An Aggressive Approach to Loop Unrolling

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 execution performance of a program is loop unrolling. The most obvious benefit of unrolling a loop is that the transformed loop usually, but not always, 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 index 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 ten to thirty 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 the 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 performance improvements of 20 to 30 percent over the simple, naive approaches employed by many production compilers.

1 INTRODUCTION

A well known programming rule of thumb is that programs spend roughly 90% of their time executing in 10% of the code [HePa90]. 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, adjusts the loop termination code and eliminates redundant branch instructions. The primary effect is a reduction in the total number of instructions executed by the CPU when the loop is executed. In addition, loop unrolling, in conjunction with other code optimizations, can increase instruction-level parallelism and improve memory hierarchy locality [Baco94, Davi94, Mahl92].

As an introduction to loop unrolling, consider the C code fragment in Figure 1a. The code computes the dot product of two vectors. When the code is executed, the body of the loop will be executed 13 times. An unrolled version of the fragment can be obtained by replicating the body of the loop three times. The resulting code is shown in Figure 1b. Since it is known that the loop is executed 13 times, the unrolled loop in Figure 1b can be improved by eliminating all but one branch instruction and folding the four increments to the index variable into a single increment by four. This improved loop is shown in Figure 1c. Since four bodies of the loop in Figure 1a are executed for each iteration of the loop in Figure 1c, the latter can execute only 12 iterations of the former. The remaining iteration is peeled off and placed before the loop. The final code is shown in Figure 1c. When run, the code in Figure 1c executes fewer instructions than the code in

```
c = 0;
for (i = 0; i < 13; i++)
  c += a[i] * b[i];
```

Figure 1a: Original loop

```
c = 0;
for (i = 0; i < 13; i += j){
  j = 0;
  c += a[i] * b[i];
  j += 1;
  if (i >= 13) goto L1;
  c += a[i + 1] * b[i + 1];
  j += 1;
  if (i >= 13) goto L1;
  c += a[i + 2] * b[i + 2];
  j += 1;
  if (i >= 13) goto L1;
  c += a[i + 3] * b[i + 3];
  j += 1;
  }
L1:
```

Figure 1b: Unrolled loop

```
c = 0;
c += a[0] * b[0];
for (i = 1; i < 13; i += 4){
  c += a[i] * b[i];
  c += a[i + 1] * b[i + 1];
  c += a[i + 2] * b[i + 2];
  c += a[i + 3] * b[i + 3];
  }
```

Figure 1c: Unrolled loop after removal the of extra branches and folding increments.

Figure 1: Rolled and Unrolled Loop

Figure 1a.The reduction in instructions executed comes from two sources:

- the number of branch instructions executed is reduced, and
- the number of increments of the index variable is reduced.

The net effect on most machines is that the code in Figure 1c executes faster than the code in Figure 1a.

When implementing loop unrolling in a production compiler, three obvious questions are: how should loop unrolling be done, when should it be done, and what kind of code bodies should it be applied to? Although loop unrolling is a well-known code improvement technique [Dong79, Weiss87, Henn90], there has not been a thorough study of this important technique provides satisfactory answers to these questions. Most previous studies are restricted to unrolling loops which have a single basic block and whose iteration counts are easily determinable at compile time. This paper answers the above questions by presenting 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. To address these shortcomings, the paper presents new algorithms for the aggressive application of loop unrolling and measurements of the effectiveness of these algorithms at reducing the execution time. It also analyzes the interactions between loop unrolling and other key code improvements.

The following section defines the frequently used terms in this paper. Section 3 presents related work and surveys unrolling techniques employed by compilers for various contemporary architectures. Section 4 examines the ordering of loop unrolling with respect to other code optimization techniques. Section 5 presents the results of a compile-time study of the size and nature of loop bodies in the benchmark set. The algorithms for unrolling loops are presented in Section 6. Section 7 discusses the interactions between loop unrolling and other common optimizations. Section 8 gives the performance increase achieved by applying loop unrolling. All the results presented in this paper are based on implementation of the algorithms in an existing retargetable back end called *vpo* [Beni88, Beni94]. Using the C front end and *vpo*, the effectiveness of the algorithms is evaluated on computer systems based on the MIPS R2000 and Motorola 68020 processors [Kan89, Moto84].

2 TERMINOLOGY

This section defines the frequently used terms in this paper.

Iteration count: The number of times a loop is executed.

Loop code: The instructions comprising the loop.

Loop branch: The loop code instruction(s) that check the loop index variable and decide if control should exit the loop. The number of instructions comprising the loop branch can vary. For example, on the MIPS R4000, a loop branch usually consists of a single instruction if a pair of registers are compared for equality or inequality. For all other comparisons on the pair, a sequence of two instructions is required. In the latter case, the loop branch consists of two instructions. 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. The loops in Figure 2a and Figure 2b are counting loops, while the loop in Figure 2c is not.

Compile-time counting loop: A counting loop whose iteration count is trivially known at compile time. Figure 2a contains an example of a compile-time counting loop.

Execution-time counting loop: A counting loop whose iteration count is not trivially known at compile time. Figure 2b contains an example of a execution-time counting loop. Some execution-time counting loops can be transformed into compile-time counting loops through application of other code improvements such as constant folding, constant propagation, copy propagation, and function inlining. The iteration count of execution-time counting loops can often be expressed in symbolic form. For example, the iteration count of the loop in Figure 2b is `n`.

```
for (i = 0; i < 12; i++)
  a[i] = i;
```

Figure 2a: A counting loop whose iteration count is determinable at compile time

```
for (i = 0; i < n; i++)
  a[i] = i;
```

Figure 2b: A counting loop whose iteration count is determinable only at execution time

```
i = 0;
for (a = head, i= 0; a != NULL; a = a->next)
  a.num = i++;
```

Figure 2c: A non-counting loop.

Figure 2: A counting and a non-counting loop.

Rolled loop: A loop prior to unrolling.

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.

Unroll factor: Our review of previous literature indicates that there is no standard term for the amount a loop is unrolled. For instance, if the loop body of a rolled loop is replicated *n* times to get an unrolled loop, then this amount has been described as *unroll depth* [Dong79] of ($n + 1$), *unroll factor* [Lo95, Baco94] of *(n + 1), unroll amount* [Freu94] of *(n + 1),* or *unroll factor* [Scho92] of *n*. Some researchers do not use any term at all [Wall92, Mahl92].

In this paper, we use the term *unroll factor* in the manner used by Schofield. Thus, if the loop body of a rolled loop is replicated *n* times to get an unrolled loop, then the unroll factor is *n*. Such a loop will have *(n + 1)* loop bodies. This definition is more precise the alternative definitions, because those definitions imply that a rolled loop has a unroll factor of one. This is misleading because the loop has not been unrolled.

Leftover iterations: 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. If the iteration count is known at compile time, then these iterations are executed as a straight line code by the code containing the unrolled loop. Otherwise, they are executed by another loop†. These iterations are called leftover iterations. In Figure 1c, the code before the unrolled loop executes the leftover iterations.

Prologue (Epilogue) code: The code for leftover iterations can be inserted either before or after the unrolled loop. The code which executes these iterations is called the *Prologue (Epilogue)* code. The prologue code executes the first few iterations of the loop while the epilogue code executes the last few iterations. A combination of prologue and epilogue code can also be used.

Innermost loop: A loop which does not contain a loop.

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

3 PREVIOUS WORK

Many researchers discuss loop unrolling as a way of decreasing loop overhead. Dongarra suggests 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 discusses loop unrolling from the perspective of automatic scheduling by the compiler [Weis87]. His study considers only Livermore loops [McMa72]. This study also discusses the effect of loop unrolling on instruction buffer size and register pressure within the loop.

Mahlke discusses optimizations which can increase instruction-level parallelism for supercomputers [Mahl92]. Loop unrolling is one of them. By analyzing loops with known bounds, they show 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 discusses the effect of loop unrolling on SPEC benchmarks and the way in which it facilitates global scheduling and insertion of the compensation code [Freu94].

In addition to reviewing the results reported in the literature, it is worthwhile to take a look at what is done in practice by examining the extent to which existing modern compilers apply loop unrolling automatically. The techniques used by C compilers on five RISC platforms were examined. The platforms were the MIPS R2000, the MIPS R4000 [Kane89], the DEC Alpha [Digi92], the IBM RS6000 [IBM90] and the SUN SPARC [Sun87]. Five native compilers and the GNU C [Stal89] compiler on each platform were evaluated. The compiler versions used are documented in the appendix.

†This is the most common option. These iterations can also be executed as a straight line code.

The native compilers on the R4000, R2000, SPARC and the DEC Alpha 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 often forgo opportunities for producing better code. The native compiler on the RS6000 unrolls counting and non-counting `for` and `while` loops, including those with complex control flow. However, it fails to eliminate redundant loop branch instructions from an unrolled counting `while` loop. The compiler does not unroll loops formed using `goto` statements. The GNU C compiler 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 commercial optimizing compilers to loop unrolling is not uniform. Also, to our knowledge, no previous work thoroughly investigates loop unrolling. Previous studies on this subject have been done in bits and pieces and only in conjunction with other optimizations. These two factors motivated us to thoroughly analyze loop unrolling and examine the issues involved. In this study only innermost counting loops are unrolled. The unroll factor for each unrolled loop is three.

4 LOOP UNROLLING: HOW AND WHEN TO DO IT

An optimizing compiler may apply loop unrolling in conjunction with other code optimizations. The question is *how* and *when* should loop unrolling be applied? The answer to *how* determines the ease of application of loop unrolling while the answer to *when* impacts the costs incurred and benefits achieved from loop unrolling. The answer to *how* also effects the decision about *when*. Therefore, the discussion of *when* follows the discussion of *how*.

Loop unrolling can be done either manually (i.e., by hand) or automatically (i.e., by the compiler). It is best done by the compiler, but it is sometimes done manually when the compiler does not support it. When done manually, it is usually performed as a source-code-to-source-code transformation primarily because source code is easier to understand than a low-level program representation. Unrolling a loop at the source-code level involves identification of loop constructs (e.g., `for`, `while`, `do-while`, etc.), determination of the loop count to ensure that it is a counting loop, replication of the loop body and the adjustment of loop count of the unrolled loop. A prologue or epilogue code may also be inserted. Using this approach, it is difficult to unroll loops formed using a `while` and `goto` statements since the loop count is not obvious. However, for all but the simplest of loops, this approach is tedious and error prone.

The other alternative is to unroll loops automatically. Automatic unrolling can be done early on source code, late on the unoptimized intermediate representation, or very late on an optimized 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 [Beni94] has shown that loop optimizations are more beneficial if they are done in the compiler back end. If unrolling is applied at the back-end level after other loop optimizations have been applied, then the shortcomings in the above two approaches are eliminated.

