# Aggressive Loop Unrolling in a Retargetable, Optimizing Compiler

JACK W. DAVIDSON and SANJAY JINTURKAR

{jwd,sj3e}@virginia.edu

Department of Computer Science, Thornton Hall

University of Virginia,

Charlottesville, VA 22903 U. S. A.

## Abstract

*A well-known code transformation for improving the run-time performance of a program is loop unrolling. The most obvious benefit of unrolling a loop is that the transformed loop usually requires fewer instruction executions than the original loop. The reduction in instruction executions comes from two sources: the number of branch instructions executed is reduced, and the control variable is modified fewer times. In addition, for architectures with features designed to exploit instruction-level parallelism, loop unrolling can expose greater levels of instruction-level parallelism. Loop unrolling is an effective code transformation often improving the execution performance of programs that spend much of their execution time in loops by 10 to 30 percent. Possibly because of the effectiveness of a simple application of loop unrolling, it has not been studied as extensively as other code improvements such as register allocation or common subexpression elimination. The result is that many compilers employ simplistic loop unrolling algorithms that miss many opportunities for improving run-time performance. This paper describes how aggressive loop unrolling is done in a retargetable optimizing compiler. Using a set of 32 benchmark programs, the effectiveness of this more aggressive approach to loop unrolling is evaluated. The results show that aggressive loop unrolling can yield additional performance increase of 10 to 20 percent over the simple, naive approaches employed by many production compilers.*

**Keywords: Loop unrolling, Compiler optimizations, Code improving transformations, Loop transformations.**

## 1 Introduction

A well known programming rule of thumb is that programs spend roughly 90% of their time executing in 10% of the code [Henn90]. Any code transformation which can reduce the time spent in these small, critical portions is likely to have a measurable, observable impact on the overall execution time of the program. This critical 10% of the code frequently consists of loops. Therefore, code improvement techniques that speed up the execution of loops are important. One such technique is *loop unrolling*. Loop unrolling replicates the original loop body multiple times and adjusts the loop termination code. The primary effect is a reduction in the total number of instructions executed by the CPU when the loop is executed. The reduction in instructions executed comes from two sources: the number of branch instructions executed is reduced, and the number of increments of the control variable is reduced. In addition, loop unrolling, in conjunction with other code optimizations, can increase instruction-level parallelism and improve memory hierarchy locality [Alex93, Baco94, Davi94, Davi95a, Mahl92].

When implementing this transformation in a production compiler, three questions regarding the way to do loop unrolling most effectively arise: when should it be done, how should it be done, and what kinds of code bodies should it be applied to? Although loop unrolling is a well-known code improvement technique [Dong79, Weiss87], there has not been a thorough study that provides definitive answers to these questions.

Possibly because of the lack of definitive answers, many production compilers use simplistic approaches when applying loop unrolling. We examined how loop unrolling was performed by various compilers on seven current architectures. The architectures were the DECstation 5000/125 (MIPS R3000 chipset), SGI RealityEngine (MIPS R4000 chipset) [Kane89], the DEC 3000 (Alpha 21064 chipset) [Digi92], the IBM model 540 (IBM RS6000 chipset) [IBM90], the SUN SPARCStation IPC (LSISIc0010 chip) [Sun87] and the SUN SPARCServer (Sun SuperSparc chipset). For each platform, we examined how the native C compiler and the GNU C compiler performed loop unrolling.

The native compilers on the R4000 (ELF 32-bit executable version 1), R2000 (version 2.1), SuperSparc (ELF 32-bit executable version 1) and the Alpha (3.11-6) unroll `for` loops which have a single basic block and do not have any function calls. These compilers do not unroll loops with complex control flow. They do not unroll loops formed using `while` and `goto` statements also. Unfortunately, as this study shows, loops with complex control flow form a sizable percentage of the loops. Consequently these compilers forgo many opportunities for producing better code. The native compiler on the RS6000 (version 3.1) does unroll loops with complex control flow. However, after unrolling it fails to eliminate redundant loop branch instructions from an unrolled `while` loop. Furthermore, the compiler does not unroll loops formed using `goto` statements. The native compiler on the SUN IPC (Version 1.143) does not unroll loops. The GNU C compiler (versions 2.2.2, 2.4.5, 2.6.3) has the same limitations as the native compiler on the RS6000. Furthermore, it does not eliminate redundant loop branch instructions from an unrolled counting `for` loop with a negative stride. The above survey of current technology indicates that the approach of existing optimizing compilers to loop unrolling is not uniform.

The lack of a thorough study of unrolling and the uneven application of unrolling in production compilers motivated us to thoroughly analyze loop unrolling and examine the issues involved. This paper presents the results of a thorough compile- and run-time analysis of loop unrolling on a set of 32 benchmarks programs. The results of the analysis show that loop unrolling algorithms that only handle loops which consist of a single basic block and whose iteration count can be determined only at compile time miss many opportunities for creating more efficient loops. Using the benchmark programs, we analyzed the effectiveness of aggressive loop unrolling on run-time performance. Our measurements show that aggressive loop unrolling can yield performance increases of 10 to 20 percent for some sets of benchmarks over the simple, naive approaches employed by many production compilers, and that for some programs increases in performance by as much as 40 to 50 percent are achieved.

