THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

AVOIDING UNCONDITIONAL JUMPS BY CODE REPLICATION

By

FRANK MUELLER

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:

Spring Semester, 1991

The members of the Committee approve the thesis of Frank Mueller defended on April 12, 1991.

David Whalley
Professor Directing Thesis

Theodore P. Baker
Committee Member

Gregory A. Riccardi
Committee Member

Acknowledgements

I want to express my gratitude to Dr. David Whalley, my major professor, for his guidance, support, patience, and promptness during my work on this thesis.

Contents

List of Tables

List of Figures

Abstract

This thesis evaluates a global optimization technique that avoids unconditional jumps by replicating code. Common programming constructs such as for-loops and conditional statements are translated to machine instructions by the use of conditional and unconditional jumps. One can reduce the number of unconditional jumps by replicating chunks of code, which reduces the number of instructions executed and may introduce new opportunities for code optimization. When implemented in the back-end, this technique can be generalized to work on conditional statements, switch statements coded as a sequence of branches, and both structured and unstructured loops. The replication method is based on the idea of finding a replacement for each unconditional branch which minimizes the growth in code size. This is achieved by choosing the shortest sequence of instructions as a replacement. In addition, the execution time of programs is improved. 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 despite increases in code size.

Chapter 1

Introduction

Unconditional branches are instructions that 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 coded using unconditional jumps, thus resulting in relatively compact code.

In recent years, code size has become less important. For instance, with the introduction of reduced instruction set computers (RISC), the code size of programs has increased since instructions are less powerful and more simply encoded. To make up for increased bus traffic due to more frequent instruction fetches, cache memory has been added for RISC architecture as well as for the traditional complex instruction set computers (CISC).

This thesis describes a method of replacing unconditional branches uniformly by replicating a sequence of instructions from the branch 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 evaluated by measurements of program traces to illustrate its advantages over traditional replication methods.

The document is structured as follows. Chapter 2 gives an overview of research on related topics. Chapter 3 describes the environment used for the experiment. Chapter 4 introduces the terminology used in the thesis, and describes the format of the intermediate representation. Chapter 5 illustrates the advantages of using code replication for optimizing various programming constructs. Chapter 6 provides an informal description of the algorithms used to implement code replication. Chapter 7 discusses the results of the implementation by comparing the measurements of numerous programs with and without code replication. Chapter 8 gives an overview of future work and Chapter 9 summarizes the results.

Chapter 2

Related Research

Several techniques to avoid unconditional jumps have been applied in optimizing compilers. For example, branch chaining changes the destination of a branch to the destination of the last jump in a chain of unconditional branches. The number of unconditional branches can be reduced further by reorganizing the order of basic blocks. Such traditional techniques, as described by Aho, Sethi, and Ullman [Ah86], can be enhanced by code replication, an optimization method described in this thesis, to remove almost all unconditional branches of a program.

When a compiler front-end emits intermediate code, it is quite common to use the termination criteria for a loop as a pre-check condition for the loop, followed by the loop body, and, instead of an unconditional branch to the pre-check, a post-check which is the inverted termination condition.[1] Nevertheless, such code improvements can only be performed whenever natural loops are found in a program source code. Unconditional jumps in unnatural loops (loops with multiple entry points) are usually not eliminated.

All techniques employed in the front-end lack generality in reducing the number of unconditional jumps. Instances of unconditional jumps cannot always be detected at the level of the parser due to its interaction with other optimizations

[1] A set of examples for loops and other occurrences of unconditional branches is given in Chapter 5.

and a lack of information about the target architecture. Consequently, front-end methods for code replication cannot catch occurrences of unconditional jumps which are introduced by the optimization phase.

Procedure inlining, an optimization method described by Davidson and Holler [Da88], often results in replicated code. 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 procedure call to a non-recursive function can be replaced by the actual code of the procedure 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 branch back to the instruction following the call.

Branches have long been recognized as causing problems for machines, and a variety of schemes have been proposed to reduce the cost of transfers of control. Delaying the execution of branches is a commonly used technique to avoid stalls in instruction pipelines [Pa85]. Using complicated hardware, branches have been folded into instructions when brought into an instruction cache [Di87]. In this approach, each instruction also contains the address of the next instruction to be executed. Thus, an unconditional branch can be folded into its preceding instruction. Conditional branch instructions contain two potential addresses for the next instruction and a static prediction bit to support prefetching towards the path which is more likely to be executed. In another study, the use of branch registers has been suggested to move the branch target address calculation out of loops and reduce delays in the pipeline [Da90-1].

Golumbic and Rainish [Go90] used the method of replicating parts of basic blocks for instruction scheduling. In their approach, it suffices to copy the number of instructions needed to avoid a pipeline delay 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 branch. Their goal is strictly to increase the pipeline utilization by filling the delay slots of conditional branches if they are likely to be taken, but not to avoid unconditional branches by code replication.

Lately, several approaches have been made to change the positional order of basic blocks based on profiling data [Mc89,Pe90]. Code was restructured such that frequently executed chains of basic blocks follow one another positionally. Thus, the number of executed instructions was reduced by introducing transfers of control to infrequently executed portions of code in remote locations.

Chapter 3
Environment for Experimentation

Two different approaches are commonly taken to reduce the number of unconditional branches. One can employ language-dependent optimizations in the front-end of a compiler or one can attempt machine-dependent optimizations in the back-end. While optimizations at an early stage take advantage of information already available from parsing, the latter approach generalizes optimizations, applies them to unstructured programming constructs, and takes architectural restrictions into consideration.

For those reasons, code replication was embedded into the back-end of a tool called Very Portable Optimizer (VPO) which reads the intermediate representation generated by the front-end called Very Portable C Compiler (VPCC). The optimizations performed in VPO include:

- branch chaining and branch minimization
- dead code elimination
- instruction selection
- local register allocation
- global register allocation by coloring
- common subexpression elimination

- dead variable elimination
- code motion
- strength reduction
- instruction scheduling

VPO itself is a part of the "Environment for Architectural Study and Experimentation (EASE)" which was designed to take static and dynamic measurements of programs and perform instruction cache analysis by cache simulation. When generating code, additional instructions are inserted to capture measurements during the execution of a program without influencing the measurements. The characteristics of instructions are obtained and stored at compile time. For dynamic measurements and cache analysis, the frequency counts and instruction information are used together to generate detailed reports of the number of instructions executed, the number of cache hits and misses, the distribution of instruction types, etc. For static measurements, a frequency count of one is assumed for each instruction in the program. A more detailed description of the environment is given by Davidson and Whalley [Da90-2].

The intermediate program representation used by the compiler in the experiment allows transformations and optimizations to occur in a machine-independent fashion. Instructions are represented by register transfer lists (RTLs). The general format of RTLs is machine-independent but the actual register transfers represent legal instructions for a specific architecture. A machine description, constructed from a context-free grammar and semantic actions, defines the set of legal instructions. This machine description is used to determine whether transformations to RTLs are legal and to translate each RTL to assembly code.

Chapter 4
Definitions

In the following sections the essential terminology is introduced, and the notation for discussing the issues of code replication in the compiler back-end is given. The terms are used in a subsequent chapter to describe the algorithms to implement code replication.

4.1 Register Transfer Lists

Throughout the following chapters, transformations on instructions of the intermediate representation of a program are described. Each instruction is denoted by a register transfer list. A more detailed description of the instruction representation is given by Benitez [Be91].

Definition 4.1 A *register transfer* denotes the assignment of a source value, `src`, to a destination location, `dst`, which represents an individual effect of an instruction and has the general form of an assignment.

`dst = src;`

The source can be a simple expression of constants, memory values, and operators. The destination denotes a memory address or a register.

Definition 4.2 A *register transfer list* (RTL) is a sequence of register transfers. The effects of the register transfers within the list are accomplished in parallel. Each RTL describes a legal instruction for a particular architecture.

The conventions used to denote RTLs for the Motorola 68020 [Mo85], the architecture used in this thesis, are summarized in Table 4.1. For example, the RTLs

```
NZ=d[0]?10;
PC=NZ<=0,L19;
```

denote a comparison between the content of data register 0 and the numeric constant 10 followed by a conditional branch to label L19. Note that the branch is only taken if the value in the register is less than or equal to 10.

For typical machines most RTLs consist of only a single register transfer. However, there are instances where a single instruction can have more than one effect. For the Motorola 68020, some instructions change the condition codes implicitly. For example, the RTL

```
NZ=d[0]?0;d[2]=d[0];
```

describes a register copy instruction which also has the effect of changing the condition codes that will be used by a conditional branch instruction.

4.2 Intermediate Representation of a Function

The following definitions describe control flow relationships between instructions within a function (intra-procedural):

Definition 4.3 A *basic block* is a sequence of consecutive RTLs with exactly one entry point, an optional label, and one exit point. Only the last RTL, at the exit point, can be a branch instruction.

Since the instructions within a basic block are executed sequentially, they are always executed the same number of times. A function or procedure call is

Table 4.1: Register Transfer Conventions for the Motorola 68020

symbol	description
Ln	label for $n \in N$ (positive numbers including 0)
PC	program counter
NZ	negative and zero flag
?	comparison operation (infix) with two operands
A[loc]	address of a memory location
B[A[loc]]	content of a memory location (byte size)
W[A[loc]]	content of a memory location (word size)
L[A[loc]]	content of a memory location (long size)
b[n]	data register (byte size) for $n \in \{0..7\}$
w[n]	data register (word size) for $n \in \{0..7\}$
d[n]	data register (long size) for $n \in \{0..7\}$
m[n]	address register for $n \in \{0..7\}$
	where m[7] is reserved for the stack pointer
	and m[6] is reserved for the frame pointer
dm[n]	address register with pre-decrement for $n \in \{0..7\}$
m[n]i	address register with post-increment for $n \in \{0..7\}$
name.	symbolic offset for a local identifier
_name	symbolic offset for a global identifier
ST	call instruction
LK	link instruction
UK	unlink instruction
MM	multiple move instruction
n	numeric constant for $n \in Z$ (positive and negative numbers)

not treated as an ordinary branch instruction since the control is transferred beyond the bounds of the current function (inter-procedural). Therefore, a call instruction can occur anywhere within a basic block.

Definition 4.4 A *legal transition* from basic block a to basic block b, denoted as $a \rightarrow b$, is determined by the last RTL r of block a and the location or label l of block b:

- If r is an unconditional branch, then the transition $a \rightarrow b$ is the only transfer of control from a and the destination of the branch has to be label l (see transition $1 \rightarrow 2$ in Figure 4.1).

- If r is a conditional branch, then there are two transitions from block a. One transition is to the basic block f which positionally follows block a (see $3 \rightarrow 4$ in Figure 4.1) and the other is to the block g whose label k is the target of the branch instruction r (see $3 \rightarrow 2$ in Figure 4.1). Furthermore, $b = g$ iff $l = k$.

- If r is a return instruction, then there is no transition from block a since only a single function is evaluated at one time (see basic block 4 in Figure 4.1).

- If r is any other instruction, then the transition $a \rightarrow b$ is the only transition and block b positionally follows block a (see transition $2 \rightarrow 3$ in Figure 4.1).

Notice that the *self-reflexive* transition $a \rightarrow a$ is legal transition.

Definition 4.5 A transition $a \rightarrow b$ is a *fall through transition* if block b positionally follows block a.

```
main() {
int i, k;
char a[10],b[10];

scanf("%k", &k);
for (i=k;i<10;i++)
  a[i]=b[i];
}
```

```
#1        m[6]=LK28;
          L[A[dm[7]]]=A[m[6]+k.];
          L[A[dm[7]]]=A[L15];
          ST=A[_scanf];
          m[7]=m[7]+8;
          d[0]=L[A[m[6]+k.]];
          m[0]=A[d[0]+m[6]+a.];
          m[1]=A[d[0]+m[6]+b.];
          PC=L18;

#2   L19
          B[A[m[0]i]]=B[A[m[1]i]];
          d[0]=d[0]+1;

#3   L18
          NZ=d[0]?10;
          PC=NZ<0,L19;

#4        m[6]=UK;
          PC=RT;
```

Figure 4.1: A simple Control Flow Graph

Definition 4.6 The *control flow graph*, CFG, of a function is the graph whose nodes are the basic blocks representing the function and whose edges are all legal transitions between any two blocks.