Loop unrolling can have an adverse impact on the performance of the unrolled loop if the latter overflows the instruction cache [Dong79, Weis87]. The performance degradation depends on the size, organization, and replacement policy of the cache. For instance, the MIPS R2000 has a 64K direct-mapped write-through instruction cache. This cache can accommodate a maximum of 16K (16384) instructions†. If

†Each instruction on MIPS R2000 requires four bytes.

all instructions in the loop fit in the instruction cache, then no *capacity misses* occur during the execution of this loop. On the other hand, if all the instructions do not fit in the instruction cache, capacity misses will occur. In such an event, some instructions will be accessed from the main memory, which will degrade performance. To measure this performance degradation, we performed an experiment with unrolled versions of a loop. The original loop contains only two instructions†. Figure 3 shows the effect of loop unrolling on performance when unroll factor is increased. The Figure indicates that as the size of the

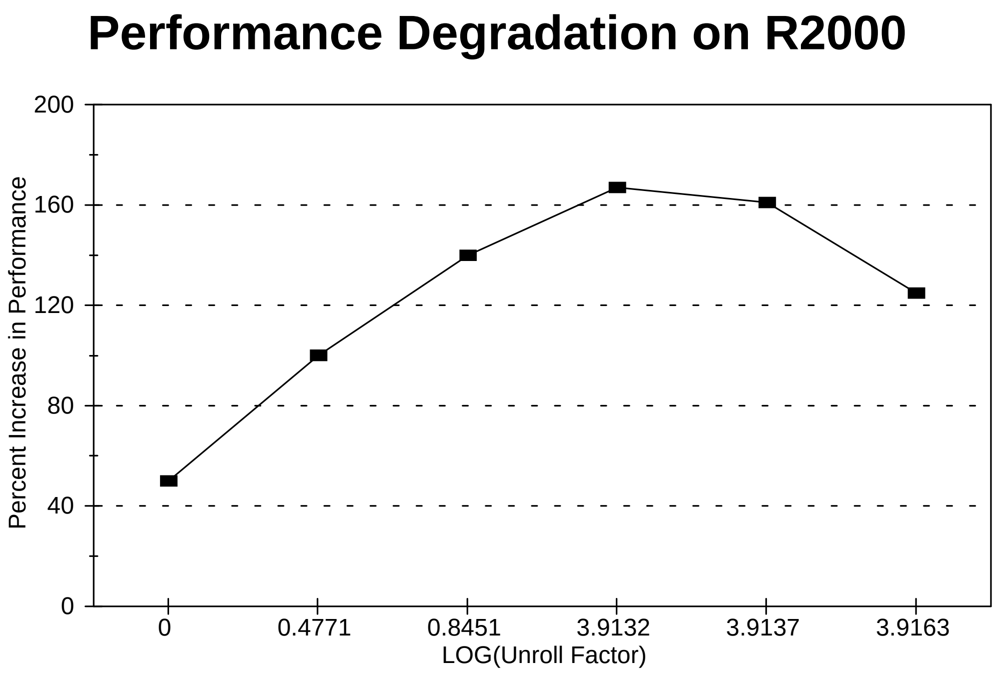


Figure 3: Degradation in performance due to overflow of the instruction cache.

unrolled loop exceeds 16384 instructions, the performance degrades. This degradation in performance will be even more pronounced if unrolling is applied in conjunction with other code replicating optimizations like function inlining [Davi88, Davi93].

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. To correctly determine the size of the unrolled loop code, the compiler needs to know the size of the rolled loop code. If unrolling is applied to either the source code or the intermediate code, then this information is either not available or is inaccurate. Some researchers use a conversion factor to estimate the number of machine-language instructions produced for a source-code line. For instance, Freudenberger [Freu94] unrolls a C loop four times and a Fortran loop eight times, thus implying that a source-code line in C will yield more assembly code line than a source-code line in Fortran. Such a factor will change from one source language to another and from one target machine to another, making retargeting of the front end as well as the back end difficult. For instance, a complex instruction set computer (CISC) is likely to require fewer machine language instructions for a piece of source code than a reduced instruction set computer (RISC). The conversion factor mentioned above fails to take the nature of the machine into consideration. Clearly, such an approach is ad-hoc in nature.

†Both the instructions complete their execution in a single cycle. To differentiate the effect of stalls due to instruction cache misses from the effect of stalls due to data cache misses and other high latency operations, we did not have any memory references or other high latency instructions in the original loop.

Another consideration in determining when and how to do loop unrolling is its impact on compilation time. Typically loop unrolling increases compilation time because more code is processed by subsequent phases of the compile. Many compilers do loop unrolling early, before applying other code improvement phases. The rationale is that organizationally it is somewhat simpler, and the full repertoire of code improvements will be applied to the unrolled code. A problem with this approach is that it is difficult to determine the most appropriate unroll factor because the size of the loop body cannot be determined until the loop has been optimized. For machines with small instruction caches, it is possible that the resulting loop could be too large. Furthermore, all subsequent code improvement phases must process the larger loop bodies thereby slowing compilation.

On the other hand, if loop unrolling is done later, after other code improvement phases are applied, it is possible to determine an unroll factor that will not cause the size of the loop to exceed the capacity of the instruction cache. In addition, we have found that performing loop unrolling later results in a lower growth in compilation time. This is because not all phases of the code improver need to be reapplied to the larger unrolled loop bodies. For example, consider the code improvement common subexpression elimination (CSE). This transformation eliminates the recomputation of values that have already been computed. If loop unrolling is applied early, the CSE phase of the compiler must process the larger loop bodies. If loop unrolling is applied late, after CSE has been done (on the smaller loop bodies), there is no need to reapply CSE. This is because loop unrolling cannot introduce new common subexpressions.

If loop unrolling is done prior to register allocation and assignment, the register allocator may overcommit the registers to the unrolled code. Due to this, it may not have 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. But if unrolling is applied after the traditional optimizations, these issues will not arise.

It is apparent that all of the above mentioned problems are addressed if unrolling is done in the back end after other traditional code improvements have been done. In the back end, a sophisticated loop detection mechanism is available. This mechanism can easily detect both structured and unstructured loops. Also, the size of the loop code at this stage is closer to its final size, and this helps in deciding the unroll factor. If unrolling is done after register allocation, the register pressure increases minimally, since only one register is required to modify the iteration count of the loop. If registers are available after all other optimizations have been applied, register renaming can be applied to the loop code to remove instruction scheduling conflicts.

5 COMPILE-TIME STUDY OF LOOPS

An analysis of loops in 32 benchmarks was performed to determine the complexity and size of loop bodies and the nature of their loop bounds. These benchmarks are a mix of UNIX utilities, synthetic benchmarks, user codes, 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 three 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. The third part of the study measures the size of loop bodies. This portion of the study gives an indication of how important the size of a loop is in determining the unroll factor. For each study, the percentages given are a percentage of the loops in that benchmark that are candidates for unrolling. A loop is a candidate for unrolling if it is an innermost counting loop.

Benchmark types	Benchmark	Description
SYNTHETIC	arraymerge	Merges two sorted arrays into a third one
	bubblesort	Sorting algorithm
	puzzle	Benchmark to test recursion and array indexing
	queens	The eight queens problem
	quicksort	Sorting algorithm
	shellsort	Sorting algorithm
	sieve	Sieve of eratosthenes
USER	cache	Cache simulation
	encode	Stores vpo's RTL files in encoded form
	sa-tsp	Travelling salesperson problem
NUMERICAL	ll3	Livermore kernel 3
	ll4	Livermore kernel 4
	ll5	Livermore kernel 5
	Ll16	Livermore kernel 16
	linpack	Floating-point benchmark
	s006	Loop kernel 006 by Kuck and associates
	s008	Loop kernel 008 by Kuck and associates
	s011	Loop kernel 011 by Kuck and associates
UNIX UTILITIES	banner	Draws a banner
	cal	Prints out a calender
	cb	C beautifier
	compact	Compresses text files
	diff	Prints out differences between two files
	grep	Searches for a string in a file
	nroff	A standard document formatter.
	od	Prints out the octal dump of a file
	sort	Sorting utility
	wc	Word count
SPEC	eqntott	PLA optimizer
	xlisp	LISP interpreter
	espresso	Boolean expression translation
	gcc	Optimizing compiler

Table 1: Description of benchmarks

5.1 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	Name	Total loops	Candidates for unrolling (percentage)	Execution-time counting loops (percentage)
SYNTHETIC	arraymerge	7	43	67
	bubblesort	5	80	100
	puzzle	50	42	19
	queens	4	100	0
	quicksort	9	44	100
	shellsort	6	50	100
	sieve	5	20	100
USER	cache	10	60	67
	encode	6	33	100
	sa-tsp	13	46	100
NUMERICAL	ll3	2	100	100
	ll4	4	50	50
	ll5	2	100	100
	ll16	3	66	50
	linpack	33	52	100
	s006	3	100	66
	s008	3	66	50
	s011	3	100	66
UNIX UTILITIES	banner	8	0	NA
	cal	9	33	0
	cb	14	0	NA
	compact	12	0	NA
	diff	67	10	100
	grep	21	9	50
	nroff	129	17	26
	od	20	10	100
	sort	54	7	75
	wc	4	0	NA
SPEC	eqntott	105	26	92
	xlisp	122	5	100
	espresso	698	12	94
	gcc	1177	18	50

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 shows the results. Column 4 is the percentage of total loops which are candidates for unrolling in the benchmark. Thus, in benchmark *arraymerge*, 43% of the seven loops are innermost counting loops. Also, 67% of the loops that can be unrolled (43% of the total number of loops) are execution-time counting loops. The results in this table show that 69% of the candidate loops in synthetic benchmarks, 89% of the candidate loops in user codes, 85% of the candidate loops in numerical benchmarks, 58% of the candidate loops in Unix utilities, and 84% of the candidate loops in SPEC benchmarks are execution-time counting loops. Thus more than 50% of candidate loops in all benchmark categories are execution-time counting loops. These statistics clearly indicate that in addition to compile-time counting loops, an optimizing compiler should also unroll execution-time counting loops.uuuu

5.2 Control-flow complexity analysis

To present our analysis of the control-flow complexity of loops in an organized manner, we developed a

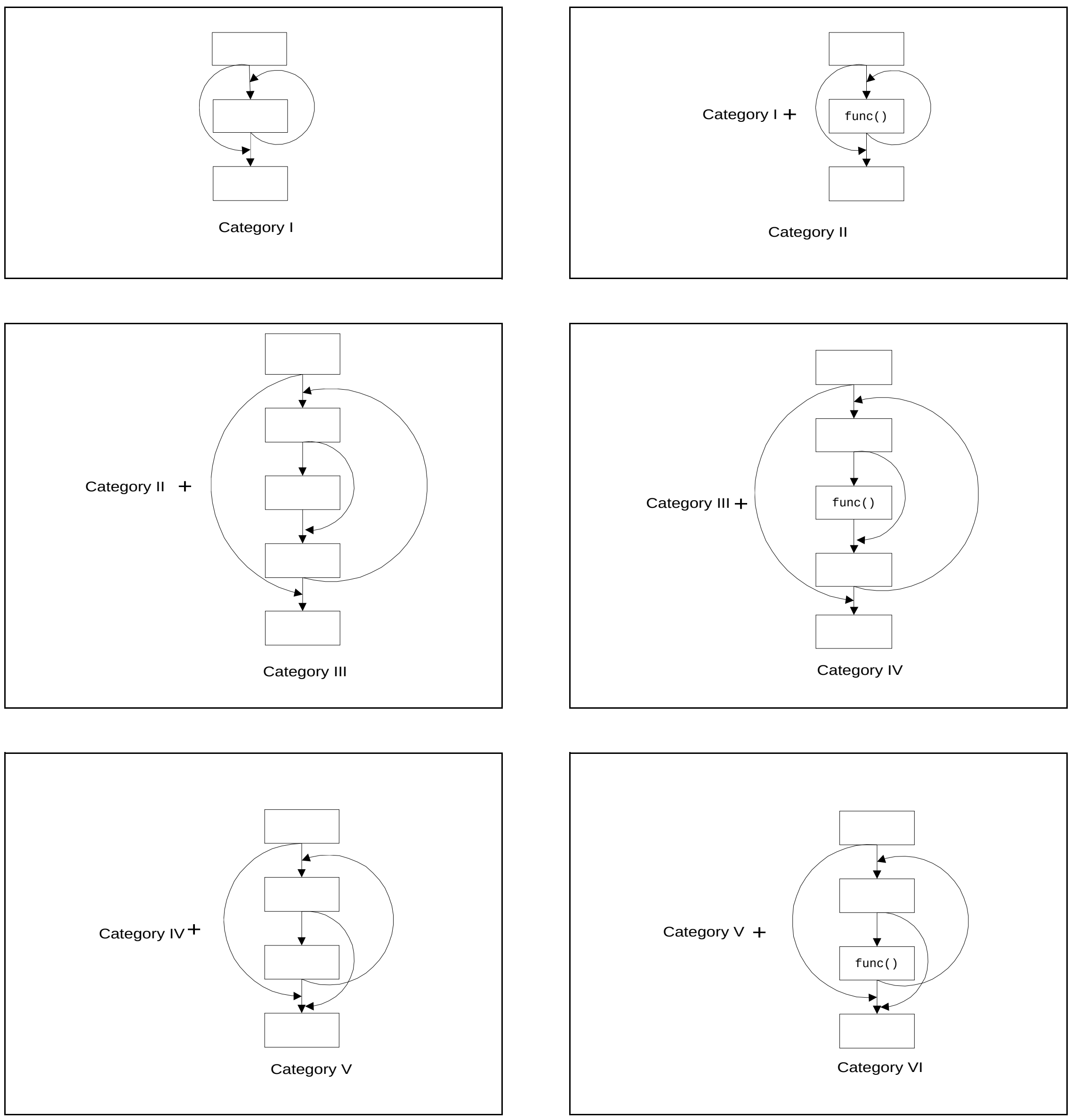


Figure 3: Categories of loops

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 3 shows this classification. The first category contains loops which have a single basic block and no function calls. The second category also contains loops which have a single basic block and function calls. The third category includes loops which have internal branches[†] but no function calls. Note that this category will allow loops which have a single basic block and 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 and column 2 contains the name of the benchmark. Column 3 contains the total number of loops in the corresponding benchmark. Columns 5 through 10 show the distribution of loops which can be unrolled in the various categories. All measurements in column 4 through 10 are percentages. Again, column 4 is the percentage of total loops in the benchmark which are candidates for unrolling.The statistics in this table support the following observations. As expected, numerical benchmarks have a uniformly high percentage of loops which can be unrolled. On the other hand, UNIX utilities have a low percentage of loops which can be unrolled. For these benchmarks, only nine percent of the loops can be unrolled. For the SPEC benchmarks, the average across the four programs is 14 percent.