## 2 Terminology

This section defines the frequently used terms in this paper.

*Loop branch*: The loop code instruction(s) that check the loop control variable and decide if control should exit the loop. The number of instructions comprising the loop branch can vary. The basic block containing these instructions is called the *loop branch block.*

*Loop body*: The loop code minus the loop branch instruction(s). A loop body may contain several basic blocks.

*Counting loop*: A loop whose iteration count can be determined either at compile time or at execution time prior to the entry into the loop code.

*Compile-time counting loop*: A counting loop whose iteration count is trivially known at compile time.

*Execution-time counting loop*: A counting loop whose iteration count is not trivially known at compile time.

*Unrolled loop:* A loop whose loop body consists of multiple copies of the loop body of a rolled loop. A loop unrolled $n$ times consists of $(n + 1)$ copies of the loop body of a rolled loop. The unroll factor is $n$.

*Prologue(Epilogue) code*: If the iteration count of a rolled loop is not an integral multiple of the *unroll factor* + *1*, then as many as *unroll factor* iterations are executed separately. These iterations are called leftover iterations. The code which executes these iterations is called the *Prologue (Epilogue)* code.

*Candidates for unrolling*: All innermost counting loops are candidates for unrolling.

## 3 How and When to Unroll

An optimizing compiler is likely to apply loop unrolling in conjunction with other code optimizations. The question is *how* and *when* should loop unrolling be applied?

Automatic unrolling can be done on source code, early on the unoptimized intermediate representation, or very late on an optimized, low-level representation of the program. If it is done at the source-code level, then typically only counting loops formed using `for` statements are unrolled. Unrolling loops formed using other control constructs is difficult since the loop count is not obvious. If automatic unrolling is applied at the intermediate-code level, then a sophisticated system to perform loop analysis is required to identify anything beyond counting loops containing more than one basic block. Introducing such a system at this level is a wasteful duplication of effort, because recent research has shown that loop optimizations are more beneficial if they are done in the compiler back end [Beni94]. Additionally, performing unrolling early has a greater impact on compilation time because more code must be processed by subsequent phases of the compiler.

Another important question concerning the implementation of loop unrolling is how many times a loop should be unrolled. Most of the benefits from unrolling are due to the elimination of branches. If loops are unrolled 15 times, then 93.75% of the branches are eliminated. Therefore, an unroll factor of 15 is sufficient to extract most of the benefits. Increasing the unroll factor further yields marginal improvement in performance. However, unconstrained unrolling can have an adverse impact on the

performance if the transformed loop overflows the instruction cache [Dong79, Weis87]. The performance degradation depends on the size, organization, and replacement policy of the cache. To make sure that the unrolled loop does not overflow the instruction cache, it is necessary for the compiler to determine the size of the unrolled loop code in terms of machine-language instructions.

Another important issue concerns register allocation and assignment. If loop unrolling is done prior to register allocation and assignment, the register allocator may overcommit the registers to the unrolled code. Consequently, there may not be enough registers available to apply other useful optimizations such as strength reduction and induction variable elimination to the code. This may lead to degradation in performance, instead of improvement.

The above issues are addressed if unrolling is applied to a low-level representation of the program late in the optimization process after other traditional code optimizations have been done. With this approach, not all phases of the optimizer need to be reapplied to the larger unrolled loop bodies which reduces the increase in compilation time. Furthermore, back ends of optimizing compilers contain sophisticated loop detection mechanisms. These mechanisms can easily detect both structured and unstructured loops. Also, at this stage the size of the loop code is closer to its final size. This along with the use of a low-level representation (i.e., machine code) allows the most appropriate unroll factor to be determined. Also, since unrolling is applied after register allocation has been done, the register pressure will not increase. Any artificial dependencies introduced by this approach can be eliminated by applying register renaming [Davi95b].

# 4 What to Unroll

An analysis of loops in 32 benchmarks was performed at compile time to determine the complexity and size of loop bodies and the nature of their loop bounds. These benchmarks are a mix of Unix utilities, user codes, synthetic benchmarks, numerical benchmarks and the C portion of the SPEC benchmark [SPEC89] suite. The benchmarks are listed in Table 1.

The compile-time study consists of two parts. The first part classifies loops on the basis of whether they are compile-time counting loops or execution-time counting loops. The second part of the study classifies loops on the basis of the complexity of the loop body. These parts of the study give an indication of the sophistication required of the unrolling mechanism in the compiler. For each study, the percentages given are a percentage of the loops in that benchmark that are candidates for unrolling.

