DESIGN AND IMPLEMENTATION OF A PORTABLE GLOBAL CODE OPTIMIZER

BY

SCOTT ALAN MAHLKE

B.S., University of Illinois, 1988

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1992

Urbana, Illinois

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Professor Wen-mei Hwu, for his guidance, his support, and, most importantly, his patience. I have learned a great deal from him during my years in graduate school, and I look forward to working together in the future.

I wish to extend my appreciation to two people, Pohua Chang and William Chen, with whom I have worked very closely in the past few years. Pohua wrote most of the IMPACT-I C compiler. He also helped with the formulation of many of the optimizations, helped with initial implementation, and assisted with many of the difficulties encountered along the way. William wrote the first code generator for the compiler and helped with much of the debugging of the optimizer. Without both of them, this project would never have been finished.

I also would like to thank three other people, Nancy Warter, and my two officemates, Sadun Anik and Tom Conte. Nancy and I have spent many long nights writing papers together. Sadun and Tom have answered my endless supply of day to day questions.

Finally, I wish to thank everyone above and all of my other friends who have made my life in graduate school very enjoyable.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

The importance of compile time code optimizations to improve code efficiency has been recognized for many years. There are two categories of code optimization, machine-dependent optimization and machine-independent optimization. Machine-dependent optimizations require information regarding the target machine's instruction set and hardware features, and are thus performed in conjunction with code generation. Machine-independent optimizations use a wide variety of transformations to improve both a programs speed and efficiency. Machine-independent optimizations are performed on a well-defined compiler intermediate code. Although the concepts concerning machine-independent code optimization are well known, few people understand the details required for implementation. In this thesis, the complete design of a portable machine-independent optimizer is presented.

The use of profile information to identify more optimization opportunities than a conventional optimizer can recognize is also investigated. An execution profile refers to the dynamic behavior of a program when executed on a given input. Conventional optimizers rely on static loop analysis to identify critical regions of a program. However, static analysis often results in large prediction errors. Profile information can accurately identify

the critical regions of a program. More aggressive conventional optimization is then possible when the critical regions are known by the compiler. The use of profile information to guide code optimization will be referred to as *profile-based code optimization*.

A portable machine-independent optimizer, *Lopti*, has been constructed as a component of the IMPACT-I C compiler [8]. The IMPACT-I C compiler is a prototype retargetable profiling compiler that currently generates code for the following machines: MIPS R2000, Sun SPARC, Intel i860, and AMD29k. All optimizations are performed on a low-level machine-independent intermediate code called *Lcode* [7].

1.1 Related Work

Code optimizers have been studied by many researchers both as independent modules, and as components of large compilers. Almost all existing high-level language compilers today perform code optimization to some degree. The PL.8 compiler project at IBM [4] accepts several source languages and generates high-quality code for various different machines. Optimizations are applied on a low-level intermediate code and are repeatedly performed to maximize their effectiveness. The Amsterdam Compiler Kit [20] consists of several components that can be combined to form compilers with different properties. The global and peephole optimizers use a pattern/replacement table that specifies how specific patterns of instruction sequences within a window can be replaced with a more efficient sequence. The GNU C compiler [19] is a public domain optimizing compiler capable of generating code for a large number of machines.

Chow has designed and built *UOPT*, a machine-independent global optimizer [11]. Included in this design is a machine-independent intermediate code that effectively supports optimization. A complete set of classical optimizations, and some additional optimization techniques are contained in *UOPT*. Chow also evaluated the performance, efficiency, and relative importance of these optimizations on several target machines.

1.2 Organization of the Thesis

This thesis is organized into six chapters. Chapter 2 gives a brief overview of the IMPACT-I C compiler. In Chapter 3, the implementation details of classical machine-independent optimizations are discussed. In Chapter 4, the use of profile information to identify additional opportunities for classical optimizations is presented. In Chapter 5, the performance of the optimizer is evaluated. Finally, some concluding remarks are made in Chapter 6.

2. OVERVIEW OF THE IMPACT-I C COMPILER

A diagram showing the major components of the IMPACT-I C compiler is shown in Figure 2.1. *Box A* contains the compiler front-end and the code generator. The compiler front end translates a C program into an intermediate code which is suitable for code optimization and code generation. The compiler front end performs appropriate lexical, syntactic, and semantic analyses on the C program. If an error is found in the lexical, syntax, or semantic analysis, the compilation process is stopped abruptly before assembly/machine code generation. The IMPACT-I C compiler generates code for several existing processor architectures: MIPS R2000, Sun SPARC, Intel i860, and AMD29k. Each code generator performs the following tasks: (1) machine-dependent code optimizations, e.g., constant preloading, instruction selection, (2) register allocation, and (3) assembly/machine code generation. The machine-independent global optimizer is shown in *Box B*.

Box C is used to obtain profile information. The input to the profiler is an intermediate code plus a set of input data. From the intermediate code, a profiler is automatically generated. The profiler is executed once with each input data set to produce a profile file. After we have obtained all profile files, they are summarized. The summarized profile

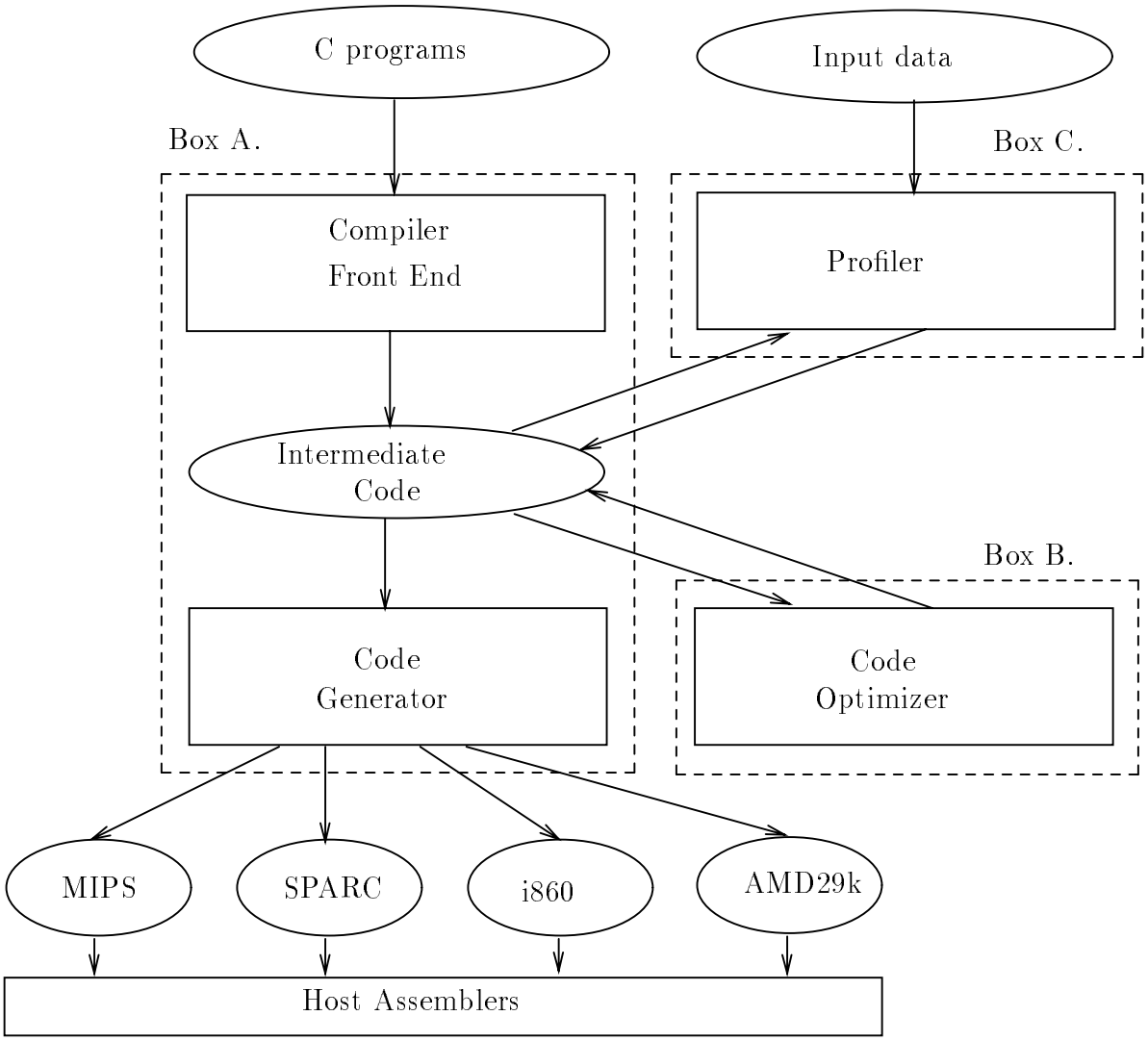

Figure 2.1: A block diagram of the IMPACT-I C compiler.

information is then integrated into the intermediate code. The global code optimizations in *Box B* are modified to use the profile information.

The compilation procedure consists of the following steps:

1. The compiler front end translates a C program into an intermediate code (*Box A*). If there is no need to perform global code optimizations goto 4; otherwise, goto 2.

2. The compiler performs classic global code optimizations (*Box B*). If there is no need to perform profile-based code optimizations goto 4; otherwise, goto 3.

3. The compiler generates a profiler and obtains profile information (*Box C*). The profile information is integrated into the intermediate code (*Box C*). The compiler applies profile-based code optimizations (*Box B*) on the intermediate code. Goto 4.

4. The compiler generates target assembly/machine code (*Box A*).

2.1 Program Representation

In optimizing compilers, a function is typically represented by a flow graph [1], where each node is a basic block and each arc is a potential control flow path between two basic blocks. A flow graph can be extended to include profile information. We define a *weighted flow graph* as a quadruple $\{V, E, count, arc\_count\}$, where each node in V is a basic block, each arc in E is a potential control flow path between two basic blocks, $count(v)$ is a function that returns the execution count of a basic block v, and $arc\_count(e)$ is a function that returns the taken count of a control flow path e.

