of procedure call overflow (P.PCO)
REFERENCE: Cragon 86B
DESCRIPTION: Probability that the number of in-processor registers required exceeds the number of registers available (value must be between 0 and 1).

MODEL: $P.PCO = \min [1, A.PC * B^{K.PC}]$

VARIABLES: A.PC: overflow rate intercept
B: number of register sets in the processor
K.PC: slope of the overflow curve

RELATIONSHIPS: SUPERCLASS: Procedure call time
SUBCLASS:

PART NAME: Pipeline Processor
MODEL NAME: Pipeline processing time
REFERENCE: Hawng p. 147
DESCRIPTION: T_k is the time to process n tasks in a k stage pipeline. k cycles are used to fill up the pipeline and complete the first task. (n-1) cycles are needed to complete the remaining (n-1) tasks. (This assumes the vector length equals the number of stages in the pipeline)

MODEL: $T_k = k + (n-1)$

VARIABLES: k: number of stages,
n: number of tasks

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Pipeline Processor
MODEL NAME: Pipeline speedup
REFERENCE: Hwang, p. 148
DESCRIPTION: Speedup of a pipeline processor over a non-pipeline processor.

MODEL:
$$S_k = \frac{T_1}{T_k} = \frac{n \cdot k}{k+(n-1)}$$

maximum $s_k \leq k$

VARIABLES: n: number of tasks
k: number of stages

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Pipeline Processor
MODEL NAME: Pipeline efficiency (EFF)
REFERENCE: Hwang p. 151
DESCRIPTION: The average fraction of pipeline stages in use.

MODEL:
$$EFF = \frac{n}{k + (n-1)} = \frac{S_k}{k}$$

VARIABLES: n: number of tasks
k: number of stages
S_k: speedup

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Pipeline Processor
MODEL NAME: Pipeline clock period (CP) and frequency (f)
REFERENCE: Hwang p. 146
DESCRIPTION: Clock period is the time delay to go through one stage in the pipeline. Frequency is the number of clock periods, or cycles, per second.

MODEL:
$$CP = t_m + t_l$$
$$f = 1/CP$$

VARIABLES: t_m: time for slowest stage
t_l: latch delay

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Pipeline Processor
MODEL NAME: Pipeline throughput rate (w)
REFERENCE: Hwang p.151
DESCRIPTION: Total tasks completed /total time periods. Ideal maximum for very large n is one output per clock = f.

MODEL:
$$w = \frac{n}{CP * k + (n-1) * CP} = \frac{EFF}{CP}$$
maximum $w = 1/CP = f$

VARIABLES: n: number of tasks
k: number of stages
CP: clock period
EFF: pipeline efficiency

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Pipeline Processor
MODEL NAME: Pipeline segment bottleneck
REFERENCE: Hwang pp. 193-4
DESCRIPTION: The maximum throughput of a pipeline is inversely proportional to the bottleneck time.

MODEL: throughput = $\frac{1}{T_{max}}$

VARIABLES: T_{max} = longest pipeline-segment time

RELATIONSHIPS: SUPERCLASS:
SUBCLASS: Pipeline segment bottleneck solution.

PART NAME: Pipeline Processor
MODEL NAME: Pipeline bottleneck solution
REFERENCE: Hwang pp. 193-194
DESCRIPTION: Subdivide the bottleneck If not subdivisible, use duplicates of the bottleneck in parallel (control is more complex than subdivide).

MODEL:

VARIABLES:

RELATIONSHIPS:SUPERCLASS: Pipeline segment bottleneck
SUBCLASS:

PART NAME: Pipeline processor
MODEL NAME: Pipeline reservation tables
REFERENCE: Hwang p. 154.
DESCRIPTION: Like a schedule: displays how the various stages are being used in successive time periods.

MODEL:

	t0	t1	t2	t3 ...
S1				
S2				
.				
.				
.				
SK				

VARIABLES: The rows correspond to pipeline stages (k),
The columns correspond to clock time units

RELATIONSHIPS:SUPERCLASS:
SUBCLASS:

PART NAME: Pipeline Processor
MODEL NAME: Dynamic Pipeline reservation tables - latency
REFERENCE: Hwang p. 204.
DESCRIPTION: Examines time delay between starting two jobs so as to avoid conflicts in which both jobs attempt to use the same stage at the same time.