## 4.1 Loop bounds analysis

Our experience is that the iteration count of the majority of loops is difficult, and sometimes impossible to determine at compile time. An iteration count often cannot be determined at compile time because the loop bounds are passed as arguments to the function containing the loop. While interprocedural analysis provides some help, loop bounds are often based on problem size which are supplied as external inputs to the program. In these cases, the iteration count cannot be determined at compile time.

| Type | Program | Description | Type | Program | Description |
|---|---|---|---|---|---|
| SYNTHETIC | arraymerge | Merges two sorted arrays | UNIX UTILITIES | banner | Draws a banner |
| | bubblesort | Sorting algorithm | | cal | Prints out a calender |
| | puzzle | Test recursion | | cb | C beautifier |
| | queens | Eight queens problem | | compact | Compresses text files |
| | quicksort | Sorting algorithm | | diff | Prints differences |
| | shellsort | Sorting algorithm | | grep | Searches for a string |
| | sieve | Sieve of eratosthenes | | nroff | Document formatter |
| USER | cache | Cache simulation | | od | Prints octal dump |
| | encode | Encodes vpo's files | | sort | Sorting utility |
| | sa-tsp | Trav. salesman problem | | wc | Word count |
| NUMERICAL | ll3 | Livermore kernel 3 | SPEC | eqntott | PLA optimizer |
| | ll4 | Livermore kernel 4 | | xlisp | LISP interpreter |
| | ll5 | Livermore kernel 5 | | espresso | Boolean expr. translation |
| | ll6 | Livermore kernel 16 | | gcc | Optimizing compiler |
| | linpack | Floating-point benchmark | | | |
| | s006 | Kernel by Kuck and assoc. | | | |
| | s008 | Kernel by Kuck and assoc. | | | |
| | s011 | Kernel by Kuck and assoc. | | | |

**Table 1:** Description of benchmarks

| Type | Candidates for unrolling (percentage) | Execution-time counting loops (percentage) |
|---|---|---|
| User | 46 | 89 |
| Unix | 9 | 35 |
| Synthetic | 54 | 69 |
| Numerical | 79 | 73 |
| SPEC | 15 | 84 |

**Table 2:** Distribution of loops based on loop bounds

To determine how important it is for a loop unrolling algorithm to handle execution-time counting loops, we measured the percentage of loops that are execution-time counting loops. Table 2 contains the results. Column 2 is the average percentage of loops which are candidates for unrolling in each benchmark in the five categories. As expected, numerical benchmarks have a high percentage of loops which can be unrolled. On the other hand, Unix utilities have a low percentage of loops which can be

unrolled. On the other hand, Unix utilities have a low percentage of loops which can be unrolled. Column 3 contains the percentage of candidate loops which are execution-time counting loops. Thus, in user codes, on an average 46 percent of loops in each benchmark are candidates for unrolling and 89 percent of these candidates are execution-time counting loops. These statistics clearly indicate that algorithms that only handle compile-time counting loops miss many opportunities for producing more efficient code.

## 4.2 Control-flow complexity analysis

For the analysis of the control-flow complexity of loops, we developed a scheme for classifying the innermost counting loops based on the complexity of their loop bodies. The classification scheme has six categories and is cumulative in nature†. Figure 1 shows this classification. The first category contains loops which have a single basic

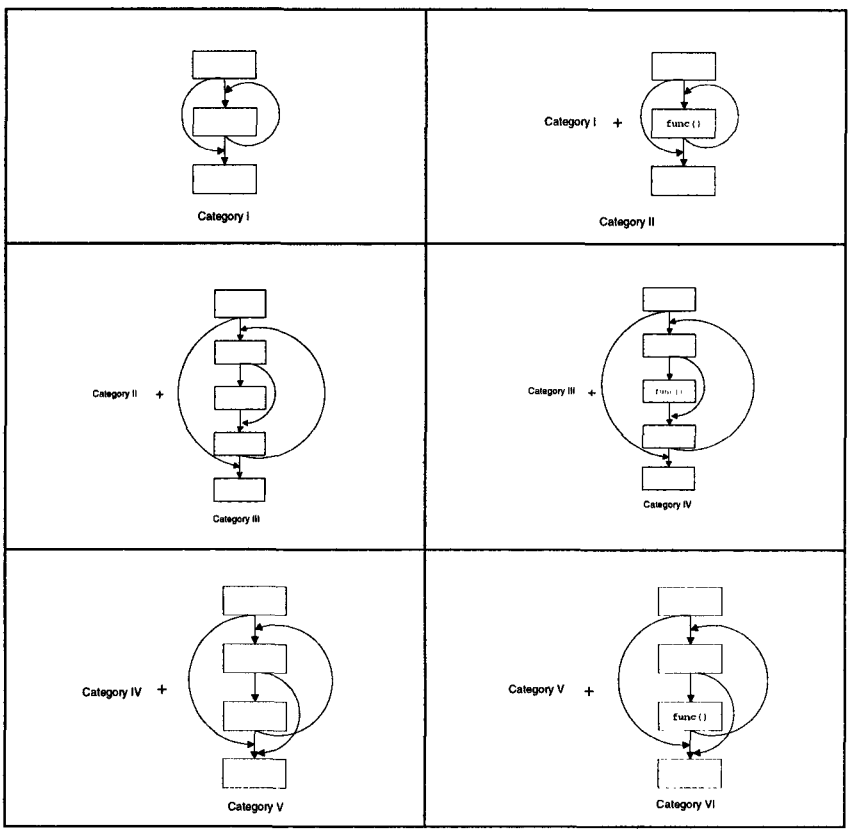


**Fig: 1.** Categories of loops.

block and no function calls. The second category contains loops which have a single

†That is, category *n* contains all the loops in category *n* - 1.

basic block and function calls. The third category includes loops which have internal branches† but no function calls. The fourth category permits function calls also. The fifth category permits loops which have multiple exits, but have no function calls. The sixth category allows function calls also.