Each basic block contains a straight-line segment of instructions. The last instruction of a basic block may be one of the following types: (1) an unconditional jump instruction,

(2) a 2-way conditional branch instruction, (3) a multiway branch instruction (*switch* statement in C), or (4) an arithmetic instruction. For simplicity, a jump-subroutine instruction is assumed to be an arithmetic instruction because it does not change the control flow within the function where the jump-subroutine instruction is defined. Except for the last instruction, all other instructions in a basic block must be arithmetic instructions that do not change the flow of control to another basic block.

The instruction set that we have chosen for our intermediate code (*Lcode*) has the following properties: (1) The opcode (operation code) set is very close to that of the host machine instruction sets, e.g., MIPS R2000 and SPARC. (2) It is a load/store architecture. Arithmetic instructions are register-to-register operations. Data transfers between registers and memory are specified by explicit memory load/store instructions. (3) The intermediate code provides an infinite number of temporary registers. This allows code optimization to be formulated independently of the machine dependent register file structures and calling conventions.

2.2 Profiler Implementation

The following information is collected with the profiler in *Box C* of Figure 2.1.

1. The number of times a program has been profiled (N).
2. The invocation count $fn\_count(f_i)$ of each function f_i.
3. The execution count $count(b_k)$ of each basic block b_k.
4. For each 2-way conditional branch instruction I, the number of times it has been taken ($taken\_count(I)$).

5. For each multiway branch instruction I, the number of times each case (cc) has been taken ($case\_count(I, cc)$).

With this information, we can annotate a flow graph to form a weighted flow graph.

Figure 2.2 shows the major components of the profiler that appears in *Box C* of Figure 2.1. Automatic profiling is provided by four tools: (1) a probe insertion program, (2) an execution monitor, (3) a program to combine several profile files into a summarized profile file, and (4) a program that maps the summarized profile data into a flow graph to generate a weighted flow graph data structure.

The profiling procedure requires five steps as shown in Figure 2.2.

(a) The probe insertion program assigns a unique ID to each function and inserts a probe at the entry point of each function. Whenever the probe is activated, it produces a $function(id)$ token. In a $function(id)$ token, id is the unique ID of the function. The probe insertion program also assigns a unique ID to each basic block within a function. Therefore, a basic block can be uniquely identified by a tuple *(function id, basic block id)*. The probe insertion program inserts a probe in each basic block to produce a $bb(fid, bid, cc)$ token every time that basic block is executed. In a $bb(fid, bid, cc)$ token, fid identifies a function, bid identifies a basic block in that function, and cc is the branch condition. The output of the probe insertion program is an annotated intermediate code.

(b) The annotated intermediate code is compiled to generate an executable program which produces a trace of tokens every time the program is executed.

(c) The execution monitor program consumes a trace of tokens and produces a profile file. We have implemented the execution monitor program in two ways. It can be a separate program which listens through a UNIX socket for incoming tokens.

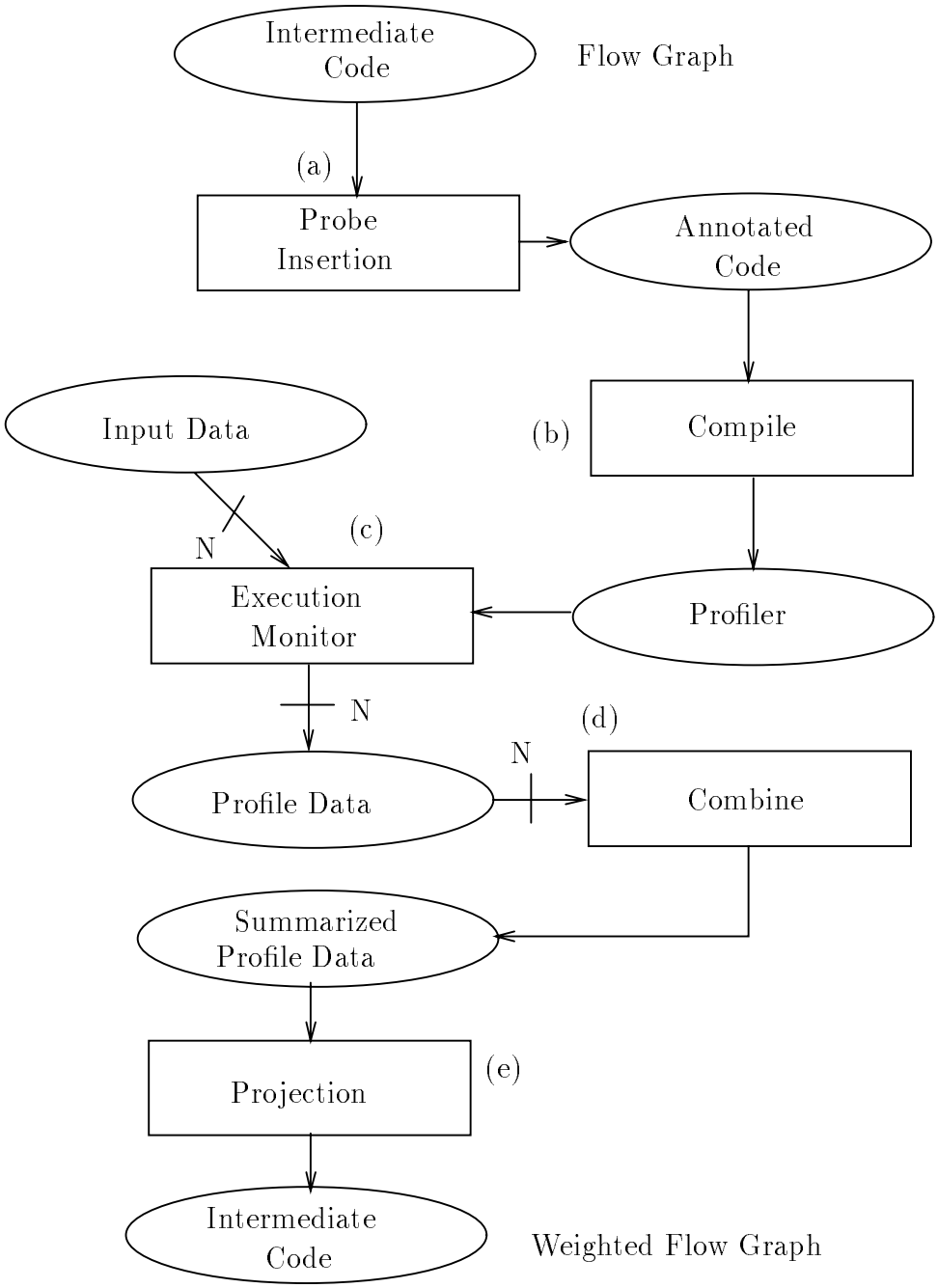

Figure 2.2: A block diagram of the profiler.

Alternatively, it can be a function which is linked with the annotated user program. The second approach is at least two orders of magnitude faster than the first approach, but may fail when the original user program contains a very large data section that prevents the monitor program from allocating the necessary memory space. Fortunately, we have not yet encountered that problem.

(d) Step (c) is repeated once for each additional input. All profile files are combined into a summarized profile file.

(e) Finally, the composite profile file is mapped into the original intermediate code. Because we have not changed the structure of the program, it is straightforward to search using the assigned function and basic block identifiers. To simplify the formulation of code optimizations, all execution counts are divided by the number of times the program has been profiled.

3. CLASSICAL OPTIMIZATION

There are three classes of classical optimizations, local optimizations, global optimizations, and loop optimizations [1] [2]. The optimizer presented in this thesis, *Lopti*, performs most standard classical optimizations. A summary of the optimizations included in *Lopti* is presented in Tables 3.1, 3.2, and 3.3. The goals of classical optimizations are to improve the execution speed and reduce the size of a program. For some of the transformations, a conflict exists between these two objectives. The main objective used in the design of *Lopti* is optimizing for speed. There have been no optimizations included that optimize only for space. In the rest of this chapter, the implementation details for each class of optimizations will be discussed. The style used to formulate all optimizations is first described.

3.1 Formulation of Code Optimizations

Each optimization will be formulated as a pattern matching rule consisting of two parts. The first part is a set of predicate functions which return true or false. An optimization opportunity is identified by all predicate functions of a rule returning true. The second part of each rule is a set of actions, which are the steps taken to perform the

Table 3.1: Local optimizations.

constant propagation
forward copy propagation
memory copy propagation
arithmetic common subexpression elimination
redundant load elimination
redundant store elimination
constant folding
strength reduction
constant combining
arithmetic operation folding
branch operation folding
operation cancellation
dead code removal
code reordering

Table 3.2: Global optimizations.

constant propagation
forward copy propagation
backward copy propagation
memory copy propagation
arithmetic common subexpression elimination
redundant load elimination
redundant store elimination
dead code removal

Table 3.3: Loop optimizations.

invariant code removal
global variable migration
branch simplification
induction variable strength reduction
induction variable elimination

code transformation. Optimization consists of selecting a set of prospective operations and testing if the predicate functions for a rule all return true. If the pattern in the rule is detected, the corresponding set of actions is performed.

A common set of notations will be used to express all optimization rules. Let $\{op(j)|j = 1 \cdots m, m \geq 1\}$ denote an ordered set of operations in each function. Most optimizations examine one or more pairs of operations to test for optimization opportunities. For each operation $op(j)$, the set of register variables that $op(j)$ modifies is denoted by $dest(j)$. The set of register variables that $op(j)$ requires as source operands is given by $src(j)$. To denote each source operand individually, $src1(j)$, $src2(j)$, $src3(j)$ to denote source operand one, two, and three are used, respectively. Finally, the operation code of $op(j)$ is given by f_j.