MODEL:

VARIABLES:

RELATIONSHIPS:SUPERCLASS:
SUBCLASS:

PART NAME: Pipeline Processor
MODEL NAME: Pipeline vector execution time
REFERENCE: Hwang, p. 220
DESCRIPTION: Time required to execute a single vector task. Broken down into overhead time and actual execution, or production, time.

MODEL: (Overhead + Production)

VARIABLES:

RELATIONSHIPS:SUPERCLASS:
SUBCLASS: Overhead time, Production time

PART NAME: Pipeline Processor
MODEL NAME: Overhead time
REFERENCE: Hwang p. 217, p. 220; Strevell
DESCRIPTION: Pipeline overhead time due to startup and flushing delays. I assume the setup time is included in the overhead time. Setup is the time to route the operands and control variables to the appropriate functional units. Hwang may be using the terms setup and startup synonymously. As far as flushing delays, it seems more natural to look at it as filling delays. You don't get any outputs while you're filling a pipeline, but you are getting outputs as you are flushing.

MODEL: (unknown)
Strevell: T.SETUP + (k-1)

VARIABLES: T.SETUP: setup time
k: number of stages
(k-1): fill time

RELATIONSHIPS: SUPERCLASS: Vector execution time
SUBCLASS:

PART NAME: Pipeline Processor
MODEL NAME: Production delay
REFERENCE: Hwang p. 220
DESCRIPTION: The execution, or production, time is simply the time between operands multiplied by the number of operands.

MODEL: T.EXE = T.LAT * N

VARIABLES: T.LAT: average latency between successive operands
N: vector length

RELATIONSHIPS: SUPERCLASS: Vector execution time
SUBCLASS:

PART NAME: Branch control
MODEL NAME: Average instructions
REFERENCE: Hwang p. 190-1
DESCRIPTION: The average number of instructions executed per instruction cycle in a pipeline processor. When p = 0 (no branching instructions), the result is n instructions per n pipeline clocks (one instruction cycle), which is the ideal case. Assumes the number of instructions waiting to be executed is very large. Branching interferes with the prefetching strategy.

MODEL:
$$= \frac{n}{1 + pq(n-1)}$$

VARIABLES: p = probability of a conditional branch
q = probability that a conditional branch is successful
n = number of pipeline clock periods per instruction cycle

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Non-pipeline Processor
MODEL NAME: Non-pipeline processing time
REFERENCE: Hwang p. 147
DESCRIPTION: Time for a non-pipeline processor to process n vectors.

MODEL: $T_1 = n \cdot k$

VARIABLES: n: number of vectors
k: number of elements/vector

RELATIONSHIPS: SUPERCLASS:
SUBCLASS:

PART NAME: Non-pipeline Processor
MODEL NAME: Non-pipeline processing time-A
REFERENCE: Bell
DESCRIPTION: Instruction processing time for a non-pipeline processor.

MODEL: $t = K_1C_1 + K_2C_2$

VARIABLES: K_1: number of microcycles expected in an instruction
C_1: microcycle time
K_2: number of memory accesses in an instruction
C_2: memory read pause (delay) time

RELATIONSHIPS: SUPERCLASS: NON.PIPELINE.PROCESSOR
SUBCLASS:

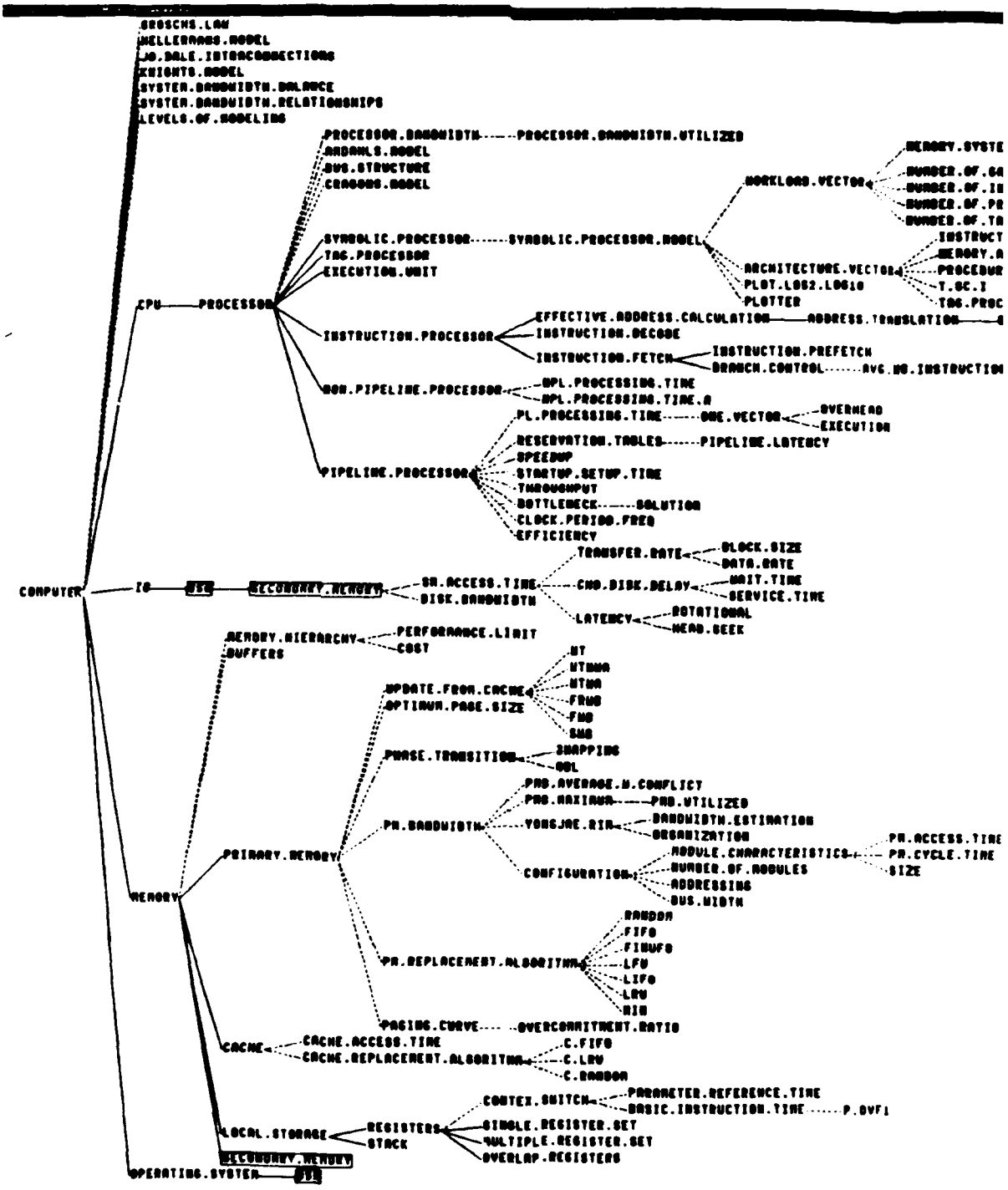