Table 3 shows the statistics related to this classification. Column 1 contains the type of benchmark. Columns 2 indicates the average percentage of loops in each benchmark in a benchmark category which are candidates for unrolling. Columns 3 through 8 show the average distribution of loops which can be unrolled in the various categories. All measurements are percentages.

From this table, it is apparent that benchmarks in all the categories have a sizable percentage of loops which have more than one basic block. This indicates that if loops with only a single basic block are unrolled, a high percentage of loops will not be considered. Consequently, unrolling loops consisting of a single basic block only limits the effectiveness of loop unrolling.

| Type | Candidates for unrolling (Percentage) | Category (Percentages) | | | | | |
|---|---|---|---|---|---|---|---|
| | | I | II | III | IV | V | VI |
| User | 46 | 39 | 67 | 78 | 78 | 100 | 100 |
| Unix | 9 | 38 | 52 | 53 | 72 | 98 | 100 |
| Synthetic | 54 | 42 | 74 | 83 | 98 | 100 | 100 |
| Numerical | 79 | 75 | 76 | 94 | 100 | 100 | 100 |
| SPEC | 15 | 29 | 37 | 52 | 66 | 89 | 100 |

**Table 3:** Distribution of loops based on control-flow complexity.

# 5 Results

To measure the impact of aggressive unrolling, we implemented an aggressive loop unroller in *vpo*, a highly optimizing back end that has been used to implement a variety of imperative languages such as Ada, C, PL/I, and Pascal. *vpo* has two characteristics that make it an ideal framework for implementing and evaluating aggressive loop unrolling algorithms. First, *vpo* performs all code improvements on a single, low-level representation called RTLs (register transfer lists) [Beni94, Davi81]. Within *vpo*'s framework, loop unrolling can be applied late in the compilation process after many other code improving transformations have been applied and detailed information about loops has been gathered by *vpo*'s analysis phase. In addition, the late application of loop unrolling and the low-level representation allows *vpo* to accurately estimate the size of loops and choose the largest unroll factor that will not cause the loop to exceed the size of the machine's instruction cache. It also means that not all phases of *vpo* need to process the larger loop bodies which minimizes the increase in compilation time. Second, *vpo*'s internal program representation and organization permits optimizations

†The targets of all conditional and unconditional branches lie inside the loop.

to be reapplied as needed. Because of this, the implementation of loop unrolling is simplified as we can rely on these phases to remove any inefficiencies introduced by unrolling.

Figure 2 contains a diagram of the organization of *vpo*. Vertical columns

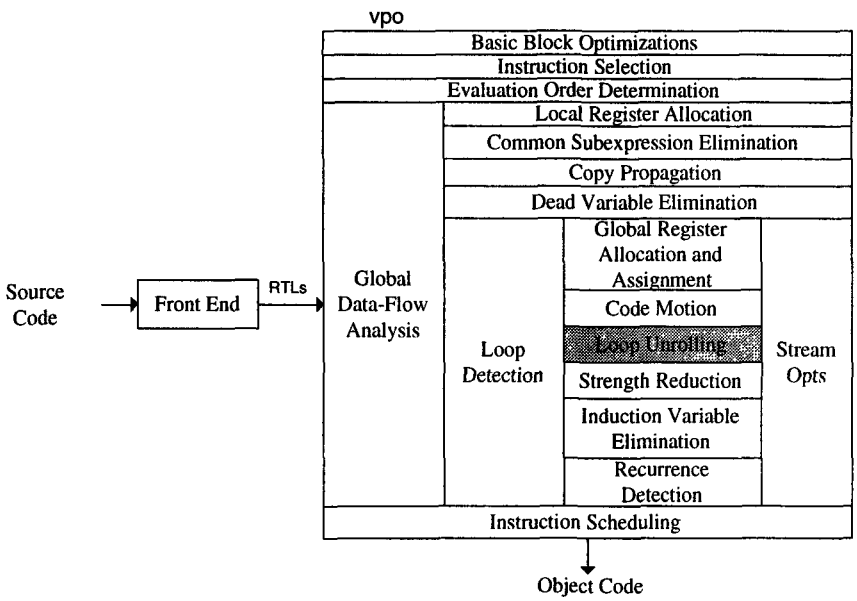


**Fig: 2.** Schematic of *vpo*-based C compiler.

represent logical phases which operate serially. Columns that are divided horizontally into rows indicate that the sub-phases of the column may be executed in an arbitrary order. For example, instruction selection may be performed at any time during the optimization process. Global data-flow analysis, on the other hand, is done after the basic block optimizations, instruction selection, and evaluation order determination, but before local register assignment, common subexpression elimination, etc.

