Avoiding Unconditional Jumps by Code Replication

Frank Mueller and David B. Whalley

Department of Computer Science, B-173
Florida State University
Tallahassee, Florida 32306-4019
e-mail: whalley@cs.fsu.edu

Abstract

This study evaluates a global optimization technique that avoids unconditional jumps by replicating code. When implemented in the back-end of an optimizing compiler, this technique can be generalized to work on almost all instances of unconditional jumps, including those generated from conditional statements and unstructured loops. The replication method is based on the idea of finding a replacement for each unconditional jump which minimizes the growth in code size. This is achieved by choosing the shortest sequence of instructions as a replacement. Measurements taken from a variety of programs showed that not only the number of executed instructions decreased, but also that the total cache work was reduced (except for small caches) despite increases in code size. Pipelined and superscalar machines may also benefit from an increase in the average basic block size.

1 Introduction

Unconditional jumps occur often in programs. Depending on the environment, execution frequencies between 4% and 10% have been reported [Pe77, Cl82]. Common programming constructs such as loops and conditional statements are translated to machine code using unconditional jumps, thus resulting in relatively compact code. Code size, however, has become less important since the introduction of caches. For instance, inlining [Da88] and loop unrolling [He90] can obtain improvements while increasing the code size.

This study describes a method of replacing unconditional jumps uniformly by replicating a sequence of instructions from the jump destination. To perform this task, an algorithm is proposed which is based on the idea of following the shortest path within the control flow when searching for a replication sequence. The effect of code replication is shown by capturing measurements from the execution of a number of programs.

The document is structured as follows. Section 2 gives an overview of research on related topics. Section 3 illustrates the advantages of using code replication for optimizing various programming constructs. Section 4 provides an informal description of the algorithms used to implement code replication. Section 5 discusses the results of the implementation by comparing the measurements of numerous programs with and without code replication. Section 6 gives an overview of future work and Section 7 summarizes the results.

2 Related Work

Several optimizations that attempt to improve code by replicating instructions have been implemented. Loop unrolling [He90] replicates the body within a loop. This reduces the number of compare and branch instructions that are executed. Also, more effective scheduling may be achieved for pipelined and multiple issue machines since some of the basic blocks comprising the loop contain more instructions after unrolling.

Inlining, an optimization method studied by Davidson and Holler [Da88], results in replicated code when a routine is inlined from more than one call site. Hwu and Chang [Hw89] used inlining techniques based on profiling data to limit the number of call site expansions and thereby avoid excessive growth. In general, a call to a non-recursive routine can be replaced by the actual code of the routine body. The procedure call can be viewed as an unconditional jump to the beginning of the body, and any return from the procedure can be viewed as an unconditional jump back to the instruction following the call.

Golumbic and Rainish [Go90] used the method of

```
i = 1;
while (i++<n)
  x[i-1] = x[i];
```

	without replication	with replication
move 1 into data reg. 1 (variable i*)*	d[1]=1;	d[0]=1;
move addr. of x[] *into addr. reg. 0*	a[0]=a[6]+x.;	d[1]=2;
L15		NZ=d[0]?L[_n];
copy data reg. 1 into data reg. 0	d[0]=d[1];	PC=NZ>=0,L16;
incr. index into x[]	a[0]=a[0]+1;	a[0]=a[6]+x.+1;
incr. data reg. 1 (variable i*)*	d[1]=d[1]+1;	L000
compare data reg. 0 with variable n	NZ=d[0]?L[_n];	B[a[0]]=B[a[0]+1];
exit loop if greater/equal	PC=NZ>=0,L16;	a[0]=a[0]+1;
move x[i] *into* x[i-1]	B[a[0]]=B[a[0]+1];	d[0]=d[1];
jump unconditionally to L15	PC=L15;	d[1]=d[1]+1;
	L16 ...	NZ=d[0]?L[_n];
		PC=NZ<0,L000;
		L16 ...

Table 1: Exit Condition in the Middle of a Loop (RTLs for 68020)

replicating parts of basic blocks for instruction scheduling to exploit potential parallelism for a super-scalar processor. In their approach, it suffices to copy the number of instructions needed to avoid a pipeline stall from the block following a conditional statement. They expand natural loops similarly by replicating instructions from the top of the loop, negating the branch condition, and inserting another unconditional jump. Their goal was to increase the number of instructions that can be issued simultaneously.

