ARMY RESEARCH LABORATORY

Effects of Loop Unrolling and Loop Fusion on Register Pressure and Code Performance

by Dale Shires

ARL-TR-1386 June 1997

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer need. Do not return it to the originator.

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5066

ARL-TR-1386 **June 1997**

Effects of Loop Unrolling and Loop Fusion on Register Pressure and Code Performance

Dale Shires
Weapons and Materials Research Directorate, ARL

Abstract

Many of today's high-performance computer processors are super-scalar. They can dispatch multiple instructions per cycle and, hence, provide what is commonly referred to as instruction-level parallelism. This super-scalar capability, combined with software pipelining, can increase processor throughput dramatically. Achieving maximum throughput, however, is nontrivial. Compilers must engage in aggressive optimization techniques, such as loop unrolling, speculative code motion, etc., to structure code to take full advantage of the underlying computer architecture. The phase-ordering implications of these optimizations are not well understood and are the subject of continuing research. Of particular interest are optimizations that enhance instruction-level parallelism. Two of these are loop unrolling and loop fusion. These are source-level optimizations that can be performed by either the programmer or the compiler. These optimizations have dramatic effects on the compiler's instruction scheduler. Performed too aggressively, these optimizations can increase register pressure and result in costly memory references. This paper details experiments performed to measure the effects of these source-level code transformations and how they influenced register pressure and code performance.

Contents

1 Introduction and Problem Statement 1

2 Experimental Methodology 2

3 Results 2

3.1 Loop Unrolling 2

3.2 Loop Fusion 5

4 Conclusion 6

References 7

A Loop Unrolling Examples 9

B Loop Fusion Examples 13

C Loop Unrolling Pipeline Success Messages 17

D Loop Fusion Pipeline Success Messages 19

Distribution List 21

Report Documentation Page 23

INTENTIONALLY LEFT BLANK

1 Introduction and Problem Statement

Modern high-performance computer platforms are capable of achieving incredible levels of code execution speed. One way they increase performance is by taking advantage of parallelism found in algorithms. To this end, many of these systems offer multiprocessor parallelism. Furthermore, many also offer software pipelining to take full advantage of low-level, or code-level, parallelism [1]. This is parallelism actually present in the way machine instructions are dispatched.

Also of paramount importance is that these machines take full advantage of their complicated memory systems. Most of the standard optimization techniques will really only provide maximum performance if the computer's memory system is being used in an efficient manner. Most shared-memory architectures have some type of memory hierarchy. The main reason memories are implemented in this fashion is to optimize the price-performance ratio, given the widening gap between central processing unit (CPU) speed and main memory performance. CPU speeds are currently doubling about every 2 or 3 years while the speed of main memory has historically doubled only about every decade. These tiered memories, with their nondeterministic behavior, are hard to manage and predict. This makes the job of the compiler's code generator that much more difficult. Memory systems have become so complicated on some architectures that slight memory reference changes on codes may speed up or slow down execution by an order of magnitude.

Since registers provide fast data access, one goal of the compiler back end is to allocate and assign registers in an effective manner. The register allocator tries to assign a register to each register candidate. Since register access is very fast, the compiler should generate code that reuses these assigned registers as much as possible. To do this, the register allocator and scheduler should work closely together [2]. This is actually a very complicated phase-ordering problem. At the least, the scheduler should order code in a way that instruction-level parallelism can be exploited, and the register allocator should give top priority to assigning a register to frequently used variables. Several source-level optimizations can be performed with the goal of increasing memory locality and instruction-level parallelism and thus assisting the code generator.

The problem, however, is that when these optimizations are pursued too aggressively, they can reach a point of diminishing returns. When the compiler starts to run out of available registers to use, register pressure is said to be high. At the point where registers are no longer available, the register allocator must actually "spill" a register's content to memory to free it for other uses [3]. On tiered memory machines, such an action can be detrimental to varying degrees. If the value is written to cache, the access time is very small, but the cache manager may still have to invalidate the cache line. This operation can cause the cache line to be rewritten to main memory. Actual main memory access can be very expensive. On the Silicon Graphics (SGI) Power Challenge architecture, this delay, though seldom reaching this point, can be as high as 90 cycles. Accordingly, the governing hypothesis for this study is that memory locality and code-level, parallelism-enhancing transformations are beneficial only to the point where register pressure becomes very high.

2 Experimental Methodology

The example codes listed in this paper were all run on the SGI Power Challenge architecture. This is a 64-bit architecture using 75-MHz MIPS R8000 processors. There are 32 64-bit floating-point registers available to the assembler. The architecture is superscalar and can dispatch up to four instructions per cycle. Prefetching is not implemented in the Power Challenge architecture. This machine uses a hierarchical memory structure like the one described previously.

Loop unrolling and loop fusion were the two transformations that were studied in this experiment. These are common transformations, and loop unrolling especially is the most heavily used transform to increase instruction-level parallelism. All programs were written in C and were compiled with the SGI MIPSpro compiler version 6.0.2. Two compile options were used. The first was `-O2`, which turns on extensive optimization. These optimizations are conservative in that they almost always provide some speedup and maintain floating-point accuracy. The second was `-O3`, which is aggressive optimization. The main consequence of `-O3` optimization is that it turns on software pipelining. The code scheduler attempts to pipeline innermost loops whenever possible.

A discovery made halfway through these trials led to a small change in the analysis of the results. In version 6.0.1 of the compiler system (running at the University of Delaware), the pipeline scheduler would give up if it could not generate a schedule without register spilling. The newer version of the compiler (running at the Army Research Laboratory), however, will still schedule pipelined loops with spill code introduced. The general hypothesis remains the same. The new twist is that spilling will limit pipelining usefulness.