A common predicate function used throughout the explanation of all optimizations is *no_danger*. This predicate returns true if there are not any operations in a given set whose side effects may adversely affect the optimization transformation. In this optimizer, all synchronization operations are assumed to be dangerous. Furthermore, if any of the operations considered in a particular optimization rule are memory operations or use parameter passing registers, all jump subroutine operations are considered to be dangerous (no interprocedural analysis is performed by the optimizer).

3.2 Local Optimization

Local optimizations refer to the optimizations done within individual basic blocks [1]. Local optimizations are independently applied to each basic block in a program, and thus use no information regarding other basic blocks. Each local optimization used in the optimizer is formulated in the following subsections.

3.2.1 Constant propagation

Local constant propagation replaces future uses of a variable in a basic block which is assigned a constant, with that constant. The optimization rule for local constant propagation is given below.

- Predicate functions
  1. $op(x)$ is a move operation.
  2. $src1(x)$ is a constant.
  3. There is a flow dependency between $op(x)$ and $op(y)$.
  4. There is no danger in $\{op(j), j = x + 1 \cdots y - 1\}$.
- Actions
  1. If $src1(y) = dest(x)$, then change $src1(y)$ to $src1(x)$.
  2. If $src2(y) = dest(x)$, then change $src1(y)$ to $src1(x)$.
  3. If $src3(y) = dest(x)$, then change $src3(y)$ to $src1(x)$.

3.2.2 Forward copy propagation

Local forward copy propagation replaces future uses of a variable in a basic block which is assigned the value of another variable, with the other variable. The optimization rule for local forward copy propagation is given below.

- Predicate functions
  1. $op(x)$ is a move operation.
  2. $src1(x)$ is a register.
  3. There is a flow dependency between $op(x)$ and $op(y)$.
  4. $src1(x)$ is not modified by $\{op(j), j = x \cdots y - 1\}$.
  5. There is no danger in $\{op(j), j = x + 1 \cdots y - 1\}$.
- Actions
  1. If $src1(y) = dest(x)$, then change $src1(y)$ to $src1(x)$.
  2. If $src2(y) = dest(x)$, then change $src2(y)$ to $src1(x)$.
  3. If $src3(y) = dest(x)$, then change $src3(y)$ to $src1(x)$.

3.2.3 Memory copy propagation

Local memory copy propagation replaces future reads of a memory variable with a register assignment if the current value of the memory variable resides in a register. Propagation must stop if the memory variable is directly or indirectly modified. The existence of indirect modifications is determined by local memory disambiguation. The optimization rule for local memory copy propagations is given below.

- Predicate functions
  1. $op(x)$ is a store operation.
  2. $op(y)$ is a load operation.
  3. $op(x)$ and $op(y)$ are compatible data types.
  4. $address(x) = address(y)$.
  5. The variables in $src(x)$ are not not modified by $\{op(j), j = x \cdots y - 1\}$.
  6. There is no write to $address(x)$ in $\{op(j), j = x + 1 \cdots y - 1\}$.
  7. There is no danger in $\{op(j), j = x + 1 \cdots y - 1\}$.
- Actions
  1. Change $op(y)$ to a move operation, with $src1(y) = src3(x)$.

3.2.4 Arithmetic common subexpression elimination

Local arithmetic common subexpression elimination removes the recomputation of an arithmetic expression whenever a previously computed value can be used. A previous computation is available if all of the source operands are not modified between computations. The optimization rule for local arithmetic common subexpression elimination is shown below.

- Predicate functions
  1. $op(x)$ is an arithmetic operation.
  2. $f_x = f_y$.

3. $src(x) = src(y)$.
4. $dest(x)$ is not the same as any of the variables in $src(x)$.
5. $dest(y)$ is not the same as any of the variables in $src(y)$.
6. The variables in $src(x)$ are not modified by $\{op(j), j = x \cdots y - 1\}$.
7. There is no danger in $\{op(j), j = x + 1 \cdots y - 1\}$.

- Actions

1. Insert a new move operation, $op(a)$, after $op(x)$, with $dest(a) = temp\_reg$ and $src1(a) = dest(x)$.
2. Change $op(y)$ to a move operation, with $src1(y) = dest(a)$; ($dest(y)$ remains unchanged).

3.2.5 Redundant load elimination

Local redundant load elimination removes load operations if there is a previous load of the same memory variable. Removal must stop, however, whenever the memory variable changes value, either directly or indirectly. The optimization rule for local redundant load elimination is shown below.

- Predicate functions

1. $op(x)$ is a load operation.
2. $f_x = f_y$.
3. $src(x) = src(y)$.
4. $dest(x)$ is not the same as any of the variables in $src(x)$.
5. $dest(y)$ is not the same as any of the variables in $src(y)$.
6. The variables in $src(x)$ are not modified by $\{op(j), j = x \cdots y - 1\}$.
7. There is no write to $address(x)$ in $\{op(j), j = x + 1 \cdots y - 1\}$.
8. There is no danger in $\{op(j), j = x + 1 \cdots y - 1\}$.

- Actions

1. Insert a new move operation, $op(a)$, after $op(x)$, with $dest(a) = temp\_reg$ and $src1(a) = dest(x)$.
2. Change $op(y)$ to a move operation, with $src1(y) = dest(a)$; ($dest(y)$ remains unchanged).

3.2.6 Redundant store elimination

Local redundant store elimination removes memory store operations if there is a previous store of the same value to the same memory variable. There can be no modifications of the memory variable or the value to be stored between the store operations to apply the optimization. The optimization rule for local redundant store elimination is given below.

- Predicate functions
  1. $op(x)$ is a store operation.
  2. $f_x = f_y$.
  3. $src(x) = src(y)$.
  4. The variables in $src(x)$ are not modified by $\{op(j), j = x \cdots y - 1\}$.
  5. There is no write to $address(x)$ in $\{op(j), j = x + 1 \cdots y - 1\}$.
  6. There is no danger in $\{op(j), j = x + 1 \cdots y - 1\}$.
- Actions
  1. Delete $op(y)$.

3.2.7 Constant folding

Local constant folding evaluates operations whose value can be determined at compile time. An operation can be evaluated if all of its operands are numeric constants. Also, algebraic identities, such as add with 0 or multiply by 0 or 1, can be used to evaluate operations at compile time. The optimization rule for local constant folding is broken down into five classes based on the characteristics of the folding. The rules are given below.

- Predicate functions for type 1
  1. $op(x)$ is an add, a subtract, a logical OR, a logical XOR, or a shift operation.
  2. $src2(x) = 0$.

- Actions for type 1
  1. $temp\_src = src1(x)$.
  2. Change $op(x)$ to a move operation, with $src1(x) = temp\_src$.
- Predicate functions for type 2
  1. $op(x)$ is a multiply, a divide, or a logical AND operation.
  2. $src1(x) = 0$.
- Actions for type 2
  1. Change $op(x)$ to a move operation, with $src1(x) = 0$.
- Predicate functions for type 3
  1. $op(x)$ is a multiply or a divide operation.
  2. $src2(x) = 1$.
- Actions for type 3
  1. $temp\_src = src1(x)$
  2. Change $op(x)$ to a move operation, with $src1(x) = temp\_src$.
- Predicate functions for type 4
  1. $op(x)$ is an arithmetic operation.
  2. The variables in $src(x)$ are numerical constants.
  3. $op(x)$ can be evaluated (e.g., divide by 0 cannot be evaluated).
- Actions for type 4
  1. $temp\_src$ = the result of f_x applied to $src(x)$.
  2. Change $op(x)$ to a move operation, with $src1(x) = temp\_src$.
- Predicate functions for type 5
  1. $op(x)$ is a conditional branch operation.
  2. The variables in $src(x)$ are numerical constants.
- Actions for type 5
  1. $temp\_src = target(x)$.
  2. cc = the result of applying the branch condition to $src(x)$.
  3. If $cc = 1$, convert $op(x)$ into a jump operation, with $target(x) = temp\_src$; otherwise, $(cc = 0)$ delete $op(x)$.

3.2.8 Strength reduction

Local strength reduction replaces expensive operations by equivalent cheaper ones. In this optimizer, three operations are considered for strength reduction, integer multiply, integer divide, and integer remainder. The optimization rule for local strength reduction broken down by operation type is shown below.

- Predicate functions for type 1
  1. $op(x)$ is an integer multiply operation.
  2. $src2(x)$ is a power of 2 integer constant.
- Actions for type 1
  1. $temp\_src = log_2(src2(x))$.
  2. Change $op(x)$ to a logical left shift operation, with $src2(x) = temp\_src$, ($dest(x)$ and $src1(x)$ remain unchanged).
- Predicate functions for type 2
  1. $op(x)$ is an integer divide operation.
  2. $src2(x)$ is a power of 2 integer constant.
- Actions for type 2
  1. $temp\_src = log_2(src2(x))$.
  2. Change $op(x)$ to an arithmetic right shift operation, with $src2(x) = temp\_src$, ($dest(x)$ and $src1(x)$ remain unchanged).
- Predicate functions for type 3
  1. $op(x)$ is an integer remainder operation.
  2. $src2(x)$ is a power of 2 integer constant.
- Actions for type 3
  1. $temp\_src = src2(x) - 1$.
  2. Change $op(x)$ to a logical AND operation, with $src2(x) = temp\_src$; ($dest(x)$ and $src1(x)$ remain unchanged).

3.2.9 Constant combining

Local constant combining merges a pair of operations each one with a constant source operand. The combination of constant source operands are evaluated at compile time, and, therefore, the pair of operations can be replaced by a single operation. The optimization rule for local constant combining is shown below.