3 Motivation

All replication techniques employed in the front-end of a compiler lack generality in reducing the number of unconditional jumps. Instances of unconditional jumps cannot always be detected due to interaction with other optimizations and a lack of information about the target architecture. Consequently, front-end methods for code replication cannot eliminate occurrences of unconditional jumps which are introduced by the optimization phase of a compiler.

The optimization evaluated in this study, code replication, was accomplished by modifying the back-end of the optimizing compiler VPO (Very Portable Optimizer) [Be88]. The algorithms to perform the optimization, except for a few small functions, are machine-independent. In general, RTLs[1] are searched for unconditional jumps. By determining the jump destination and using control flow information, a subset of the basic blocks in the function can be replicated, replacing the unconditional jump. Such an optimization can be applied to almost all occurrences of unconditional jumps. The following sections give examples of instances where code replication can be applied to applications written in C.

[1] Register Transfer Lists (RTLs) represent the effects of instructions of a target machine.

3.1 Loops

For while-loops, the front-end VPCC (Very Portable C Compiler) [Da89] generates intermediate code with an unconditional jump at the end of the loop. This unconditional transfer can be replaced by the instructions testing the termination condition of the loop with the termination condition reversed.

The intermediate code produced by the front-end for for-loops with an unknown number of iterations includes an unconditional transfer of control preceding the loop to the instructions comprising the termination condition, a portion of code placed at the end of the loop. This unconditional jump can also be replaced by the code which tests for the inverse termination condition. Thus, the code checking the termination condition would appear before the loop and at the end of the loop.

Often, the replication of the termination condition of while and for loops is performed by optimizing compilers. But when the exit condition is placed in the middle of a loop, most compilers do not attempt a replacement for the unconditional jump. An example for such a situation is given in Table 1. Code replication is used to replace the unconditional jump by the RTLs between label L15 and the conditional branch. The conditional branch is reversed and a new loop header is introduced at label L000. Afterwards, other optimizations such as common subexpression elimination are applied. In this example, one unconditional jump per loop iteration is saved. The method proposed in this study handles these cases as well as unstructured loops, which are typically not recognized as loops by an optimizer.

```
if (i>5)
  i = i / n;
else
  i = i * n;
return(i);
```

	without replication	with replication
compare i *and 5*	NZ=L[a[6]+i.]?5;	NZ=L[a[6]+i.]?5;
branch if less/equal	PC=NZ<=0,L22;	PC=NZ<=0,L22;
load i *into data reg. 0*	d[0]=L[a[6]+i.];	d[0]=L[a[6]+i.];
divide data reg. 0 by n	d[0]=d[0]/L[a[6]+n.];	d[0]=d[0]/L[a[6]+n.];
store data reg. 0 into i	L[a[6]+i.]=d[0];	L[a[6]+i.]=d[0];
jump unconditionally to L23	PC=L23;	a[6]=UK;
	L22	PC=RT;
load i *into data reg. 0*	d[0]=L[a[6]+i.];	L22
multiply data reg. 0 by n	d[0]=d[0]*L[a[6]+n.];	d[0]=L[a[6]+i.];
store data reg. 0 into i	L[a[6]+i.]=d[0];	d[0]=d[0]*L[a[6]+n.];
	L23	L[a[6]+i.]=d[0];
restore old frame pointer	a[6]=UK;	a[6]=UK;
return from subroutine	PC=RT;	PC=RT;

Table 2: If–Then–Else Statement (RTLs for 68020)

3.2 Conditional Statements

For the if-then-else construct an unconditional jump is generated at the end of the then-part to jump over the else-part. This unconditional jump can also be eliminated by code replication. There are two execution paths possible which are joined at the end of the if-then-else construct. The two execution paths can be separated completely or their joining can be at least deferred by replicating the code after the if-then-else construct, so that the unconditional jump is replaced by the copied instructions. Table 2 shows an example of replicating code where the two execution paths return from the function separately.[2] The method of code replication used for conditional statements can also be applied to *break* and *goto* statements, and conditional expressions in the C language (`expr?expr:expr`).