Thus, the CFG is a directed graph possibly including cyclic dependencies. Figure 4.1 shows the CFG, basic blocks, and RTLs for a simple function with one loop.

Definition 4.7 The *transitive closure* of a directed graph is a graph with edges from any node a to node b if there exists a sequence of transitions a to b denoted as $a \xrightarrow{*} b$.

Thus, the transitive closure of a graph indicates whether one node can be reached from another. In Figure 4.1, the transition $1 \xrightarrow{*} 4$ is introduced since there exists a sequence $1 \rightarrow 3 \rightarrow 4$, but there is no sequence $4 \xrightarrow{*} 1$.

A transitive closure is called *reflexive* if it includes connections of the form $a \xrightarrow{*} a$ for any node a. The graph of a *non-reflexive* transitive closure excludes these edges.

Finally, the terminology to describe loops in a program is defined.

Definition 4.8 In the CFG, a basic block a *dominates* block b if all paths from the first basic block in the function to block b include block a.

Notice that any basic basic block dominates itself.

Definition 4.9 A transition $a \rightarrow b$ is called a *back edge*, if b dominates a.

Definition 4.10 A *natural loop* is a set of basic blocks with a single entry block h, called the *header* block, which dominates all other blocks in the loop and there exists a back edge $a \rightarrow h$ for a block a within the loop.

Notice that a natural loop can have more than one back edge. A loop which has more than one entry point is called an *unnatural loop*.

Definition 4.11 The *preheader* block p of a natural loop is the only predecessor of the header block h from which the loop can be entered. Thus, $p \rightarrow h$ is the only transition to h from outside the loop.

Preheaders sometimes have to be created to perform optimizations such as code motion and strength reduction that move RTLs out of a loop.

Chapter 5
Motivation

The optimization evaluated in this thesis, code replication, was accomplished by modifying the optimizer of VPO. The algorithms to perform the optimization, except for a few small functions, are machine-independent. In general, RTLs are searched for unconditional jumps. By determining the branch destination and using the control flow information already available in the back-end, a subset of the basic blocks in the function can be replicated, replacing the unconditional branch. Such an optimization can be applied for all loop constructs (for, while, do-while) as well as some conditional statements (if-then-else) and other transfers of control within a function (break, goto). The following sections give examples of instances where code replication can be applied.

5.1 Loops

For while-loops with an unknown number of iterations at compile time, the front-end, VPCC, generates intermediate code with an unconditional transfer of control 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. Table 5.1 illustrates the layout of while-loops before and after code replication.

Similarly, the intermediate code produced by the front-end for for-loops with an unknown number of iterations has an unconditional transfer of control to the

Table 5.1: While–Loop

```
i = 1;
while (a[i]) {
  a[i-1] = a[i];
  i++;
}
```

without replication	with replication
`m[0]=A[1+m[6]+a.];`	`NZ=B[A[1+m[6]+a.]]?0;`
`m[1]=m[0];`	`PC=NZ==0,L16;`
`L15`	`m[0]=A[1+m[6]+a.-1];`
`NZ=B[A[m[0]]]?0;`	`L000`
`PC=NZ==0,L16;`	`B[A[m[0]]]=B[A[m[0]+1]];`
`B[A[m[0]+-1]]=B[A[m[1]i]];`	`m[0]=m[0]+1;`
`m[0]=m[0]+1;`	`NZ=B[A[m[0]+1]]?0;`
`PC=L15;`	`PC=NZ!=0,L000;`
`L16 ...`	`L16 ...`

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 replicated code would appear before the loop and at the end of the loop (see Table 5.2). This optimization method can also be applied to nested loops.

Table 5.2: For–Loop

for (i=k;i<10;i++) a[i]=b[i];	
without replication	with replication
d[0]=L[A[m[6]+k.]];	d[0]=L[A[m[6]+k.]];
	NZ=d[0]?10;
	PC=NZ>=0,L0001;
m[0]=A[d[0]+m[6]+a.];	m[0]=A[d[0]+m[6]+a.];
m[1]=A[d[0]+m[6]+b.];	m[1]=A[d[0]+m[6]+b.];
PC=L18;	L19
L19	B[A[m[0]i]]=B[A[m[1]i]];
B[A[m[0]i]]=B[A[m[1]i]];	d[0]=d[0]+1;
d[0]=d[0]+1;	NZ=d[0]?10;
L18	PC=NZ<0,L19;
NZ=d[0]?10;	L0001 ...
PC=NZ<0,L19;	
...	

Traditionally, the replication of the termination condition discussed so far 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 branch. An example for such a situation is given Table 5.3. The method proposed in this thesis handles these cases as well as unnatural loops.

In fact, one unconditional branch per loop iteration is saved in this particular example.

Table 5.3: Exit Condition in the Middle of a Loop

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

without replication	with replication

<pre>   d[1]=1;
     m[0]=A[1+m[6]+a.-1];
L15
     d[0]=d[1];
     m[0]=m[0]+1;
     d[1]=d[1]+1;
     NZ=d[0]?L[A[_n]];
     PC=NZ>=0,L16;
     B[A[m[0]]]=B[A[m[0]+1]];
     PC=L15;
L16 ...</pre>
	<pre> d[0]=1;
	     d[1]=2;
	     NZ=d[0]?L[A[_n]];
	     PC=NZ>=0,L16;
	     m[0]=A[2+m[6]+a.-1];
	L000
	     B[A[m[0]]]=B[A[m[0]+1]];
	     m[0]=m[0]+1;
	     d[0]=d[1];
	     d[1]=d[1]+1;
	     NZ=d[0]?L[A[_n]];
	     PC=NZ<0,L000;
	L16 ...</pre>

5.2 Conditional Statements

The if-then-else construct imposes problems for code replication. Generally, an unconditional branch is only generated at the end of the if-part to jump over the else-part. 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 branch is replaced by the copied instructions. Table 5.4 shows how the two execution paths return from the function separately.

Table 5.4: If–Then–Else Statement

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

without replication	with replication
NZ=L[A[m[6]+i.]]?5;	NZ=L[A[m[6]+i.]]?5;
PC=NZ<=0,L22;	PC=NZ<=0,L22;
d[0]=L[A[m[6]+i.]];	d[0]=L[A[m[6]+i.]];
d[0]=d[0]/L[A[m[6]+n.]];	d[0]=d[0]/L[A[m[6]+n.]];
L[A[m[6]+i.]]=d[0];	L[A[m[6]+i.]]=d[0];
PC=L23;	m[6]=UK;
L22	PC=RT;
d[0]=L[A[m[6]+i.]];	L22
d[0]=d[0]*L[A[m[6]+n.]];	d[0]=L[A[m[6]+i.]];
L[A[m[6]+i.]]=d[0];	d[0]=d[0]*L[A[m[6]+n.]];
L23	L[A[m[6]+i.]]=d[0];
m[6]=UK;	m[6]=UK;
PC=RT;	PC=RT;

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.

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`).

5.3 Sources for other Optimizations

Code replication creates new opportunities for global optimizations by modifying the control flow of a function. The examples given in the following paragraphs cover instances of constant folding, common subexpression elimination, and code motion.

5.3.1 Constant Folding of Comparisons and Conditional Branches

After applying code replication, sources for constant folding may be introduced which did not exist before. The example in Table 5.5 illustrates a case of constant folding. By performing code replication, the control flow at two conditional statements is reorganized. As a side-effect, two conditional jumps depend on the comparison of two constants. After applying constant folding, one conditional branch can be eliminated and the other one can be replaced by an unconditional branch. Notice that after constant folding the code between the RTL `PC=L19;` and label L19 cannot be reached anymore. Also, the conditional branch to L17 can instead be chained to L19. After performing dead code elimination and branch chaining, the unreachable code and the unconditional branch are eliminated. Overall, three jump instructions and one test can be avoided.

This example may not seem to be realistic. Nevertheless, slightly more complex cases can be caused by code replication when two execution paths are joined. The if-condition is often used to check for special cases and collapses after replication and constant folding as shown in Table 5.5, while the else-part covers the general case limiting the sources for further optimization. In addition, replicating code at loops sometimes results in constant comparisons

Table 5.5: Constant Folding for If–Then–Else Statements

```
if (j==0)
  i = 0;
else
  i = 1;
if (i==0)
  a=b[i];
else
  a=c[j];
return(a);
```

1. before replication

```
      d[0]=L[A[m[6]+j.]];
      NZ=d[0]?0;
      PC=NZ!=0,L17;
      d[0]=0;
      PC=L18;

L17
      d[0]=1;
L18
      NZ=d[0]?0
      PC=NZ!=0,L19;
      b[0]=B[A[d[0]+m[6]+b.]];
      PC=L20;
L19
      m[0]=L[A[m[6]+j.]];
      m[1]=A[m[6]+c.];
      m[0]=m[0]+m[1];
      b[0]=B[m[0]]];
L20
      m[6]=UK;
      PC=RT;
```

2. after replication

```
      d[0]=L[A[m[6]+j.]];
      NZ=d[0]?0;
      PC=NZ!=0,L17;
      NZ=0?0;
      PC=NZ!=0,L19;
      b[0]=B[A[m[6]+b.]];
      m[6]=UK;
      PC=RT;
L17
      NZ=1?0;
      PC=NZ!=0,L19;
      b[0]=B[A[1+m[6]+b.]];
      m[6]=UK;
      PC=RT;
L19
      b[0]=B[A[d[0]+m[6]+c.]];
      m[6]=UK;
      PC=RT;
```

3. after constant folding

```
      d[0]=L[A[m[6]+j.]];
      NZ=d[0]?0;
      PC=NZ!=0,L17;
      b[0]=B[A[m[6]+b.]];
      m[6]=UK;
      PC=RT;
L17
      PC=L19;
      b[0]=B[A[1+m[6]+b.]];
      m[6]=UK;
      PC=RT;
L19
      b[0]=B[A[d[0]+m[6]+c.]];
      m[6]=UK;
      PC=RT;
```

4. after dead code elimination

```
      d[0]=L[A[m[6]+j.]];
      NZ=d[0]?0;
      PC=NZ!=0,L19;
      b[0]=B[A[m[6]+b.]];
      m[6]=UK;
      PC=RT;
L19
      b[0]=B[A[d[0]+m[6]+c.]];
      m[6]=UK;
      PC=RT;
```

in the preheader, and the discussed variation on constant folding can be applied as well. An example for such a case is discussed in the next chapter (see Table 6.3).

5.3.2 Common Subexpression Elimination

In conjunction with code replication, common subexpression elimination can usually remove instructions when a value is assigned to a register, followed by an unconditional branch. If the replication sequence uses the register, the use is often 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. The example in Table 5.5 illustrates such a case. The assignment of the value zero to register `d[0]` is only used to index the array `b`. After replication and constant folding, the RTL initializing `d[0]` is simply removed and the index zero for array `b` is omitted.

Another example is given in Table 5.6. The first column in the table indicates the basic block number. In the example, block 8 is copied as a result of replacing the unconditional branch in block 3. Since block 3 dominates block 8, common subexpression elimination removes the RTLs in block 8 that modify the value of `d[1]`. As the measurements in Chapter 7 show, this optimization results in a tremendous improvement since there are two less instructions in the inner loop, and the inner loop is executed very frequently. One should note that a more sophisticated code motion algorithm could have moved up the the two RTLs even without code replication. Code motion in VPO was designed to implement the algorithm described by Aho, Sethi, and Ullman [Ah86] and does not handle instances of multiple assignments to the same object.

5.3.3 Relocating the Preheader of Loops

The example shown in Table 5.6 also illustrates two instances where redundant executions of instructions are avoided for an execution path. Code motion is performed after code replication. This results in a new location of the preheaders, blocks 2 and 4, for the two nested natural loops. The fall-through transition from block 1 to the outer loop allows an RTL to be moved to basic block 2. Similarly, block 4 contains an RTL which resides in block 3 without code replication. If a loop is never executed because the conditional jump preceding the loop is taken, the RTL following that branch would not be executed, resulting in an overall savings of the number of instructions executed. The Tables 5.1 and 5.2 include similar instances.

5.4 Unremovable Branches

A few instances of unconditional branches cannot be removed at all as the following discussion illustrates.