3 Results

3.1 Loop Unrolling

Loop unrolling replicates the body of a loop some number of times known as the unrolling factor. Loop unrolling has the ability to increase performance in two ways. First, it reduces loop overhead by performing less compare and branch instructions. Second, it increases work performed in the resulting larger loop body by allowing more opportunity for optimization and register usage. Most of the increase in performance speed on the SGI is because multiplication and addition instructions may be overlapped in the multiple instruction cycle.

A simple 2-D matrix multiply code fragment was used to test unrolling effects on the R8000 processor. This code is listed in Appendix A. Four unrolled versions of the matrix multiply were implemented in different functions. There is a caveat. The author does not claim this code to be the best version of matrix multiply possible. Simply, the base version is straightforward and provides a good example of unrolling for memory locality. Other C codes with loop reordering and splitting will undoubtedly come closer in reaching near-theoretical peak on the SGI architecture than these versions. The function MM_basic is the basic matrix multiply loop. The optimizer unrolled the inner loop four times when this was compiled. The hand-coded unrolling of the other functions was performed on the outer and middle loop nests. The exact unrolling can be seen in the code listing in Appendix A. The optimizer did not unroll the inner loop in these cases.

Various data were collected during program execution. The results are displayed in Table 1. Column one lists the function name. Columns "-O2" and "-O3" list the run times of the code compiled with the two flags, respectively. The rest of the columns pertain only to the executable compiled with -O3 optimization. "Cycles/Iteration" lists how many computer cycles were required to perform one complete iteration of the inner loop. For instance,

Table 1: Matrix multiply performance and other statistics.

	Run Time (sec)		-O3 Compiled Executable		
Function	-O2	-O3	Cycles/Iteration	FLOPS (%)	Memory (%)
MM_basic	94.37	91.84	0.67	33	100
MM_unroll_1	20.17	14.59	0.58	85	71
MM_unroll_2	13.38	10.45	0.56	88	66
MM_unroll_3	14.23	14.28	0.67	74	44

in MM_unroll_2, both the outer and middle loops were unrolled four times. The inner loop, therefore, actually completes 16 iterations each time it is executed. The scheduler reported this loop to be pipelined with a steady-state of nine cycles per iteration. In this case, the number reported in the table is derived from dividing the steady state number of cycles by the total number of computations performed by one iteration of the inner loop. FLOPS gives the compiler-calculated rate of floating-point operations per second based on the MIPS R8000's ability to perform two such operations per cycle. Memory lists the percentage of peak memory references achieved. The maximum is two each cycle.

As evident from the timing profiles, the function MM_basic is perhaps the worst way of performing a matrix multiply. This poor performance results from the inefficient way in which memory is being utilized. The best way to check on memory performance is through profiling. Two profiling mechanisms are available on the SGI operating system: `prof` and `pixie`. Comparison of their outputs tells on a procedure-by-procedure basis how well the memory system is performing. `Prof` uses program counter sampling to collect data. It interrupts the code periodically and records the location of the program counter. The condensed `prof` output is listed as follows:

```
samples   time(%)       cum time(%)       procedure (file)

  28046 2.8e+02s( 34.1) 2.8e+02s( 34.1)       MM_basic
  18403 1.8e+02s( 22.4) 4.6e+02s( 56.6)    MM_unroll_1
  18236 1.8e+02s( 22.2) 6.5e+02s( 78.8)    MM_unroll_2
  17263 1.7e+02s( 21.0) 8.2e+02s( 99.8)    MM_unroll_3
```

In contrast, `pixie` instruments the code with counters at the beginning and end of basic blocks. It counts only the number of cycles the program executes and does not account for cache misses, bank conflicts, etc. The abbreviated `pixie` output is given as follows:

```
       cycles(%)  cum %     secs     instrns     calls procedure(file)

14883848018(28.12)  28.12   198.45 29765772028         1 MM_basic
12760324018(24.10)  52.22   170.14 26840486028         1 MM_unroll_1
12630242018(23.86)  76.08   168.40 26610363028         1 MM_unroll_2
12540096818(23.69)  99.77   167.20 26484145228         1 MM_unroll_3
```

Optimizers can affect the accuracy of profiling. Therefore, the profiled executables were created with optimizations disabled.

In the best case, `pixie` is reporting that MM_basic should complete in about 198 seconds. `Prof` is showing that it is taking about 280 seconds. This shows that the current structure of the code is not working well with the memory system. The unrolled code fragments dramatically illustrate the advantages of loop unrolling. In these cases, loop unrolling was the means to achieve register (or loop) blocking. By unrolling the various loops, loads and stores for several array elements were highly reduced. The memory system performed much better, as evident from the run time as well as the closely matched times given in the `prof` and `pixie` profiles.

Getting the maximum benefits from a compiler usually requires having a detailed knowledge of the many optional flags to control the fine points in the compiling process. The MIPSpro compiler is no different. With standard options, the compiler could not pipeline the loop body for MM_unroll_3 because the loop body was too long. The compile option `-SWP:body_ins=250` was used to increase the maximum size of a loop body that would be considered for software pipelining.

Loop unrolling led to great speed increases. Unrolling with pipelining allowed the basic matrix multiply to execute at 33% efficiency. Without unrolling, efficiency is only around 10% of the maximum throughput. Loop unrolling with the goal of register blocking achieved even greater results. The software pipeliner, which allows differing loop iterations to overlap, was able to achieve speedup over standard `-O2` optimization in almost every case.