3.3 Sources for other Optimizations

Code replication also creates new opportunities for global optimizations by modifying the control flow of a function. The following paragraphs describe new opportunities for constant folding, instruction selection, common subexpression elimination, and code motion.

3.3.1 Constant Folding of Comparisons and Conditional Branches

After applying code replication, sources for constant folding may be introduced which did not exist before. For example, conditional branches, dependent on the comparison of two constants, may be introduced by changing the control flow during code replication. Depending on the result of the constant comparison, such a conditional branch can either be eliminated or replaced by an unconditional jump. In the later case, dead code elimination may remove instructions following the unconditional jump which can no longer be reached.

3.3.2 Elimination of Instructions

In conjunction with code replication, common subexpression elimination can often combine instructions when an initial value is assigned to a register, followed by an unconditional jump. If the replication sequence uses the register, the use is replaced by the initial value so that the assignment to the register becomes redundant if there are no further uses or sets of the register. Similarly, instruction selection may combine instructions when the head of a loop is moved. For example, the second RTL in the replicated code of Table 1 is a simplification of the first and fifth RTLs in the code before replication.

3.3.3 Relocating the Preheader of Loops

After code replication the execution of some instructions may be avoided for an execution path. For instance, code motion is performed after an initial pass of code replication. This may result in a new location of the preheaders of loops. Thus, if a loop is not executed because the conditional branch preceding the loop is taken, the instructions in the preheader (following that branch) would not be executed. This may result in considerable savings when loops are nested. An example is the RTL preceding label L000 in the replicated code of Table 1 which is the loop preheader.

[2] Notice that nested if-then-else statements can cause code to be replicated very often, thus resulting in an disproportional growth in code size relative to the original code size.