- Predicate functions for type 1
  1. $op(x)$ is an add or a subtract operation.
  2. $src2(x)$ is a numerical constant.
  3. $dest(x)$ is not the same as any of the variables in $src(x)$.
  4. $op(y)$ is either a load, a store, an add, or a subtract, a comparison, or a conditional branch operation.
  5. $src2(y)$ is a numerical constant.
  6. There is a flow dependency between $op(x)$ and $op(y)$.
  7. Either $src1(y)$ or $src2(y)$ are the same as $dest(x)$; (this check is necessary because the flow dependency between $op(x)$ and $op(y)$ could be for $src3(y)$ in the case that $op(y)$ is a store operation).
  8. $src1(x)$ is not modified by $\{op(j), j = x \cdots y - 1\}$.
  9. There is no danger in $\{op(j), j = x + 1 \cdots y - 1\}$.
- Actions for type 1
  1. If $op(x)$ is a subtract operation, $val1 = -src2(x)$; otherwise, $val1 = src2(x)$.
  2. If $op(y)$ is a subtract, a comparison, or a conditional branch operation, $val2 = -src2(y)$; otherwise, $val2 = src2(y)$.
  3. Change $src1(y)$ to $src1(x)$.
  4. If $op(y)$ is a subtract operation, change $src2(y)$ to $-(val1 + val2)$; otherwise, change $src2(y)$ to $val1 + val2$.
- Predicate functions for type 2
  1. $op(x)$ is an integer add or an integer subtract operation.
  2. $dest(x) = src1(x)$, e.g., $op(x)$ is an increment/decrement operation.
  3. $src2(x)$ is a numerical constant.
  4. $op(y)$ is an integer add or an integer subtract operation.

5. $dest(y) = src1(y)$
6. $src2(y)$ is a numerical constant.
7. There is a flow dependency between $op(x)$ and $op(y)$.
8. $dest(x)$ is not used by $\{op(j), j = x + 1 \cdots y - 1\}$.
9. There is no danger in $\{op(j), j = x + 1 \cdots y - 1\}$.

- Actions for type 2

1. If $op(x)$ is a subtract operation, $val1 = -src2(x)$; otherwise, $val1 = src2(x)$.
2. If $op(y)$ is a subtract operation, $val2 = -src2(y)$; otherwise, $val2 = src2(y)$.
3. If $op(x)$ is a subtract operation, change $src2(x)$ to $-(val1 + val2)$; otherwise, change $src2(x)$ to $val1 + val2$.
4. Delete $op(y)$.

3.2.10 Arithmetic operation folding

Local arithmetic operation folding merges a pair of operations whose functionalities can be combined into 1 operation. The pairs of operations are limited to arithmetic and memory operations for this optimization. The optimization rule for local arithmetic operation folding is given below.

- Predicate functions for type 1

1. $op(x)$ is an add operation.
2. $op(y)$ is a load or a store opcode.
3. $src2(y) = 0$.
4. $dest(x)$ is not the same as any of the variables in $src(x)$.
5. There is a flow dependency between $op(x)$ and $op(y)$.
6. Either $src1(y)$ or $src2(y)$ are the same as $dest(x)$.
7. The variables in $src(x)$ are not modified by $\{op(j), j = x \cdots y - 1\}$.
8. There is no danger in $\{op(j), j = x + 1 \cdots y - 1\}$.

- Actions for type 1

1. Change $src1(y)$ to $src1(x)$.
2. Change $src2(y)$ to $src2(x)$.

- Predicate functions for type 2

  1. $op(x)$ is a comparison operation.
  2. $op(y)$ is an integer equal comparison (EQ) or an integer not equal comparison (NE) operation.
  3. Either $src1(y)$ or $src2(y)$ is equal to 0.
  4. There is a flow dependency between $op(x)$ and $op(y)$.
  5. The variables in $src(x)$ are not modified by $\{op(j), j = x \cdots y - 1\}$.
  6. There is no danger in $\{op(j), j = x + 1 \cdots y - 1\}$.

- Actions for type 2

  1. If $op(y)$ is an NE operation, change $op(y)$ to a move operation, with $src1(y) = dest(x)$, ($dest(y)$ remains unchanged).
  2. Otherwise, ($op(y)$ is a eq operation); change $op(y)$ to a comparison operation with opcode opposite to that of $op(x)$, with $src1(y) = src1(x)$ and $src2(y) = src2(x)$.

- Predicate functions for type 3

  1. $op(x)$ is an OR, an AND, or an XOR operation.
  2. $op(y)$ is an XOR operation.
  3. Either $src1(y)$ or $src2(y)$ is -1.
  4. There is a flow dependency between $op(x)$ and $op(y)$.
  5. The variables in $src(x)$ are not modified by $\{op(j), j = x \cdots y - 1\}$.
  6. There is no danger in $\{op(j), j = x + 1 \cdots y - 1\}$.

- Actions for type 3

  1. If $op(x)$ is an OR operation, change $op(y)$ to a NOR operation, with $src1(y) = src1(x)$ and $src2(y) = src2(x)$.
  2. If $op(x)$ is an AND operation, change $op(y)$ to a NAND operation, with $src1(y) = src1(x)$ and $src2(y) = src2(x)$.
  3. If $op(x)$ is an XOR operation, change $op(y)$ to a NXOR operation, with $src1(y) = src1(x)$ and $src2(y) = src2(x)$.

3.2.11 Branch operation folding

Local branch operation folding merges a comparison operation and a branch operation into a single compare and branch operation. The optimization rule used for local branch operation folding is presented below.

- Predicate functions
  1. $op(x)$ is a comparison operation.
  2. $op(y)$ is a branch equal to (beq) or branch not equal to (bne).
  3. There is a flow dependency between $op(x)$ and $op(y)$.
  4. Either $src1(y)$ or $src2(y)$ is 0.
  5. The variables in $src(x)$ are not modified by $\{op(j), j = x \cdots y - 1\}$.
  6. There is no danger in $\{op(j), j = x + 1 \cdots y - 1\}$.
- Actions
  1. If $op(y)$ is a bne operation, change $op(y)$ to a conditional branch operation with opcode corresponding to that of $op(x)$, with $src1(y) = src1(x)$ and $src2(y) = src2(x)$; ($target(y)$ remains unchanged).
  2. Otherwise, ($op(y)$ is a beq operation), change $op(y)$ to a conditional branch operation with opcode opposite to that of $op(x)$, with $src1(y) = src1(x)$ and $src2(y) = src2(x)$; ($target(y)$ remains unchanged).

3.2.12 Operation cancellation

Local operation cancellation removes one or more operations when the effects of one negate those of another. Opportunities for this optimization are not usually present as a result of the user program; however, they can result after inline function expansion and loop induction variable elimination. The optimization rule for local operation cancellation is shown below.

- Predicate functions for type 1
  1. $op(x)$ is an integer add or an integer subtract operation.
  2. $dest(x) = src1(x)$, e.g., $op(x)$ is an increment/decrement operation.

3. $op(y)$ is the inverse operation of $op(x)$, (acceptable combinations for f_x and f_y are add/sub and sub/add).
4. $dest(y) = src1(y)$.
5. $dest(x) = dest(y)$.
6. $src2(x) = src2(y)$.
7. There is a flow dependency between $op(x)$ and $op(y)$.
8. There is an output dependency between $op(x)$ and $op(y)$.
9. $src2(x)$ is not modified by $\{op(j), j = x \cdots y - 1\}$.
10. There is no danger in $\{op(j), j = x + 1 \cdots y - 1\}$.

- Actions for type 1

1. Delete $op(x)$.
2. Delete $op(y)$.

- Predicate functions for type 2

1. $op(x)$ is an add or subtract operation.
2. $dest(x)$ is not the same as any of the variables in $src(x)$.
3. $op(y)$ is the inverse operation of $op(x)$.
4. There is a flow dependency between $op(x)$ and $op(y)$.
5. If $op(x)$ is the subtract, then either $src1(y)$ or $src2(y)$ must be the same as $src2(x)$.
6. If $op(y)$ is the subtract, then either $src1(x)$ or $src2(x)$ must be the same as $src2(y)$.
7. The variables in $src(x)$ are not modified by $\{op(j), j = x \cdots y - 1\}$.
8. There is no danger in $\{op(j), j = x + 1 \cdots y - 1\}$.

- Actions for type 2

1. If $op(x)$ is a subtract operation, then change $op(y)$ to a move, with $src1(y) = src1(x)$; ($dest(y)$ remains unchanged).
2. Otherwise, ($op(y)$ is a subtract operation); one of the following four possibilities results:
   (a) If $src1(x) = src1(y)$, then change $src1(y)$ to 0 and $src2(y)$ to $src2(x)$.
   (b) If $src2(x) = src2(y)$, then change $src1(y)$ to 0 and $src2(y)$ to $src1(x)$.
   (c) If $src1(x) = src2(y)$, then change $op(y)$ to a move operation, with $src1(y) = src2(x)$.
   (d) If $src2(x) = src2(y)$, then change $op(y)$ to a move operation, with $src1(y) = src1(x)$.

3.2.13 Dead code removal

Local dead code removal removes operations that can be immediately identified as superfluous. There are two cases in which this happens. The first is an assignment operation which has the same source and destination. The second is the complementary situation where memory variables are concerned. The optimization rule for local dead code removal is shown below.

- Predicate functions for type 1
  1. $op(x)$ is a move operation.
  2. $dest(x) = src1(x)$.
- Actions for type 1
  1. Delete $op(x)$.
- Predicate functions for type 2
  1. $op(x)$ is a load operation.
  2. $op(y)$ is a store operation.
  3. $op(x)$ and $op(y)$ are compatible data types.
  4. $address(x) = address(y)$.
  5. $dest(x) = src3(y)$.
  6. There is a flow dependency between $op(x)$ and $op(y)$.
  7. The variables in $src(x)$ are not modified by $\{op(j), j = x \cdots y - 1\}$.
  8. There is no write to $address(x)$ in $\{op(j), j = x + 1 \cdots y - 1\}$.
  9. There is no danger in $\{op(j), j = x + 1 \cdots y - 1\}$.
- Actions for type 2
  1. Delete $op(y)$.