Sometimes loop preheaders are introduced to move invariant instructions out of the loop when code motion and strength reduction is applied. Whenever the header of a natural loop is preceded by a fall-through transition of the same loop, the preheader cannot be placed positionally in front of the header, but rather has to be placed at a remote location. Thus, such a preheader is called a *remote preheader*. The last instruction of a remote preheader is an unconditional jump to the top of the loop. Such a branch is a candidate for an initial pass of code replication. Remote preheaders are explicitly excluded

Table 5.6: Function BubbleSort() in bubblesort.c.

#	without replication	with replication

1	m[6]=LK12;
	L[A[m[7]]]=MM7168;
	d[4]=L[A[m[6]+Elements.]];
	m[0]=A[4+_Array];
	PC=L56;
		m[6]=LK12;
		L[A[m[7]]]=MM7168;
		d[4]=L[A[m[6]+Elements.]];
		NZ=d[4]?2;
		PC=NZ<0,L07;

2		m[0]=A[4+_Array];

3	L57
	d[3]=1;
	d[2]=1;
	m[1]=m[0];
	PC=L60;
		L57
		d[3]=1;
		d[2]=1;
		d[1]=d[4];
		d[1]=d[1]-1;
		NZ=d[3]?d[1];
		PC=NZ>0,L59;

4		m[1]=m[0];
5	L61 d[0]=L[A[m[1]]]; NZ=d[0]?L[A[m[1]+4]]; PC=NZ<=0,L58;	L61 d[0]=L[A[m[1]]]; NZ=d[0]?L[A[m[1]+4]]; PC=NZ<=0,L58;
6	L[A[m[1]]]=L[A[m[1]+4]]; L[A[m[1]+4]]=d[0]; d[3]=0;	L[A[m[1]]]=L[A[m[1]+4]]; L[A[m[1]+4]]=d[0]; d[3]=0;

7	L58
	m[1]=m[1]+4;
	d[2]=d[2]+1;
		L58
		m[1]=m[1]+4;
		d[2]=d[2]+1;

8	L60
	d[1]=d[4];
	d[1]=d[1]-1;
	NZ=d[2]?d[1];
	PC=NZ<=0,L61;
		NZ=d[2]?d[1];
		PC=NZ<=0,L61;

9	NZ=d[3]?0; PC=NZ!=0,L07;	L59 NZ=d[3]?0; PC=NZ!=0,L07;
10	d[4]=d[1];	d[4]=d[1];
11	L56 NZ=d[4]?2; PC=NZ>=0,L57;	NZ=d[4]?2; PC=NZ>=0,L57;
12	L07 NL=MM7168,L[A[m[7]]]; m[6]=UK; PC=RT;	L07 NL=MM7168,L[A[m[7]]]; m[6]=UK; PC=RT;

from repeated replication to avoid the situation where a loop is replicated and a remote preheader is created which causes the same replication to be applied *ad infinitum.*

In Figure 5.1, an example of a loop is given without a preheader and after a preheader has to be created in a remote position. Blocks 3 and 4 comprise the loop. Also, block 3 falls through into block 4 and causes the preheader, block 0, to be placed at a remote location. The last instruction in block 0 is an unconditional branch to the header of the loop. Without the preheader, the loop could be entered by taking the conditional branch from block 1.

There are other instances when an unconditional branch cannot be removed. Indirect branches are explicitly excluded from being replicated at this implementation stage. Thus, any unconditional branch to the basic block containing the indirect branch cannot be optimized. This deficiency can be overcome by allowing indirect jumps and their jump tables to be replicated and the branch addresses to be recalculated.

In general, an unconditional branch cannot be replaced if a replication sequence does not exist. For endless loops for example, there is no transfer of control to a return statement.

Finally, code replication is not applied to other types of unconditional transfers of control such as function calls and return statements. Function calls and return statements are beyond the range of global optimization methods which operate across the basic block of a function but can be handled by inlining.

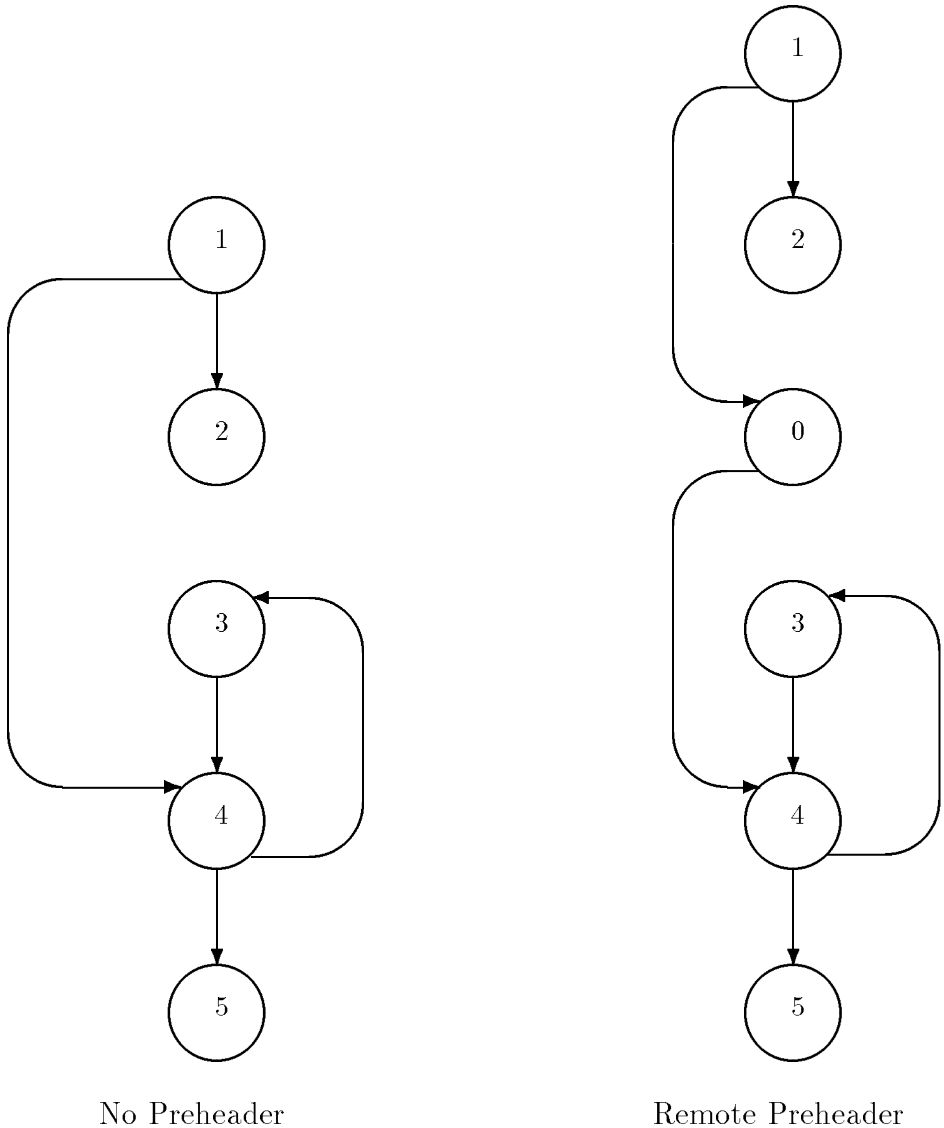


Figure 5.1: Introducing a Remote Preheader

Chapter 6

Algorithms

Two algorithms have been developed to evaluate code replication for this thesis. JUMPS is the algorithm for replacing unconditional branches by code replication. Occasionally, common subexpression elimination after code replication results in the comparison of two constants preceding a conditional branch. Therefore, CONSTS, an algorithm for constant folding at conditional branches, was also developed.

6.1 Code Replication

The algorithm JUMPS searches basic blocks for unconditional branches. Once an unconditional branch is found, the control flow of the program is traversed to replicate either the termination condition of a loop or any other sequence of basic blocks that includes no further unconditional branches. Thus, the algorithm JUMPS is a generalized technique of code replication applicable at any unconditional branch.

The traversal of basic blocks was chosen in order to detect all loops. When an unconditional branch is encountered, a search process is initiated to find either a return statement or a branch to the basic block positionally following the unconditional branch. The basic blocks leading to the end of a replication sequence are collected in a list if the search process was successful. The collected blocks are then inserted in the order of the traversal, and the control flow between the blocks is adjusted. In other words, the jump condition for conditional branches may have to be reversed and the label changed. Unconditional branches can be eliminated where the next block in the replication list is identical to the branch destination. In fact, there are no unconditional branches allowed in the replicated code after adjusting the control flow.

The example in Table 6.1 illustrates the sources for the two distinct replication sequences. The basic block number is indicated in the first column of Table 6.1; the other columns show the sequence of RTLs before replication occurs, after replication towards the return statement is performed, and, alternatively, after loop replication is attempted. The corresponding control flow graph is shown in Figure 6.1.

The replication for the two unconditional branches is performed as follows:

- The first problem in this example is how the unconditional branch in block 4 preceding label L90 can be replaced by replicating code.

  Favoring returns: The first sequence, terminated by the return statement, consists of the basic blocks 1 and 6. After replacing the unconditional branch with the RTLs of blocks 1 and 6, the control flow is adjusted in the copied blocks 3 and 4: The jump condition of block 3 is inverted and the label L09 is generated for block 2.

  Favoring loops: The replication sequence resulting in a loop consists of the basic blocks 1 and 2. The unconditional branch in block 4 is replaced by a copy of the two blocks, and the control flow is updated: The jump condition in the new block 4 is reversed and the label L08 is generated for the new block 3 to jump back to the top of the loop.

Table 6.1: Excerpt from Function wcp() in wc.c.

#	without replication	favor returns	favor loops
1	L83 NZ=B[A[m[2]]]?0; PC=NZ==0,L02;	NZ=B[A[m[2]]]?0; PC=NZ==0,L02;	NZ=B[A[m[2]]]?0; PC=NZ==0,L02;
2	b[0]=B[A[m[2]i]]; NZ=b[0]?99; PC=NZ==0,L90;	L09 b[0]=B[A[m[2]i]]; NZ=b[0]?99; PC=NZ==0,L90;	L09 b[0]=B[A[m[2]i]]; NZ=b[0]?99; PC=NZ==0,L90;
	...	...	L08 ...
3		NZ=B[A[m[2]]]?0; PC=NZ!=0,L09;	NZ=B[A[m[2]]]?0; PC=NZ==0,L02;
4	PC=L83;	m[6]=UK; PC=RT;	b[0]=B[A[m[2]i]]; NZ=b[0]?99; PC=NZ!=0,L08 ;
5	L90 L[A[dm[7]]]=d[4]; ST=A[_ipr]; m[7]=m[7]+4; PC=L83;	L90 L[A[dm[7]]]=d[4]; ST=A[_ipr]; m[7]=m[7]+4; NZ=B[A[m[2]]]?0; PC=NZ!=0,L09;	L90 L[A[dm[7]]]=d[4]; ST=A[_ipr]; m[7]=m[7]+4; NZ=B[A[m[2]]]?0; PC=NZ!=0,L09;
6	L02 m[6]=UK; PC=RT;	m[6]=UK; PC=RT;	L02 m[6]=UK; PC=RT;
		L02 m[6]=UK; PC=RT;	

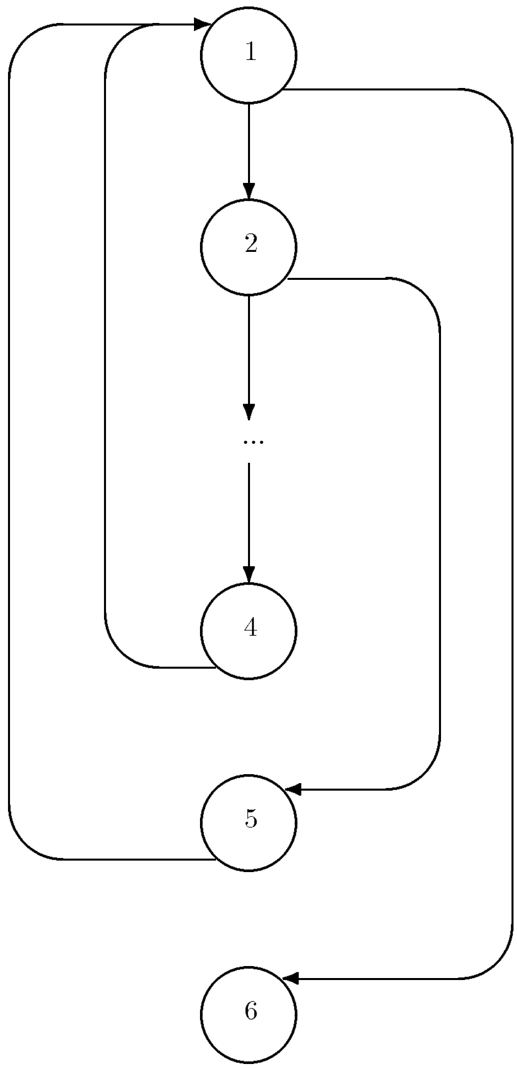


Figure 6.1: Control Flow Graph for Excerpt from Function wcp() in wc.c.