To determine the impact of aggressive loop unrolling, we measured the increase in performance due to unrolling. We measured both execution cycles/dynamic instruction counts and actual CPU time. While measures of execution cycles/dynamic instruction counts are useful in understanding the effects of code improvements, they do not take into account other system effects such as memory traffic, cache performance, and pipeline stalls which can effect overall execution time. Furthermore, most users are concerned with how much faster their program runs when a code improvement is applied. Consequently, we felt it important to collect both types of measurements.

We gathered our measurements on a R3000-based DECstation Model 5000/125 and Motorola 68020 based Sun-3/200 [Moto84]. These two architectures were chosen because they represent two ends of the computer architecture spectrum: the DECstation

is a RISC architecture while the Sun-3 is a CISC architecture. The measurements for the performance increase in execution cycles were taken only on DECstation, while the measurements of the performance increase in CPU times have been taken on the DECstation as well as the Sun-3.

As described in Section 3, our loop unrolling algorithm automatically determines the best unroll factor $n$, where $0 \leq n \leq 15$, to use for each loop. For the DECstation 5000, all loops were unrolled fifteen times. On the Sun-3, however, the small instruction cache size (64 words) limited the amount of unrolling possible.

## 5.1 Performance increase

### 5.1.1 Reduction in cycle count

Measurements of execution cycles of all benchmarks except *xlisp*, *espresso* and *gcc* were taken using *pixie*†, an architecture evaluation tool for MIPS processors [Kane89]. The unit of measurement is cycles. Performance of benchmarks *gcc*, *espresso* and *xlisp* were measured using *ease* [Davi90], a tool to evaluate architectures. One of the measures provided by *ease* is dynamic instruction counts. It works at the assembly language level, and therefore, does not count `no-ops`.††

To determine whether handling execution-time counting loops is important, we measured the performance increase of the benchmarks when only compile-time loops were unrolled and when both compile-time and execution-time loops were unrolled. Figure 3 contains the graph showing the percentage increase using each approach. The graph shows that unrolling algorithms that only handle compile-time counting loops are much less effective than algorithms that also handle execution-time counting loops.

A second set of measurements was performed to determine the benefits of handling loops with complex control flow. Using the categories described in Figure 1, we measured the percentage increase in performance for each set of benchmarks as loops with increasingly complex control-flow are handled. Figure 4 shows the percentage performance increase for each set of benchmarks and average performance increase across benchmark categories. We measured performance increase for each benchmark set because we were interested to see if the percentage increase depended on the type of benchmark.

In user codes, the benchmarks *encode* and *cache* slow down due to unrolling. This is because the unrolled loops in the two benchmarks are not executed enough number of times to amortize the cost of loop unrolling. *Encode* has an execution-time counting loop whose iteration count is less than sixteen. Since the unroll factor used on the DECstation is 15, only the epilogue code is executed, and the unrolled portion of the code is not executed at all. The overhead incurred in computing the iteration count of the unrolled loop slows down the benchmark. If the unroll factor had been lower, then the unrolled loop would have been executed, yielding benefits. Thus, increasing the unroll factor can have a negative impact on the performance.The benchmark *cache* slows down when loops with a single basic block and function calls are unrolled because the unrolled loop is not executed enough times to amortize the cost of

†The code breaks if compiled with *pixie*.

††On R2000, scheduling is done by the assembler, and therefore, no-ops are inserted by it.

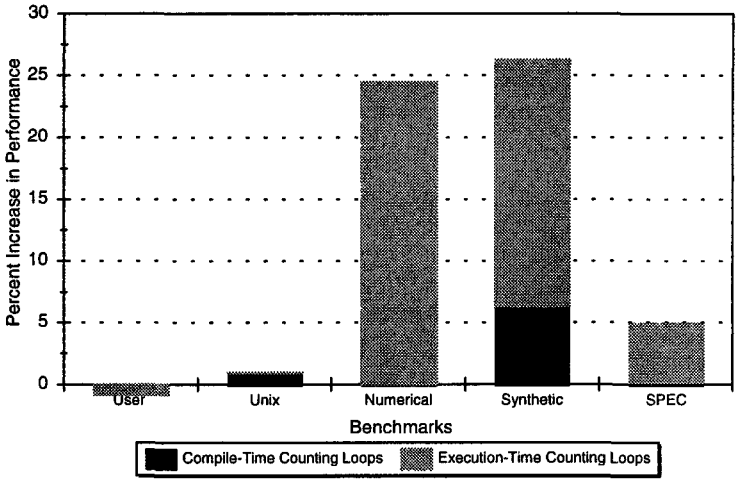


**Fig: 3.** Performance increase due to unrolling compile-time counting loops and execution-time counting loops on MIPS R2000.

calculating the loop iteration count and the extra conditional branch. In Unix utilities, the performance of *cal*, *diff* and *nroff* improves because of unrolling. The performance of other utilities does not improve because they spend a majority of their execution time in non-counting loops.

For numerical benchmarks and synthetic benchmarks, substantial performance increase occurs from unrolling loops with internal branches and multiple exits. For instance, the performance of bubblesort improves by approximately 50% when loops with complex control-flow are unrolled. Similarly, the performance of the benchmark *s008* increases by over 50% when loops with internal branches are unrolled. If loops with complex control-flow are not unrolled, these benefits would not be attained.