Except for the numerical benchmarks, a number of benchmarks in other categories have 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 with only a single basic block limits the effectiveness of loop unrolling. Another interesting statistic is that in the synthetic benchmarks and the UNIX utilities, a high percentage of loops have internal branches and function calls. This is because a large body of code consists of function calls whose execution is contingent upon the result of conditional statements enclosing the calls.

Type	Name	Total loops	Cand. for unrolling	Category I	Category II	Category III	Category IV	Category V	Category VI
SYNTHETIC	arraymerge	7	43	67	100	100	100	100	100
	bubblesort	5	80	0	50	75	100	100	100
	puzzle	50	42	76	76	86	86	100	100
	queens	4	100	50	75	75	100	100	100
	quicksort	9	44	0	50	75	100	100	100
	shellsort	6	50	0	67	67	100	100	100
	sieve	5	20	100	100	100	100	100	100
USER	cache	10	60	17	50	83	83	100	100
	encode	6	33	50	50	50	50	100	100
	sa-tsp	13	46	50	100	100	100	100	100

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

†That is, category *n* contains all the loops in category *n* - 1.
†The targets of all conditional and unconditional branches lie inside the loop.

Type	Name	Total loops	Cand. for unrolling	Category I	Category II	Category III	Category IV	Category V	Category VI
NUMERICAL	ll3	2	100	100	100	100	100	100	100
	ll4	4	50	100	100	100	100	100	100
	ll5	2	100	100	100	100	100	100	100
	ll16	3	66	50	50	100	100	100	100
	linpack	33	52	71	77	82	100	100	100
	s006	3	100	66	66	100	100	100	100
	s008	3	66	50	50	100	100	100	100
	s011	3	100	66	66	66	100	100	100
UNIX UTILITIES	banner	8	0	NA	NA	NA	NA	NA	NA
	cal	9	33	33	100	100	100	100	100
	cb	14	0	NA	NA	NA	NA	NA	NA
	compact	12	0	NA	NA	NA	NA	NA	NA
	diff	67	10	29	29	29	43	100	100
	grep	21	9	50	50	50	50	100	100
	nroff	129	17	13	35	39	39	87	100
	od	20	10	50	50	50	100	100	100
	sort	54	7	50	50	50	100	100	100
	wc	4	0	NA	NA	NA	NA	NA	NA
SPEC	eqntott	105	26	52	56	67	74	96	100
	xlisp	122	5	0	17	17	33	67	100
	espresso	698	12	25	28	57	77	99	100
	gcc	1177	18	40	47	67	80	95	100

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

5.3 Loop size analysis

The third part of the compile-time study measured the size of the loop code. The results of this study are presented in Table 4. The size of the loop code is measured by the back end of *vpo-vpcc* [Beni89] compiler for the MIPS R2000 after all the traditional optimizations have been performed. This measurement is used to calculate the unroll factor. As with Table 2, only innermost counting loops are considered. The study reveals some interesting facts. All the loops which can be unrolled have a size less than 256 instructions. In fact, except for the loops in *linpack* and SPEC benchmarks, all others have size less than or equal to 64 instructions! This means that on a machine with cache of size 64K, most of the loops can be safely unrolled 256 times without overflowing the instruction cache. Thus, as cache sizes increase, the unroll factor can be increased without any adverse effect on the performance of the instruction cache.

6 LOOP UNROLLING: ALGORITHMS

In this section, algorithms to unroll loops are presented. These algorithms are presented in a semi-set notation.

Type	Name	Total Loops	Cand. for unrolling	Size ≤ 2	Size ≤ 4	Size ≤ 8	Size≤16	Size≤32	Size≤64	Size≤128	Size≤256
SYNTHETIC	arraymerge	7	43	0	0	25	50	100	100	100	100
	bubblesort	5	80	0	0	25	50	100	100	100	100
	puzzle	50	42	0	0	10	95	100	100	100	100
	queens	4	100	0	0	50	75	75	100	100	100
	quicksort	9	44	0	0	25	50	100	100	100	100
	shellsort	6	50	0	0	33	67	100	100	100	100
	sieve	5	20	0	0	100	100	100	100	100	100
USER	cache	10	60	0	0	67	83	100	100	100	100
	encode	6	33	0	0	100	100	100	100	100	100
	sa-tsp	13	46	0	0	33	50	100	100	100	100
NUMERICAL	ll3	2	100	0	0	0	100	100	100	100	100
	ll4	4	50	0	0	0	100	100	100	100	100
	ll5	2	100	0	0	0	100	100	100	100	100
	ll16	3	66	0	0	0	50	50	50	100	100
	linpack	33	52	0	0	12	53	71	88	94	100
	s006	3	100	0	0	33	100	100	100	100	100
	s008	3	66	0	0	50	100	100	100	100	100
	s011	3	100	0	0	33	100	100	100	100	100
UNIX UTILITIES	banner	8	0	NA	NA	NA	NA	NA	NA	NA	NA
	cal	9	33	0	33	100	100	100	100	100	100
	cb	14	0	NA	NA	NA	NA	NA	NA	NA	NA
	compact	12	0	NA	NA	NA	NA	NA	NA	NA	NA
	diff	67	10	0	0	43	86	86	100	100	100
	grep	21	9	0.00	0.00	50	50	100	100	100	100
	nroff	129	17	0	0	70	91	100	100	100	100
	od	20	10	0	0	50	100	100	100	100	100
	sort	54	7	0.00	25	50	75	100	100	100	100
	wc	4	0	NA	NA	NA	NA	NA	NA	NA	NA
SPEC	eqntott	105	26	0	11	41	74	89	96	100	100
	xlisp	122	5	0	17	50	83	100	100	100	100
	espresso	698	12	0	0	12	48	85	98	99	100
	gcc	1177	18	0	5	31	76	89	99	99	100

Table 4: Compile-time study of loops to measure size of loop bodies.

6.1: Unrolling compile-time counting loops

Compile-time counting loops consisting of a single basic block are simple to unroll. If the loop is unrolled *n* times, and the iteration count is not a multiple of the unroll factor, then, in addition to the unrolled loop, a prologue or epilogue to the loop is added which executes the first few or the last few iterations of the original loop. For instance, if the iteration count of a loop is 10 and the unroll factor is four, then two bodies of the rolled loop are inserted as prologue or epilogue. For reasons explained later, it is better to place these rolled loop bodies as epilogues to the unrolled loop.

The algorithm for unrolling such loops is presented in Figure 5. At line 3, a register needed to modify the loop count is acquired. At line 8, a new basic block is created and the loop branch instruction(s) is moved to this basic block. To insert the epilogue code, function *CopyAndInsertBlock()* is called at line 17. The epilogue code is inserted below the loop branch block. At lines 23 through 26, the loop is unrolled. Again, the function *CopyAndInsertBlock()* is called. This function also adjusts the control flow of the loop to take the replicated blocks into account. These replicated blocks are placed below the loop body block, in reverse order of their generation.

Unrolling a compile-time counting loop has no run-time overhead. If the unroll factor is *n*, then the number of branch instructions and increments to the loop count executed are reduced by *n* **.**

6.2: Unrolling execution-time counting loops

Unrolling execution-time counting loops is more complicated. For such loops, only the symbolic value of the iteration count is available at compile time. Therefore, it is not known if this iteration count is a multiple of the (*unroll factor + 1*). To unroll such loops, an additional loop to execute the leftover iterations must be

```
1   proc UnrollCompileTimeCountingLoop(Loop, UnrollFactor) is
2       // Get a register. If necessary, free up a register by spilling values not used inside the loop to memory.
3       GetRegister(Loop)
4       // Calculate LeftoverIterations.
5       LeftOverIterations ← Loop.Iterations % (UnrollFactor + 1)
6       // Find the branch instruction(s) for the loop and move it to a new basic block. This block is called
7       // BranchBlock. Other block(s) in the loop is (are) called BodyBlock(s).
8       Loop.BranchBlock ← CreateNewBranchBlock(Loop.CodeBlock)
9       // Modify the termination condition for the branch instruction of the unrolled loop, so that the
10      // iteration count of unrolled loop is a multiple of (UnrollFactor + 1).
11      ModifyLoopBranch(Loop.BranchBlock)
12      i ← LeftOverIterations
13      // Insert eplilogue code. Since the IterationCount is known at compile time, LeftOverIterations
14      // copies of the Loop.BodyBlock are inserted below the BranchBlock.
15      InsertionPoint ← Loop.BranchBlock
16      while i ≠ 0 do
17          CopyAndInsertBlock(Loop.BodyBlock, InsertionPoint)
18          i ← i – 1
19      endwhile
20      i ← UnrollFactor
21      // Insert UnrollFactor copies of Loop.BodyBlock into the rolled loop to form the unrolled loop.
22      InsertionPoint ← Loop.BodyBlock
23      while i ≠ 0 do
24          CopyAndInsertBlock(Loop.BodyBlock, InsertionPoint)
25          i ← i – 1
26      endwhile
27  endproc
```

Figure 5: Algorithm to unroll compile-time counting loops.

inserted. There are two choices: the code to execute the leftover iterations can be inserted as a prologue to the main loop or as an epilogue. For the prologue case, the inserted loop executes the first few iterations of the original loop, while in the epilogue case the inserted loop executes the last few iterations. Both the alternatives are shown in Figure 6.

```
for (i = m; i < n; i++){
  ...
  }
```

Figure 6a: Original loop

```
rem = (n - m) % unrollfactor + 1;
for (i = m; i < m + rem; i++){
  ...
  }
for (i = m + rem; i < n;
  i += unrollfactor + 1){
  ...
  }
```

Figure 6b: Leftover iterations inserted as prologue to the loop.

```
maxval = n - unrollfactor;
for (i = m; i < max_val;
     i += unrollfactor + 1){
  ...
  }
for (j = i; j < n; j++) {
  ...
  }
```

Figure 6c: Leftover iterations inserted as epilogue to the loop.

Figure 6: Two alternatives to inserting the loop which executes the leftover iterations. Here, `n` is greater than `m`.

Alternative 6b is more expensive than alternative 6c because an expensive mod or division operation is required to calculate the iteration count of the loop. This operation can be replaced by cheaper shifts if the unroll factor is a power of two, and the loop bounds are of unsigned type. But in general, a division operation is required. On the other hand, alternative 5c requires only a subtraction. This overhead can play a significant role if the unrolled loop is either nested inside another loop or is inside a function which is executed frequently. This cost can be even higher if the unrolled loop is not executed frequently enough to amortize the cost of this test. The performance of the benchmark *Cache* suffers if the leftover iterations are inserted as prologue to the unrolled loop.