If two replication sequences have the same length, as in this case, loops are actually favored because back edges are more likely to be followed in the control flow.

- The second problem in this example is how the unconditional branch in block 5 preceding label L02 can be replaced by replicating code.

  Favoring returns: The replication sequence terminated by the return statement consists of the basic blocks 1 and 6 again. Thus, the two blocks are copied and the control flow is adjusted as before. The old version of block 6 with label L02 is still the last block positionally.

  Favoring loops: The replication sequence favoring loops includes block 1 only. After replicating block 1 and replacing the unconditional branch, the control flow is adjusted by inverting the conditional branch to create a fall-through transition between blocks 5 and 6.

  Notice that although the replication sequences differ, the final replacement for the second unconditional branch results in a loop starting at label L09 for both favoring loops and favoring returns. Nevertheless, the replication sequence for loops is shorter in this example and is preferred over the sequence favoring returns. In fact, the block structure in Table 6.1 illustrates the redundancy of creating a copy of block 6 (favoring returns) where a simple fall-through transition results in more compact code (favoring loops).

In general, the task of code replication is to find a sequence of basic blocks to replace an unconditional branch. One would have to examine all paths between any two blocks in order to determine which path results in the best replication sequence. This can be accomplished by algorithms such as depth-first search or breadth-first search which have an exponential complexity.

In order to avoid algorithms with a high degree of complexity, it was decided to use some initial assumptions. 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 basic blocks.[1]

Finding the shortest path in a graph with n nodes can be accomplished by using Warshall's algorithm for calculating the transitive closure of a graph [Wa62] which has a complexity of $O(n^3)$ where n denotes the number of nodes.[2]

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 together with their jump tables 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

[1] In a future implementation, it is intended to use the number of RTLs rather than the number of blocks to determine the shortest path

[2] Several improvements to the algorithm have been suggested to cut down on the complexity. Among the more recent efforts, an algorithm with an average complexity of $O(n^2 * lg(n))$ has been proposed by Bloniaz, Fischer, and Meyer [Bl76], and Schnorr described an algorithm running in $O(n + m^*)$ time where m^* is the expected number of edges in the transitive closure [Sc77]. The latter approach is preferable for relatively sparse connections. The original algorithm as suggested by Warshall was chosen for this thesis. For a depth-bounded search with a small depth on the other hand, an exhaustive search in a breadth-first or depth-first manner might be just as useful. But it was decided not to limit the work to a depth boundary thereby missing some lengthy replication sequences.

block traversals [Fl62]. In the end, the matrix can be used to look up the shortest path between two arbitrary basic blocks in the table without recalculating the control flow after each replication.

For the example in Table 6.1, with the control flow graph of Figure 6.1, the resulting connections are shown in Table 6.2. Each entry indicates the next block in the replication sequence and, in parentheses, the number of blocks left in the replication sequence. The latter information can be used for heuristics to decide which replication sequence should be preferred. The length of a path has to be available when setting up the table in order to decide if a connection is the shortest connection found up to this point and thus needs to be recorded.

Table 6.2: Connections between Basic Blocks

from/to	1	2	4	5	6
1	–	2 (0)		2 (1)	6 (0)
2	5 (1)	–		5 (0)	5 (2)
4	1 (0)	1 (1)	–	1 (2)	1 (2)
5	1 (0)	1 (1)		–	1 (1)
6	–	–	–	–	–

The replication sequence resulting in a loop for the unconditional branch in block 4 is found by examining the entry in row 4 and column 5. Informally, an attempt is made to "fill the gap" between the blocks 4 and 5. In other words, the question "Is there a connection between block 4 and 5 which results in a sequence of basic blocks with fall-through transitions?" can be answered by examining entry (4,5). This entry indicates that block 1 is the first block in the sequence and that the sequence will consist of 2 blocks. Having collected block 1, the new problem is to find a connection between block 1 and 5. By examing entry (1,5), block 2 is collected next. There is a direct connection between block 2 and block 5 since entry (2,5) indicated a remaining length of 0 so that the sequence is complete with blocks 1 and 2.

The alternative replication sequence for block 4 favoring returns is found similarly by examining the entries (4,6) and (1,6). The replication sequence then consists of blocks 1 and 6. In this case, both replication sequences for block 4 have a length of 2, and an empirical decision is made to favor loops. This decision is based on the assumption that creating loops is more likely to support the actual control flow at execution time since back edges are very likely to be taken.

The algorithm JUMPS is divided into different phases:

1. Initially, the information required to find the shortest sequence of basic blocks to replace an unconditional branch is set up.

2. In the second step, the basic blocks within a function are traversed sequentially and unconditional branches are replaced as follows. Either a sequence of blocks that ends with a return from the subroutine is replicated (*favoring returns*), or a sequence of blocks is chosen linking the current block containing the unconditional branch with the block positionally following the unconditional jump (*favoring loops*). The sequence of blocks to be replicated will fall through to the next block (see Table 6.1 and Table 6.2). At this point, heuristics can be plugged in to make the choice between these two options.

   In some cases, only one possible path for replication can be found because certain blocks might not be reachable from the current block, and in a few cases there might not be any replication possible at all.

3. Once a sequence of basic blocks is replicated, the control flow is adjusted accordingly. New labels are introduced, and the destinations of conditional branches are modified. In addition, all unconditional branches that are found in the replicated code can be eliminated since the sequence of blocks replicated had to follow the control flow with fall-through transitions. Recall that unconditional branches cannot occur in replicated code. Figure 6.2 shows the pseudo-code for adjusting the control flow.

4. As a result of the replication process, code which cannot be reached by the control flow anymore can be sometimes found. Therefore, dead code elimination is invoked to delete all these blocks. An example was given previously in Table 5.5.

6.2 Improvements

The method of strictly using the shortest path to find replication sequences discussed in the last section has the deficiency that natural loops are not necessarily preserved.

Figure 6.3 shows the control flow graph of a program portion before and after replication. An if-then-else statement consisting of blocks 1, 2, and 3, is followed by a natural loop. An if-then statement (blocks 5 and 6) is embedded in the loop. The dotted arc between block 2 and 4 indicates an unconditional branch.

```
let A be a block in the original code;
let F be the block positionally following A (fall through);
let B be the destination block of a conditional branch in A;
let A', B', and F' be replications of A, B, and F respectively;
let N' be a copied block positionally following A';

if (basic block A' is terminated by an unconditional branch) {
  remove the branch in A';
  create a fall-through transition to the following block N';
}
if (the basic block A' ends with a conditional branch)
  if (the next block N' is a copy of block B) {
    reverse the branch condition in A';
    if (block F was replicated)
      change the branch destination in A' to block F';
    else
      change the branch destination in A' to block F;
  }
  else {
    if (block B was replicated)
      change the branch destination in A' to block B';
    else
      change the branch destination in A' to block B;
  }

if (the basic block A' does not end with a branch instruction)
  create a fall-through transition to the block N';

if (the A' is the last block in the replication sequence)
  if (the last instruction in A' is a return statement)
    do not create any transitions to other blocks;
  else
    fall through to the block positionally following;
```

Figure 6.2: Adjusting the Control Flow

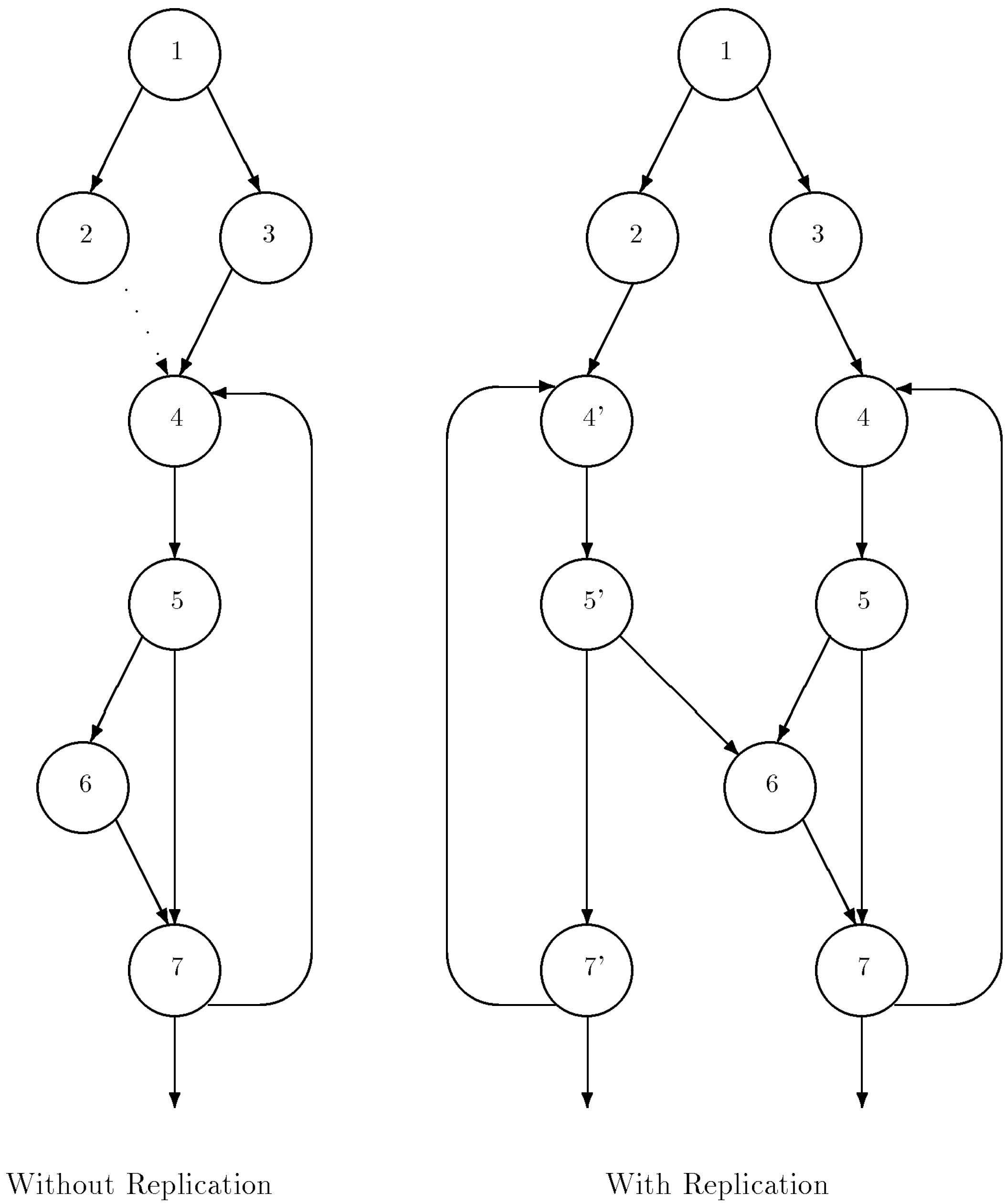


Figure 6.3: Interference with Natural Loops

After replication, a copy of the loop is created below block 2. Block 6 was not copied since only the blocks on the shortest path were collected for replication. Thus, the original loop consisting of the blocks 4 to 7 is no longer a natural loop since the loop can be entered either from block 4 or from block 6. Nevertheless, the replicated blocks form a natural loop with block 4', 5', and 7'.

Since unnatural loops are not recognized as loops, some valuable optimization techniques such as code motion and strength reduction cannot be applied. Thus, it is undesirable to introduce unnatural loops. Furthermore, had block 6 been included in the replication sequence, two independent natural loops would have resulted as illustrated in the second graph of Figure 6.3. The original loop with blocks 4 to 6 would be preserved, and a one-to-one copy consisting of block 4' to 6' would be introduced as before.