For SPEC benchmarks, the performance of benchmark *eqntott* improves by approximately 19% when loops with multiple exits and internal branches are unrolled. This improvement would not have been obtained if only the loops with a single basic block were to be unrolled. The performance of benchmark *espresso* is marginally better if loops with multiple basic blocks are not unrolled. This is because this benchmark contains a number of execution-time counting loops with multiple basic blocks which have an iteration count of zero or one and the overhead incurred to execute the unrolled loop is not amortized. Benchmark *xlisp* has no improvement since the execution of this benchmarks is dominated by non-counting loops, while *gcc* improves by about 0.7 percent.

The combined result shows that while unrolling loops with a single basic block is very beneficial, unrolling loops with complex control-flow is even more beneficial. The performance of numerical and synthetic benchmarks increases by an additional

12% and 17% respectively, when loops with internal branches are unrolled. The performance of SPEC benchmarks increases by about 3% when loops with multiple exits are unrolled. Clearly, unrolling loops with complex control-flow increases performance for various benchmarks.

### 5.1.2 Reduction in execution times

This section presents the increase in performance computed using execution times on the DECstation and the 68020-based Sun-3. To measure execution time, the Unix

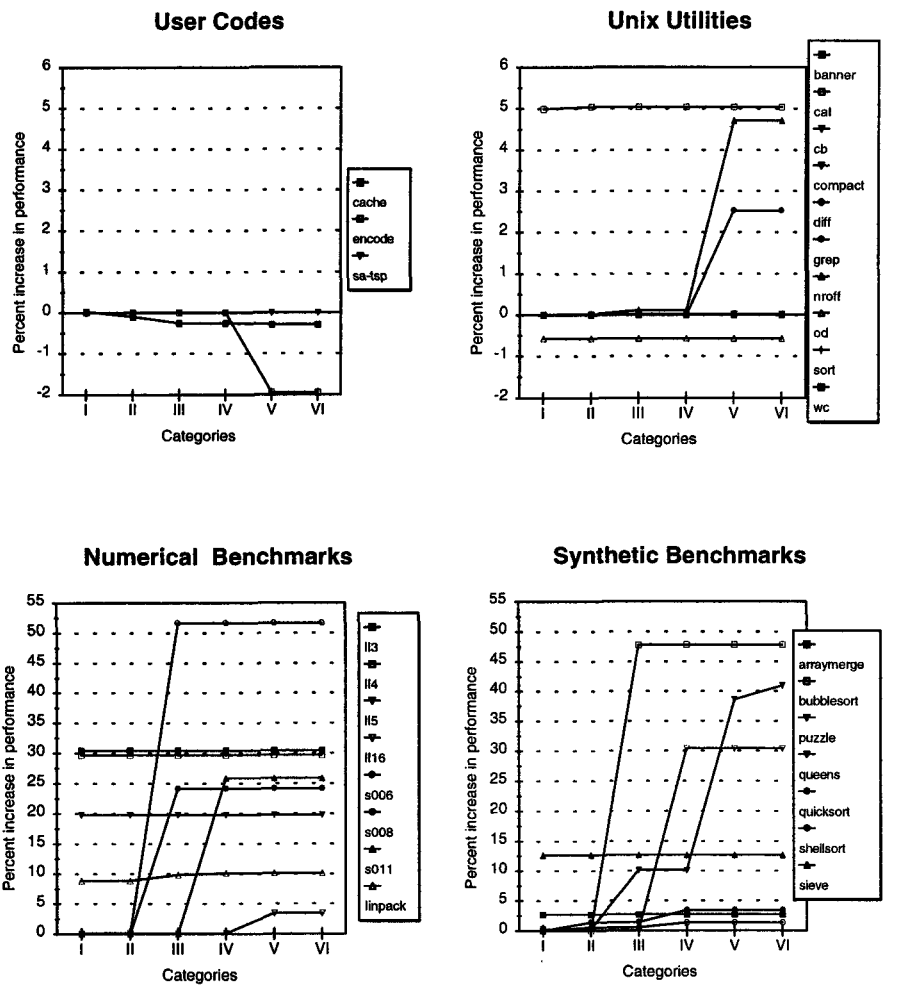


**Fig: 4.** Percentage increase in performance for benchmarks in all categories by loop body complexity on MIPS R2000.

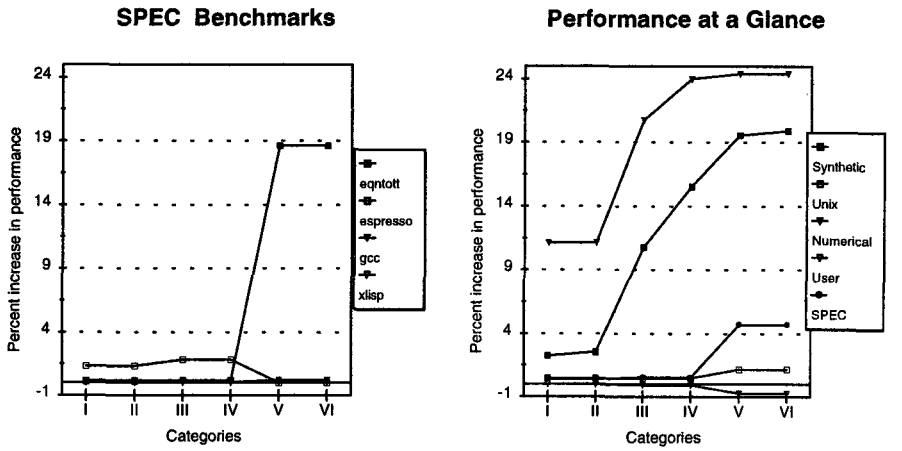


**Fig: 4.** Percentage increase in performance for benchmarks in all categories by loop body complexity on MIPS R2000.

command */bin/time* was used, and the *user* portion of the execution time was reported for each benchmark when loops are rolled and unrolled. Each benchmark was executed on a lightly loaded machine five times, the highest and the lowest measurements were dropped, and the average of the remaining three measurements was computed. The average of performance increase in each category is reported in this paper. Detailed results are available in a technical report [Davi95b].

Table 3 contains the measurements of the percentage increase in execution time. Column 2 gives the average percent increase in performance for a benchmark in each benchmark category on the Sun-3. Column 3 contains the percent increase in performance on DECstation. These measurements indicate that loop unrolling increases

| Type | Performance increase (%) on M68020 (User Time) | Performance increase (%) on DEC R2000 (User Time) |
|---|---|---|
| User | -0.74 | -0.44 |
| Unix | 0.90 | 0.47 |
| Synthetic | 15.75 | 17.42 |
| Numerical | 2.61 | 21.44 |
| SPEC | 0.07 | 3.47 |

**Table 4:** Execution-time improvement from unrolling loops on 68020 and MIPS R2000.

performance of benchmarks on the DECstation and the Sun-3. The performance increase is comparable for non-numerical benchmarks on both the machines. For

numerical benchmarks, which perform floating-point computations, the benefits are higher on the DECstation. Loop unrolling eliminates conditional branch instructions and redundant increments to the induction variable. As a side effect, the unrolled loop has an extra counter which is required by the loop which executes the leftover iterations. The combined cost of these instructions is significant when compared to the total cost of floating-point instructions inside the loops in numerical benchmarks on the DECstation. On the other hand, the cost of conditional branches and associated instructions required to maintain the loop is not significant when compared to the floating-point instructions inside the loops on the Sun-3†. Therefore, the benefits from the elimination of branch instructions and redundant increments is higher on the DECstation. Loop unrolling will result in larger performance increase on the DECstation (a RISC architecture) if register renaming is applied along with it because the instruction pipeline will be better utilized [Davi95b]. In general, the benefits from loop unrolling, to a large extent, are contingent on the cost of branch instructions to other instructions inside the loop body.