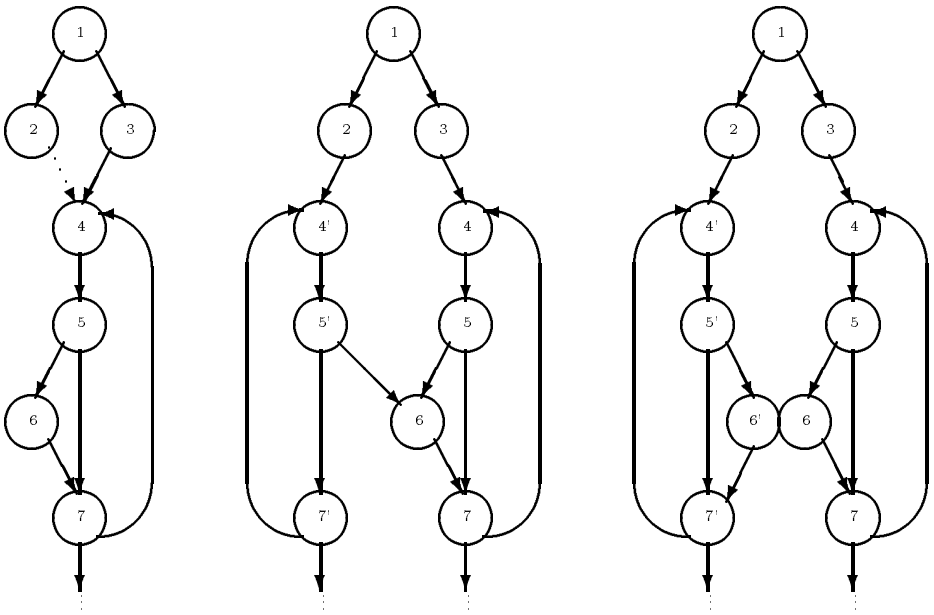

Figure 1: Interference with Natural Loops

4 JUMPS: An Algorithm for Code Replication

The task of code replication is to find a sequence of basic blocks to replace an unconditional jump. In order to avoid algorithms with a high degree of complexity, it was decided to make the initial assumption that only the shortest path between two basic blocks is examined. This constraint is motivated by the goal of limiting the size of code introduced by the replication process. The shortest path is determined with respect to the number of RTLs. Finding the shortest path in the control-flow graph with n blocks is accomplished by using Warshall's algorithm for calculating the transitive closure of a graph [Wa62] which has a complexity of $O(n^3)$. First, all legal transitions between any two distinct basic blocks are collected in a matrix. This initial pass creates a copy of the control flow graph but it excludes self-reflexive transitions and, optionally, other edges whose control flow is excluded explicitly. For example, the replication of indirect jumps has not yet been implemented at this point. Then, the non-reflexive transitive closure is calculated for all nodes with respect to the shortest path. The transitivity relation between two nodes is only recorded if it is the shortest connection found so far in terms of the number of RTLs in the traversed blocks [Fl62]. In the end, the matrix can be used to look up the shortest path between two arbitrary basic blocks in the table without having to recalculate it after each replication. The algorithm JUMPS is divided into the following steps:

1. Initially, the matrix used to find the shortest sequence of basic blocks to replace an unconditional jump is set up.
2. In the second step, the basic blocks within a function are traversed sequentially and unconditional jumps are replaced as follows. Either a sequence of blocks that ends with a return from the routine is replicated (*favoring returns*), or a sequence of blocks is chosen linking the current block containing the unconditional jump with the block positionally following the unconditional jump (*favoring loops*). In the latter case, the last block to be replicated will fall through to the next block. At this point, heuristics can be used to make the choice between these two options.
3. If a collected block (i.e. a block chosen for replication in the previous step) was detected to be the header of a natural loop and the block collected previously was not inside the same loop, then all blocks inside this loop are included in the replication sequence in their positional order. The example in Figure 1 has an unconditional jump from block 2 to block 4 before replication. Without replicating block 6, the original loop would have two entry points and would be unstructured.
4. Once a sequence of basic blocks is replicated, the control flow is adjusted accordingly. A conditional branch is reversed in the replicated path if the path does not follow the fall-through transition. New labels are introduced, and the destinations of conditional branches are modified. In addition, all

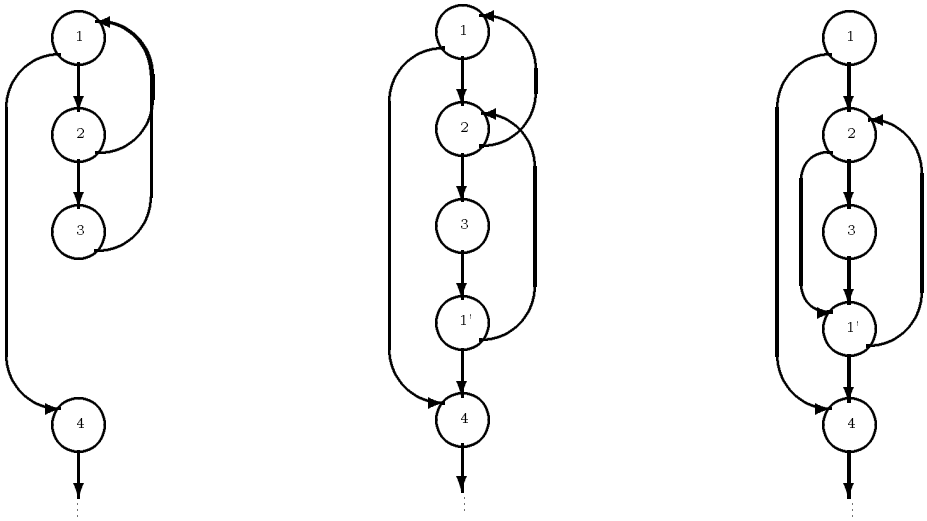

Figure 2: Partial Overlapping of Natural Loops

unconditional jumps that are found in the replicated code can be eliminated since the sequence of replicated blocks had to follow the control flow with fall-through transitions.

5. The adjustment of the control flow is extended, once again to preserve the structure of loops. When a replication is initiated from a block inside a loop, a portion of the loop can be copied without introducing unnatural loops. In addition, the control flow of all blocks in the loop which were not copied but branch conditionally to a block which was copied, is changed to the copied block. If a block occurs twice in the replication sequence, forward branches (positionally down) are favored over branches back to a previous block. These modifications are needed to avoid the introduction of natural loops which partially overlap and would complicate standard loop optimizations. An example is given in Figure 2. There is an unconditional transfer from block 3 to block 1 in the initial control flow. By changing the target of the conditional branch in block 2 after replication, the introduction of partially overlapping loops is avoided in the adjusted control flow.