This deficiency motivated a variation on the implementation discussed in the last section. When determining the replication sequence in the algorithm for code replication, the blocks on the shortest path are collected. A variation in the collection method resolves the above problem.

An informal description of the modifications to the algorithm JUMPS is given below.

1. When traversing the shortest path, any block on this path is collected as before (see algorithm JUMPS). If a collected block 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. Then, the process of collecting blocks follows the original path again.

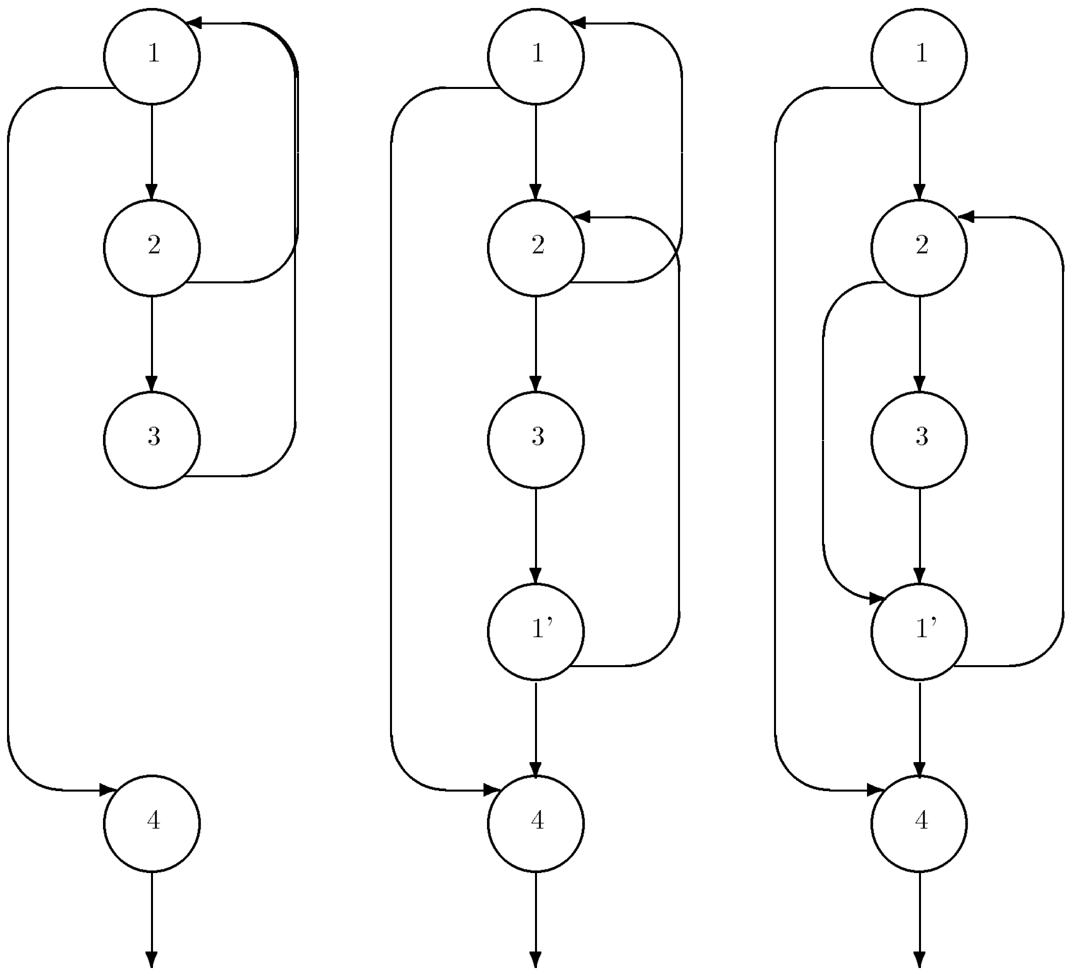
2. In addition, the adjustment of the control flow is extended in two ways:

   - 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. This distinction is needed to avoid the partial overlapping of natural loops.

An example for partially overlapping loops is given in Figure 6.4. After code replication, the unconditional jump at the end of block 3 is removed and a fall-through transition to the copied block 1' is added. The control flow of the back-edges results in two natural loops, the first one consisting of blocks 1, 2, 3, and 1', and the second loop including blocks 2, 3, and 1'. Block 2 is the header of the second loop but has a back edge to the header of the first loop so that the back edges are partially overlapping. After adjusting the control flow, the problem of overlapping loops is resolved. Then, there is only one loop (blocks 2, 3, and 1') since the former transition 2→1 was changed to the transition 2→1'.

Usually, partially overlapping loops do not occur in programs. With conventional optimizations, partially overlapping loops can only be caused by goto statements. Loop optimization methods cannot take complete advantage of overlapping loops for various reasons. First, the nesting level of such loops



Initial Control Flow After Replication Adjusted Control Flow

Figure 6.4: Partial Overlapping of Natural Loops

is difficult to represent accurately which can effect global register allocation. Second, loop optimizations such as code motion and strength reduction are generally implemented to work on one loop at a time. But overlapping loops may have to be handled differently to take into account changes which affect both loops. Since optimizing compilers may not optimize partially overlapping loops well, it was decided to eliminate such loops by adjusting the control flow as discussed previously.

Notice that the improved version of the algorithm JUMPS requires complete loops to be copied, which results in an additional increase of the code size by about 3%. The results of the improved version of JUMPS are evaluated in the next chapter.

6.3 Removal of Constant Comparisons

CONSTS is an algorithm eliminating the comparison of two constants. The list of basic blocks is traversed in their positional order and searched for conditional branches. If a conditional branch is encountered, the compiler checks if the comparison instruction setting the condition codes was based on the comparison of two constants or on the comparison of the same two objects. If this is the case, the compare instruction is eliminated and, depending on the branch condition, the branch is either removed if it was never taken, or replaced by an unconditional branch if the conditional branch was always taken. If an unconditional branch is introduced, it is subject for removal by a later iteration of code replication.

The example shown in Table 6.3 illustrates the initial replication of the conditional branch to replace the unconditional branch at the end of the loop.

The first conditional branch is moved into the preheader of the loop and, after common subexpression elimination (CSE) is performed, can be eliminated since the branch is never taken due to the comparison between the constants 100 and 1000.

Table 6.3: Excerpt from Function main() in quicksort.c.

without replication	after JUMPS	after CONSTS
d[2]=100;	d[2]=100;	d[2]=100;
...	...	...
L23	NZ=d[2]?1000;	
NZ=d[2]?1000;	PC=NZ>0,L02;	
PC=NZ>0,L02;	L00	L00
...(loop body)	...(loop body)	...(loop body)
d[2]=d[2]+100;	d[2]=d[2]+100;	d[2]=d[2]+100;
	NZ=d[2]?1000;	NZ=d[2]?1000;
PC=L23;	PC=NZ<=0,L00;	PC=NZ<=0,L00;
L02	L02	
m[6]=UK;	m[6]=UK;	m[6]=UK;
PC=RT;	PC=RT;	PC=RT;

6.4 Integration into an Optimizing Compiler

This section summarizes the different algorithms for code replication and their relation to further opportunities for optimizations. Furthermore, the order of the performed optimizations is given.

Two different algorithms for code replication were implemented in the experimental environment. One handles replication for natural loops only. The other, algorithm JUMPS, replicates code whenever an unconditional jump is encountered which can be replaced.

1. Algorithm LOOPS: Unconditional jumps are often needed to either enter a natural loop or jump back to the header of such a loop. This unconditional branch is replaced by the termination condition of the loop. The replicated condition has to be inverted if copied into the preheader of the loop. In fact, by replicating the termination condition, the opportunities for other optimizations such as common subexpression elimination are improved. Depending on the original layout of the loop, either one unconditional jump is removed at the entry point, or one unconditional branch is saved per loop iteration. This algorithm also handles nested loops.

2. Algorithm JUMPS: This algorithm was discussed in the previous sections of this chapter. It is a generalized approach which attempts to replace any occurrences of unconditional branches by replicating code. Again, sources for further optimizations such as common subexpression elimination and register coloring are produced.

The traditional approach, algorithm LOOPS, was compared with the algorithm JUMPS with respect to its impact on optimizing the code of programs.

In addition, an algorithm CONSTS to perform constant folding at conditional branches was implemented. This algorithm may introduce unconditional branches which then become a candidate for removal by the code replication algorithm JUMPS.

The code replication algorithms 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 branches. 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.

After most unconditional branches have been removed, instruction selection is performed followed by the common optimization phases such as register coloring, common subexpression elimination, code motion, and strength reduction.

Then, CONSTS is invoked to replace or eliminate comparisons of constants together with the associated branches.

In order to replace all unconditional branches generated by CONSTS or introduced by remote preheaders, code replication is reinvoked. If any improvements to the RTLs were applied since register coloring was performed, a set of optimizations including register coloring, common subexpression elimination, and code replication, will be reapplied.

The order in which the optimizations are applied is summarized in Figure 6.5.

```
branch chaining;
dead code elimination;
jump minimization by reordering basic blocks;
code replication (JUMPS or LOOPS alternatively);
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 (CONSTS);
  code replication (JUMPS or LOOPS alternatively);
  dead code elimination;
} while (change);
```

Figure 6.5: Order of Optimizations

Chapter 7

Measurements

Dynamic and static measurements were taken from a number of well-known benchmarks, UNIX utilities, and one application (see Table 7.1). The code was generated for the Motorola 68020 processor [Mo85]. Also, the impact of code replication on instruction caching was investigated. All measurements include 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 (algorithm CONSTS). Library routines could not be measured since the source code was not available to be compiled by VPO.

Each program was tested with three different sets of optimizations:

- SIMPLE: Only the standard optimizations were performed.
- LOOPS: Standard optimizations and replication within natural loops with respect to the termination condition (algorithm LOOPS) were invoked.
- ALL: All possible replications (algorithm JUMPS) including the improvements discussed in the last chapter were performed together with the standard optimizations.

Table 7.1: Test Set of C Programs

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	ackerman	ackerman function
	bubblesort	sort numbers
	matmult	matrix multiplication
	sieve	iteration
	queens	8-queens problem
	quicksort	sort numbers (iterative)
User code	mincost	VLSI circuit partitioning

7.1 Static and Dynamic Behavior

Table 7.2 shows the number of unconditional branches relative to the total number of instructions for the static and dynamic measurements. Only unconditional direct jumps to a designated label are counted while indirect branches, call instructions, and return statements are excluded. The number of unconditional branches is reduced by about 36.9% dynamically when LOOPS was applied, and with code replication practically no unconditional branches are left. Thus, code replication results in a reduction of instructions executed by at least the number of unconditional branches which could be avoided dynamically.

The unconditional branches of the program "grep" which could not be removed by code replication are transfers to a basic block including an indirect branch. Since indirect branches were excluded from being replicated, no repli-

Table 7.2: Percent of Unconditional Branches

program	static			dynamic		
	SIMPLE	LOOPS	ALL	SIMPLE	LOOPS	ALL
cal	1.86	0.89	0.00	3.16	0.07	0.00
quicksort	3.67	1.18	0.00	1.14	0.11	0.00
wc	6.94	4.10	0.00	7.07	7.07	0.00
grep	8.39	7.29	0.59	3.40	3.38	1.69
sort	6.87	3.91	0.00	12.22	8.94	0.00
od	6.34	4.80	0.00	5.78	3.45	0.00
mincost	5.08	2.76	0.00	2.50	1.40	0.00
bubblesort	4.38	0.70	0.00	0.05	0.00	0.00
ackerman	2.78	2.78	0.00	3.86	3.86	0.00
matmult	4.11	0.00	0.00	0.21	0.00	0.00
banner	5.65	1.89	0.00	4.12	2.47	0.00
sieve	1.43	0.00	0.00	8.53	0.00	0.00
compact	5.42	4.51	0.00	2.53	1.21	0.00
queens	1.06	1.06	0.00	0.05	0.05	0.00
deroff	9.95	9.05	0.03	7.21	7.18	0.18
average	4.93	3.00	0.04	4.12	2.61	0.12

cation sequence can be found for the unconditional transfers to this basic block.

The columns SIMPLE in Table 7.3 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. The code size in bytes is shown in Table A.1 in Appendix A. When LOOPS is applied, the number of instructions increases by only 5.4%. With generalized code replication, on the other hand, about an average of 42.5% more instructions are generated.