From the data presented in the above sections, it is clear loop unrolling can be a very effective code improvement. Furthermore, to be most effective, loop unrolling algorithms must handle loops with complex control flow and loops whose the iteration count is not known at compile time. For some programs, performance improvements as high as 20 to 50 percent can be achieved when loops are unrolled aggressively. Also, loop unrolling does not result in excessive increase in the size of executable code [Davi95b].

Thus, loop unrolling is similar to many other code improvements, which affects only a subset of the programs to which it is applied. It is most beneficial when it is applied aggressively to unroll execution-time counting loops and loops with complex control-flow.

# 6 Previous Work

Many researchers have presented loop unrolling as a way of decreasing loop overhead. Dongarra suggested manual replication of the code body for loops written in FORTRAN [Dong79]. Array subscripts and loop increments are adjusted to reflect that the loop has been unrolled. Weiss discussed loop unrolling from the perspective of automatic scheduling by the compiler [Weis87]. His study considers only Livermore loops. This study also discussed the effect of loop unrolling on instruction buffer size and register pressure within the loop.

Mahlke discussed optimizations which can increase instruction-level parallelism for supercomputers [Mahl92]. Loop unrolling is one of them. By analyzing loops with known bounds, they showed that if register renaming is applied after loop unrolling, the execution time of the loop decreases. In trace-scheduling and global compaction methodology [Fish83, Freu94], loop unrolling is a key feature. Freudenberger discussed the effect of loop unrolling on SPEC benchmarks and the way in which it facilitates global scheduling and insertion of the compensation code [Freu94].

†The relative cost of a floating-point instruction, when compared to a conditional branch instruction, is higher on Sun-3 than on the DECstation.

# 7 Summary

While loop unrolling is a well-known code improvement, there has been little discussion in the literature of the issues that must be addressed to perform loop unrolling most effectively. This paper addresses this deficiency. Through extensive compile- and run-time analyses of a set of 32 benchmark programs the paper analyzes the loop characteristics that are important when considering loop unrolling. One factor analyzed was the importance of handling loops where the loop bounds are not known at compile time. The analysis shows that most loops that are candidates for unrolling have bounds that are not known at compile time (i.e., execution-time counting loops). Consequently, an effective loop unrolling algorithm must handle execution-time counting loops. Another factor analyzed was the control-flow complexity of loops that are candidates for unrolling. The analysis shows that unrolling loops with complex control-flow is as important as unrolling execution-time counting loops. For some benchmark programs significant improvements can be gained if loops with complex control flow are unrolled. Because handling such loops does not significantly impact compilation time or unduly complicate the loop unrolling algorithms, our conclusion is that an aggressive compiler should unroll such loops.