Unrolling does reach a point of maximum usefulness in these test cases. With each function, more and more unrolling was done in order to promote register reuse and instruction-level parallelism. MM_unroll_2 has extensive unrolling but does not produce any spill code. Implementation of MM_unroll_3 however, produces extensive spilling. A quick check of the statistics reported in Table 1 graphically show that the point has been reached at which unrolling is harming execution speed. For MM_unroll_3, the cycles/iteration is higher and the FLOP rate is smaller than those figures reported for MM_unroll_2. The actual output from the scheduler is listed next.[1]

```
#<swps>      Not unrolled before pipelining
#<swps>   27 cycles per iteration
#<swps>   40 flops        ( 74% of peak) (madds count as 1)
#<swps>   40 madds        ( 74% of peak)
#<swps>   24 mem refs     ( 44% of peak)
#<swps>    3 integer ops  (  5% of peak)
#<swps>   67 instructions ( 62% of peak)
#<swps>   32 fgr registers used.
#<swps>   29 restores introduced.
#<swps>   14 possible stall cycles
#<swps>   14 min possible stall cycles
```

As expected, register spilling does hurt the speed of execution in this case. A massive amount of spills and restores has been added by the scheduler, and even pipelining cannot hide the resultant delays.

In the pipeline message, there is a statement saying 14 possible stall cycles may exist. Most stalls on this processor occur due to the floating-point unit and the integer unit of

[1]Complete pipeliner messages for the example codes are listed in Appendices C and D.

the CPU becoming unsynchronized. Several factors may lead to this occurrence. Indirect addressing, such as `a[b[i]]`, will cause the lookup of `b[i]` to complete before the load/store can begin. Multidimensional arrays, prevalent in these examples, lead to similar problems. These integer unit operations paired together with the many floating-point multiplication operations may be the reason the compiler is warning of worst-case synchronization stalls. How much of the degradation in MM_unroll_3 is attributable to stall cycles and how much is attributable to spilling is hard to determine.

3.2 Loop Fusion

Loop fusion is a process where two or more adjacent loops are merged into a single loop. Loop fusion has the potential to increase performance by reducing loop overhead and increasing instruction-level parallelism.

A somewhat contrived example was used to test fusion on the R8000 processor. The loops were deliberately designed to give variables long live ranges and hence to make things as difficult as possible for the scheduler to achieve scheduling, not to mention pipelining, without introduction of some spill code. The code is listed in Appendix B. The loops in the NotFused function are named loop1, loop2, and loop3. Table 2 lists the execution results.

Table 2: Loop Fusion Performance and Other Statistics.

	Run Time (sec)		-O3 compiled executable		
Function	-O2	-O3	Cycles/Iteration	FLOPS (%)	Memory (%)
NotFused	0.97	0.81			
loop1			18.0	33	100
loop2			11.0	68	95
loop3			11.0	68	95
Fused	0.67	1.26	48.0	43	53

If the loops are not pipelined, the fused loop does indeed outperform the three separate loops. The multiple compare and branch instructions executed in the three loops can be extremely costly because they often interfere with maximum instruction issue per cycle. In this case, the reduction in loop overhead increased code speed by 1.4. The fused loops did create a small amount of spill code, but the effects seem to be negligible compared to cycles lost on compare and branch instructions.

For pipelining, however, the spill code seemed to cause a greater problem. The pipeliner reported numerous potential problems:

```
#<swps>      Not unrolled before pipelining
#<swps>    48 cycles per iteration
#<swps>    42 flops        ( 43% of peak) (madds count as 1)
#<swps>     0 madds        (  0% of peak)
#<swps>    51 mem refs     ( 53% of peak)
#<swps>     6 integer ops  (  6% of peak)
#<swps>    99 instructions ( 51% of peak)
#<swps>    32 fgr registers used.
#<swps>     3 spills 5 restores introduced.
```

```
#<swps>    25 possible stall cycles
#<swps>    11 min possible stall cycles
#<swps>    26 min cycles required for resources
#<swps>    48 cycle schedule register allocated.
#<swps>    30 min cycles required for resources with additional memory refs.
#<swps>    30 min cycles required for recurrences with additional memory refs.
```

Stalls are once again present, and this time there is a warning about the number of cycles required to deal with resources and recurrences with memory references. All of these problems seem to have a very bad cumulative effect on the final performance. This code was not written with any regard to memory locality. The three loops taken separately and pipelined performed fairly well, but could still not perform as well as the fused, nonpipelined loop. The spill code in the pipelined fused loop has put extreme burdens on the memory system and has caused a severe loss of performance.

4 Conclusion

To be of maximum usefulness, the scheduler of a compiler must be able to fully take into account the extremely complicated memory systems in most of today's shared-memory, high-performance computers. As has been shown from these examples, transformations to increase memory locality as well as reduce loop overhead and promote instruction-level parallelism can be extremely advantageous. They are, however, extremely interrelated, and promoting one often takes place with the detriment of the other.

Is there a best choice for ordering these transformations or some way of knowing how much of one to perform? Building such knowledge into the compiler will be very difficult. Optimal scheduling is itself an NP-complete problem, and predicting memory system behavior is difficult. Building extensive information about the memory system into the compiler will undoubtedly greatly increase compile time and with the nondeterministic behavior of the memory system still have the potential to not be totally accurate. Those codes worthy of extensive analysis and optimization will probably be best served by having compilers that generate detailed messages about the actions they took that will allow the programmer to make more informed choices about optimizing source-level code structure. It seems that only through profiling and modifying code by hand can maximum performance be achieved on a per-architecture basis. Some general conclusions are noteworthy, however:

- Loop unrolling is very efficient at promoting instruction-level parallelism.
- Loop fusion is very efficient at removing costly compare and branch instructions and may be more efficient than pipelining in some cases.
- Large loop bodies with somewhat random or erratic memory access patterns will seldom benefit from pipelining. These loops will either be better off not pipelined or distributed and then pipelined if possible.
- Codes written that take into account the memory system should in most cases benefit from pipelining.
- Loop unrolling to promote register reuse is only efficient to just prior to the point where spill code must be introduced.