3.2.14 Code reordering

Local code reordering rearranges a pair of operations to enable additional local optimization opportunities to be exposed. An optimization may not be possible because a

variable is modified between the two operations considered for optimization. However, it may be possible to reorder the operations to enable the optimization to occur. The optimization rule for local code reordering to enable forward copy propagation (type 1) and arithmetic operation folding (type 2) is given below.

- Predicate functions for type 1
  1. $op(x)$ is a move opcode.
  2. $src1(x)$ is a register.
  3. $src1(x)$ is not the same as $dest(x)$.
  4. There is a flow dependency between $op(x)$ and $op(y)$.
  5. $src1(x)$ is modified by $\{op(j), j = x \cdots y - 1\}$; let $op(a)$ denote the first modification of $src1(x)$.
  6. There is no danger in $\{op(j), j = x + 1 \cdots y - 1\}$.
  7. Can either move $op(y)$ above $op(a)$, (this is true when there are no dependencies between $op(y)$ and $\{op(j), j = a \cdots y - 1\}$), or $op(a)$ below $op(y)$, (this is true when there are no dependencies between $op(a)$ and $\{op(j), j = a + 1 \cdots y\}$).
- Actions for type 1
  1. If can move $op(y)$ above $op(a)$, move $op(y)$ above $op(a)$; otherwise, move $op(a)$ below $op(y)$.
- Predicate functions for type 2
  1. $op(x)$ is an integer add or an integer subtract operation.
  2. $dest(x) = src1(x)$, e.g., $op(x)$ is an increment/decrement operation.
  3. $src2(x)$ is a numerical constant.
  4. $op(y)$ is either a load, a store, a comparison, an add, or a subtract operation.
  5. $src2(y)$ is a numerical constant.
  6. There is a flow dependency between $op(x)$ and $op(y)$.
  7. Either $src1(y)$ or $src2(y)$ are the same as $dest(x)$.
  8. $dest(x)$ and $dest(y)$ are not the same.
  9. There is no danger in $\{op(j), j = x + 1 \cdots y - 1\}$.
  10. Can either move $op(y)$ above $op(x + 1)$ or $op(x)$ below $op(y - 1)$.

- Actions for type 2

  1. If $op(x)$ is a subtract operation, $val1 = -src2(x)$; otherwise, $val1 = src2(x)$.
  2. If $op(y)$ is a subtract or a comparison operation, $val2 = -src2(y)$; otherwise, $val2 = src2(y)$.
  3. If $op(y)$ is a subtract operation, change $src2(y)$ to $-(val1 + val2)$; otherwise, change $src2(y)$ to $val1 + val2$.
  4. If can move $op(y)$ above $op(x + 1)$, move $op(y)$ above $op(x)$; otherwise, move $op(x)$ below $op(y)$.

3.3 Global Optimization

Global optimization expands the scope of many local optimizations to the function level [17]. Dataflow information needs to be gathered to make many of the optimization decisions. The steps necessary to perform dataflow analysis and global optimization are discussed in the following sections.

3.3.1 Data flow analysis

Dataflow analysis is summary information distributed to each basic block regarding the previous and future uses of program variables [1] [3]. Dataflow information is collected by iteratively solving systems of equations that relate information at various points in a function. Three sets of data flow information are used by the global and loop optimizations, live variable information, available definition information, and available expression information.

The program variables live at each point in a program are determined by performing live variable analysis. A variable is live at a certain point if the value it currently holds may possibly be used in the future. Four sets of dataflow information are used to calculate live variable information, $def()$, $use()$, $in()$, and $out()$. Each of these sets is associated with every basic block in a function. The $def()$ set of a basic block is the set of variables

assigned a value in that basic block. The $use()$ set of a basic block is the set of variables used as a source operand in that basic block before they are defined. The $in()$ set of a basic block is the set of variables live just before the basic block is entered. Finally, the $out()$ set of a basic block is the set of variables live just after the block is exited. The $use()$ and $def()$ sets can be determined by a single scan of every basic block in a function. The $in()$ and $out()$ sets for each basic block, bb_i, are determined by the following equations,

$$in(bb_i) = use(bb_i) + (out(bb_i) - def(bb_i)) \tag{3.1}$$

$$out(bb_i) = \bigcup(in(bb_j)), bb_j \ a \ successor \ of \ bb_i \tag{3.2}$$

These equations need to be iterated until a stable solution is found. For completeness, an algorithm for performing live variable analysis is given in Figure 3.1.

The definitions of a variable available at each point in a program are determined by performing available definition analysis. A definition d is available at a point p if there is a path from d to p, and d is not destroyed along that path. A definition of a variable is destroyed by another definition of that variable. Four dataflow sets are used to calculate available definition information, $a\_gen()$, $a\_kill()$, $a\_in()$, and $a\_out()$. The $a\_gen()$ set of a basic block comprises the definitions in the basic block that reach the end of the block. The $a\_kill()$ set of a basic block contains all definitions that are destroyed by definitions in the block. The $a\_in()$ set of a basic block consists of all definitions that reach the entry point of the block. The $a\_out()$ set of a basic block consists of all definitions that reach the exit point of the block. Before $a\_gen()$ and a_kill can be calculated, all operations which define a variable and all operations which use a variable must be determined. Two sets to hold this information are associated with each program variable, $op\_def()$ and $op\_use()$. An algorithm to calculate this information is shown in Figure 3.2.

```
compute_live_var(function)
{
    /* Compute def and use sets for each basic block */
    for each basic block in function, bb_i {
        def(bb_i) = 0
        use(bb_i) = 0
        for each operation in bb_i, oper_j {
            def(bb_i) = def(bb_i) + {dest(oper_j)}
            for each variable in src(oper_j), src_k {
                if src_k is used before defined in bb_i then
                    use(bb_i) = use(bb_i) + {src_k}
            }
        }
    }
    /* Initialize in of each basic block to 0 */
    for each basic block in function, bb_i {
        in(bb_i) = 0
    }
    /* Iteratively compute in and out for each basic block */
    change = 1
    while (change ≠ 0) {
        change = 0
        for each basic block in function, bb_i {
            old_in = in(bb_i)
            out(bb_i) = ⋃(in(bb_j)), bb_j a successor of bb_i
            in(bb_i) = use(bb_i) + (out(bb_i) − def(bb_i))
            diff = in(bb_i) − old_in
            if (diff ≠ 0) then
                change = change + 1
        }
    }
}
```

Figure 3.1: Algorithm for computing live variables.

```
compute_operand_information(function)
{
    /* Initialize op_def and op_use of each operand to 0 */
    for each variable used as a source or destination operand in function, operand_i {
        op_def(operand_i) = 0
        op_use(operand_i) = 0
    }
    /* Compute op_def and op_use for each operand */
    for each basic block in function, bb_i {
        for each operation in bb_i, oper_j {
            op_def(dest(oper_j)) = op_def(dest(oper_j)) + {oper_j}
            for each variable in src(oper_j), src_k {
                op_use(src_k) = op_use(src_k) + {oper_j}
            }
        }
    }
}
```

Figure 3.2: Algorithm for computing all definitions and uses of an operand.

The $a\_gen()$ and $a\_kill()$ can now be determined by a single pass over each basic block. The $a\_in()$ and $a\_out()$ are found by the following equations:

$$a\_in(bb_i) = \bigcap(a\_out(bb_j)), bb_j \ a \ predecessor \ of \ bb_i \tag{3.3}$$

$$a\_out(bb_i) = a\_gen(bb_i) + (a\_in(bb_i) - a\_kill(bb_i)) \tag{3.4}$$

These equations need to be iterated until a stable solution is found. An algorithm for performing available definition analysis is given in Figure 3.3.

The expressions available at each point in a program are determined by performing available expression analysis. An expression is the evaluation of one or more source operands. For example, the operation $a = b + c$, contains the expression $b + c$. An expression, e, is available at a point p if there is a path from e to p, and e is not destroyed along that path. An expression is destroyed by a definition to any of its source operands. Four dataflow sets are used to calculate available expression information, $e\_gen()$, $e\_kill()$, $e\_in()$, and $e\_out()$. The $e\_gen()$ set of a basic block contains the expressions in the basic block that reach the end of the block. The $e\_kill()$ set of a basic block consists of all expressions that are destroyed by definitions in the basic block. The $e\_in()$ set of a basic block consists of all expressions that reach the entry point of the block. The $e\_out()$ set of a basic block contains all expressions that reach the exit point of the block. The $op\_def()$ and $op\_use()$ sets discussed in the previous subsection are similarly used here. The $e\_gen()$ and $e\_kill()$ can be determined by a single pass over each basic block. Then, $e\_in()$ and $e\_out()$ for each basic block, bb_i, are determined by the following equations:

$$e\_in(bb_i) = \bigcap(e\_out(bb_i)), bb_j \ a \ predecessor \ of \ bb_i \tag{3.5}$$

$$e\_out(bb_i) = e\_gen(bb_i) + (e\_in(bb_i) - e\_kill(bb_i)) \tag{3.6}$$

These equations need to be iterated until a stable solution is found. An algorithm for performing available expression analysis is shown in Figure 3.4.

```
compute_available_def(function)
    /* Compute a_gen and kill sets for each basic block */
    for each basic block in function, bb_i {
        a_gen(bb_i) = 0
        a_kill(bb_i) = 0
        for each operation in bb_i, oper_j {
            a_g = {oper_j}
            a_k = op_def(dest(oper_j)) - {oper_j}
            a_gen(bb_i) = a_g + (a_gen(bb_i) - a_k)
            a_kill(bb_i) = a_k + (a_kill(bb_i) - a_g)
        }
    }
    /* Initialize a_in and a_out of each basic block */
    U = the set of all operations in function
    a_in(bb_0) = 0
    a_out(bb_0) = a_gen(bb_0)
    for each basic block in function, bb_i, (i ≠ 0) {
        a_in(bb_i) = 0
        a_out(bb_i) = U - a_kill(bb_i)
    }
    /* Iteratively compute a_in and a_out sets for each basic block */
    change = 1
    while (change ≠ 0) {
        for each basic block in function, bb_i, (i ≠ 0) {
            old_a_in = a_in(bb_i)
            old_a_out = a_out(bb_i)
            a_in(bb_i) = ⋂(a_out(bb_j)), bb_j a predecessor of bb_i
            a_out(bb_i) = a_gen(bb_i) + (a_in(bb_i) - a_kill(bb_i))
            diff_in = old_a_in - a_in(bb_i)
            diff_out = old_a_out - a_out(bb_i)
            if (diff_in ≠ 0)
                change = change + 1
            if (diff_out ≠ 0)
                change = change + 1
        }
    }
}
```