Furthermore, inserting the code to execute the leftover iterations as prologue code loop skews the alignment of memory references. The benefits from memory-access coalescing [Davi94] are dependent on the alignment of the memory references. So it is important that this pattern not be skewed. Our experience is that the starting address of array references which are passed as parameters are usually aligned at a proper boundary. If the loop which executes the leftover iterations is inserted as a prologue, then the sequence of memory references which can be coalesced will not be aligned at a proper boundary, and memory access coalescing will yield less benefits.

While inserting epilogue code is necessary for many loops, it does represent additional loop overhead. The overhead comes from:

- an extra conditional branch instruction which checks if the epilogue code will be executed, and
- extra instructions to calculate the iteration count of the unrolled loop.

This overhead can affect whether the unrolled loop runs faster than the rolled loop. If the unrolled loop body is not executed sufficient number of times, then the cost of this overhead is not amortized and the performance degrades instead of improving. *Espresso* is one benchmark which suffers due to this effect.

The algorithm to unroll loops when the loop count must be determined at execution time is presented in Figure 7. At line 9, a replica of the loop code is inserted. This code forms the epilogue which executes

```
1  proc UnrollExecutionTimeCountingLoop(Loop, UnrollFactor) is
2      // Get a register. If necessary, free up a register by spilling values not used inside the loop to memory.
3      GetRegister(Loop)
4      // Make a copy of the Loop.CodeBlock and insert it below the original loop. This forms the
5      // epilogue code to execute the Leftover iterations. The iteration count of this code is modified later.
6      // Since the iteration count is available in symbolic form only, the eplilogue code is a loop and not a
7      // straightline code.
8      InsertionPoint $\leftarrow$ Loop.BranchBlock
9      CopyAndInsertBlocks(Loop.CodeBlock, InsertionPoint)
10     // Move the branch instruction(s) to a new basic block.
11     CreateNewBranchBlock(Loop.CodeBlock)
12     // Insert copies of the Loop.BodyBlock into the rolled loop to form the unrolled loop.
13     InsertionPoint $\leftarrow$ Loop.BodyBlock
14     $i \leftarrow$ UnrollFactor
15     while $i \neq 0$ do
16         CopyAndInsertBlock(Loop.BodyBlock, InsertionPoint)
17         $i \leftarrow i - 1$
18     endwhile
19     // Insert a conditional branch which determines if the control should enter the epilogue code.
20     InsertBranch(Loop.EplilogueCodeBlock)
21     // Modify the conditional branch which determines if control should enter the original loop to reflect
22     // the fact that it has been unrolled. The control enters the unrolled loop only if the iteration
23     // count of the loop is greater than the UnrollFactor.
24     ModifyBranch(Loop.CodeBlock)
25     // Insert instructions to compute the iteration count of the unrolled loop into the preheader.
26     // Also, modify the Loop.BranchBlock so that this iteration count is taken into account.
27     InsertIterationCount(Loop.CodeBlock, UnrollFactor)
28 endproc
```

Figure 7: Algorithm to unroll execution-time counting loops.

the leftover iterations. Note that at this point, there is no separate branch block to contain the branch instruction. This is because the entire loop code has to be replicated to form the epilogue. A new branch block is created at line 11. At lines 15 through 18, the loop is unrolled by inserting the copies of the loop body. At line 20, a branch is inserted to check if the control should enter the epilogue code. At line 24, the branch to check if the control should enter the unrolled loop is modified to reflect the fact that the loop has been unrolled. At lines 27, the iteration count of the unrolled loop code is modified.

The algorithm for the function *CopyAndInsertBlock()* is given in Figure 8. This procedure replicates the basic blocks and inserts them into the loop. This procedure is used when the loop code consists of a single basic block.

6.3: Unrolling loops with multiple basic blocks

A loop can have multiple basic blocks if it has internal branches or multiple exits or a combination of the two. Unrolling loops with multiple basic blocks is more complicated than the previous two cases, but is doable.

If a loop with multiple basic blocks is to be unrolled, then the routine *CopyAndInsertBlock()* has to be modified. This is because the control flow becomes complicated. The modified *CopyAndInsertBlock()* routine is presented in Figure 9. Unlike the earlier version of this routine, there are multiple basic blocks to

```
1  proc CopyAndInsertBlock(Block, InsertionPoint) is
2      // Make a duplicate of this block. Note that here, the block is a piece of straightline code. The branch
3      // condition present earlier has been stripped off and moved to a new basic block. Our branch
4      // optimizer will merge these blocks to form a single basic block. the left pointer points to the
5      // successor in the control flow graph.
6      Block.Duplicate ← CopyBasicBlock(Block)
7      Block.Duplicate.Duplicate ← Block
8      Block.Duplicate.Left ← Block.Left
9      Block.Left ← Block.Duplicate
10     // Place the basic block below the indicated insertion point.
11     PlaceBelowInsertionPoint(Block.Duplicate, InsertionPoint)
12 endproc
```

Figure 8: Algorithm to copy and insert basic blocks

replicate each time the loop is unrolled. This is done at lines 6 through 8. At line 12, the control flow between the replicated blocks is adjusted to mirror the control flow of the original loop body. At lines 16 through 17, a set of consecutive replicated basic blocks are collected for insertion into the loop data structure. The control flow between the existing blocks in the loop code and the newly replicated blocks is set up by the routine *FixControlFlow()*. If all the basic blocks in the loop code are consecutive (i.e., the static placement of the blocks), then there is only one such set of blocks. This is the most common case.

Line 24 places the first such sequence in the loop below the insertion point. If multiple such sequences are to be placed, then line 27 finds an appropriate location to place each such sequence. An appropriate such location is one which causes minimum disruption in control flow.

The routine *FixControlFlow()* is presented in Figure 10. At lines 5 through 6, the left and right pointers of the duplicate basic blocks are assigned the targets of the left and right pointers of the original basic block. At lines 9 through 10, the right pointer of the duplicate block is changed if the original block ends in a conditional jump. In such a case, the right pointer of the duplicated block points to the duplicate of the target of the right pointer of the original block. At lines 15 through 17, the left pointer of the duplicate block is changed if the original block ends in an unconditional jump and the target of this jump has been duplicated. In such a case, the left pointer of the duplicated block points to the duplicate of the target of the left pointer of the original block. If the original basic block just falls through, and the target has not been duplicated, then an unconditional jump to the bottom of the replicated basic block is added. The target of this jump is the same as the target of the left pointer of the original basic block. This is done at lines 24 through 25.

The above approach introduces some redundant unconditional jumps, which are eliminated by the branch optimizer†. When unrolling loops with multiple basic blocks, it is necessary that control flow not be altered in a way that extra branches which can not be eliminated are introduced into the unrolled loop. Otherwise, the very purpose of unrolling is forfeited. To avoid extra branches, the algorithm inserts the replicated blocks textually following the block that contains the loop branch rather than textually following some other basic block. In the latter case, extra branches will remain in order to keep the correct control flow.

†Our optimizer allows us to reapply various phases on demand.

```
proc CopyAndInsertBlock(BlockSet, InsertionPoint) is
    BlockSequenceSet ← ∅
    // Make a copy of all the blocks belonging to the BlockSet. BlockSet contains all the basic
    // blocks forming the loop body. The duplicate pointer of a block and its copy point to one another.
    // The duplicate pointers of all other blocks point to null.
    ∀ Block ∈ BlockSet do
        Block.Duplicate ← CopyBasicBlock(Block)
        Block.Duplicate.Duplicate ← Block
    endfor
    // Now set up the control flow
    ∀ Block ∈ BlockSet do
        FixControlFlow(Block.Duplicate)
        // Find a sequence of blocks which need to be inserted into the loop. More than one such
        // sequence exists if all the basic blocks forming the loop code are not textually consecutive. The
        // up pointer of a block points to its immediate textual predecessor.
        if (¬Block.Up.Duplicate)
            BlockSequenceSet ← BlockSequenceSet ∪ Block.Duplicate
        endif
    endfor
    // Place each sequence determined above, below the indicated insertion point. The insertion point
    // for the first sequence is passed to this function as an argument.
    BlockSequence ← BlockSequenceSet.First
    BlockSequenceSet ← BlockSequenceSet – BlockSequence
    PlaceBelowInsertionPoint(BlockSequence , InsertionPoint)
    // For all other sequences, find an appropriate insertion point.
    ∀ BlockSequence ∈ BlockSequenceSet do
        InsertionPoint ← FindAppropriateInsertionPoint(Loop.Code)
        PlaceBelowInsertionPoint(BlockSequence, InsertionPoint)
    endfor
endproc
```

Figure 9: Modified CopyAndInsertBlock which is used while unrolling loops with multiple basic blocks.

7 INTERACTIONS WITH OTHER OPTIMIZATIONS

Applied in isolation, loop unrolling generally yields code that runs faster. However, like most code improvements, loop unrolling interacts with other code transformations. These interactions affect the quality of the final code. The nature of these interactions (i.e., whether they result in better or worse code) helps determine when and how to apply unrolling. These interactions are discussed below.

7.1 Constant propagation

Constant propagation is the replacement of all instances of a variable that is assigned a constant with the constant itself. Constant propagation can convert an execution-time counting loop into a compile-time counting loop. As noted earlier, unrolling the latter is simpler than unrolling the former and yields more efficient code.

Figure 11 contains an example illustrating the interaction between constant propagation and loop unrolling. It contains two versions of the Livermore Kernel 3. Figure 11a is an compile-time counting loop,

```
proc FixControlFlow(Block) is
    // If a block ends with an conditional branch, then the left pointer points to the target of a “not taken”
    // branch, while the right pointer points to the target of the “taken branch”. Otherwise, the left pointer
    // points to the successor in the control flow graph. The right pointer is null.
    Block.Left ← Block.Duplicate.Left
    Block.Right ← Block.Duplicate.Right
    // If the block is terminated by a a conditional branch whose target has a duplicate, then correct the right
    // pointer. This occurs when the loop body has internal branches.
    if (Block.Duplicate.Right.Duplicate) then
        Block.Right ← Block.Duplic.Right.Duplicate
    endif
    if (Block.Status = JUMP) then
        // If the block is terminated by an unconditional branch whose target has a duplicate, then correct
        // the left pointer.This occurs when the loop body has internal branches.
        if (Block.Duplicate.Left.Duplicate) then
            Block.Left ← Block.Duplicate.Left.Duplicate
        endif
    // Otherwise, this block falls through
    else
        // If the is not duplicated then insert an unconditional jump. This occurs when the body blocks
        // of a loop are replicated or when the blocks in a loop are not textually consecutive. Unconditional
        // jumps in the first case are completely eliminated by our branch optimizer by merging the blocks,
        // while unconditional jumps in the second case are sometimes eliminated by target code replication.
        if (¬Block.Duplicate.Left.Duplicate) then
            InsertUnconditionalBranch(Block, Block.Left)
        endif
    endif
endproc
```

Figure 10: Routine to fix the control flow among the replicated basic blocks while unrolling loops with mu basic blocks.

```
for (i = 0; i < 100; i++)
  q += z[k] * x[k];
```

Figure 11a: Loop bounds known at compile time.

```
m = 100;
for (i = 0; i < m; i++)
  q += z[k] * x[k];
```

Figure 11b: Loop bounds can be known at compile time by constant propagation.

Figure 11: Livermore Kernel 3. The native compiler on R4000-based RealityEngine^2 will unroll the loop in 10a but not the loop in 10b.

while the loop in Figure 11b is an execution-time counting loop. As we noted earlier, compilers that apply simple loop unrolling algorithms early will unroll the loop in Figure 11a but not the loop in Figure 11b. For example, the native compiler on the Silicon Graphics R4000-based RealityEngine^2 unrolls the loop in Figure 11a, but does not unroll the loop in Figure 11b. This is because this compiler unrolls loops of nesting depth one only if it is a compile-time counting loop. However, if constant propagation is applied, the loop in Figure 11b can be transformed into a compile-time counting loop that can be trivially and efficiently unrolled. Apparently, this does not happen on the RealityEngine^2 native compiler because loop unrolling is done before constant propagation.