References

[1] J. L. Lo and S. J. Eggers. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. In *ACM SIGPLAN 1995*, pages 151–162, 1995.

[2] C. Norris and L. Pollock. An experimental study of several cooperative register allocation and instruction scheduling strategies. In *MICRO 28*. The 28th International Symposium on Micro Architecture, November 1995.

[3] G. Chaitin, M. Auslander, and A. Chandra. Register allocation via coloring. In *Computer Languages*, volume 6, pages 47–57. 1981.

INTENTIONALLY LEFT BLANK

A Loop Unrolling Examples

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#define N 1000
#define L 480
#define M 1000

double a[M][L], b[L][N], c[M][N];

void init() {
   int i, j, k;

   for (i=0; i<M; i++)
      for (k=0; k<L; k++)
      a[i][k] = drand48();

   for (k=0; k<L; k++)
      for (j=0; j<N; j++)
      b[k][j] = drand48();

   for (i=0; i<M; i++)
      for (j=0; j<N; j++)
         c[i][j] = 0.0;

   }


void MM_basic() {
   int i, j, k;

   StartTimer();

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

   StopTimer();

   printf("%f\n", c[1][1]);

   }


void MM_unroll_1() {
   int i, j, k;

   StartTimer();

   for (j=0; j<N; j+=2)
      for (k=0; k<L; k+=6)
         for (i=0; i<M; i++) {
            c[i][j+0] = c[i][j+0] + a[i][k+0] * b[k+0][j+0];
            c[i][j+0] = c[i][j+0] + a[i][k+1] * b[k+1][j+0];
            c[i][j+0] = c[i][j+0] + a[i][k+2] * b[k+2][j+0];
            c[i][j+0] = c[i][j+0] + a[i][k+3] * b[k+3][j+0];
            c[i][j+0] = c[i][j+0] + a[i][k+4] * b[k+4][j+0];
            c[i][j+0] = c[i][j+0] + a[i][k+5] * b[k+5][j+0];

            c[i][j+1] = c[i][j+1] + a[i][k+0] * b[k+0][j+1];
```

```
            c[i][j+1] = c[i][j+1] + a[i][k+1] * b[k+1][j+1];
            c[i][j+1] = c[i][j+1] + a[i][k+2] * b[k+2][j+1];
            c[i][j+1] = c[i][j+1] + a[i][k+3] * b[k+3][j+1];
            c[i][j+1] = c[i][j+1] + a[i][k+4] * b[k+4][j+1];
            c[i][j+1] = c[i][j+1] + a[i][k+5] * b[k+5][j+1];

            }

   StopTimer();

   printf("%f\n", c[1][1]);

   }


void MM_unroll_2() {
   int i, j, k;

   StartTimer();

   for (j=0; j<N; j+=4)
      for (k=0; k<L; k+=4)
         for (i=0; i<M; i++) {
            c[i][j+0] = c[i][j+0] + a[i][k+0] * b[k+0][j+0];
            c[i][j+0] = c[i][j+0] + a[i][k+1] * b[k+1][j+0];
            c[i][j+0] = c[i][j+0] + a[i][k+2] * b[k+2][j+0];
            c[i][j+0] = c[i][j+0] + a[i][k+3] * b[k+3][j+0];

            c[i][j+1] = c[i][j+1] + a[i][k+0] * b[k+0][j+1];
            c[i][j+1] = c[i][j+1] + a[i][k+1] * b[k+1][j+1];
            c[i][j+1] = c[i][j+1] + a[i][k+2] * b[k+2][j+1];
            c[i][j+1] = c[i][j+1] + a[i][k+3] * b[k+3][j+1];

            c[i][j+2] = c[i][j+2] + a[i][k+0] * b[k+0][j+2];
            c[i][j+2] = c[i][j+2] + a[i][k+1] * b[k+1][j+2];
            c[i][j+2] = c[i][j+2] + a[i][k+2] * b[k+2][j+2];
            c[i][j+2] = c[i][j+2] + a[i][k+3] * b[k+3][j+2];

            c[i][j+3] = c[i][j+3] + a[i][k+0] * b[k+0][j+3];
            c[i][j+3] = c[i][j+3] + a[i][k+1] * b[k+1][j+3];
            c[i][j+3] = c[i][j+3] + a[i][k+2] * b[k+2][j+3];
            c[i][j+3] = c[i][j+3] + a[i][k+3] * b[k+3][j+3];

            }

   StopTimer();
   printf("%f\n", c[1][1]);

   }


void MM_unroll_3() {
   int i, j, k;

   StartTimer();

   for (j=0; j<N; j+=10)
      for (k=0; k<L; k+=4)
         for (i=0; i<M; i++) {
            c[i][j+0] = c[i][j+0] + a[i][k+0] * b[k+0][j+0];
            c[i][j+0] = c[i][j+0] + a[i][k+1] * b[k+1][j+0];
            c[i][j+0] = c[i][j+0] + a[i][k+2] * b[k+2][j+0];
            c[i][j+0] = c[i][j+0] + a[i][k+3] * b[k+3][j+0];

            c[i][j+1] = c[i][j+1] + a[i][k+0] * b[k+0][j+1];
```

```
            c[i][j+1] = c[i][j+1] + a[i][k+1] * b[k+1][j+1];
            c[i][j+1] = c[i][j+1] + a[i][k+2] * b[k+2][j+1];
            c[i][j+1] = c[i][j+1] + a[i][k+3] * b[k+3][j+1];