Figure 3.3: Algorithm for computing available definitions.

```
compute_available_expr(function)
    /* Compute e_gen and e_kill sets for each basic block */
    for each basic block in function, bb_i {
        e_gen(bb_i) = 0
        e_kill(bb_i) = 0
        for each operation in bb_i, oper_j {
            e_k = op_use(dest(oper_j))
            if oper_j in e_k then
                e_g = 0
            else
                e_g = {oper_j}
            e_gen(bb_i) = e_g + (e_gen(bb_i) - e_k)
            e_kill(bb_i) = e_k + (e_kill(bb_i) - e_g)
        }
    }
    /* Initialize e_in and e_out of each basic block */
    U = the set of all operations in function
    e_in(bb_0) = 0
    e_out(bb_0) = e_gen(bb_0)
    for each basic block in function, bb_i, (i ≠ 0) {
        e_in(bb_i) = 0
        e_out(bb_i) = U - e_kill(bb_i)
    }
    /* Iteratively compute e_in and e_out sets for each basic block */
    change = 1
    while (change ≠ 0) {
        for each basic block in function, bb_i, (i ≠ 0) {
            old_e_in = e_in(bb_i)
            old_e_out = e_out(bb_i)
            e_in(bb_i) = ⋂(e_out(bb_j)), bb_j a predecessor of bb_i
            e_out(bb_i) = e_gen(bb_i) + (e_in(bb_i) - e_kill(bb_i))
            diff_in = old_e_in - e_in(bb_i)
            diff_out = old_e_out - e_out(bb_i)
            if (diff_in ≠ 0)
                change = change + 1
            if (diff_out ≠ 0)
                change = change + 1
        }
    }
```

Figure 3.4: Algorithm for computing available expressions.

3.3.2 Constant propagation

Global constant propagation replaces future uses of a variable which is assigned a constant, by that constant. If all subsequent uses of the variable can be replaced by the constant, the assignment operation becomes unnecessary and can be eliminated by later dead code removal. Constant propagation must stop, however, when a new value could be assigned to the variable. This is determined by examining the data flow information at each use of the variable. The optimization rule for constant propagation is given below.

- Predicate functions
  1. $op(x)$ is a move operation.
  2. $src1(x)$ is a constant.
  3. At least 1 of the variables in $src(y)$ matches $dest(x)$.
  4. $op(x)$ is in $a\_in(bb(y))$.
  5. $dest(x)$ is not modified by $\{op(j), j = first(bb(y)) \cdots y - 1\}$.
  6. There is no danger in $\{op(j), j = x + 1 \cdots y - 1\}$. (There may be many of these paths; this predicate implies no danger along any of the possible paths.)
- Actions
  1. If $src1(y) = dest(x)$, then change $src1(y)$ to $src1(x)$.
  2. If $src2(y) = dest(x)$, then change $src1(y)$ to $src1(x)$.
  3. If $src3(y) = dest(x)$, then change $src3(y)$ to $src1(x)$.

3.3.3 Forward copy propagation

Global forward copy propagation replaces future uses of a variable x that is assigned the value of another variable ($x = y$), with the other variable, y. If all subsequent uses of x can be replaced by y, the assignment operation becomes unnecessary and can be eliminated by later dead code removal. Forward copy propagation must stop, however, if either variable is possibly modified. The optimization rule for global forward copy propagation is given below.

- Predicate functions
  1. $op(x)$ is a move operation.
  2. $src1(x)$ is a register.
  3. At least 1 of the variables in $src(y)$ matches $dest(x)$.
  4. $op(x)$ is in $a\_in(bb(y))$.
  5. $dest(x)$ is not modified by $\{op(j), j = first(bb(y)) \cdots y-1\}$.
  6. $op(x)$ is in $e\_in(bb(y))$.
  7. The variables in $src(x)$ are not modified by $\{op(j), j = first(bb(y)) \cdots y-1\}$.
  8. There is no danger in $\{op(j), j = x+1 \cdots y-1\}$.
- Actions
  1. If $src1(y) = dest(x)$, then change $src1(y)$ to $src1(x)$.
  2. If $src2(y) = dest(x)$, then change $src2(y)$ to $src1(x)$.
  3. If $src3(y) = dest(x)$, then change $src3(y)$ to $src1(x)$.

3.3.4 Backward copy propagation

Global backward copy propagation is similar to forward copy propagation; however, the variables are propagated backwards instead of forwards. Two variables may have the same value at some point in the program (e.g., $x = y$); however, it may not be possible to replace all uses of x by y. Thus, the assignment operation can not be eliminated after forward copy propagation. In this case, it may be possible to change the destination of operations which define y to x. If all definitions of y that reach the assignment operation can be changed, the assignment operation can be eliminated. The optimization rule for global backward copy propagation is given below.

- Predicate functions
  1. $bb(x) = bb(y)$.
  2. $dest(x)$ is a register.
  3. $op(x)$ is not in $a\_out(bb(y))$
  4. $op(y)$ is a move operation.

5. $dest(y)$ is a register.
6. There is a flow dependency between $op(x)$ and $op(y)$.
7. $dest(y)$ is not modified in $\{op(j), j = x \cdots y-1\}$.
8. $dest(y)$ is not used in $\{op(j), j = x+1 \cdots y-1\}$.
9. If $dest(y)$ is redefined in $bb(y)$, then there are no uses of $src1(y)$ after the first redefinition.
10. There is no danger in $\{op(j), j = x+1 \cdots y-1\}$.

- Actions

1. $old\_dest = dest(x)$.
2. $new\_dest = dest(y)$.
3. If $dest(y)$ is redefined in $bb(y)$, let $redef$ be the first operation succeeding $op(y)$ that redefines $dest(y)$; otherwise, $redef = last(bb(y))$.
4. Delete $op(y)$.
5. For each $\{op(j), j = x+1 \cdots redef\}$, change all uses of old_dest to new_dest.

3.3.5 Memory copy propagation

Global memory copy propagation replaces a load of a memory variable with a register assignment if the current value of the memory variable exists in a register. Propagation stops whenever there is a possible direct or indirect modification of the memory variable. Indirect modifications are detected by performing global memory disambiguation. The optimization rule for global memory copy propagation is shown below.

- Predicate functions

1. $op(x)$ is a store operation.
2. $op(y)$ is a load operation.
3. $op(x)$ and $op(y)$ are compatible data types.
4. $address(x) = address(y)$.
5. $op(x)$ is in $e\_in(bb(y))$.
6. The variables in $src(x)$ are not modified by $\{op(j), j = first(bb(x)) \cdots y-1\}$.

7. There is no write to $address(x)$ in $\{op(j), j = x + 1 \cdots y - 1\}$. (This set of instructions may not be unique; this predicate implies no write to $address(x)$ for all possible instructions between $op(x)$ and $op(y)$.)
8. There is no danger in $\{op(j), j = x + 1 \cdots y - 1\}$.

- Actions

1. Change $op(y)$ to a move operation, with $src1(y) = src3(x)$.

3.3.6 Arithmetic common subexpression elimination

Global arithmetic common subexpression elimination removes the recomputation of an expression whenever a previously computed value can be used. A previous computation of the expression can be used if all of the source operands are never modified between computations. The optimization rule for global arithmetic common subexpression elimination is presented below.

- Predicate functions

1. $op(x)$ is an arithmetic operation.
2. $f_x = f_y$.
3. $src(x) = src(y)$.
4. $dest(x)$ is not the same as any of the variables in $src(x)$.
5. $dest(y)$ is not the same as any of the variables in $src(y)$.
6. $op(x)$ is in $e\_in(bb(y))$.
7. The variables in $src(x)$ are not modified by $\{op(j), j = first(bb(y)) \cdots y - 1\}$.
8. There is no danger in $\{op(j), j = x + 1 \cdots y - 1\}$.

- Actions

1. Insert a new move operation, $op(a)$, after $op(x)$, with $dest(a) = temp\_reg$ and $src1(a) = dest(x)$.
2. Change $op(y)$ to a move operation, with $src1(y) = dest(a)$; ($dest(y)$ remains unchanged).

3.3.7 Redundant load elimination

Global redundant load elimination removes unnecessary memory load operations if there is a previous load of the same memory variable. There can be no possible modifications, either direct or indirect, of the memory variable between the load operations to apply the optimization. The optimization rule for global redundant load elimination is given below.