Using the benchmark programs and a C compiler that implements the algorithms for loop unrolling, the effectiveness of the code transformation at improving run-time efficiency was measured. Our measurements show that aggressive loop unrolling can yield run-time performance increases of 10 to 20 percent for some sets of benchmarks over a simple and naive approach, and that for some programs increases in performance by as much as 40 to 50 percent are achieved.

# Acknowledgements

This work was supported in part by National Science Foundation grants CCR-9214904 and MIP-9307626. We also thank Mark Bailey and Bruce Childers for their feedback.

# References

[Alex93] Alexander, M. J., Bailey, M. W., Childers, B. R., Davidson, J. W., and Jinturkar, S., "Memory Bandwidth Optimizations for Wide-Bus Machines", *Proceedings of the 25th Hawaii International Conference on System Sciences*, Maui, HA, January 1993, pp. 466-475.

[Baco94] Bacon, D. F., Graham, S. L., and Sharp, O. J., "Compiler Transformations for High-Performance Computing", *ACM Computing Surveys*, **26**(4), Dec. 1994, pp. 345-420.

[Beni94] Benitez, M. E. and Davidson, J. W., "The Advantages of Machine-Dependent Global Optimizations", *Proceedings of the Conference on Programming Languages and System Architecture*, Springer Verlag Lecture Notes in Computer Science, Zurich, Switzerland, March 1994, pp. 105-124.

[Davi81] Davidson, J. W., and Fraser, C. W., "The Design and Application of a Retargetable Peephole Optimizer", *ACM Transactions on Programming Languages and Systems*, **2**(2), April 1980, pp. 191-202.

[Davi90] Davidson, J. W. and Whalley, D. B., "Ease: An Environment for Architecture Study and Experimentation", *Proceedings of the 1990 ACM Sig-*

*metrics Conference on Measurement and Modelling of Computer Systems*, Boulder, CO, May 1990, pp. 259-260.

[Davi94] Davidson, J. W. and Jinturkar, S., "Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses", *Proceedings of SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994, pp 186-195.

[Davi95a] Davidson, J. W. and Jinturkar, S., "An Aggressive approach to Loop Unrolling", available as University of Virginia Technical Report # CS-95-26.

[Davi95b] Davidson, J. W. and Jinturkar, S., "Improving Instruction-level Parallelism by Loop Unrolling and Dynamic Memory Disambiguation", *Proceedings of the 28th International Symposium on Microarchitecture,* Ann Arbor, MI, Nov 1995, pp 125-134.

[Digi92] *Alpha Architecture Handbook*, Digital Equipment Corporation, Boston, MA, 1992.

[Dong79] Dongarra, J.J. and Hinds, A. R., "Unrolling Loops in Fortran", *Software-Practice and Experience*, **9**(3), Mar. 1979, pp. 219-226.

[Fish84] Fisher, J. A., Ellis, J. R., Ruttenberg, J. C. and Nicolau, A., "Parallel Processing: A Smart Compiler and a Dumb Machine", *Proceedings of the SIGPLAN'84 Symposium on Compiler Construction*, Montreal, Canada, June 1984, pp. 37-47.

[Freu94] Freudenberger, S. M., Gross, T. R. and Lowney, P. G., "Avoidance and Suppression of Compensation Code in a Trace Scheduling Compiler", *ACM Transactions on Programming Languages and Systems*, **16**(4), July 1994, pp. 1156-1214.

[Henn90] Hennessy, J. L. and Patterson, D. A., *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc, San Mateo, CA, 1990.

[IBM90] IBM RISC System/6000 Technology, Austin, TX, 1990.

[Kane89] Kane, G., "MIPS RISC Architecture", Prentice-Hall, Englewood Cliffs, NJ, 1992.

[Mahl92] Mahlke, S. A., Chen, W. Y., Gyllenhaal, J. C. and Hwu, W. W., "Compiler Code Transformations for Superscalar-Based High-Performance Systems", *Proceedings of Supercomputing '92,* Portland, OR, Nov. 1992, pp. 808-817.

[Moto84] *MC68020 32-Bit Microprocessor User's Manual*, Prentice-Hall, Englewood Cliffs, N.J.

[Stal89] Stallman, R. M., *Using and Porting GNU CC*, Free Software Foundation, Cambridge, MA, 1989.

[Sun87] *The SPARC Architecture Manual*, Version 7, Sun Microsystems Corporation, Mountain View, CA, 1987.

[Weis87] Weiss, S, and Smith, J. E., "A Study of Scalar Compilation Techniques for Pipelined Supercomputers", *Proceedings of Second International Conference on Architectural Support for Programming Languages and Operating Systems*", Palo Alto, CA, Oct. 1987, pp. 105-109.