6. Even with replication of entire natural loops, it is still possible for this algorithm to introduce new loops which are unstructured. Therefore, the control flow graph is checked to determine if it is still reducible. The replicated code is removed if it introduced a non-reducible flow graph. In this case, replication may be attempted using the longer block sequence (see step 2). Reducibility has to be preserved to ensure that the same optimizations can be applied to loops as without replication.

The algorithm JUMPS is applied to a function for each unconditional jump until no more unconditional jumps can be replaced. As a result of the replication process, blocks which cannot be reached by the control flow anymore can sometimes occur. Therefore, dead code elimination is invoked to delete these blocks.

5 Measurements

Static and dynamic frequency measurements and instruction cache performance measurements were taken from a number of well-known benchmarks, UNIX utilities, and one application (see Table 3). The code was generated for the Motorola 68020/68881 processor and

Class	Name	Description
Utilities	banner	banner generator
	cal	calendar generator
	compact	file compression
	deroff	remove nroff constructs
	grep	pattern search
	od	octal dump
	sort	sort or merge files
	wc	word count
Benchmarks	bubblesort	sort numbers
	matmult	matrix multiplication
	sieve	iteration
	queens	8-queens problem
	quicksort	sort numbers (iterative)
User code	mincost	VLSI circuit partitioning

Table 3: Test Set of C Programs

		static			dynamic		
		SIMPLE	LOOPS	JUMPS	SIMPLE	LOOPS	JUMPS
Sun SPARC	average	3.74%	2.40%	0.03%	3.28%	1.89%	0.10%
	std. deviation	1.78%	1.99%	0.12%	2.71%	2.56%	0.30%
Motorola 68020	average	5.08%	3.42%	0.04%	4.14%	2.47%	0.13%
	std. deviation	2.49%	2.83%	0.15%	3.48%	3.36%	0.43%

Table 4: Percent of Instructions that are Unconditional Jumps

the Sun SPARC processor, a RISC architecture. For the SPARC processor, delay slots after transfers of control were filled. The standard code optimization techniques such as branch chaining, instruction selection, register coloring, common subexpression elimination, constant folding, code motion, strength reduction, and constant folding at conditional branches were applied on the measured code. Library routines could not be measured since the source code was not available to be compiled by VPO.

The measurements were collected by using EASE (Environment for Architectural Study and Experimentation) [Da90-2] which is designed to measure the code produced by the optimizer VPO. When generating code, additional instructions are inserted to capture measurements during the execution of a program.

Each program was tested with three different sets of optimizations:

- SIMPLE: Only the standard optimizations were performed.
- LOOPS: Unconditional jumps preceding a loop or at the end of the loop are replaced by the termination condition of the loop and the replicated condition is reversed. Depending on the original layout of the loop, either one unconditional jump is removed at the entry point, or one unconditional jump is saved per loop iteration. This optimization is often implemented in conventional optimizers.
- JUMPS: This is the algorithm discussed previously. It is a generalized approach which attempts to replace any occurrences of unconditional jumps by replicating code.

5.1 Integration into an Optimizing Compiler

The code replication algorithms (JUMPS and LOOPS) are integrated into the optimizing back-end of the VPO compiler in the following manner. After performing initial branch optimizations such as branch chaining, code replication is performed to reduce the remaining number of unconditional jumps. When JUMPS is used for code replication, the compile-time overhead for the replication process itself is minimal, but the following optimization stages process more RTLs. The impact of LOOPS on the compile time is minimal.