- Predicate functions
  1. $op(x)$ is a load operation.
  2. $f_x = f_y$.
  3. $src(x) = src(y)$.
  4. $dest(x)$ is not the same as any of the variables in $src(x)$.
  5. $dest(y)$ is not the same as any of the variables in $src(y)$.
  6. $op(x)$ is in $e\_in(bb(y))$.
  7. The variables in $src(x)$ are not modified by $\{op(j), j = first(bb(y)) \cdots y-1\}$.
  8. There is no write to $address(x)$ in $\{op(j), j = x+1 \cdots y-1\}$. (There may be many of these paths; this predicate implies no writes to $address(x)$ along any of the possible paths.)
  9. There is no danger in $\{op(j), j = x+1 \cdots y-1\}$.
- Actions
  1. Insert a new move operation, $op(a)$, after $op(x)$, with $dest(a) = temp\_reg$ and $src1(a) = dest(x)$.
  2. Change $op(y)$ to a move operation, with $src1(y) = dest(a)$; ($dest(y)$ remains unchanged).

3.3.8 Redundant store elimination

Global redundant store elimination removes unnecessary memory store operations if there is a previous store of the same value to the same address. There can be no possible modifications of either the memory variable or the value to be stored between the store operations. The optimization rule for global redundant store elimination is given below.

- Predicate functions
  1. $op(x)$ is a store operation.
  2. $f_x = f_y$.
  3. $src(x) = src(y)$.
  4. $op(x)$ is in $e\_in(bb(y))$.
  5. The variables in $src(x)$ are not modified by $\{op(j), j = first(bb(y)) \cdots y - 1\}$.
  6. There is no write to $address(x)$ in $\{op(j), j = x + 1 \cdots y - 1\}$.
  7. There is no danger in $\{op(j), j = x + 1 \cdots y - 1\}$.
- Actions
  1. Delete $op(y)$.

3.3.9 Dead code removal

Dead code removal removes operations whose destination variable is not subsequently used in the program. Dead code removal does not usually result from the programmer inserting unnecessary code, but results after other local and global optimizations. An algorithm to detect and perform global dead code removal is given below.

- Predicate functions
  1. $op(x)$ can be deleted.
  2. $dest(x)$ is a register.
  3. There are no flow dependencies from $op(x)$.
  4. $op(x)$ is not in $a\_out(bb(x))$.
- Actions
  1. Delete $op(x)$.

3.4 Loop Optimization

A set of special optimizations exists to be performed on loops within a program. Most programs spend much of their execution time within loops, thus a large amount

of speedup can be expected from an effective loop optimizer. Loops have two important properties. First, a loop has a unique entry point, which is referred to as the *header*. No block in the loop may be executed unless the *header* has been previously executed. Thus, there may be no side entrances into a loop considered for optimization. Second, there is at least one path from a basic block within the loop back to the *header*. A path of this type is referred to as a *backedge*. Identifying loops and the steps necessary for loop optimization are discussed in the following sections.

3.4.1 Identifying loops

Identifying loops in a control flow graph can be broken down into three steps, finding dominators, loop detection, and loop merging.

A basic block, bb_i, dominates a basic block, bb_j, if all paths from the initial basic block to bb_j go through bb_i [1]. Thus, every basic block dominates itself. For completeness, an algorithm for computing dominators is given in Figure 3.5. The algorithm iteratively computes $dom(bb_i)$, which is the set of basic blocks which dominate bb_i. Each function must have a unique entry block which will be designated as bb_0.

After the dominators for each basic block have been determined, *backedges* can be recognized by a control flow arc from a basic block, bb_j, to another basic block, bb_i, in which bb_i dominates bb_j [1]. The basic blocks in the loop are then all basic blocks which can reach bb_j without going through bb_i. An algorithm to detect *backedges* and find all basic blocks contained within the loop defined by the *backedge* is presented in Figure 3.6.

Loop optimization can be complicated if two or more loops can have the same *header*. Therefore, all loops with the same *header* are combined into a single loop, containing multiple *backedges*. It is possible that more conservative loop optimization decisions result after merging; however, this approach was chosen to reduce the complexity of loop

```
find_dominator(function)
{
        U = the set of all basic blocks in function
        dom(bb_0) = {bb_0}
        for all basic blocks in function, bb_i, (i ≠ 0) {
                dom(bb_i) = U
        }
        change = 1
        while (change ≠ 0) {
                change = 0
                for all basic blocks in function, bb_i, (i ≠ 0) {
                        old = dom(bb_i)
                        dom(bb_i) = ⋂(dom(bb_j)), bb_j a predecessor of bb_i
                        dom(bb_i) = dom(bb_i) + {bb_i}
                        diff = old − dom(bb_i)
                        if (diff ≠ 0) then change = change + 1
                }
        }
}
```

Figure 3.5: Algorithm for finding dominators.

```
add_blocks(loop, bb, header)
{
        for each predecessor block of bb, pred_i {
                if (pred_i = header) then continue
                if (pred_i.visited = 1) then continue
                pred_i.visited = 1
                loop.loop_bb = loop.loop_bb + {pred_i}
                add_blocks(loop, pred_i, header)
        }
}

loop_det(function)
{
        for each basic block in function, bb_i {
                for each successor block of bb_i, succ_i {
                        if (succ_i in dom(bb_i)) then {
                                Create a element in the loop data structure, loop_i
                                loop_i.loop_bb = {succ_i}
                                loop_i.header = succ_i
                                if (bb_i ≠ succ_i) {
                                        loop_i.loop_bb = loop_i.loop_bb + {bb_i}
                                        set bb_j.visited to 0 for all j
                                        add_blocks(loop_i, bb_i, succ_i)
                                }
                        }
                }
        }
}
```

Figure 3.6: Algorithm for detection of single *backedge* loops.

optimization. An algorithm for merging loops with the same header block is given in Figure 3.7.

```
merge_loops(function)
{
    for each loop in function, loop_i {
        for each loop in function, loop_j, (j ≠ i) {
            if (loop_i.header = loop_j.header) {
                loop_i.loop_bb = loop_i.loop_bb + loop_j.loop_bb
                remove loop_j from loop data structure for function
            }
        }
    }
}
```

Figure 3.7: Algorithm for merging loops with the same header block.

3.4.2 Creating loop preheaders

A loop *preheader* is a basic block which is used to contain operations that are to be executed once every time the loop is invoked. A loop has a *preheader* if the *header* has one predecessor block which is not in the loop, and that predecessor block has one successor, the *header*. A *preheader* can be created for loops which do not have one by inserting a new basic block before the *header*; then, all control transfers to the *header* which are not from basic blocks in the loop are redirected to the *preheader*. A single control transfer is then added between the *preheader* and the *header*.

3.4.3 Invariant code removal

The basic principle of invariant code removal is to move operations whose source operands do not change within the loop to the loop's preheader block. Operations of this type are then executed only once each time the loop is invoked, rather than on every iteration. The optimization rule for loop invariant code removal is shown below.

- Predicate functions
  1. $op(x)$ can be moved; (branch, subroutine call, and synchronization operations are operations which can not be moved).
  2. The variables in $src(x)$ are not modified in the loop.
  3. $op(x)$ is the only operation in the loop which modifies $dest(x)$.
  4. There are no uses of $dest(x)$ in $\{op(j), j = first(bb(x)) \cdots x\}$.
  5. For all basic blocks, $bb(j)$, in the loop except $bb(x)$; if $dest(x)$ is used in the basic block, then $op(x)$ is in $a\_in(bb(j))$.
  6. For all basic blocks, $bb(j)$, from which the loop can be exited, if $dest(x)$ is in $out(bb(j))$, then $op(x)$ is in $a\_out(bb(j))$.
  7. If $op(x)$ is not executed on every iteration of the loop, then $op(x)$ must not possibly cause an exception. (Memory, floating-point, and integer divide operations are the most common operations which cannot be removed unless they are executed on every iteration; note page faults are not treated as exceptions for this predicate.)
  8. If $op(x)$ is a load or a store operation, then there are no writes to $address(x)$ in the loop.
  9. There is no danger in the loop.
- Actions
  1. Move $op(x)$ to after the last operation in the preheader block of the loop.

3.4.4 Global variable migration

Global variable migration moves a frequently accessed memory variable (globally declared scalar variable, array element, or structure element) into a register for the duration

of the loop. Loads and stores to a migrated variable within the loop are replaced by register accesses. A load instruction is inserted into the loop preheader to initialize the register, and a store is placed at each loop exit to update memory after the execution of the loop. The optimization rule for loop global variable migration is given below.

- Predicate functions
  1. $op(x)$ is a load or a store operation.
  2. The variables in $address(x)$ are not modified in the loop.
  3. If $op(x)$ is not executed on every iteration of the loop, then $op(x)$ must not possibly cause an exception.
  4. All memory operations, $op(j)$, in the loop whose address can equal $address(x)$ must have $address(j) = address(x)$.
  5. There is no danger in the loop.
- Actions
  1. Insert a new load operation, $op(a)$, after the last operation in the preheader of the loop, with $dest(a) = temp\_reg$ and $address(a) = address(x)$.
  2. Insert a new store operation, $op(b)$, with $address(b) = address(x)$ and $src3(b) = temp\_reg$, as the first operation of each basic block that can be immediately reached, bb_i, when the loop is exited. (Note, if bb_i can be reached from a basic block not in the loop, a new basic block is created to insert the store operation, and a jump operation is inserted at the end of the new block back to bb_i. Then, all branches in the loop whose target is bb_i are redirected to the new block. This procedure is necessary to preserve correctness for all possible paths of control flow entering bb_i.)
  3. Change all load operations in the loop, $op(j)$, with $address(j) = address(x)$ to move operations, with $src(j) = temp\_reg$; ($dest(j)$ remains unchanged).
  4. For all store operations in the loop, $op(j)$, with $address(j) = address(x)$, let $temp\_src = src3(j)$ and change $op(j)$ to a move operation with $dest(j) = temp\_reg$ and $src1(j) = temp\_src$.