            c[i][j+2] = c[i][j+2] + a[i][k+0] * b[k+0][j+2];
            c[i][j+2] = c[i][j+2] + a[i][k+1] * b[k+1][j+2];
            c[i][j+2] = c[i][j+2] + a[i][k+2] * b[k+2][j+2];
            c[i][j+2] = c[i][j+2] + a[i][k+3] * b[k+3][j+2];

            c[i][j+3] = c[i][j+3] + a[i][k+0] * b[k+0][j+3];
            c[i][j+3] = c[i][j+3] + a[i][k+1] * b[k+1][j+3];
            c[i][j+3] = c[i][j+3] + a[i][k+2] * b[k+2][j+3];
            c[i][j+3] = c[i][j+3] + a[i][k+3] * b[k+3][j+3];

            c[i][j+4] = c[i][j+4] + a[i][k+0] * b[k+0][j+4];
            c[i][j+4] = c[i][j+4] + a[i][k+1] * b[k+1][j+4];
            c[i][j+4] = c[i][j+4] + a[i][k+2] * b[k+2][j+4];
            c[i][j+4] = c[i][j+4] + a[i][k+3] * b[k+3][j+4];

            c[i][j+5] = c[i][j+5] + a[i][k+0] * b[k+0][j+5];
            c[i][j+5] = c[i][j+5] + a[i][k+1] * b[k+1][j+5];
            c[i][j+5] = c[i][j+5] + a[i][k+2] * b[k+2][j+5];
            c[i][j+5] = c[i][j+5] + a[i][k+3] * b[k+3][j+5];

            c[i][j+6] = c[i][j+6] + a[i][k+0] * b[k+0][j+6];
            c[i][j+6] = c[i][j+6] + a[i][k+1] * b[k+1][j+6];
            c[i][j+6] = c[i][j+6] + a[i][k+2] * b[k+2][j+6];
            c[i][j+6] = c[i][j+6] + a[i][k+3] * b[k+3][j+6];

            c[i][j+7] = c[i][j+7] + a[i][k+0] * b[k+0][j+7];
            c[i][j+7] = c[i][j+7] + a[i][k+1] * b[k+1][j+7];
            c[i][j+7] = c[i][j+7] + a[i][k+2] * b[k+2][j+7];
            c[i][j+7] = c[i][j+7] + a[i][k+3] * b[k+3][j+7];

            c[i][j+8] = c[i][j+8] + a[i][k+0] * b[k+0][j+8];
            c[i][j+8] = c[i][j+8] + a[i][k+1] * b[k+1][j+8];
            c[i][j+8] = c[i][j+8] + a[i][k+2] * b[k+2][j+8];
            c[i][j+8] = c[i][j+8] + a[i][k+3] * b[k+3][j+8];

            c[i][j+9] = c[i][j+9] + a[i][k+0] * b[k+0][j+9];
            c[i][j+9] = c[i][j+9] + a[i][k+1] * b[k+1][j+9];
            c[i][j+9] = c[i][j+9] + a[i][k+2] * b[k+2][j+9];
            c[i][j+9] = c[i][j+9] + a[i][k+3] * b[k+3][j+9];

            }

   StopTimer();

   printf("%f\n", c[1][1]);

   }

main() {
   init();
   MM_basic();
   MM_unroll_1();
   MM_unroll_2();
   MM_unroll_3();
   }
```

INTENTIONALLY LEFT BLANK

B Loop Fusion Examples

```
#include <stdio.h>

#define N 1000
#define M 1000

double x1[N];
double x2[N];
double x3[N];
double x4[N];
double x5[N];
double x6[N];
double x7[N];
double x8[N];
double x9[N];
double x10[N];
double x11[N];
double x12[N];

double y1[N];
double y2[N];
double y3[N];
double y4[N];
double y5[N];
double y6[N];
double y7[N];
double y8[N];
double y9[N];
double y10[N];
double y11[N];
double y12[N];

double z1[N];
double z2[N];
double z3[N];
double z4[N];
double z5[N];
double z6[N];
double z7[N];
double z8[N];
double z9[N];
double z10[N];
double z11[N];
double z12[N];

double a1[N];
double a2[N];
double a3[N];
double a4[N];
double a5[N];
double a6[N];
double a7[N];
double a8[N];
double a9[N];

double b1[N];
double b2[N];
double b3[N];
double b4[N];
double b5[N];
double b6[N];
```

```
double c1[N];
double c2[N];
double c3[N];

void NotFused() {
   int i, j;

   StartTimer();

   for (j=0; j<M; j++) {

      /* loop1 */
      for (i=0; i<N; i++) {
         x9[i] = x8[i] - x1[i];
         x10[i] = x7[i] - x2[i];
         x11[i] = x6[i] - x3[i];
         x12[i] = x5[i] - x4[i];
         y9[i] = y8[i] - y1[i];
         y10[i] = y7[i] - y2[i];
         y11[i] = y6[i] - y3[i];
         y12[i] = y5[i] - y4[i];
         z9[i] = z8[i] - z1[i];
         z10[i] = z7[i] - z2[i];
         z11[i] = z6[i] - z3[i];
         z12[i] = z5[i] - z4[i];
         }

      /* loop2 */
      for (i=0; i<N; i++) {
         a1[i] = x9[i] + x10[i] + x11[i] + x12[i];
         a2[i] = y9[i] + y10[i] + y11[i] + y12[i];
         a3[i] = z9[i] + z10[i] + z11[i] + z12[i];
         a4[i] = x9[i] + x12[i];
         a5[i] = x10[i] + x11[i];
         a6[i] = y9[i] + y12[i];
         a7[i] = y10[i] + y11[i];
         a8[i] = z9[i] + z12[i];
         a9[i] = z10[i] + z11[i];
         }