Figure 3 summarizes the order in which the different optimization phases are invoked. Code replication is performed at an early stage so that the later optimizations can take advantage of the simplified control flow. In order to replace all unconditional jumps generated by constant folding at conditional branches or introduced by remote preheaders, code replication is reinvoked repeatedly. The final invocation of code replication replaces those unconditional jumps which remained in the code because replication would have resulted in a non-reducible flow graph.

```
branch chaining;
dead code elimination;
reorder basic blocks to minimize jumps;
code replication (either JUMPS or LOOPS);
dead code elimination;
instruction selection;
register assignment;
if (change)
  instruction selection;
do {
  register allocation by register coloring;
  instruction selection;
  common subexpression elimination;
  dead variable elimination;
  code motion;
  strength reduction;
  recurrences;
  instruction selection;
  branch chaining;
  constant folding at conditional branches;
  code replication (either JUMPS or LOOPS);
  dead code elimination;
} while (change);
filling of delay slots for RISCs;
```

Figure 3: Order of Optimizations

5.2 Static and Dynamic Behavior

Table 4 shows the number of unconditional jumps relative to the total number of instructions for the static and dynamic measurements. The number of unconditional jumps is reduced by 40-42% dynamically when LOOPS was applied, and with code replication prac-

tically no unconditional jumps are left. Thus, code replication results in a reduction of instructions executed by at least the number of unconditional jumps which could be avoided dynamically.

The few unconditional jumps left after code replication are due to indirect jumps, infinite loops, and interactions with other optimization phases (such as code motion) that may introduce unconditional jumps. Paths containing indirect jumps are excluded from replication in the current implementation. Infinite loops do not provide any opportunity to replace the unconditional branch. And interactions with other optimization phases are treated conservatively to avoid the potential of replication *ad infinitum*.

Table 5 illustrates the static and dynamic behavior of the programs. The columns SIMPLE indicate the total number of instructions and the other columns represent the change in the number of instructions relative to the SIMPLE version of each program. The static change is proportional to the growth of the code size. When LOOPS is applied, the number of instructions increases by only 2.6-4.0%. With generalized code replication, on the other hand, an average of about 53% more instructions are generated. The decrease in the number of instructions executed for LOOPS is less than half the decrease for JUMPS.

For the SPARC about 1.5 more instructions are found between branches after code replication was applied and 50% of the executed no-op instructions were eliminated. Thus, the opportunities for instruction scheduling may improve if code replication is applied for a pipelined machine. Also, in a multiple-issue processor more potential parallelism may be found [Ri72].

5.3 Impact on Instruction Caching

The cache performance was tested for cache sizes of 1Kb, 2Kb, 4Kb, and 8Kb. For each different cache size a direct-mapped cache with 16 bytes per line was simulated. Both the miss ratio and the fetch cost were measured in the experiment. The estimation of the fetch cost is based on the assumption that misses are ten times as expensive as hits. Thus, fetch cost is calculated as follows:

$$\begin{aligned} fetch\ cost \quad = \quad & cache\ hits * cache\ access\ time + \\ & cache\ misses * miss\ penalty \end{aligned}$$

where the cache access time is 1 time unit and the miss penalty is 10 units of time. Context-switches were simulated by invalidating the entire cache every 10,000 units of time. The estimates for the cache access time, the miss penalty, and the context-switching interval were adopted from Smith's cache studies [Sm82]. Notice that the overall fetch cost can decrease while the miss ratio increases for the same program. This can be explained by the reduced number of instructions executed after replication and illustrates the short-comings of the miss ratio as a measurement when the code in a program changes. (This observation was first made by [?].)

Table 6 shows the change of the miss ratio and fetch cost for varying sizes of direct-mapped caches. Each set of measurements with the same configuration is related to the corresponding values of the SIMPLE version. For example, for a 1Kb cache with context switches on the SPARC, the difference between the miss ratio of LOOPS and the miss ratio of the SIMPLE version was -0.05%, a slight decrease of misses.

The impact of context switching was minimal, and the miss ratio only increased slightly with context switching on.

For small caches, code replication (JUMPS) may be outperformed by loop replication (LOOPS). A program may initially fit in the cache, but after code replication is applied, it might not fit anymore. Therefore, capacity misses can be introduced. For example, for a 1Kb cache about 1% additional misses were caused by instruction fetches. But for larger caches the miss ratio changes only slightly.