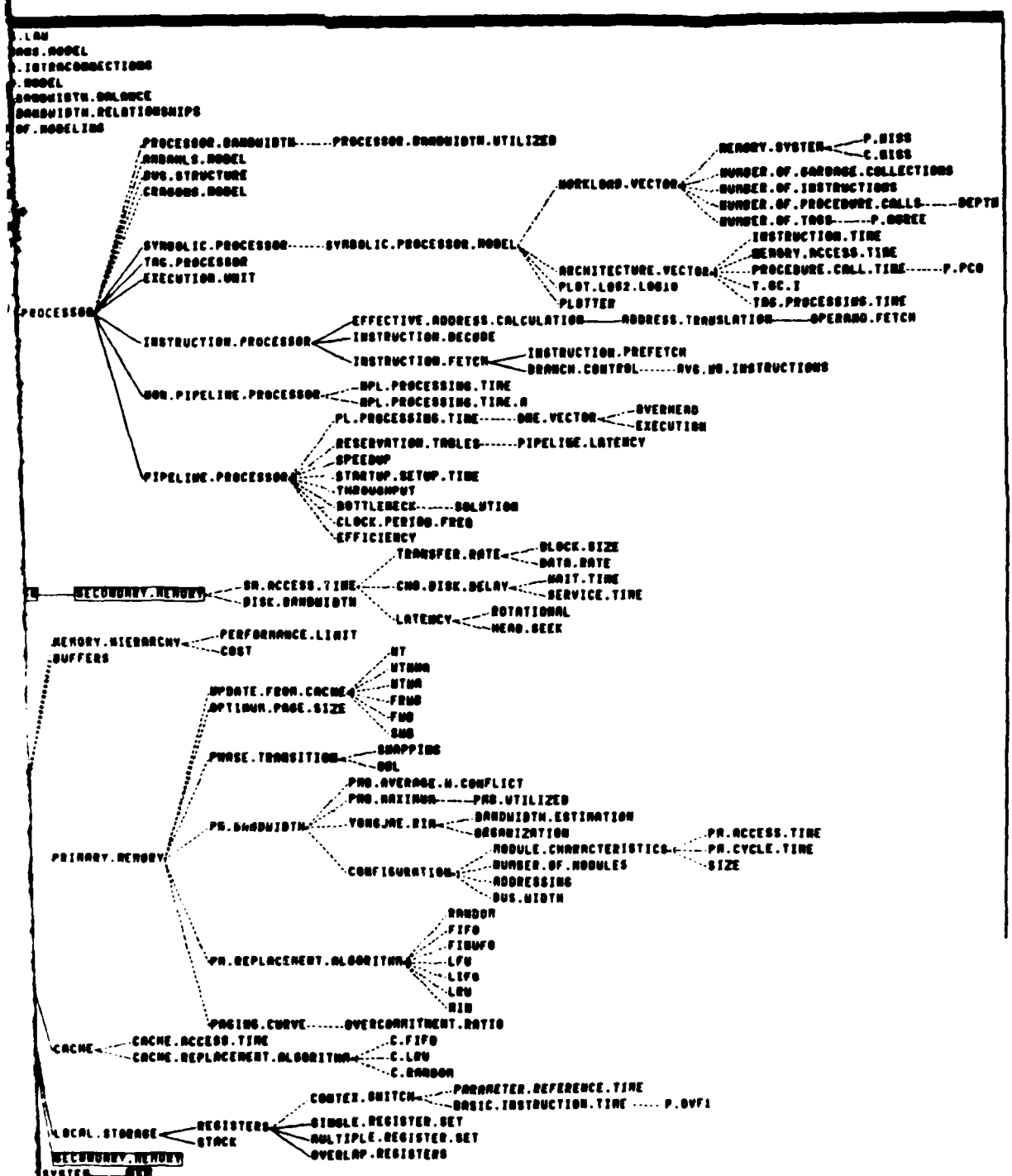