7.2 Branch optimizations

Branch optimizations are transformations that attempt to remove unnecessary branches. Loop unrolling in conjunction with constant propagation can introduce unnecessary branches. It is important that branch optimizations be performed after loop unrolling.

Consider the loops in Figure 12. The original loop is shown in Figure 12a. The outer loop is a compile-time counting loop, while the inner loop is an execution-time counting loop. Interestingly, in this example, the inner loop is unrolled by the native compiler on the RealityEngine[2]. The rationale appears to be that it is not profitable to unroll execution-time counting loops of nest depth one, but it is worthwhile to unroll execution-time counting loops that are inner loops.

Figure 12b shows the unrolled loop. At this point, constant propagation has not been applied. Consequently, the compiler produces two inner loops. One is the prologue loop described in Section 6.2, while the second is the main unrolled loop. After constant propagation, the compiler produces the final loop shown in Figure 12c. Notice, that the first inner loop will never be executed and could be eliminated. However, apparently the branch optimization of the compiler is not invoked following the other transformations and the zero-trip loop remains in the final code.

```
m = 100;
for (k = 0; k < 5; k++)
  for (j = 0; j < m; j++)
     q += z[k][j] * x[k][j];
```

Figure 12a: Original loop

```
for (k = 0; k < 5; k++) {
  rem = m % 4;
  for (j = 0; j < rem; j++)
     q += z[k][j] * x[k][j];
  for (j = rem; j < m; j += 4) {
     q += z[k][j] * x[k][j];
     q += z[k][j+1] * x[k][j+1];
     q += z[k][j+2] * x[k][j+2];
     q += z[k][j+3] * x[k][j+3];
  }
}
```

Figure 12b: Unrolled loop before constant propagation.

```
for (k = 0; k < 5; k++) {
  for (j = 0; j < 0; j++)
     q += z[k][j] * x[k][j];
  for (j = 0; j < 100; j += 4) {
     q += z[k][j] * x[k][j];
     q += z[k][j+1] * x[k][j+1];
     q += z[k][j+2] * x[k][j+2];
     q += z[k][j+3] * x[k][j+3];
  }
}
```

Figure 12c: Unrolled loop after constant propaga tion.

Figure 12: Effect of unrolling loops before and after constant propagation.

7.3 Function inlining

Inline function expansion replaces frequently invoked function calls with the function body itself. Function inlining increases the spatial locality and decreases the number of function calls. This subject has been investigated by previous researchers [Davi88, Hwu89, McFa91, Davi93, Chen93]. Inlining increases the size of a basic block, which facilitates better application of traditional optimizations like dead code elimination, constant propagation [Beni94, Davi88], and instruction scheduling. But function inlining, like loop unrolling, increases the size of the code which may have an adverse effect on the performance of the instruction cache.

Chen [Chen93] suggests that function inlining decreases the miss ratio due to increased sequentiality of instructions for caches in the range 1-4K, while it increases the miss ratio for caches in the range 8-32K due to increase in the size of the working set. It has no adverse effect on the performance of large instruction caches (64K onwards). When inlining is applied in conjunction with loop unrolling, the cache miss ratio increases but schemes such as load forwarding exploit the increased sequentiality of instructions and reduce negative effects on the cache.

Since both loop unrolling and function inlining increase the size of the code, it is necessary that the order of their application be such that maximum benefits are derived with minimum adverse effects. The dilemma is how to order these two optimizations? There are four cases to be considered. They are shown in

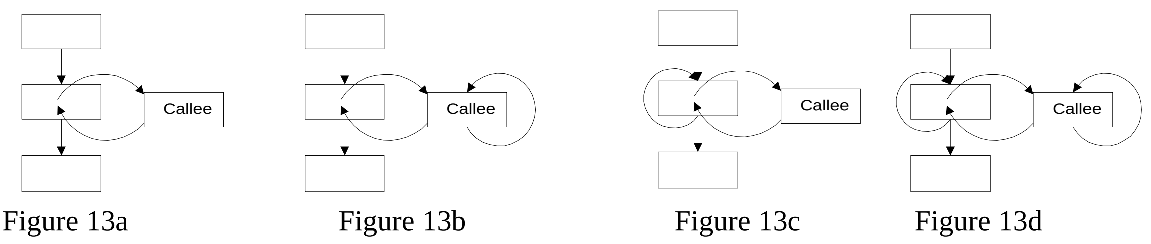


Figure 13: Four cases where inlining may have to be applied. In three of these cases, unrolling can also be done.

Figure 13. Each subfigure in Figure 13 shows a caller and a callee. In Figure 13a neither the caller nor the callee have loops. Therefore, inlining is unlikely to pay, while unrolling cannot be applied. In Figure 13b, the callee contains a loop†. Since the callee is executed only once, the benefit from inlining is unlikely to be high. Instead, unrolling can be applied. In Figure 13c, the call site is in a loop. From the results shown later, it is clear that loops which have function calls in them do not benefit much from unrolling, since a majority of the time is spent in the callee. Therefore, it is beneficial to first inline the function call and then apply unrolling. Furthermore, if loop unrolling is applied before function inlining, it may increase in the size of the loop body in the caller to such an extent that it may preclude the application of function inlining.

In Figure 13d, the call site is in a loop and the callee also contains a loop. Chen [Chen93] has shown that inlining does not have an adverse effect on the performance of large instruction caches. Taking this into consideration, the function can be inlined without any adverse effects. Inlining will convert the loop enclosing the caller from an innermost loop into an outer loop since the callee also contains a loop. Consequently, the loop previously enclosing the call site will not be unrolled. Also, if the loop bounds of the loop in the callee are passed as arguments to it, then, this loop can be converted into a loop whose iteration count can be determined at compile time by the application of constant propagation. Unrolling this loop will be cheaper. Also, some dead code may be eliminated [Beni94]. Thus, unrolling after inlining is easier and likely to yield better code.

7.4 Register Allocation

Loop unrolling also interacts with register allocation. If loop unrolling is done before register allocation, the register allocator sees a larger loop body. In an attempt to avoid artificial dependencies, the register allocator may overcommit the registers to different operands. This will increase the register pressure inside the loop [Weis87]. Increased register pressure may lead to spilling, and inability to apply other code improvements. This may actually degrade the performance, rather than improving it.

†To simplify the discussion, the callee is considered to have only one loop.

However if unrolling is done after register allocation, the replicated code is a mirror image of the original loop body. The replicated code will use the same set of registers as the original loop body. With this approach, only one additional register is required to maintain the iteration count of the unrolled loop. The remaining registers are available for use in applying other code improvements.

7.5 Instruction scheduling

Instruction scheduling attempts to reorder instructions so that the pipeline performance is improved [Henn90]. Instruction scheduling can improve the performance of an unrolled loop [Lo95]. However, if register renaming is applied after register allocation, the ability of an instruction scheduler to reorder instructions in an unrolled loop is limited by artificial dependencies created by naive reuse of registers by loop unrolling. These artificial dependencies can be eliminated by applying register renaming [Mahl92] in software while the instructions are being scheduled at the very end of the compilation process. Since the loop has already been optimized, the structure of the loop code is fixed due to which register renaming needs to be applied only once. This approach, in conjunction with dynamic-memory disambiguation, improves instruction-level parallelism on modern superscalar and VLIW processors [Davi95].

7.6 Architecture dependence

Loop unrolling is an architecture-dependent optimization. As mentioned earlier, in isolation, the major gain in performance from loop unrolling is from the reduction in loop overhead which includes the execution of a conditional branch. On modern pipelined processors, the penalty paid for executing conditional branches is high since a conditional branch disrupts the flow of instructions in the pipeline. The penalty paid to execute the conditional branch varies across architectures, primarily because their pipeline structure varies. This makes the benefit due to loop unrolling highly architecture dependent.

All our experiments have been performed on the R2000 and the M68k. This architecture has no branch prediction, and has a branch latency of two cycles. The instruction in the branch delay slot is executed whether or not the branch is taken. If this delay slot is filled with a useful computation, then the branch latency is hidden. On the other hand, if no useful computation is inserted in the delay slot, a `nop` is placed. If such a loop is unrolled n times, then as many as $2n$ instructions per iteration of the unrolled loop are eliminated.

Benefits from unrolling will be different on the RS6000 [IBM90]. This architecture uses branch prediction. A correctly predicted branch does not incur any penalty, while a mispredicted branch can incur up to a three cycle penalty. All the conditional branches on this architecture are predicted not taken. This means that most of the loop branches will be mispredicted. Unrolling a loop n times can thus reduce up to $3n$ cycles. An optimization used to reduce the number of mispredicted branches in a loop on the RS6000 inverts the loop branch, so that the common case for it is a fall through. This conditional branch is followed by an unconditional branch which transfers the control to the head of the loop [Bern92]. To avoid a stall, there should be a minimum of two non-branch instructions between these two branches, which may or may not be possible. If these instructions are not available, then unrolling such a loop can save as many as $4n$ instructions per iteration of the unrolled loop. Bernstien [Bern92] also indicates that codes with high branch intensity will suffer on the RS6000. This means that loops with internal branches and multiple exits can suffer. Therefore, such loops will benefit disproportionately from unrolling on this architecture.

On the Alpha AXP systems [DEC92], branch prediction is static and done by the compiler. All backward branches are predicted taken, and all the forward branches are predicted not taken. Calder [Cald94] indicates that even a correctly predicted taken branch will incur a two cycle penalty. Since loop branches are predicted taken, unrolling the loop n times can eliminate as much as a $2n$ cycle penalty.

8 LOOP UNROLLING: RESULTS

To determine the impact of aggressive loop unrolling, we performed measurements for the increase in code size and the percent increase in performance. For performance improvement, 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 other system effects such as memory traffic, cache performance, and pipeline stalls which can effect overall execution time. Furthermore, most users are concerned only by the amount by which their programs run faster due to the application of the code improvement technique. Consequently, we felt it important to gather both types of measurements.

We gathered our measurements on a R2000-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 increase in the code size and 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.

To extract maximum benefits from loop unrolling, we have attempted to unroll the loops aggressively. Figure 14 shows percent loop branches eliminated from counting loops as a function of the unroll factor. From the figure it is apparent that if loops are unrolled 15 times, then 93.75% of the branches are eliminated. Most of the benefits from unrolling are accrued due to the elimination of branches. Therefore, an unroll factor of 15 is sufficient to extract most of the benefits. Increasing the unroll factor further will improve benefits marginally. Our results shown later support this hypothesis. But such a big unroll factor should be used only if the unrolled loop will not overflow the instruction cache, thus causing capacity cache misses. As mentioned earlier, the cache on the DECstation is big enough to accommodate loops which have been unrolled 15 times. Therefore, we use a constant unroll factor of 15 for this machine. On the other hand, the Sun-3 has a 64 word instruction cache. To avoid capacity cache misses, we unroll a loop until the its size does not exceed 64 instructions. Since we apply unrolling the back end level, we are able to better able to estimate the size of the unrolled loop.

8.1 Increase in code size of the executable

It is important that a code optimization which intends to improve the performance of a program not increase the size of the executable code significantly. In certain environments like embedded processors, the increase in the size of the executable can offset the gains provided by unrolling. With this point of view, we have measured the increase in size of executable code of each benchmark due to loop unrolling on the DECstation. Table 5 shows the percent increase in the size of the executable code.

8.2 Performance increase

8.2.1 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

†The code breaks if compiled with *pixie*.
††On R2000, scheduling is done by the assembler, and therefore, no-ops are inserted by it.