The change of the code size is directly related to the total number of unconditional branches and the cause of the unconditional branch. The program "deroff" initially has 570 unconditional branches, and almost all of these bran-

Table 7.3: Total Number and Percent Change of Instructions

program	static instructions			dynamic instructions executed		
	SIMPLE	LOOPS	ALL	SIMPLE	LOOPS	ALL
cal	323	+4.33%	+13.93%	36,290	−3.09%	−3.11%
quicksort	245	+3.67%	+18.37%	536,566	−0.39%	−3.96%
wc	173	+12.72%	+63.01%	421,038	−0.00%	−4.58%
grep	775	+4.39%	+74.97%	1,309,586	−0.03%	−3.41%
sort	1,558	+13.16%	+56.35%	902,075	−8.94%	−12.51%
od	1,198	+2.67%	+71.45%	1,980,808	−2.59%	−10.30%
mincost	906	+4.08%	+31.13%	302,062	−1.53%	−5.76%
bubblesort	137	+3.65%	+2.92%	20,340,231	−18.92%	−18.92%
ackerman	72	+0.00%	+2.78%	2,239,579	−0.00%	−3.86%
matmult	146	+3.42%	+3.42%	4,891,507	−0.21%	−0.21%
banner	177	+19.77%	+38.98%	2,473	−1.66%	−8.21%
sieve	70	+1.43%	+1.43%	1,759,088	−8.53%	−8.53%
compact	1,143	+2.89%	+78.65%	10,602,159	−0.42%	−4.16%
queens	94	+0.00%	+12.77%	189,518	−0.00%	−0.05%
deroff	5,730	+5.08%	+167.61%	360,051	−0.42%	−7.12%
average	850	+5.42%	+42.52%	3,058,202	−3.12%	−6.31%

ches are avoided by code replication. The result, however, was an increase of the number of instructions by a factor of 2.7. Most of the unconditional jumps in the program were not due to loops, and the resulting replication sequences occasionally became lengthy.

The dynamic measurements in Table 7.3 show the change of the number of instructions executed. There is a 6.3% dynamic decrease when code replication is invoked, while LOOPS exceeds a 3.1% reduction. Compared with the simple approach of eliminating unconditional branches in case of natural loops, the number of instructions executed is reduced by another 3.2% in average. Thus, only about half of the dynamic improvement is due to the traditional method of removing unconditional branches at loops. The rest of the gain occurs when the generalized version of code replication is applied.

Table 7.4 indicates the average number of instructions executed between two branch instructions in the first column (SIMPLE) and its change in percent for LOOPS and ALL. A 28.3% increase in the number of instructions executed between branches is measured so that an average of 6.5 instructions is found between branches after code replication was applied. Thus, the opportunities for instruction scheduling may improve if code replication is applied in a RISC environment. Or, in a large instruction word (LIW) environment more potential parallelism can be found due to the increase of the basic block size [Ri72]. Also, instruction caches may perform more efficiently when instructions are requested from consecutive memory locations and branches occur less frequently.

Table 7.4: Total Number and Percent Change of Instructions between Branches

program	SIMPLE	LOOPS	ALL
cal	5.90	−2.03%	−13.56%
quicksort	5.47	+10.24%	+7.13%
wc	4.35	+0.00%	+173.79%
grep	5.50	+0.00%	+29.64%
sort	4.90	−8.37%	+46.33%
od	6.05	+9.42%	+34.55%
mincost	5.23	+10.13%	+19.31%
bubblesort	5.96	−21.64%	−21.48%
ackerman	2.71	+0.00%	+16.97%
matmult	9.71	+5.05%	+5.05%
banner	5.23	+22.18%	+33.65%
sieve	3.45	−6.67%	−6.67%
compact	6.33	+10.90%	+23.38%
queens	4.40	+0.00%	+0.23%
deroff	2.93	+0.00%	+76.11%
average	5.21	+1.95%	+28.30%

7.2 Impact on Instruction Caching

The cache performance was tested for cache sizes of 1Kb, 2Kb, 4Kb, and 8Kb. The number of lines were 64, 128, 256, and 512, respectively. 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 when instruction prefetching is performed. Thus, fetch cost is calculated as follows:

$$fetch\ cost = cache\ hits * cache\ access\ time + cache\ misses * miss\ penalty$$

where the cache access time is 1 time unit and the miss penalty is 10 units of time.

Context-switches were simulated, invalidating the entire cache every 10000 units of work with respect to the fetch cost. 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 cost of instruction fetching can decrease while the miss ratio increases for the same program. Such a behavior 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 dynamic behavior of a program changes.

The relatively small cache sizes were chosen to simulate a realistic environment where programs do not fit completely in the cache. Recall that library routines were not measured so that the simulated cache sizes correspond to larger caches as found in a realistic environment.

The simulation of instruction caching required the insertion of trace code for each program. Many instruction sets support different branch instructions for short jumps and for long jumps. Often, the selection of the branch instruction is made by the assembler, and a compiler might not be able to analyze the actual size of all instructions. To avoid inaccurate results, a two-pass compilation was implemented for the cache analysis. In the first pass, the size of each instruction is determined by the assembler without inserting trace code. Rather than inspecting the object code and using a table lookup method, which may be rather complicated for a highly encoded architecture, a label is inserted before and after each instruction. The assembler is used to determine the size of each instruction, the difference between each pair of labels. In the second pass, trace instructions are inserted into the program code but the instruction sizes are actually taken from a file generated by the first pass.

A collection of tables showing the cache measurements and the program sizes can be found in the Appendices A and B. The following cache configurations were varied:

- *size:* Caches with sizes of 1Kb, 2Kb, 4Kb, and 8Kb were investigated.
- *context switching:* Measurements were taken with and without simulating context switching to evaluate the impact on multi-tasking systems and single-tasking system such as personal computers.
- *alignment:* Unaligned instructions and aligned instructions were simulated. The size of aligned instructions was set to 4 bytes. Aligned instructions simulate the impact of caching on reduced instruction sets.

Table 7.5 shows the change of the fetch cost for a 1Kb direct-mapped cache. The measurements of the SIMPLE version were related to the programs where replication occurred only at loops and those where code replication was performed. In other words, each set of measurements is compared to the corresponding values of the SIMPLE version with the same configuration.

Table 7.5: Percent Change in Instruction Fetch Cost for 1Kb Direct-Mapped Cache

	unaligned instructions				aligned instructions			
context sw	on		off		on		off	
program	LOOPS	ALL	LOOPS	ALL	LOOPS	ALL	LOOPS	ALL
cal	−1.78	−3.26	−1.72	−3.41	−2.72	−2.39	−2.97	−1.90
quicksort	−3.01	−0.75	−2.98	−0.82	−0.37	−3.82	−0.38	−3.83
wc	+0.00	−6.38	+0.00	−6.99	+0.01	−3.88	+0.01	−4.55
grep	+4.78	−1.00	+4.86	−0.73	+0.02	+26.40	−0.02	+26.29
sort	+7.44	−12.67	+7.05	−13.07	−21.55	−27.24	−21.69	−27.49
od	−3.89	+14.19	−3.68	+15.34	−14.33	−29.33	−14.59	−29.19
mincost	+1.16	+13.89	+1.14	+14.94	+2.90	+8.90	+3.13	+9.22
bubblesort	−25.98	−25.95	−25.92	−25.89	−18.91	−18.91	−18.92	−18.92
ackerman	+0.00	−0.02	+0.00	−0.02	+0.00	−3.77	+0.00	−3.86
matmult	−0.10	−0.10	−0.19	−0.19	−0.21	−0.21	−0.21	−0.21
banner	−0.11	−6.22	−0.11	−6.22	−1.18	−8.07	−1.18	−8.07
sieve	−11.64	−11.64	−11.65	−11.65	−8.52	−8.52	−8.53	−8.53
compact	+2.21	+25.02	+2.06	+25.51	−13.62	−8.58	−14.70	−8.84
queens	+0.00	+12.89	+0.00	+12.60	+0.00	+0.20	+0.00	−0.03
deroff	−1.22	+2.67	−0.88	+3.56	+0.71	−2.40	+0.30	−2.10
average	−2.14	+0.04	−2.13	+0.20	−5.18	−5.44	−5.32	−5.47

For unaligned instructions, code replication does not change the average instruction fetch cost significantly while a reduction of 2.1% is found for LOOPS. The findings for code replication can be explained by an increase in the size of some loops such that a loop occupies more lines after applying code replication than before. Thus, each time the instructions of such a loop are fetched, additional hits or misses will be caused by the replicated lines of the loop (see

Figure 7.1). Notice that some programs initially fit in the cache, but after code replication is applied, they might not fit anymore. Therefore, capacity misses can also be introduced.

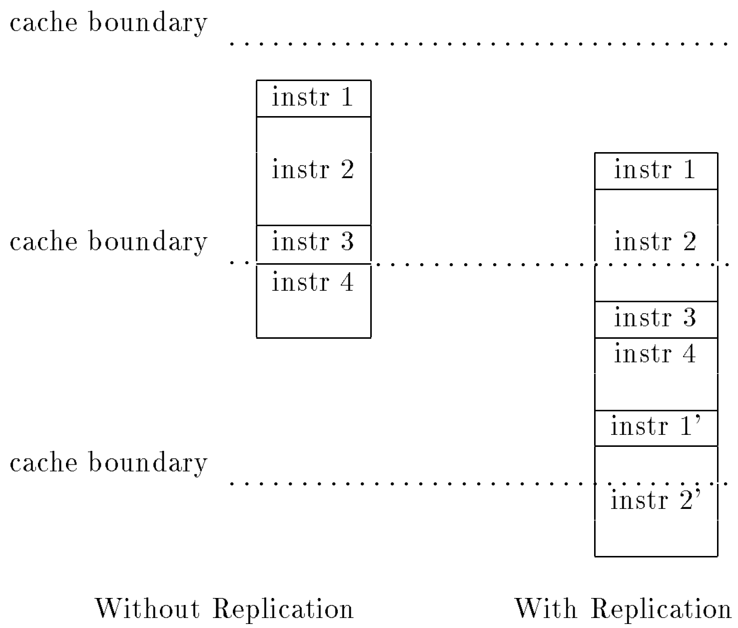


Figure 7.1: Instruction Alignment in a Cache

An unaligned instruction can also span a line boundary causing two cache references, one for each line, whenever the instruction has to be fetched (see instructions 2 and 2' in the replicated code of Figure 7.1). With instruction alignment, an instruction spanning two adjacent lines can be avoided if the size of all instructions are the same and if the line size is an integer multiple of the instruction size. Therefore, aligned instructions were simulated by fixing the instruction size to 4 bytes.

For example, the instruction fetch cost for the program "compact" increases by 25% for unaligned instructions in a 1Kb cache. But 16% of the instruction fetches are due to alignment problems where instructions span more than one line. Table C.1 in the Appendix C illustrates that the percentage of instructions causing alignment problems increased by 6% for generalized code replication compared to the SIMPLE version.[1] In fact, after aligning the instructions the fetch cost for the program "compact" is reduced by 8.5%. The program "bubblesort" is an example for a drastically reduced fetch cost for unaligned instructions with JUMPS being applied, but at the same time 8.6% less instructions caused alignment problems compared to SIMPLE. In this case, the reduction in fetch cost for unaligned instructions is partially due to instructions spanning more than one line. Overall, unaligned instructions can cause unpredictable cache measurements because their impact can dominate the changes of cache measurements caused by code replication.

For aligned instructions, both replication techniques (LOOPS and ALL) reduce the instruction fetch cost by about 5%. The generalized replication algorithm JUMPS performs slightly better than LOOPS. This result is more conclusive since unpredictable side-effects of instructions spanning more than one cache line cannot occur.

But not all programs improved with respect to the fetch cost: Although the number of instructions executed for the program "grep" decreased by 3.4% after applying JUMPS, the fetch cost increased by 26% for aligned instructions and the static number of instructions increased by 75%. In this case, the cost of the extra misses outweights the reduction in the number of instructions executed.

[1] The values in Table C.1 are derived by calculating the difference between the number of references of unaligned instructions and the number of references of aligned instructions. Then, these additional references are related to the total number of references of unaligned instructions resulting in the "percent of additional instructions." The last two columns represent the difference of columns 2, 3 and columns 2, 4 respectively.

This shows that even if the number of instructions executed decreases there can be a negative impact on the cache performance mostly due to expanded loops.