      /* loop3 */
      for (i=0; i<N; i++) {
         b1[i] = a1[i] + a2[i] + x1[i] + x2[i] + x7[i];
         b2[i] = a2[i] + a3[i] + x3[i] + x4[i] + y7[i];
         b3[i] = a3[i] + a1[i] + x5[i] + x6[i] + z7[i];
         c1[i] = y1[i] + a1[i];
         c2[i] = y2[i] + a2[i];
         c3[i] = y3[i] + a3[i];
         }

      }

   StopTimer();

   printf("%f\n", c3[1]);

   } /* end NotFused */


void Fused() {
   int i, j;

   StartTimer();

   for (j=0; j<M; j++)
```

```
      for (i=0; i<N; i++) {
         x9[i] = x8[i] - x1[i];
         x10[i] = x7[i] - x2[i];
         x11[i] = x6[i] - x3[i];
         x12[i] = x5[i] - x4[i];
         y9[i] = y8[i] - y1[i];
         y10[i] = y7[i] - y2[i];
         y11[i] = y6[i] - y3[i];
         y12[i] = y5[i] - y4[i];
         z9[i] = z8[i] - z1[i];
         z10[i] = z7[i] - z2[i];
         z11[i] = z6[i] - z3[i];
         z12[i] = z5[i] - z4[i];
         a1[i] = x9[i] + x10[i] + x11[i] + x12[i];
         a2[i] = y9[i] + y10[i] + y11[i] + y12[i];
         a3[i] = z9[i] + z10[i] + z11[i] + z12[i];
         a4[i] = x9[i] + x12[i];
         a5[i] = x10[i] + x11[i];
         a6[i] = y9[i] + y12[i];
         a7[i] = y10[i] + y11[i];
         a8[i] = z9[i] + z12[i];
         a9[i] = z10[i] + z11[i];
         b1[i] = a1[i] + a2[i] + x1[i] + x2[i] + x7[i];
         b2[i] = a2[i] + a3[i] + x3[i] + x4[i] + y7[i];
         b3[i] = a3[i] + a1[i] + x5[i] + x6[i] + z7[i];
         c1[i] = y1[i] + a1[i];
         c2[i] = y2[i] + a2[i];
         c3[i] = y3[i] + a3[i];
         }

   StopTimer();

   printf("%f\n", c3[1]);

   } /* end Fused */

main() {

   NotFused();
   Fused();

   } /* end main */
```

INTENTIONALLY LEFT BLANK

C Loop Unrolling Pipeline Success Messages

```
#<swps>
#<swps> Pipelined loop line 23 steady state
#<swps>
#<swps>     4 unrollings before pipelining
#<swps>     2 cycles per 4 iterations
#<swps>     0 flops        (  0% of peak) (madds count as 2)
#<swps>     0 flops        (  0% of peak) (madds count as 1)
#<swps>     0 madds        (  0% of peak)
#<swps>     4 mem refs     (100% of peak)
#<swps>     2 integer ops  ( 50% of peak)
#<swps>     6 instructions ( 75% of peak)
#<swps>     0 short trip threshold
#<swps>     3 ireg registers used.
#<swps>     1 fgr register used.
#<swps>
#<swps>
#<swps> Pipelined loop line 36 steady state
#<swps>
#<swps>     4 unrollings before pipelining
#<swps>     6 cycles per 4 iterations
#<swps>     8 flops        ( 33% of peak) (madds count as 2)
#<swps>     4 flops        ( 33% of peak) (madds count as 1)
#<swps>     4 madds        ( 33% of peak)
#<swps>    12 mem refs     (100% of peak)
#<swps>     3 integer ops  ( 25% of peak)
#<swps>    19 instructions ( 79% of peak)
#<swps>     2 short trip threshold
#<swps>     7 ireg registers used.
#<swps>    14 fgr registers used.
#<swps>
#<swps>     6 possible stall cycles
#<swps>     6 min possible stall cycles
#<swps>
#<swps>
#<swps> Pipelined loop line 53 steady state
#<swps>
#<swps>       Not unrolled before pipelining
#<swps>     7 cycles per iteration
#<swps>    24 flops        ( 85% of peak) (madds count as 2)
#<swps>    12 flops        ( 85% of peak) (madds count as 1)
#<swps>    12 madds        ( 85% of peak)
#<swps>    10 mem refs     ( 71% of peak)
#<swps>     3 integer ops  ( 21% of peak)
#<swps>    25 instructions ( 89% of peak)
#<swps>     4 short trip threshold
#<swps>    11 ireg registers used.
#<swps>    27 fgr registers used.
#<swps>
#<swps>
#<swps> Pipelined loop line 84 steady state
#<swps>
#<swps>       Not unrolled before pipelining
#<swps>     9 cycles per iteration
#<swps>    32 flops        ( 88% of peak) (madds count as 2)
#<swps>    16 flops        ( 88% of peak) (madds count as 1)
#<swps>    16 madds        ( 88% of peak)
#<swps>    12 mem refs     ( 66% of peak)
#<swps>     3 integer ops  ( 16% of peak)
#<swps>    31 instructions ( 86% of peak)
#<swps>     2 short trip threshold
#<swps>     7 ireg registers used.
```

```
#<swps>    32 fgr registers used.
#<swps>
#<swps>     8 min cycles required for resources
#<swps>     9 cycle schedule register allocated.
#<swps>
#<swps>
#<swps> Pipelined loop line 120 steady state
#<swps>
#<swps>       Not unrolled before pipelining
#<swps>    27 cycles per iteration
#<swps>    80 flops        ( 74% of peak) (madds count as 2)
#<swps>    40 flops        ( 74% of peak) (madds count as 1)
#<swps>    40 madds        ( 74% of peak)
#<swps>    24 mem refs     ( 44% of peak)
#<swps>     3 integer ops  (  5% of peak)
#<swps>    67 instructions ( 62% of peak)
#<swps>     2 short trip threshold
#<swps>     7 ireg registers used.
#<swps>    32 fgr registers used.
#<swps>
#<swps>    29 restores introduced.
#<swps>    14 possible stall cycles
#<swps>    14 min possible stall cycles
#<swps>
```

D Loop Fusion Pipeline Success Messages