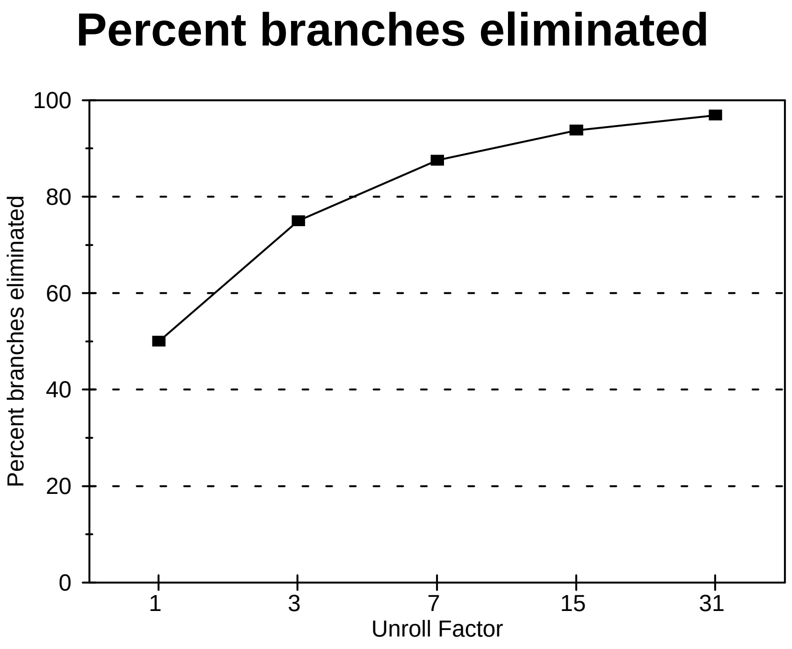


Figure 14: Performance increase due to unrolling compile-time counting loops and execution-time counting loops on MIPS R2000.

Type	Benchmark	(%) Inc. in size	Type	Benchmark	(%) Inc. in size	Type	Benchmark	(%) Inc. in size	Type	Benchmark	(%0 Inc. in size
SYNTHETIC	arraymerge	0.62	NUMERICAL	ll3	17.16	UNIX UTILITIES	banner	1.50	SPEC	eqntott	7.95
	bubblesort	5.42		ll4	19.22		cal	2.83		espresso	41.98
	puzzle	8.50		ll5	27.43		cb	0.00		gcc	19.24
	queens	8.08		ll16	148.90		compact	0.00		xlisp	0.00
	quicksort	6.46		s006	39.66		diff	3.72			
	shellsort	4.55		s008	21.59		grep	2.24			
	sieve	5.61		s011	40.73		nroff	1.70			
USER	cache	1.88		linpack	3.66		od	4.23			
	encode	1.94					sort	9.75			
	sa-tsp	3.16					wc	1.79			

Table 5: Increase in the size of executable code on MIPS R2000.

compile-time and execution-time loops were unrolled. Figure 15 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 were performed to determine the benefits of handling loops with complex control flow. Using the categories described in Figure 4, we measured the percentage increase in performance for each set of benchmarks as loops increasingly complex control flow are handled. Figures 16 through 18 show the percentage increase for each set of benchmarks. We measured performance increase

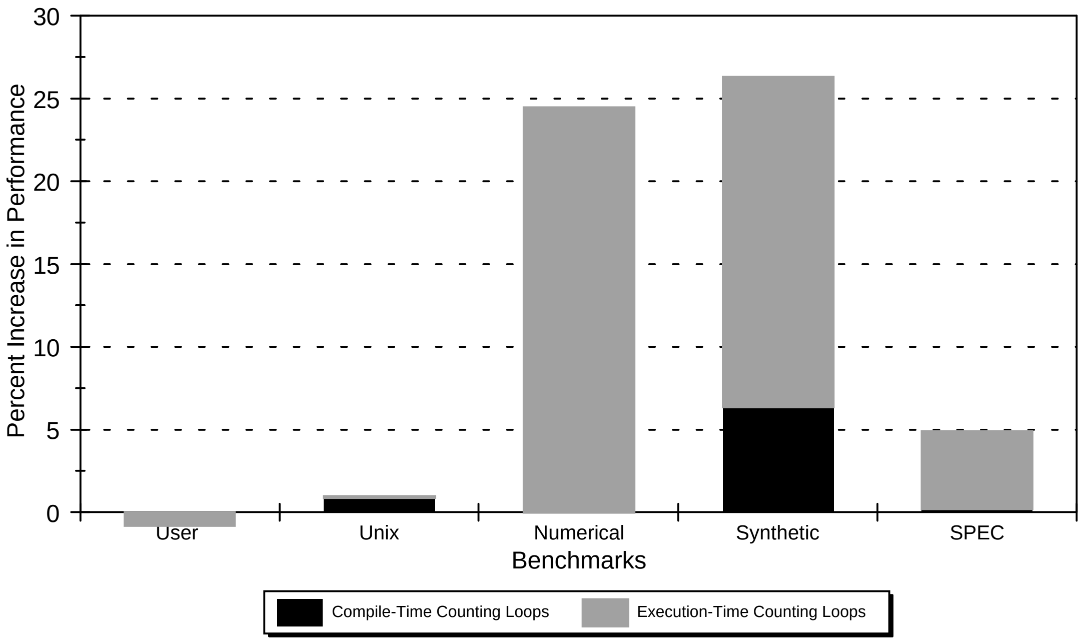


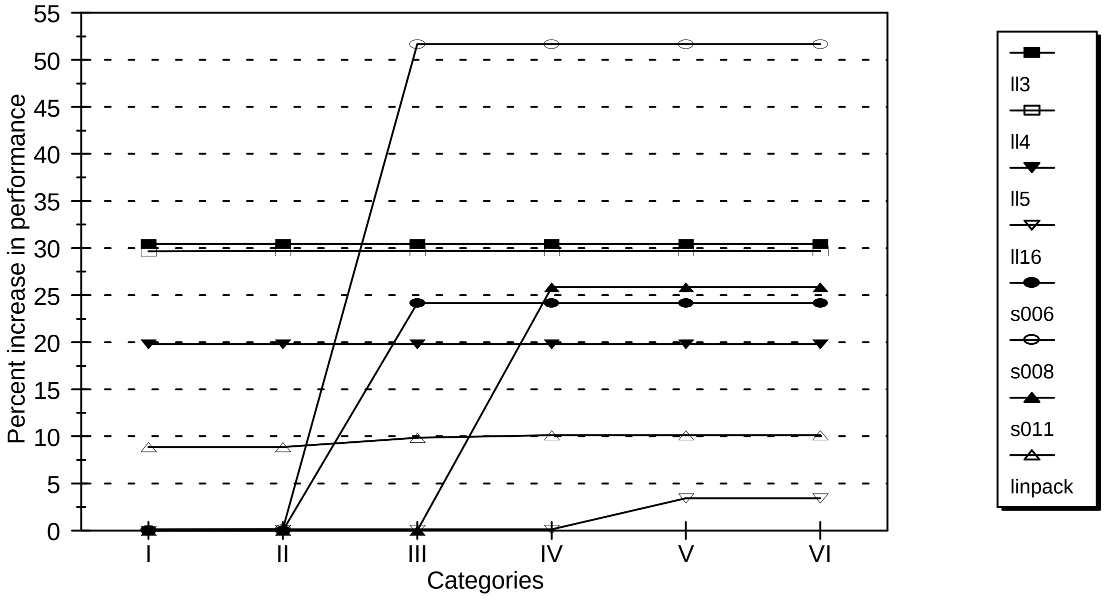
Figure 15: Performance increase due to unrolling compile-time counting loops and execution-time counting loops on MIPS R2000.

for each benchmark set because we were interested to see if the percentage increase depended on the type of benchmark. Figure 19 shows the increase in performance when all benchmarks were considered.

Figure 16 presents the performance increase for numerical and synthetic benchmarks. For numerical benchmarks, the performance increase is the best for kernel loops. For some kernel loops, the performance increase comes from category I, which indicates that it is enough to unroll loops with only a single basic block and no function calls. For some others, the improvement comes from unrolling loops which contain internal branches and multiple exits. For the synthetic benchmarks, substantial gains are made by unrolling loops which contain internal branches and multiple exits. Unlike the numerical benchmarks, unrolling loops with a single basic block only is not very beneficial for synthetic benchmarks.

Figure 17 presents the performance increase for user codes and UNIX utilities. In user codes, the benchmarks *encode* and *Cache* slow down due to 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, the unrolled loop 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 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 these utilities spend a majority of their execution time in non-counting loops. This is totally in contrast to loops in numerical codes, where a majority of the time is spent in counting loops.

Numerical Benchmarks



Synthetic Benchmarks

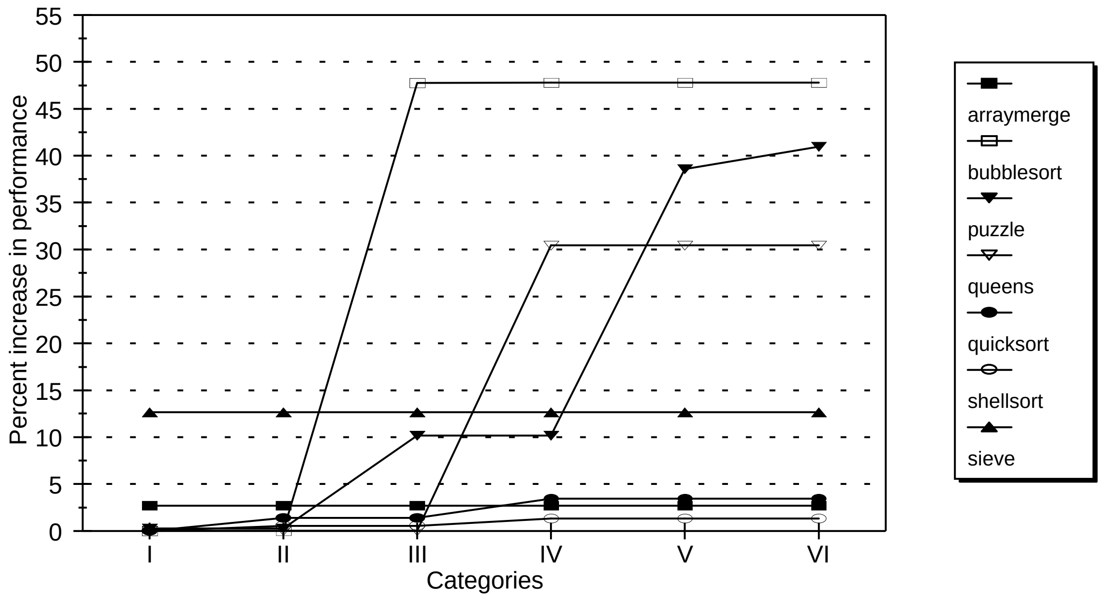
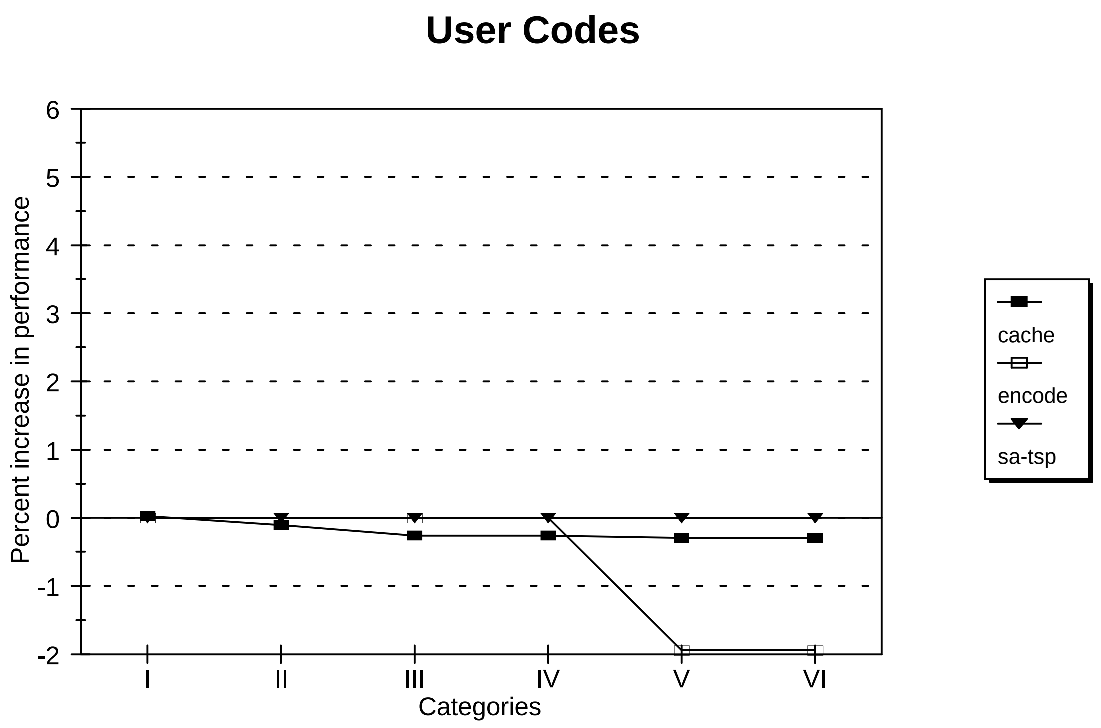


Figure 16: Percentage performance increase for Numerical and Synthetic Benchmarks by loop body complexity on MIPS R2000.