Figure A-1 Taxonomy of Performance Models



re A-1 Taxonomy of Performance Models

BIBLIOGRAPHY

Allen, A.O. "Queueing Models of Computer Systems," Computer, Vol. 13, No. 4, April 1980, pp. 13-24.

Amdahl, Gene M. "Validity of the single processor approach to achieving large scale computing capabilities," Spring Joint Conference, 1967, pp. 483-485.

Baer, J.L. Computer Systems Architecture, Computer Science Press, 1980.

Bard, Yonatnan; Sauer, Charles H. "IBM Contributions to Computer Performance Modeling," IBM Journal of Research and Development, Vol. 25, No. 5, September, 1981, pp. 562-569.

Becker, G. "How to speak the performance lingo (or at least understand it)," CMG XV International Conference on the Management and Performance Evaluation of Computer Systems, Conference Proceedings, 4 Dec., 1984.

Bell, C. Gordon, Mudge, J. Craig, McNamara, John E., Computer Engineering: A DEC view of hardware systems design, Bedford, MA: Digital Press, 1978.

Bucci, Giacomo; Maio, Dario. "Merging Performance and Cost Benefit Analysis in Computer System Evaluation," Computer, September 1982, pp. 23-31.

Brooks, Frederick P. Jr. The Mythical Man-Month: Essays on Software Engineering, Addision-Wesley, 1975.

Browne, J.C. "Understanding Execution Behavior of Software Systems," Computer, July 1984, pp. 83-87.

Cragon, Harvey G. "An Evaluation of Code Space Requirements and Performance of Various Architectures," Computer Architecture News, Vol. 7, No. 5, Feb 1979, pp. 5-21.

Cragon, Harvey G. "Performance Modeling and Evaluation," viewgraphs from a presentation, Feb. 7, 1986.

Cragon, Harvey G. [86B], "Processor and Workload Modeling," unpublisned paper, July 10, 1986.

Cragon, Harvey G. [86C], classnotes in Advanced Computer Architecture, Spring 1986, unpublished, University of Texas, Austin, Texas.

Ein-dor, P. "Grosch's Law Revisited: CPU Power and the Cost of Computation," Communications of the ACM, Vol. 28, No. 2, Feb. 1985, pp. 142-151.

Electronics, "A Simple Design May Pay Off Big for Hewlett-Packard," Electronics , March 3, 1986.

Fikes, R., Kehler, T. "The Role of Frame-Based Representation in Reasoning," Communications of the ACM, Vol. 28, No. 9, Sept. 1985 pp. 904-920.

Halladay, S. "An Architectural Overview of the Performance Analysis Process," CMG XV International Conference on the Management and Performance Evaluation of Computer Systems, Conference Proceedings, 4 Dec., 1984.

Hasegawa, M., Shiegei, Y. "High-Speed Top-of-Stack Scheme for VLSI Processor," 1985 IEEE, 0149-7111/85/0000/0048, pp. 48-55.

Hayes-Roth, Frederick, Waterman, Donald A., Lenat, Douglas B. Building Expert Systems, Addison-Wesley, 1983.

Hitchcock, C.Y. III, Sprunt, H.M. "Analyzing Multiple Register Sets," 1985 IEEE, 0149-7111/85/0000/0055, pp. 55-63.

Hwang, K., Briggs, F.A. Computer Architecture and Parallel Processing, McGraw-Hill Book Company, 1984.

Kavi, K.M., Cragon, H.G. "A Conceptual Framework For the Description and Classification of Computer

Architecture," CH 1879-6/83/0000/0010, 1983 IEEE, pp. 10-19.

KEE Users Manual [KEE], Version 2.1, July 18, 1985, Intellicorp, Inc.

KEE ActiveImages Manual [KEE AI], Version 2.1, October 17, 1985, Intellicorp, Inc.

KEE Release Notes [KEE RN], Version 2.1, September 10, 1985, Intellicorp, Inc.

Kidder, Tracy. The Soul of a New Machine, Boston: Little, Brown and Company, 1981.

Kimbleton, Stephen R. "The Role of Computer System Performance Models in Performance Evaluation," Communications of the ACM, Vol. 15, No. 7, July 1972, pp. 586-590.

King, W.F. III, Smith, S.E., Wladawsky, I. "Effects of Serial Programs in Multiprocessing Systems," Serial Programs, July 1974, pp. 303-308.

Knight Kenneth E. "Changes in Computer Performance," Datamation, September 1966, pp. 40-54.

Knight, Kenneth E. "Evolving Computer Performance 1963-1967," Datamation, January 1968, pp. 31-35.

Kobayashi, M. "Dynamic Profile of Instruction Sequences for the IBM System/370," 1983 IEEE, 0018-9340/83/0900-0859, pp. 859-861.

Lee, J.K.F., Smith, A.J. "Branch Prediction Strategies and Branch Target Buffer Design," Computer, Jan. 1984, pp. 6-22.

Levine, A.P. "An Expert System for Computer Performance Modeling: design issues," CMG XV International Conference on the Management and Performance Evaluation of Computer Systems, Conference Proceedings, 4 Dec., 1984.

Lubeck, O., Moore, J., Mendez, P. "A Benchmark Comparison of Three Supercomputers: Fujitsu VP-200, Hitachi S810/20, and Cray X-MP/2" Computer, Dec 1985, pp. 10-23.