```
#<swps>
#<swps> Pipelined loop line 75 steady state
#<swps>
#<swps>       Not unrolled before pipelining
#<swps>    18 cycles per iteration
#<swps>    12 flops        ( 16% of peak) (madds count as 2)
#<swps>    12 flops        ( 33% of peak) (madds count as 1)
#<swps>     0 madds        (  0% of peak)
#<swps>    36 mem refs     (100% of peak)
#<swps>     5 integer ops  ( 13% of peak)
#<swps>    53 instructions ( 73% of peak)
#<swps>     1 short trip threshold
#<swps>     9 ireg registers used.
#<swps>    21 fgr registers used.
#<swps>
#<swps>    18 possible stall cycles
#<swps>    18 min possible stall cycles
#<swps>
#<swps>
#<swps> Pipelined loop line 90 steady state
#<swps>
#<swps>       Not unrolled before pipelining
#<swps>    11 cycles per iteration
#<swps>    15 flops        ( 34% of peak) (madds count as 2)
#<swps>    15 flops        ( 68% of peak) (madds count as 1)
#<swps>     0 madds        (  0% of peak)
#<swps>    21 mem refs     ( 95% of peak)
#<swps>     2 integer ops  (  9% of peak)
#<swps>    38 instructions ( 86% of peak)
#<swps>     3 short trip threshold
#<swps>    17 ireg registers used.
#<swps>    29 fgr registers used.
#<swps>
#<swps>    10 possible stall cycles
#<swps>    10 min possible stall cycles
#<swps>
#<swps>
#<swps> Pipelined loop line 102 steady state
#<swps>
#<swps>       Not unrolled before pipelining
#<swps>    11 cycles per iteration
#<swps>    15 flops        ( 34% of peak) (madds count as 2)
#<swps>    15 flops        ( 68% of peak) (madds count as 1)
#<swps>     0 madds        (  0% of peak)
#<swps>    21 mem refs     ( 95% of peak)
#<swps>     2 integer ops  (  9% of peak)
#<swps>    38 instructions ( 86% of peak)
#<swps>     3 short trip threshold
#<swps>    19 ireg registers used.
#<swps>    21 fgr registers used.
#<swps>
#<swps>    10 possible stall cycles
#<swps>    10 min possible stall cycles
#<swps>
#<swps>
#<swps> Pipelined loop line 126 steady state
#<swps>
#<swps>       Not unrolled before pipelining
#<swps>    48 cycles per iteration
#<swps>    42 flops        ( 21% of peak) (madds count as 2)
#<swps>    42 flops        ( 43% of peak) (madds count as 1)
```