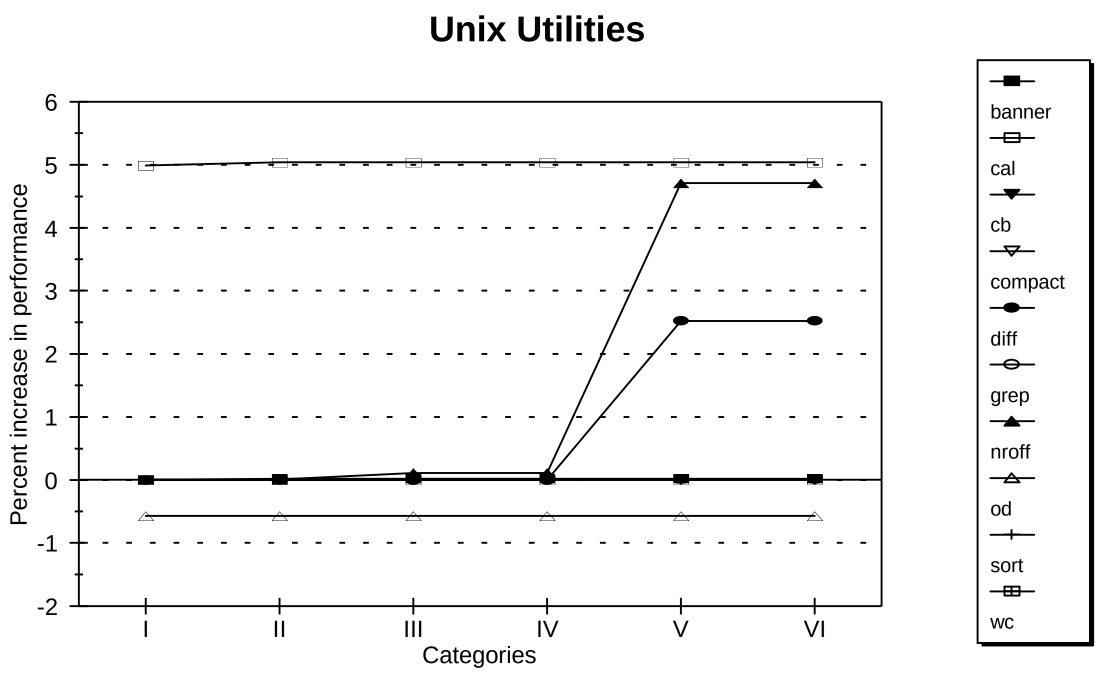


Figure 17: Percentage performance increase for User Codes and Unix Utilities by loop body complexity on MIPS R2000.

Figure 18 presents the performance increase for SPEC benchmarks. The performance of benchmark *eqntott* improves by approximately 19% when loops with multiple exits and internal branches are unrolled. This benchmark has a loop in function *cmppt()* which is executed 85% of the time, and unrolling this loop is very profitable. The performance of benchmark *espresso* is 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. These loops are unrolled. Because the iteration count is less than sixteen, the execution of the code containing unrolled loops requires the execution of an additional conditional branch for each invocation of the loop, which slows down the code. 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.

Figure 19 presents the summary of results. It is clear that unrolling loops with a single basic block and

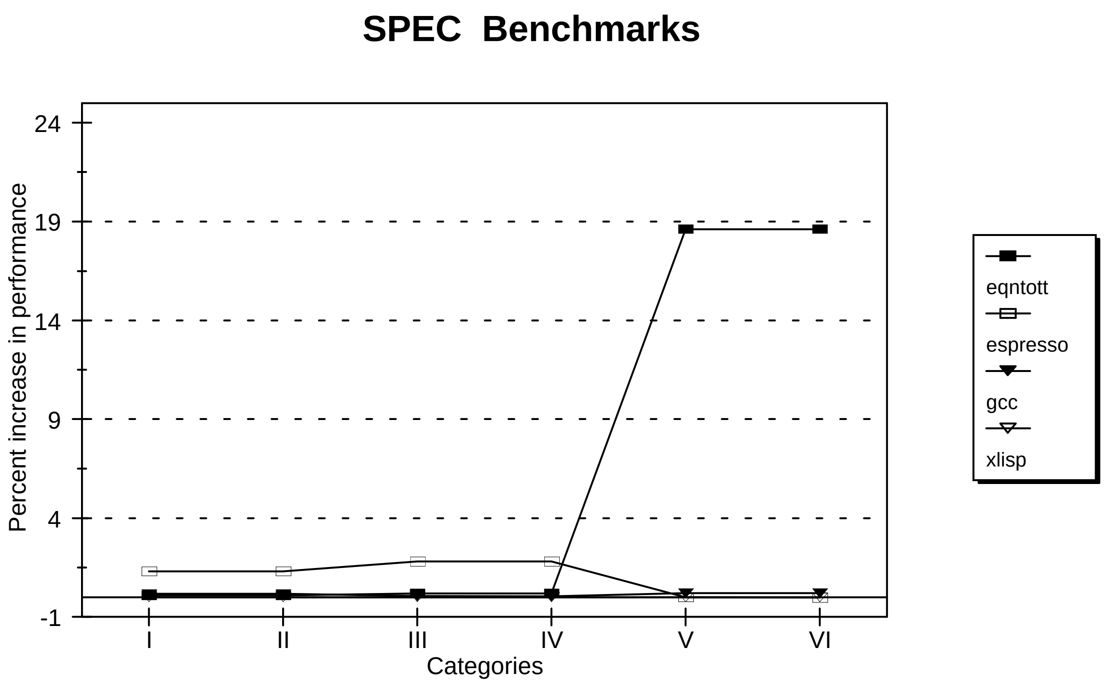


Figure 18: Percentage performance increase for SPEC benchmarks on MIPS R2000.

no function calls is very beneficial. When loops with function calls are also unrolled, there is little or no improvement in the performance. Unrolling such loops is therefore, not profitable. Unrolling loops with internal branches is profitable. Also, unrolling loops with internal branches and function calls appears to be worth the effort, unlike unrolling loops with a single basic block and function calls only, as can be seen from results of synthetic benchmarks. This intuitively makes sense, since loops with internal branches are likely to execute a function call only some of the time. Consequently, the execution cost of the function call does not dominate the execution cost of the loop. Unrolling loops with multiple exits and function calls does not yield much benefit. In this case the cost of executing the function call dominates the cost of execution of the unrolled loop.

The combined result shows that unrolling is most beneficial for kernel loops, which is not surprising. But it also makes a substantial difference in the performance of synthetic benchmarks, and the SPEC benchmark *eqntott*. The performance of all benchmarks except user codes improves when loops with multiple basic blocks are unrolled. It is important to keep in mind that except for kernel loops, all other

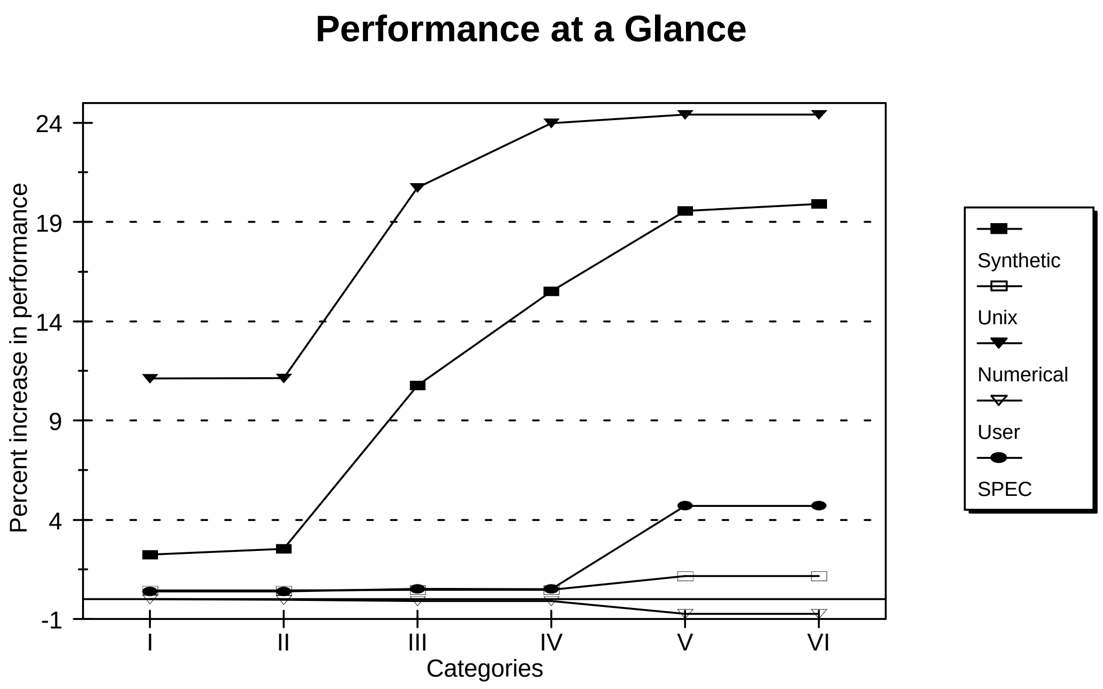


Figure 19: Average performance increase for Unix Utilities, Synthetic Benchmarks, Numerical benchmarks, User Codes and SPEC benchmarks on MIPS R2000.

benchmarks represent big pieces of code, and the performance improvements have been measured over the execution of the entire benchmark.

Figure 20 shows the average increase in performance for various benchmarks when the unroll factor is varied. This figure supports our decision that an unroll factor of 15 is sufficient to extract most of the benefits, and there is a minimal increase in performance when the unroll factor is increased beyond that. In fact, the performance of SPEC benchmark deteriorates when the loops are unrolled 31 times. The major contribution to this is from the benchmark *Eqntott*. The performance degrades because the unrolled loops are not executed enough number of times, and the overhead of calculating the iteration count of the unrolled loop and executing the loop set up code of the leftover loop dominate the execution time.

8.2.2 Reduction in execution times

This section presents the percent increase in performance in execution time on the DECstation and the 68020-based Sun-3. To measure execution time, the UNIX 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 five times, the highest and the lowest measurements were dropped, and the average of the other three measurements was reported. Although the machines primarily performed these measurements, some routine network activities and minimal interactive usage did occur. This may have caused some additional instruction cache misses.