The impact on context switching was minimal. Recall that each test set was compared to the corresponding SIMPLE version with the same configuration. Comparable changes in the measurements were found with and without context switching. In fact, the miss ratio only increased slightly with context switching on.

The tables listing the cache miss ratio in the Appendices A and B show that the misses increase by 0.02-0.84% in average for unaligned instructions depending on the cache size. For replication at loops, the increase varies between 0.02% and 0.14%. When instructions are aligned, the miss ratio recorded for code replication is almost the same as the miss ratio for the SIMPLE version. For a 2Kb cache a 0.02-0.24% reduction in the ratio has even been found for generalized code replication. If replication at loops is applied only, the reduction of the miss ratio varies between 0.42% and 0% for aligned instructions. Recall that the miss ratio can be a misleading measurement when the number of instructions changes. For example, for a 1Kb cache with context switching and aligned instructions, the program "banner" has a 0.21% increase in the miss ratio while the fetch cost decreases by 8.1% due to the reduced number of instructions executed after code replication.

The technique of code replication causes instructions to be laid out differently. Before replication, the instruction before an unconditional branch will be executed shortly before the instruction at the target address of the jump but their locations can be distant. After code replication, the two instructions will be executed sequentially as before but they now occupy adjacent memory locations. Thus, 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.

As discussed earlier, the code replication technique increases the size of some loop bodies so that more instructions can be found in a loop after code replication is applied. If a particular block inside a loop was duplicated, the execution may alternate between the original block and its duplicate. Thus, the temporal locality of a program can be degraded.

Nevertheless, code replication reduces the total number of instructions executed such that the average fetch cost is actually reduced for aligned instructions in particular.

Chapter 8

Future Work

Currently, indirect branches are excluded from replication. The algorithm for code replication could be extended to copy indirect jumps, the associated jump tables, and adjust the control flow accordingly. But not only the branch table would have to be replicated, all basic blocks which can be the destination of the indirect branch would have to be copied. The effect of such a massive replication has to be evaluated to determine if replication at indirect jumps is really desirable.

The replication sequence for unconditional branches is determined by finding the shortest path with respect to the number of basic blocks in the current implementation. Instead, the number of RTLs could be used as a cost function to determine the shortest path. The effect of this approach should be compared to the current results.

Invoking code replication at a later stage in the optimizing phase of the compiler might reduce the compile-time overhead considerably. It should be determined whether sources for other optimizations are reduced by such a change and whether the taken measurements are affected.

The measurements presented in the last chapter show that the cache work can be reduced on processors with a uniform instruction alignment. Thus, the impact of code replication on reduced instruction sets should be evaluated to get more information about the impact of code replication on RISC architectures. Furthermore, the elimination of unconditional branches may improve the schedulability of instructions. The execution of branch instructions introduces problems with pipelines because the destination is not known at the time of the fetch of the next instruction. This problem can be resolved by delaying the effect of branches by one or two pipeline cycles and scheduling non-interfering instructions to fill the unused pipeline slots. Instruction scheduling can also be used to handle delays of load and store instructions. For example, an attempt is made to move a load instruction for a register away from the next use of the same register to avoid interlocks caused by the delayed fetch. If no candidates for instruction scheduling can be found, the advantages of pipelines cannot be fully exploited. The average number of instructions between branches increases with the use of code replication. Thus, it becomes easier to fill delay slots for certain reduced instructions which improves the utilization of an instruction pipeline.

Lately, several attempts have been made to reorganize the positional order of basic blocks based on profiling data [Mc89,Pe90]. Code replication could take advantage of profiling data in two ways. First, the selection of basic blocks for a replication sequence can be modified to favor frequently executed blocks rather than the shortest path. Second, profiling data can be used to decide if a particular replication should not be performed because the unconditional branch being eliminated is rarely executed.

Chapter 9

Conclusions

A new global optimization method called code replication was developed which can be applied to eliminate almost all unconditional branches in a program. The resulting programs are executing 6.3% less instructions in average, thus generalized replication outperforms replication techniques applied to loops only. The number of instructions between branches is increased by 28.3% on the average, so that the opportunities for instruction scheduling in a RISC environment are improved. The cache work decreases by 5-6% for aligned instructions but actually increases slightly for unaligned instructions. The static number of instructions increases by an average of 42.5%. The direct compile time overhead for the replication process itself was found to be minimal, but following optimization stages must process more RTLs, which increases the overall compile time of the optimizer by an average of 72.5% (depending on the number of replicated RTLs).

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. Even more gain is expected by the technique when applied in conjunction with profiling data. The results of the test set also show that the replication of the jump condition at natural loops on its own, commonly performed in optimizing compilers, results in about 50% of the gains which can be achieved by generalized code replication.

Appendix A

Cache Measurements for Unaligned Instructions

Table A.1: Program Size in Bytes and Percent Change for Unaligned Instructions

	size in bytes			change of size	
program	SIMPLE	LOOPS	ALL	LOOPS	ALL
cal	1,172	1,206	1,300	2.90%	10.92%
quicksort	824	848	964	2.91%	16.99%
wc	664	722	1,084	8.73%	63.25%
grep	2,532	2,668	4,674	5.37%	84.60%
sort	5,272	5,912	8,268	12.14%	56.83%
od	4,126	4,244	7,208	2.86%	74.70%
mincost	3,172	3,312	4,254	4.41%	34.11%
bubblesort	498	512	510	2.81%	2.41%
ackerman	284	284	288	0.00%	1.41%
matmult	558	570	570	2.15%	2.15%
banner	512	610	714	19.14%	39.45%
sieve	260	266	266	2.31%	2.31%
compact	4,216	4,338	7,574	2.89%	79.65%
queens	314	314	354	0.00%	12.74%
deroff	21,524	22,658	61,112	5.27%	183.92%
average	3,062	3,231	6,609	4.93%	44.36%

Table A.2: Change of Miss Ratio (Context Switch, Unaligned)

cache size	1Kb		2Kb		4Kb		8Kb	
program	LOOPS	ALL	LOOPS	ALL	LOOPS	ALL	LOOPS	ALL
cal	+0.00	+0.04	+0.00	+0.04	+0.00	+0.04	+0.00	+0.04
quicksort	−0.00	+0.01	−0.00	+0.01	−0.00	+0.01	−0.00	+0.01
wc	+0.00	+0.08	+0.00	+0.08	+0.00	+0.08	+0.00	+0.08
grep	−0.01	+0.08	−0.01	+0.02	−0.01	+0.02	−0.01	+0.02
sort	+1.52	+0.06	+0.04	+0.41	+0.04	+0.06	+0.04	+0.06
od	−0.60	+4.03	−0.84	+3.20	+0.01	−0.01	+0.01	−0.01
mincost	+0.60	+3.56	+0.85	+2.69	+0.05	+0.33	+0.05	+0.33
bubblesort	−0.01	−0.01	−0.01	−0.01	−0.01	−0.01	−0.01	−0.01
ackerman	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
matmult	+0.01	+0.01	+0.01	+0.01	+0.01	+0.01	+0.01	+0.01
banner	+0.09	+0.29	+0.09	+0.29	+0.09	+0.29	+0.09	+0.29
sieve	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
compact	+0.00	+3.74	+1.57	+0.64	+1.80	+0.24	+0.04	+0.24
queens	+0.00	+0.03	+0.00	+0.03	+0.00	+0.03	+0.00	+0.03
deroff	+0.18	+0.71	+0.06	+0.09	+0.16	+0.12	+0.10	+0.02
average	+0.12	+0.84	+0.12	+0.50	+0.14	+0.08	+0.02	+0.07

Table A.3: Change of Miss Ratio (No Context Switch, Unaligned)

cache size	1Kb		2Kb		4Kb		8Kb	
program	LOOPS	ALL	LOOPS	ALL	LOOPS	ALL	LOOPS	ALL
cal	+0.01	+0.02	+0.01	+0.02	+0.01	+0.02	+0.01	+0.02
quicksort	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
wc	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
grep	−0.00	+0.11	+0.00	+0.00	−0.00	+0.00	−0.00	+0.00
sort	+1.45	+0.01	+0.00	+0.37	+0.00	+0.01	+0.00	+0.01
od	−0.55	+4.09	−0.82	+3.08	+0.00	+0.00	+0.00	+0.00
mincost	+0.58	+3.61	+0.80	+2.44	+0.00	+0.02	+0.00	+0.02
bubblesort	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
ackerman	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
matmult	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
banner	+0.09	+0.29	+0.09	+0.29	+0.09	+0.29	+0.09	+0.29
sieve	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
compact	−0.02	+3.68	+1.57	+0.42	+1.75	−0.00	−0.00	+0.00
queens	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
deroff	+0.22	+0.79	+0.07	+0.10	+0.23	+0.15	+0.17	+0.02
average	+0.12	+0.84	+0.11	+0.45	+0.14	+0.03	+0.02	+0.02

Table A.4: Percent Change of Instruction Fetch Cost (Context Switch, Unaligned)

cache size	1Kb		2Kb		4Kb		8Kb	
program	LOOPS	ALL	LOOPS	ALL	LOOPS	ALL	LOOPS	ALL
cal	−1.78	−3.26	−1.78	−3.26	−1.78	−3.26	−1.78	−3.26
quicksort	−3.01	−0.75	−3.01	−0.75	−3.01	−0.75	−3.01	−0.75
wc	+0.00	−6.38	+0.00	−6.38	+0.00	−6.38	+0.00	−6.38
grep	+4.78	−1.00	+4.77	−1.51	+4.77	−1.51	+4.77	−1.51
sort	+7.44	−12.67	−4.62	−9.94	−4.62	−12.68	−4.62	−12.68
od	−3.89	+14.19	−5.57	+9.56	+0.14	−9.89	+0.14	−9.89
mincost	+1.16	+13.89	+4.53	+14.99	−1.93	−3.31	−1.93	−3.31
bubblesort	−25.98	−25.95	−25.98	−25.95	−25.98	−25.95	−25.98	−25.95
ackerman	+0.00	−0.02	+0.00	−0.02	+0.00	−0.02	+0.00	−0.02
matmult	−0.10	−0.10	−0.10	−0.10	−0.10	−0.10	−0.10	−0.10
banner	−0.11	−6.22	−0.11	−6.22	−0.11	−6.22	−0.11	−6.22
sieve	−11.64	−11.64	−11.64	−11.64	−11.64	−11.64	−11.64	−11.64
compact	+2.21	+25.02	+13.79	+6.28	+17.76	+3.64	+2.56	+3.67
queens	+0.00	+12.89	+0.00	+12.89	+0.00	+12.89	+0.00	+12.89
deroff	−1.22	+2.67	−2.18	−2.28	−1.38	−2.08	−1.87	−2.87
average	−2.14	+0.04	−2.13	−1.62	−1.86	−4.48	−2.90	−4.53

Table A.5: Percent Change of Instruction Fetch Cost (No Context Switch, Unaligned)

cache size	1Kb		2Kb		4Kb		8Kb	
program	LOOPS	ALL	LOOPS	ALL	LOOPS	ALL	LOOPS	ALL
cal	−1.72	−3.41	−1.72	−3.41	−1.72	−3.41	−1.72	−3.41
quicksort	−2.98	−0.82	−2.98	−0.82	−2.98	−0.82	−2.98	−0.82
wc	+0.00	−6.99	+0.00	−6.99	+0.00	−6.99	+0.00	−6.99
grep	+4.86	−0.73	+4.88	−1.66	+4.87	−1.66	+4.87	−1.66
sort	+7.05	−13.07	−4.95	−10.25	−4.95	−13.08	−4.95	−13.08
od	−3.68	+15.34	−5.64	+9.48	+0.06	−9.81	+0.06	−9.81
mincost	+1.14	+14.94	+4.67	+14.64	−2.28	−5.75	−2.28	−5.76
bubblesort	−25.92	−25.89	−25.92	−25.89	−25.92	−25.89	−25.92	−25.89
ackerman	+0.00	−0.02	+0.00	−0.02	+0.00	−0.02	+0.00	−0.02
matmult	−0.19	−0.19	−0.19	−0.19	−0.19	−0.19	−0.19	−0.19
banner	−0.11	−6.22	−0.11	−6.22	−0.11	−6.22	−0.11	−6.22
sieve	−11.65	−11.65	−11.65	−11.65	−11.65	−11.65	−11.65	−11.65
compact	+2.06	+25.51	+14.35	+4.78	+18.30	+1.53	+2.20	+1.58
queens	+0.00	+12.60	+0.00	+12.60	+0.00	+12.60	+0.00	+12.60
deroff	−0.88	+3.56	−2.08	−2.23	−0.73	−1.71	−1.18	−2.83
average	−2.13	+0.20	−2.09	−1.85	−1.82	−4.87	−2.92	−4.94