MacDougall, M.H. "Instruction Level Program and Processor Modeling," Computer, July 1984, pp. 14-24.

Norton, A., Pfister, G.F. "A Methodology for Predicting Multiprocessor Performance," *1985 IEEE, 0190-3918/85/0000/0772* pp. 772-781.

Parnas, D.L. "A Technique for Software Module Specification with Examples," Communication of the ACM, Vol. 15, No. 5, May 1972, pp. 330-336.

Parnas, D.L. "On the Criteria To Be Used in Decomposing Systems into Modules," Communications of the ACM, Vol. 15, No. 12, December 1972, pp. 1053-1058.

Sauer, C.H., Chandy, K.M., "Approximate Solution of Queueing Models," Vol. 13, No. 4, April 1980, pp. 25-31.

Sauer, Charles H., Chandi, K. Mani. Computer Systems Performance Modeling, Englewood Cliffs, NJ: Prentice-Hall, 1981.

Shankar, K.S. "Data Structures, Types, and Abstractions," Vol. 13, No. 4, April 1980, pp. 47-54.

Smith, J.E. "A Study of Branch Prediction Strategies," 1981 IEEE, 0149-7111/81/0000/0135, pp. 135-148.

Spargins, J. "Analytical Queueing Models," Computer, Vol. 13, No. 4, April 1980, pp. 9-11.

Stroebel, Gary J., Baxter, Randy D., Denney, Michael J., "A Capacity Planning Expert System for IBM System/38," Computer, July, 1986.

Trivedi, K.S., Kinicki, R.E. "A Model For Computer Configuration Design," Vol. 13, No. 4, April 1980, pp. 47-54.

Welch, T.A. "Analysis of Memory Hierarchies for Sequential Data Access," Computer, May 1979, pp. 19-26.

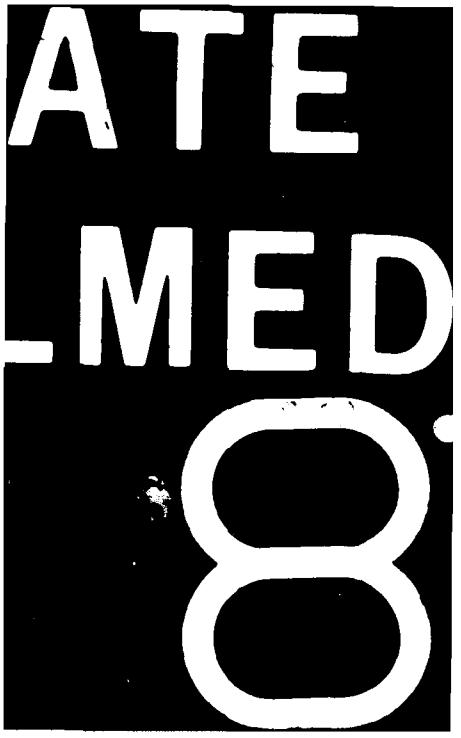
Welch, Terry A. "Memory Hierarchy Configuration Analysis," IEEE Transactions on Computers, Vo. C-27, No. 5, May 1978, pp. 408-413.

VITA

Michael William Strevell, son of John William Strevell and Jane Marie Strevell, was born in Albany, New York on April 23, 1955. He grew up in Dayton, Ohio and graduated from Carroll High School in 1973. He entered the United States Air Force Academy in Colorado Springs, Colorado in 1973. He graduated from the Academy with a B.S. degree in Electrical Engineering and a commission in the U.S. Air Force in 1977. He completed Pilot Training in 1978, and flew B-52H intercontinental nuclear bombers in Minot, N.D. until 1983. During that time he also completed an M.S. degree in Industrial Engineering and Management at North Dakota State University. From 1983 to 1986 he was a project manager in Computer Integrated Manufacturing at Wright-Patterson AFB, Ohio. In January, 1986 he entered The Graduate School of The University of Texas. His research interests include Computer Architecture, Artificial Intelligence, and Expert Systems.

Permanent address: 736 Willow Road
Naperville, IL 60540

This thesis was typed by Papers-To-Go, Austin, Texas.


ATE
MED
8