```
#<swps>     0 madds        (  0% of peak)
#<swps>    51 mem refs     ( 53% of peak)
#<swps>     6 integer ops  (  6% of peak)
#<swps>    99 instructions ( 51% of peak)
#<swps>     1 short trip threshold
#<swps>    16 ireg registers used.
#<swps>    32 fgr registers used.
#<swps>
#<swps>     3 spills 5 restores introduced.
#<swps>    25 possible stall cycles
#<swps>    11 min possible stall cycles
#<swps>
#<swps>    26 min cycles required for resources
#<swps>    48 cycle schedule register allocated.
#<swps>    30 min cycles required for resources with additional memory refs.
#<swps>    30 min cycles required for recurrences with additional memory refs.
#<swps>
```

NO. OF COPIES	ORGANIZATION
2	DEFENSE TECHNICAL INFORMATION CENTER DTIC DDA 8725 JOHN J KINGMAN RD STE 0944 FT BELVOIR VA 22060-6218
1	HQDA DAMO FDQ DENNIS SCHMIDT 400 ARMY PENTAGON WASHINGTON DC 20310-0460
1	CECOM SP & TRRSTRL COMMCTN DIV AMSEL RD ST MC M H SOICHER FT MONMOUTH NJ 07703-5203
1	PRIN DPTY FOR TCHNLGY HQ US ARMY MATCOM AMCDCG T M FISETTE 5001 EISENHOWER AVE ALEXANDRIA VA 22333-0001
1	PRIN DPTY FOR ACQUSTN HQS US ARMY MATCOM AMCDCG A D ADAMS 5001 EISENHOWER AVE ALEXANDRIA VA 22333-0001
1	DPTY CG FOR RDE HQS US ARMY MATCOM AMCRD BG BEAUCHAMP 5001 EISENHOWER AVE ALEXANDRIA VA 22333-0001
1	ASST DPTY CG FOR RDE HQS US ARMY MATCOM AMCRD COL S MANESS 5001 EISENHOWER AVE ALEXANDRIA VA 22333-0001

NO. OF COPIES	ORGANIZATION
1	DPTY ASSIST SCY FOR R&T SARD TT F MILTON THE PENTAGON RM 3E479 WASHINGTON DC 20310-0103
1	DPTY ASSIST SCY FOR R&T SARD TT D CHAIT THE PENTAGON WASHINGTON DC 20310-0103
1	DPTY ASSIST SCY FOR R&T SARD TT K KOMINOS THE PENTAGON WASHINGTON DC 20310-0103
1	DPTY ASSIST SCY FOR R&T SARD TT B REISMAN THE PENTAGON WASHINGTON DC 20310-0103
1	DPTY ASSIST SCY FOR R&T SARD TT T KILLION THE PENTAGON WASHINGTON DC 20310-0103
1	OSD OUSD(A&T)/ODDDR&E(R) J LUPO THE PENTAGON WASHINGTON DC 20301-7100
1	INST FOR ADVNCD TCHNLGY THE UNIV OF TEXAS AT AUSTIN PO BOX 202797 AUSTIN TX 78720-2797
1	DUSD SPACE 1E765 J G MCNEFF 3900 DEFENSE PENTAGON WASHINGTON DC 20301-3900
1	USAASA MOAS AI W PARRON 9325 GUNSTON RD STE N319 FT BELVOIR VA 22060-5582

NO. OF
COPIES ORGANIZATION

1 CECOM
PM GPS COL S YOUNG
FT MONMOUTH NJ 07703

1 GPS JOINT PROG OFC DIR
COL J CLAY
2435 VELA WAY STE 1613
LOS ANGELES AFB CA 90245-5500

1 ELECTRONIC SYS DIV DIR
CECOM RDEC
J NIEMELA
FT MONMOUTH NJ 07703

3 DARPA
L STOTTS
J PENNELLA
B KASPAR
3701 N FAIRFAX DR
ARLINGTON VA 22203-1714

1 SPCL ASST TO WING CMNDR
50SW/CCX
CAPT P H BERNSTEIN
300 O'MALLEY AVE STE 20
FALCON AFB CO 80912-3020

1 USAF SMC/CED
DMA/JPO
M ISON
2435 VELA WAY STE 1613
LOS ANGELES AFB CA 90245-5500

1 US MILITARY ACADEMY
MATH SCI CTR OF EXCELLENCE
DEPT OF MATHEMATICAL SCI
MDN A MAJ DON ENGEN
THAYER HALL
WEST POINT NY 10996-1786

1 DIRECTOR
US ARMY RESEARCH LAB
AMSRL CS AL TP
2800 POWDER MILL RD
ADELPHI MD 20783-1145

NO. OF
COPIES ORGANIZATION

1 DIRECTOR
US ARMY RESEARCH LAB
AMSRL CS AL TA
2800 POWDER MILL RD
ADELPHI MD 20783-1145

3 DIRECTOR
US ARMY RESEARCH LAB
AMSRL CI LL
2800 POWDER MILL RD
ADELPHI MD 20783-1145

ABERDEEN PROVING GROUND

2 DIR USARL
AMSRL CI LP (305)

NO. OF COPIES	ORGANIZATION
	ABERDEEN PROVING GROUND
12	DIR, USARL ATTN: AMSRL-WM-M, D. VIECHNICKI AMSRL-WM-MD, W. ROY D. SHIRES (10 CP)

INTENTIONALLY LEFT BLANK.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1997	Final, Nov 96 - Jan 97

4. TITLE AND SUBTITLE
Effects of Loop Unrolling and Loop Fusion on Register Pressure and Code Performance

5. FUNDING NUMBERS
78M841

6. AUTHOR(S)
Dale Shires

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: AMSRL-WM-MD
Aberdeen Proving Ground, MD 21005-5069

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-1386

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

10.SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

Many of today's high-performance computer processors are super-scalar. They can dispatch multiple instructions per cycle and, hence, provide what is commonly referred to as instruction-level parallelism. This super-scalar capability, combined with software pipelining, can increase processor throughput dramatically. Achieving maximum throughput, however, is nontrivial. Compilers must engage in aggressive optimization techniques, such as loop unrolling, speculative code motion, etc., to structure code to take full advantage of the underlying computer architecture. The phase-ordering implications of these optimizations are not well understood and are the subject of continuing research. Of particular interest are optimizations that enhance instruction-level parallelism. Two of these are loop unrolling and loop fusion. These are source-level optimizations that can be performed by either the programmer or the compiler. These optimizations have dramatic effects on the compiler's instruction scheduler. Performed too aggressively, these optimizations can increase register pressure and result in costly memory references. This paper details experiments performed to measure the effects of these source-level code transformations and how they influenced register pressure and code performance.

14. SUBJECT TERMS	15. NUMBER OF PAGES
loop transformations, compiler optimization, register allocation, scheduling	23
	16. PRICE CODE

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

INTENTIONALLY LEFT BLANK.

USER EVALUATION SHEET/CHANGE OF ADDRESS

This Laboratory undertakes a continuing effort to improve the quality of the reports it publishes. Your comments/answers to the items/questions below will aid us in our efforts.

1. ARL Report Number/Author ARL-TR-1386 (Shires) Date of Report June 1997

2. Date Report Received _____

3. Does this report satisfy a need? (Comment on purpose, related project, or other area of interest for which the report will be used.) _____

4. Specifically, how is the report being used? (Information source, design data, procedure, source of ideas, etc.) _____

5. Has the information in this report led to any quantitative savings as far as man-hours or dollars saved, operating costs avoided, or efficiencies achieved, etc? If so, please elaborate. _____

6. General Comments. What do you think should be changed to improve future reports? (Indicate changes to organization, technical content, format, etc.) _____

CURRENT ADDRESS

_____
Organization

_____
Name E-mail Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

7. If indicating a Change of Address or Address Correction, please provide the Current or Correct address above and the Old or Incorrect address below.

OLD ADDRESS

_____
Organization

_____
Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

(Remove this sheet, fold as indicated, tape closed, and mail.)
(DO NOT STAPLE)

DEPARTMENT OF THE ARMY

OFFICIAL BUSINESS

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO 0001,APG,MD

POSTAGE WILL BE PAID BY ADDRESSEE

DIRECTOR
US ARMY RESEARCH LABORATORY
ATTN AMSRL WM MD
ABERDEEN PROVING GROUND MD 21005-5066