Code replication places instructions together which are likely to be executed in a sequence but increases the distance between conditional branch instructions and their branch destinations. Nevertheless, the program's spatial locality can be improved by replicating code. Overall, code replication reduces the total number of instructions executed such that the average fetch cost is actually reduced except for small caches.

6 Future Work

The algorithm for code replication could be extended to copy indirect jumps and, for some architectures, their jump tables. If the table has to be replicated, the target addresses within the jump table should be revised. In either case, the jump destinations do not need to be copied. Thus, an indirect jump could terminate a replication sequence and provide yet another alternative besides replication paths favoring returns and favoring loops (see step 2 of algorithm JUMPS).

Furthermore, the increase in code size could be reduced by limiting the maximum length of a replication sequence to a specified number of RTLs. The improvements in the dynamic behavior of programs may drop slightly for this case while the performance of small caches should benefit.

Sun SPARC						
program	static instructions			dynamic instructions executed		
	SIMPLE	LOOPS	JUMPS	SIMPLE	LOOPS	JUMPS
cal	338	+3.25%	+21.89%	37,237	−2.95%	−3.15%
quicksort	321	+5.61%	+50.16%	836,404	−2.86%	−14.21%
wc	209	+0.96%	+58.37%	540,158	−0.00%	−1.96%
grep	968	+4.24%	+79.34%	1,930,791	−0.04%	−3.57%
sort	1,966	+4.63%	+89.17%	1,181,960	−0.71%	−10.49%
od	1,352	+4.59%	+95.19%	2,336,014	−8.84%	−10.22%
mincost	1,068	+6.84%	+30.99%	335,750	−0.59%	−3.91%
bubblesort	175	+7.43%	+5.14%	29,071,668	−0.05%	−0.07%
matmult	218	+4.59%	+3.67%	14,403,714	−0.08%	−0.28%
banner	169	+7.69%	+66.27%	2,565	−1.68%	−10.25%
sieve	93	+3.23%	+3.23%	2,184,965	−13.73%	−13.73%
compact	1,491	+1.07%	+75.18%	13,409,945	−1.94%	−4.86%
queens	114	+0.00%	+7.89%	263,518	−0.00%	−0.03%
deroff	7,987	+1.50%	+204.98%	448,581	−0.01%	−3.13%
average	1,176	+3.97%	+56.53%	4,784,519	−2.39%	−5.71%
Motorola 68020						
program	static instructions			dynamic instructions executed		
	SIMPLE	LOOPS	JUMPS	SIMPLE	LOOPS	JUMPS
cal	323	+3.72%	+24.77%	36,290	−3.09%	−3.17%
quicksort	245	+3.67%	+37.96%	536,566	−0.39%	−3.96%
wc	173	+0.58%	+56.65%	421,038	−0.00%	−5.32%
grep	775	+3.35%	+80.90%	1,309,586	−0.03%	−3.44%
sort	1,558	+3.98%	+63.67%	902,075	−1.49%	−12.43%
od	1,198	+2.92%	+85.73%	1,980,808	−9.45%	−10.30%
mincost	906	+3.20%	+35.98%	302,062	−1.10%	−5.13%
bubblesort	137	+3.65%	+2.92%	20,340,231	−18.92%	−18.92%
matmult	146	+3.42%	+3.42%	4,891,507	−0.21%	−0.21%
banner	177	+3.95%	+55.93%	2,473	−1.42%	−13.34%
sieve	70	+1.43%	+1.43%	1,759,088	−8.53%	−8.53%
compact	1,143	+0.70%	+73.93%	10,602,159	−1.54%	−5.26%
queens	94	+0.00%	+12.77%	189,518	−0.00%	−0.05%
deroff	5,730	+1.06%	+155.17%	360,051	−0.03%	−7.05%
average	905	+2.55%	+49.37%	3,116,675	−3.30%	−6.94%