Appendix B

Cache Measurements for Aligned Instructions

Table B.1: Program Size in Bytes and Percent Change for Aligned Instructions

	size in bytes			change of size	
program	SIMPLE	LOOPS	ALL	LOOPS	ALL
cal	1,288	1,340	1,464	4.04%	13.66%
quicksort	976	1,012	1,156	3.69%	18.44%
wc	688	776	1,124	12.79%	63.37%
grep	3,068	3,204	5,388	4.43%	75.62%
sort	6,224	7,044	9,712	13.17%	56.04%
od	4,768	4,884	8,016	2.43%	68.12%
mincost	3,616	3,748	4,712	3.65%	30.31%
bubblesort	544	564	560	3.68%	2.94%
ackerman	284	284	292	0.00%	2.82%
matmult	580	600	600	3.45%	3.45%
banner	696	836	972	20.11%	39.66%
sieve	276	280	280	1.45%	1.45%
compact	4,536	4,668	8,040	2.91%	77.25%
queens	376	376	424	0.00%	12.77%
deroff	22,872	24,032	61,224	5.07%	167.68%
average	3,386	3,577	6,931	5.39%	42.24%

Table B.2: Change of Miss Ratio (Context Switch, Aligned)

cache program	1Kb LOOPS	 ALL	2Kb LOOPS	 ALL	4Kb LOOPS	 ALL	8Kb LOOPS	 ALL
cal	+0.05	+0.09	+0.05	+0.07	+0.05	+0.07	+0.05	+0.07
quicksort	+0.00	+0.02	+0.00	+0.02	+0.00	+0.02	+0.00	+0.02
wc	+0.00	+0.08	+0.00	+0.08	+0.00	+0.08	+0.00	+0.08
grep	+0.01	+3.54	+0.00	+0.02	+0.00	+0.02	+0.00	+0.02
sort	−2.57	−2.40	+0.02	+0.39	+0.02	+0.04	+0.02	+0.04
od	−2.47	−4.56	−2.43	−5.81	−0.01	+0.27	−0.01	−0.02
mincost	+0.96	+3.31	+0.25	+3.39	+0.03	+0.31	+0.03	+0.31
bubblesort	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
ackerman	+0.00	+0.01	+0.00	+0.01	+0.00	+0.01	+0.00	+0.01
matmult	−0.00	−0.00	−0.00	−0.00	−0.00	−0.00	−0.00	−0.00
banner	+0.06	+0.21	+0.06	+0.21	+0.06	+0.21	+0.06	+0.21
sieve	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
compact	−2.36	−0.64	−1.17	−1.38	−0.58	−1.12	+0.02	+0.25
queens	+0.00	+0.03	+0.00	+0.03	+0.00	+0.03	+0.00	+0.03
deroff	+0.09	+0.62	+0.16	+0.05	+0.16	+0.03	−0.01	+0.09
average	−0.42	+0.02	−0.20	−0.20	−0.02	−0.00	+0.01	+0.07

Table B.3: Change of Miss Ratio (No Context Switch, Aligned)

cache size program	1Kb LOOPS	 ALL	2Kb LOOPS	 ALL	4Kb LOOPS	 ALL	8Kb LOOPS	 ALL
cal	+0.01	+0.14	+0.01	+0.02	+0.01	+0.02	+0.01	+0.02
quicksort	+0.00	+0.02	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
wc	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
grep	+0.00	+3.46	+0.00	+0.03	+0.00	+0.00	+0.00	+0.00
sort	−2.55	−2.41	+0.00	+0.36	+0.00	+0.01	+0.00	+0.01
od	−2.45	−4.39	−2.34	−5.62	+0.00	+0.32	+0.00	+0.00
mincost	+0.97	+3.27	+0.22	+3.11	+0.00	+0.03	+0.00	+0.02
bubblesort	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
ackerman	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
matmult	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
banner	+0.06	+0.21	+0.06	+0.21	+0.06	+0.21	+0.06	+0.21
sieve	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
compact	−2.45	−0.66	−1.21	−1.70	−0.60	−1.40	−0.00	+0.00
queens	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
deroff	+0.04	+0.64	+0.13	−0.04	+0.14	−0.10	−0.10	−0.04
average	−0.42	+0.02	−0.21	−0.24	−0.03	−0.06	−0.00	+0.02

Table B.4: Percent Change of Instruction Fetch Cost (Context Switch, Aligned)

cache size	1Kb		2Kb		4Kb		8Kb	
program	LOOPS	ALL	LOOPS	ALL	LOOPS	ALL	LOOPS	ALL
cal	−2.72	−2.39	−2.72	−2.53	−2.72	−2.53	−2.72	−2.53
quicksort	−0.37	−3.82	−0.37	−3.82	−0.37	−3.82	−0.37	−3.82
wc	+0.01	−3.88	+0.01	−3.88	+0.01	−3.88	+0.01	−3.88
grep	+0.02	+26.40	+0.01	−3.21	+0.01	−3.23	+0.01	−3.23
sort	−21.55	−27.24	−3.93	−9.27	−3.93	−11.98	−3.93	−11.98
od	−14.33	−29.33	−14.68	−35.64	−3.33	−8.27	−3.33	−10.47
mincost	+2.90	+8.90	−0.03	+16.47	−1.58	−4.18	−1.58	−4.18
bubblesort	−18.91	−18.91	−18.91	−18.91	−18.91	−18.91	−18.91	−18.91
ackerman	+0.00	−3.77	+0.00	−3.77	+0.00	−3.77	+0.00	−3.77
matmult	−0.21	−0.21	−0.21	−0.21	−0.21	−0.21	−0.21	−0.21
banner	−1.18	−8.07	−1.18	−8.07	−1.18	−8.07	−1.18	−8.07
sieve	−8.52	−8.52	−8.52	−8.52	−8.52	−8.52	−8.52	−8.52
compact	−13.62	−8.58	−7.74	−13.59	−4.52	−13.05	−0.01	−3.16
queens	+0.00	+0.20	+0.00	+0.20	+0.00	+0.20	+0.00	+0.20
deroff	+0.71	−2.40	+1.29	−6.76	+1.32	−6.88	−0.11	−6.43
average	−5.18	−5.44	−3.80	−6.77	−2.93	−6.47	−2.72	−5.93

Table B.5: Percent Change of Instruction Fetch Cost (No Context Switch, Aligned)

cache size	1Kb		2Kb		4Kb		8Kb	
program	LOOPS	ALL	LOOPS	ALL	LOOPS	ALL	LOOPS	ALL
cal	−2.97	−1.90	−2.97	−2.92	−2.97	−2.92	−2.97	−2.92
quicksort	−0.38	−3.83	−0.38	−3.95	−0.38	−3.95	−0.38	−3.95
wc	+0.01	−4.55	+0.01	−4.55	+0.01	−4.55	+0.01	−4.55
grep	−0.02	+26.29	−0.02	−3.14	−0.02	−3.39	−0.02	−3.39
sort	−21.69	−27.49	−4.05	−9.45	−4.05	−12.24	−4.05	−12.24
od	−14.59	−29.19	−14.71	−35.87	−3.23	−7.77	−3.23	−10.31
mincost	+3.13	+9.22	−0.11	+16.33	−1.76	−6.30	−1.76	−6.34
bubblesort	−18.92	−18.92	−18.92	−18.92	−18.92	−18.92	−18.92	−18.92
ackerman	+0.00	−3.86	+0.00	−3.86	+0.00	−3.86	+0.00	−3.86
matmult	−0.21	−0.21	−0.21	−0.21	−0.21	−0.21	−0.21	−0.21
banner	−1.18	−8.07	−1.18	−8.07	−1.18	−8.07	−1.18	−8.07
sieve	−8.53	−8.53	−8.53	−8.53	−8.53	−8.53	−8.53	−8.53
compact	−14.70	−8.84	−8.40	−16.03	−4.94	−15.67	−0.22	−5.13
queens	+0.00	−0.03	+0.00	−0.03	+0.00	−0.03	+0.00	−0.03
deroff	+0.30	−2.10	+1.13	−7.40	+1.17	−7.94	−0.87	−7.40
average	−5.32	−5.47	−3.89	−7.11	−3.00	−6.95	−2.82	−6.39

Appendix C

Measurements to Evaluate Instruction Alignment

Table C.1: Additional Instruction Fetches for Unaligned Instructions for 1Kb Cache without Context Switch

	percent of all instructions			difference to SIMPLE	
program	SIMPLE	LOOPS	ALL	LOOPS	ALL
cal	+6.46	+7.69	+6.02	+1.23	−0.44
quicksort	+3.97	+1.41	+7.00	−2.57	+3.03
wc	+10.15	+10.15	+7.79	−0.00	−2.36
grep	+3.34	+7.86	+5.05	+4.52	+1.70
sort	+5.19	+4.30	+4.28	−0.89	−0.91
od	+3.48	+6.65	+4.01	+3.17	+0.53
mincost	+9.64	+9.17	+10.23	−0.47	+0.59
bubblesort	+8.66	+0.03	+0.07	−8.63	−8.60
ackerman	+0.02	+0.02	+3.86	+0.00	+3.83
matmult	+7.85	+7.87	+7.87	+0.02	+0.02
banner	+1.08	+1.90	+2.32	+0.82	+1.24
sieve	+11.65	+8.53	+8.53	−3.12	−3.12
compact	+3.32	+5.61	+9.71	+2.28	+6.38
queens	+6.01	+6.01	+16.57	+0.00	+10.56
deroff	+12.49	+10.06	+16.15	−2.42	+3.67
average	+6.22	+5.82	+7.30	−0.40	+1.08

References

[Ah86] A. V. Aho, R. Sethi, J. D. Ullman, *Compilers – Principles, Techniques, and Tools*, Addison-Wesley, 1986

[Be91] M. E. Benitez, *Register Transfer Standard*, Computer Science Report No. RM-91-01, University of Virginia, Charlottesville, Virginia, March 1991

[Bl76] Bloniaz, Fischer, Meyer, *A note on the average time to compute transitive closures*, Automata Languages and Programming, Editors: Michaelson, Milner, Edinburgh University Press, 1976

[Br76] J. Bruno, R. Sethi, *Code Generation for a One-Register Machine*, Journal of the ACM, Vol. 23, No. 3, July 1976, pp. 502-510

[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

[Da90-1] J. W. Davidson, D. B. Whalley, *Reducing the Cost of Branches by Using Registers*, 4th International Conference on Architectural Support for Programming Languages and Operating Systems, April 1990, pp. 182-191

[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

[Di87] D. R. Ditzel, H. R. McLellan, *Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero*, Proceedings of the 14th Annual Symposium on Computer Architecture, 1987, pp. 2-9

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

[Fu89] Stephen B. Furber, *VLSI RISC Architecture and Organization*, Marcel Dekker, 1989

[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

[Mc89] S. McFarling, *Program Optimization for Instruction Caches*, 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, April 1989, pp. 183-191

[Mo85] Motorola, *MC68020 32-Bit Microprocessor User's Manual*, Prentice-Hall Inc., 1985

[Pa85] D. A. Patterson, *Reduced Instruction Set Computers*, Communications of the ACM, Vol. 28, No. 1, January 1985, pp. 8-21

[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

[Pe90] K. Pettis, R. C. Hansen, *Profile Guided Code Positioning*, Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, Vol. 25, No. 6, June 1990, pp. 16-27

[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

[Sc77] C.-P. Schnorr, *An Algorithm for Transitive Closure with Linear Expected Time*, Computer Science Lecture Series, Springer, 1977, pp. 329-338

[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

Bibliographical Sketch

Frank Mueller was born on March 24, 1966 in Berlin, Germany. He received his Bachelor of Science, majoring in Computer Science from the Technische Universität Berlin, Germany, in 1987. In 1991, he graduated from Florida State University with a Master of Science in Computer Science. He is currently seeking his Ph.D. at Florida State University. His professional interests include the areas of compilers, real-time systems, and neural networks. He is a member of the ACM.