Table 6 contains the measurements of the percentage increase in execution time. Column 3 gives the percent increase in performance on the 68020-based machine. Column 4 contains the percent increase in performance on DECstation. The results indicates that loop unrolling is beneficial on both the architectures. Benchmarks *puzzle* and *queens* perform better on the Sun-3, because our optimizer does an unsatisfactory job of moving some loop-invariant expressions outside the rolled loop in these benchmarks; but when the loop is unrolled, then the common-subexpression eliminator eliminates multiple copies of these

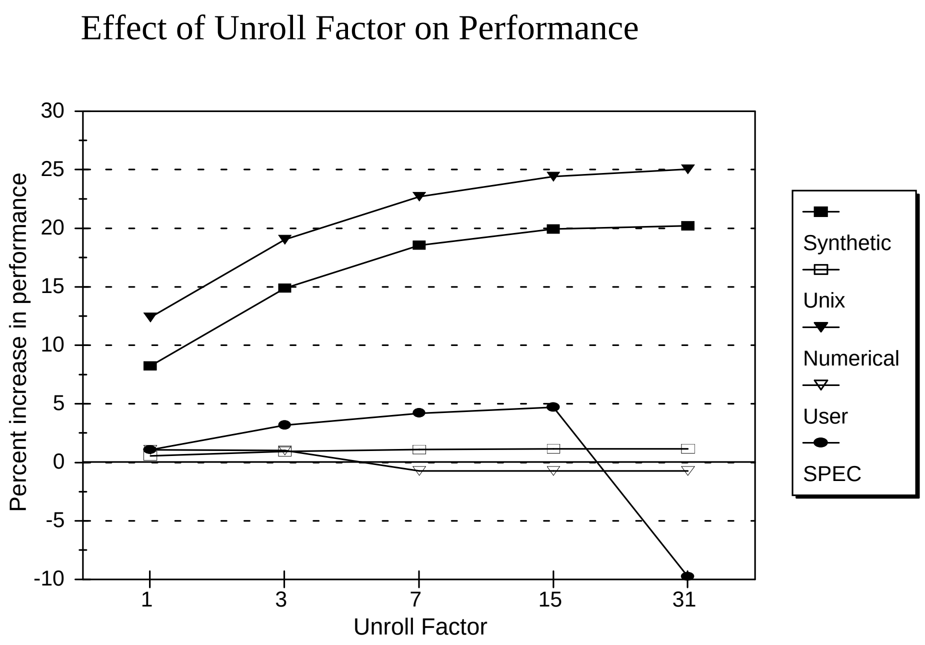


Figure 20: Percentage performance increase for all benchmarks with various unroll factors on R2000.

expressions, due to which the benefits increase. The benchmark *bubblesort* performs better on the DECstation than on the Sun-3 because the Sun-3 has fewer data registers to keep temporary values in them. In absence of sufficient data registers, the values have to be loaded from the memory, which is an expensive operation. Loop unrolling does not improve this activity. Therefore, the performance increase in limited. Performance increase in the benchmark *eqntott* on the Sun-3 is quite low when compared to the performance increase on DECstation. The Sun-3 has very few volatile registers due to which the function *cmppt()* uses more non-volatile registers. As a result, these non-volatile registers have to be stored and restored every time the function is called. The execution of the store and restore instructions is not improved by loop unrolling. Therefore, the performance increase is limited. In addition to this, this benchmark suffers from compulsory cache misses, which also effects performance. The benefits from loop unrolling are higher on the DECstation for the numerical benchmarks, which operate on double-precision floating-point data. 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.

To compare the benefits obtained in cycles and CPU time on DECstation, column 5 contains the measurements for increase in performance in cycles while Column 6 contains the difference in the percent increase in cycles and CPU times. From these measurements, it appears that the results obtained using the two metrics is mostly consistent, except in a few cases. In the benchmark *queens*, there is a noticeable difference. This benchmark is recursive in nature, and operates on global arrays. This difference can be attributed to compulsory cache misses and conflict cache misses between the memory locations used by the

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

Type	Benchmarks	Performance increase(%) on M68020 (User time)	Performance increase(%) on MIPS R2000 (User Time)	Performance increase(%) on MIPS R2000 (cycles)	Difference between performance increase in cycles and user time on MIPS R2000
SYNTHETIC	arraymerge	3.90	1.43	2.69	1.25
	bubblesort	25.24	45.82	47.81	1.99
	puzzle	62.35	39.95	40.97	1.02
	queens	8.63	20.17	30.45	10.29
	quicksort	0.34	3.30	3.45	0.14
	shellsort	0.63	1.28	1.31	0.03
	sieve	9.21	10.00	12.69	2.69
USER	cache	-3.11	0.27	-0.30	-0.56
	encode	0.73	-1.23	-1.94	-0.71
	sa-tsp	0.14	-0.36	-0.00	0.35
NUMERICAL	ll3	4.55	21.09	30.45	9.36
	ll4	0.24	29.34	29.68	0.34
	ll5	13.45	19.36	19.81	0.45
	ll16	4.11	1.48	3.45	1.97
	s006	-1.03	23.88	24.15	0.27
	s008	3.57	50.88	51.68	0.80
	s011	-1.89	25.11	25.84	0.73
	linpack	2.11	0.39	10.14	9.75
UNIX UTILITIES	banner	0.00	0.00	0.00	-2.32
	cal	5.99	2.45	5.04	2.59
	cb	0.00	0.00	0.00	0.00
	compact	0.00	0.00	0.00	0.00
	diff	0.13	2.66	2.52	-0.13
	grep	0.11	-1.63	0.00	1.63
	nroff	3.88	1.24	4.71	3.47
	od	-1.06	0.00	-0.57	-0.57
	sort	0.00	0.00	0.00	0.00
	wc	0.00	0.00	0.00	0.00
SPEC	eqntott	3.48	12.75	18.61	5.87
	espresso	-3.07	0.44	-0.02	-0.46
	gcc	-0.15	0.70	0.21	-0.49
	xlisp	0.00	0.00	0.00	0.00

Table 6: Execution-time improvement from unrolling on the Sun-3 68020 and MIPS R2000.

global arrays and the memory locations used to store and restore the non-volatile registers. A similar discrepancy exists for benchmarks *ll3* and *linpack*. The performance increase in cycles is higher than the performance increase in CPU time. We believe that the tool *pixie* underestimates the impact of interlocks in the floating-point pipeline on the DECstation. Insertion of an interlock appears to have a more negative impact than estimated by *pixie*. Similarly, the performance of the benchmark *eqntott* is worse than predicted by the measurements of instruction counts. This difference is attributed to compulsory and conflict cache misses.

These measurements indicate that loop unrolling improves 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, the benefits are higher on the DECstation, since floating point instructions are cheaper† as compared to the Sun-3 (A CISC architecture). Loop unrolling will result in larger performance increase on the DECstation (a RISC architecture) if register renaming is applied along with it, since then the instruction pipeline will be better utilized [Davi95]. 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 where 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. For most programs, the improvement is more modest. For a few programs, loop unrolling applied without the benefit of profile information can make programs run slower. Besides, it does not result in excessive increase in the size of the executable code.

Thus, loop unrolling is similar to many other code improvements. It only affects a subset of the programs to which it is applied. For some programs, the effect is small and for others it will be large. Thus, a good optimizing compiler uses a collection of code improvements where each transformation yields a modest benefit most of the time and a large benefit occasionally.

9 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. To not do so misses many opportunities for improving the run-time efficiency of the program.

In a similar vein, the control-flow complexity of loops that are candidates for unrolling was analyzed. The analysis shows that handling loops with complex control flow is not as key a factor as handling execution-time counting loops. However, 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.

Finally, a key consideration in implementing unrolling is when in the compilation process to apply the transformation. Many compilers apply the transformation early before other code improvements are applied. The rationale is that it makes implementing loop unrolling simpler (it is not unlike simple textual substitution) and that it is beneficial to apply other code improvements after loop unrolling. Our analysis indicates that it is best to apply loop unrolling later in the compilation process. There two major reasons for

†In comparison to cost of other instructions on the machine.

this conclusion. First, the impact on compilation time is minimized. Unrolling early produces larger loops that later, expensive phases of the optimizer must process. Second, applying unrolling later in the compilation process means that loops that are initially execution-time counting loops are transformed into compile-time counting loops. Better code results from unrolling compile-time counting loops.

The paper presented algorithms that address the above issues. The algorithms handle execution-time counting loops and loops with complex control flow. Furthermore, the transformations are applied after most code improvements. Using the benchmark programs and a C compiler that implements the algorithms described, the effectiveness of the algorithms at improving run-time efficiency was measured. The measurements show that aggressive loop unrolling can yield improvements in run-time performance as high as 50 percent for some benchmarks while the average improvement over all benchmarks is approximately 10 percent. We conclude that aggressively unrolling loops is clearly worth the effort.

REFERENCES

[Aho86] Aho, A., Sethi, R., and Ullman, J. D., *Compilers Principles, Techniques and Tools*, Addison-Wesley, Reading, MA, 1986.

[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*, Mauii, 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.

[Beni88] Benitez, M. E. and Davidson, J. W., “A Portable Global Optimizer and Linker”, *Proceedings of SIGPLAN ‘88 Conference on Programming Language Design and Implementation*, Atlanta, GA, June 1988, pp. 329-338.

[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.

[Cald94] Calder, B. and Grunwald, D., “Reducing Branch Costs via Branch Alignment”, *Proceedings of Sixth International conference on Architectural Support for Programming Languages and Operating Systems*”, San Jose, CA, Oct. 1994, pp. 242-251.

[Chen93] Chen, W. Y., Chung, P. P., Conte, T. M. and Hwu., W. W., “The Effect of Code Expanding Optimizations on Instruction Cache Design”, *IEEE Transactions on Computers*, 42(9), Sept. 1993, pp. 1045-1057.

[Davi88] Davidson, J. W., and Holler, A. M., “A Study of a C Function Inliner”, *Software-Practice & Experience*, 18(8), Aug. 1988, pp. 775-790.

[Davi90] Davidson, J. W. and Whalley, D. B., “Ease: An Environment for Architecture Study and Experimentation”, *Proceedings of the 1990 ACM Sigmetrics Conference on Measurement and Modelling of Computer Systems*, Boulder, CO, May 1990, pp. 259-260.

[Davi93] Davidson, J. W., and Holler, A. M., “Subprogram Inlining: A Study of its effect on program execution time”, *IEEE Transactions on Software Engineering*, 18(2), Feb. 1992, pp. 89-101.

[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.

[Davi95] Davidson, J. W. and Jinturkar, S., “Improving Instruction-level Parallelism by Loop Unrolling and Dynamic Memory Disambiguation”, *Submitted to 28th International Conference on Microarchitecture.*

[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.

[Hwu89] Hwu, W. W., and Chang, P. H., “Inline function expansion for compiling C programs”, *Proceedings of SIGPLAN ‘89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989, pp. 146-157.

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

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

[Lo95] Lo, L. J., and Eggers, S., J., “Improving Balanced Scheduling with Compiler Optimizations that increase Instruction-level Parallelism”, *Proceedings of SIGPLAN ‘89 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995, pp. 151-162.

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

[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.

[McMa72] McMohan, F. H, *The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range,* Lawrence Livermore National Laboratory, Livermore, CA, 1986.

[McFa91] McFarling, S, “Procedure Merging with Instruction Caches”, *Proceedings of SIGPLAN ‘91 Conference on Programming Language Design and Implementation*, Ontario, Canada, June 1991, pp. 71-79.

[Nico87] Nicolau, A., “Loop Quantization or Unwinding Done Right”, *First International Conference on SuperComputing*, Athens, Greece, June 1987, pp. 294-306.

[Scho89] Schofield, C.F., *Optimizing Fortran Programs*, Ellis Horwood Limited, Chichester, UK, 1989.

[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.

[Wall93] Wall, D. W., “Limits of Instruction-Level Parallelism”, *WRL Research Report,* 93/6, Digital Equipment Corporation, Palo Alto, CA, 1993.

[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.

APPENDIX:

The following table provides the version of C compiler’s used for various comparisions in this paper:

Architecture	Native C Compiler (cc)	GNU C Compiler (gcc)
MIPS R2000	2.1	2.4.5
MIPS R4000	ELF 32-bit executable version 1	2.2.2
DEC Alpha	3.11-6	2.6.3
SUN Sparc	ELF 32-bit executable version 1	2.4.5
IBM R6000	3.1	2.4.5

Table 7: The versions of C compiler used.