Table 5: Number of Static and Dynamic Instructions

	cache size	1Kb		2Kb		4KB		8Kb	
processor	context sw.	LOOPS	JUMPS	LOOPS	JUMPS	LOOPS	JUMPS	LOOPS	JUMPS
		Cache Miss Ratio							
Sun	on	−0.05%	+1.07%	−0.22%	−0.07%	+0.03%	+0.25%	+0.01%	+0.11%
SPARC	off	−0.03%	+1.07%	−0.22%	−0.08%	+0.03%	+0.21%	+0.01%	+0.07%
Motorola	on	+0.08%	+1.26%	+0.04%	+0.75%	+0.01%	+0.09%	+0.01%	+0.07%
68020	off	+0.08%	+1.25%	+0.03%	+0.70%	+0.01%	+0.05%	+0.01%	+0.03%
		Instruction Fetch Cost							
Sun	on	−2.73%	+3.44%	−3.80%	−5.24%	−2.26%	−2.94%	−2.40%	−3.98%
SPARC	off	−2.64%	+3.68%	−3.87%	−5.33%	−2.24%	−3.13%	−2.47%	−4.30%
Motorola	on	−3.07%	+1.69%	−3.26%	−0.63%	−3.58%	−5.13%	−3.57%	−5.30%
68020	off	−3.04%	+1.86%	−3.28%	−0.71%	−3.61%	−5.48%	−3.60%	−5.66%

Table 6: Percent Change in Miss Ratio and Instruction Fetch Cost for Direct-Mapped Caches

7 Conclusions

A new global optimization method called code replication was developed which can be applied to eliminate almost all unconditional jumps in a program. The resulting programs are executing 5.7-6.9% less instructions in average. The number of instructions between branches is increased by 1.5 instructions on the average on the SPARC, so that the opportunities for instruction scheduling for pipelined or multi-issue machines are improved. The cache work decreases by about 4% except for small caches. The static number of instructions increases by an average of 53%. The generalized technique of code replication should be applied in the back-end of highly optimizing compilers if the execution time but not the program size is the major concern. The results of the test set also show that the replication of only the branch condition at natural loops, commonly performed in optimizing compilers, results in about 45% of the dynamic instruction savings which can be achieved by generalized code replication.

References

[Be88] M. E. Benitez, J. W. Davidson *A Portable Global Optimizer and Linker*, Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation, Atlanta, GA, June 1988, pp. 329-338

[Cl82] D. W. Clark, H. M. Levy, *Measurement and Analysis of Instruction Use in the VAX-11/780*, Proceedings of the 9th Annual Symposium on Computer Architecture, April 1982, pp. 9-17

[Da88] J. W. Davidson, A. M. Holler, *A Study of a C Function Inliner*, Software, Vol. 18, No. 8, August 1988, pp. 775-790

[Da89] J. W. Davidson, D. B. Whalley, *Quick Compilers Using Peephole Optimizations*, Software – Practice and Experience, Vol. 19, No. 1, January 1989, pp. 195-203

[Da90-2] J. W. Davidson, D. B. Whalley, *Ease: An Environment for Architecture Study and Experimentation*, Proceedings of the SIGMETRICS 1990 Conference on Measurement and Modeling of Computer Systems, May 1990, pp. 259-260

[Fl62] R. W. Floyd, *Algorithm 97: Shortest Path*, Communications of the ACM, Vol. 5, No. 6, June 1962, p. 345

[Go90] M. C. Golumbic, V. Rainish, *Instruction Scheduling beyond Basic Blocks*, IBM Journal of Research Development, Vol. 34, No. 1, January 1990, pp. 93-97

[He90] J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1990

[Hw89] W. W. Hwu, P. P. Chang, *Inlining Function Expansion for Compiling C Programs*, Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation, Vol. 24, No. 5, June 1989, pp. 246-257

[Pe77] B. L. Peuto, L. J. Shustek, *An Instruction Timing Model of CPU Performance*, Proceedings of the 4th Annual Symposium on Computer Architecture, March 1977, pp. 165-178

[Ri72] E. M. Riseman, C. C. Foster, *The Inhibition of Potential Parallelism by Conditional Jumps*, IEEE Transactions on Computers, Vol. 21, No. 12, December 1972, pp. 1405-1411

[Sm82] A. J. Smith, *Cache Memories*, Computing Surveys, Vol. 14, No. 3, September 1982, pp. 473-530

[Wa62] S. Warshall, *A Theorem on Boolean Matrices*, Journal of the ACM, No. 9, 1962, pp. 11-12