3.4.5 Branch simplification

Some current processors (MIPS R2000) have machine instructions only for branch equal to (beq) and branch not equal to (bne). All other conditional branch operations

are broken down into two operations, a compare followed by a beq or bne. Branch simplification substitutes bne operations for conditional branches that would be broken down into two operations. Although branch simplification is a machine-dependent optimization, many opportunities for this optimization are lost if induction variable elimination is first performed. Thus, including branch simplification with other machine-dependent optimizations in a pass following machine-independent optimization is not effective, thus it has been included within the machine independent optimizer and is enabled for a group of target processors. The optimization rule for two types of loop branch simplification is presented below.

- Predicate functions for type 1
  1. $op(x)$ is a branch greater than (bgt), a branch greater than or equal (bge), a branch less than (blt), or a branch less than or equal (ble).
  2. $src1(x)$ is basic induction variable in the loop.
  3. $src2(x)$ is a numeric constant.
  4. $inc(src1(x))$ is a numeric constant.
  5. $init_val(src1(x))$ is a numeric constant.
  6. $target(x)$ is the loop header block.
  7. If $inc(src1(x))$ is positive, then $op(x)$ is a blt or a ble operation; otherwise, $op(x)$ is a bgt or bge operation.
  8. If $inc(src1(x))$ is positive, then $init_val(src1(x))$ is smaller than $src2(x)$; otherwise, $init_val(src1(x))$ is larger than $src2(x)$.
- Actions for type 1
  1. Calculate the new branch limit as follows:
     (a) $t1 = (src2(x) - init_val(src1(x))) \div inc(src1(x))$. (integer divide)
     (b) $t2 = t1 * inc(src1(x)) + init_val(src1(x))$.
     (c) If $op(x)$ is a ble or bge operation, or $t2 \neq src2(x)$, then $new\_limit = t2 + inc(src1(x))$; otherwise, $new\_limit = t2$.
  2. Change $op(x)$ to a branch not equal to (bne) operation, with $src2(x) = new\_limit$; ($src1(x)$ and $target(x)$ remain unchanged).
- Predicate functions for type 2

1. $op(x)$ is a bgt, bge, blt, or ble operation.
2. $src1(x)$ is basic induction variable in the loop.
3. $inc(src1(x))$ is either 1 or -1.
4. $src2(x)$ is not modified in the loop.
5. If $inc(src1(x))$ is positive, then $op(x)$ is a blt or a ble operation; otherwise, $op(x)$ is a bgt or bge operation.
6. If $inc(src1(x))$ is positive, then $init_val(src1(x))$ is always smaller than $src2(x)$; otherwise, $init_val(src1(x))$ is always larger than $src2(x)$. (Since $init_val(src1(x))$ and/or $src2(x)$ are not constants, this predicate returns true if all possible entries to the loop are preceded by a test of this condition.)

- Actions for type 2

1. If $op(x)$ is a ble or a bge operation, then insert a new add operation, $op(a)$, after the last operation in the loop preheader, with $dest(a) = new\_limit$, $src1(a) = src2(x)$, and $src2(a) = 1$; otherwise, $new\_limit = src2(x)$.
2. Change $op(x)$ to a bne operation, with $src2(x) = new\_limit$; ($src1(x)$ and $target(x)$ remain unchanged).

3.4.6 Induction variable strength reduction

A basic induction variable of a loop is a variable in which all modifications of the variable in a loop are increments by a constant amount. Induction variable strength reduction replaces variables whose value is a linear function (add, subtract, multiply) of a basic induction variable with a new basic induction variable. The linear function is thus replaced by an increment operation. Induction variable strength reduction is effective for two reasons. First, multiply operations in the loop can be converted to addition operations. Second, flow dependency chains within the loop are broken with this optimization, often leading to more freedom for later code scheduling; (this is why strength reduction is applied to add and subtract operations along with multiply operations). The optimization rule for loop induction variable strength reduction is given below.

- Predicate functions

1. $op(x)$ is a multiply, a logical left shift, an add, or a subtract operation.
2. $src1(x)$ is a basic induction variable in the loop.
3. $src2(x)$ is not modified in the loop.
4. $op(x)$ is the only operation in the loop which modifies $dest(x)$.
5. $dest(x)$ is not the same as any of the variables in $src(x)$.
6. $dest(x)$ is a register.
7. There is no danger in the loop.

- Actions

1. Insert a new operation having the same opcode as $op(x)$, $op(a)$, after the last operation in the loop preheader, with $dest(a) = new\_var$, $src1(a) = src1(x)$, and $src2(a) = src2(x)$.
2. Insert a new operation having the same opcode as $op(x)$, $op(b)$, after the last operation in the loop preheader, with $dest(b) = new\_inc$, $src1(b) =$ increment value of $src1(x)$, and $src2(b) = src2(x)$.
3. Create a new add operation, $op(c)$, with $dest(c) = new\_var$, $src1(c) = new\_var$, and $src2(c) = new\_inc$.
4. Insert a copy of $op(c)$ after each increment of $src1(x)$ in the loop.
5. Change $op(x)$ to a move operation with $src1(x) = new\_var$; ($dest(x)$ remains unchanged).

3.4.7 Induction variable elimination

Induction variable elimination removes unnecessary basic induction variables from a loop. Induction variables are trivially unnecessary if they are not used in the loop and not live when the loop is exited (type 1). Other induction variables can be eliminated by substituting all uses of a basic induction variable with another basic induction variable. Elimination of this type is broken down, based on the characteristics of the candidate basic induction variable for elimination and the basic induction variable to be substituted for all its uses. Four classes of induction elimination by substitution have been identified: the induction variables have the same increment and same initial value (type 2); the

induction variables have the same increment and their initial values are a known constant offset from one another (type 3); the induction variables have the same increment (type 4); the induction variables have different increments (type5). Each successive type also represents a more costly elimination; therefore, all eliminations of type 1 should be attempted, then type 2, and so forth. The optimization rule for loop induction variable elimination is shown below.

- Predicate functions for type 1
  1. $dest(x)$ is a basic induction variable in the loop.
  2. $dest(x)$ is not in any $in(bb(j))$, where $bb(j)$ is a block which can be immediately reached when the loop is exited.
  3. $dest(x)$ is not used in the loop; (increment operations with dest() and src1() equal to $dest(x)$ are not considered by this predicate).
- Actions for type 1
  1. Delete all increment operations in the loop, $op(j)$ with $dest(j) = dest(x)$.
- Predicate functions for type 2
  1. $dest(x)$ is a basic induction variable in the loop.
  2. $dest(y)$ is a basic induction variable in the loop.
  3. $dest(x)$ is not the same as $dest(y)$.
  4. $dest(x)$ is not in any $in(bb(j))$, where $bb(j)$ is a block which can be immediately reached when the loop is exited.
  5. $dest(x)$ and $dest(y)$ are basic induction variables in the same family; (two basic induction variables are in the same family if all increments of one are preceded/succeeded by a corresponding increment of the other).
  6. $inc(dest(x)) = inc(dest(y))$.
  7. $init\_val(dest(x)) = init\_val(dest(y))$.
  8. For each basic block with a definition of $dest(x)$, there are no uses of $dest(x)$ between the leading definition of either $dest(x)$ or $dest(y)$ and the last definition of either $dest(x)$ or $dest(y)$.
  9. There is no danger in the loop.
- Actions for type 2

1. Delete all increment operations in the loop, $op(j)$ with $dest(j) = dest(x)$.
2. Replace all uses of $dest(x)$ as a source operand in the loop with $dest(y)$.

- Predicate functions for type 3

1. $dest(x)$ is a basic induction variable in the loop.
2. $dest(y)$ is a basic induction variable in the loop.
3. $dest(x)$ is not the same as $dest(y)$.
4. $dest(x)$ is not in any $in(bb(j))$, where $bb(j)$ is a block which can be immediately reached when the loop is exited.
5. $dest(x)$ and $dest(y)$ are basic induction variables in the same family.
6. $inc(dest(x)) = inc(dest(y))$.
7. $init\_val(dest(x)) - init\_val(dest(y))$ is a constant.
8. All operations in the loop, $op(j)$, which use $dest(x)$ as a source operand are move, add, subtract, load, store, comparison, or branch operations, and have only $src1(j) = dest(x)$, e.g., $src2(j)$ and $src3(j)$ are not the same as $dest(x)$. Note, if $op(j)$ is communicative with respect to $src1(j)$ and $src2(j)$, $src1(j)$ and $src2(j)$ can be exchanged to expose more optimization opportunities.
9. All operations in the loop, $op(j)$, with $src1(j) = dest(x)$, have a numerical constant as $src2(j)$.
10. For each basic block with a definition of $dest(x)$, there are no uses of $dest(x)$ between the leading definition of either $dest(x)$ or $dest(y)$ and the last definition of either $dest(x)$ or $dest(y)$.
11. There is no danger in the loop.

- Actions for type 3

1. Delete all increment operations in the loop, $op(j)$, with $dest(j) = dest(x)$.
2. $offset = init\_val(dest(x))$ - $init\_val(dest(y))$.
3. For all operations in the loop, $op(j)$, with $src1(j) = dest(x)$, let $old\_src2 = src2(j)$, and perform the following:
   (a) $src1(j) = dest(y)$.
   (b) If $op(j)$ is a move operation, change $op(j)$ to an add operation, with $src2(j) = offset$; ($dest(j)$ and $src1(j)$ remain unchanged).
   (c) If $op(j)$ is an add, a load, or a store operation, then change $src2(j)$ to $old\_src2 + offset$.
   (d) If $op(j)$ is a subtract, a comparison, or a conditional branch operation, then change $src2(j)$ to $old\_src2 - offset$.

- Predicate functions for type 4

  1. $dest(x)$ is a basic induction variable in the loop.
  2. $dest(y)$ is a basic induction variable in the loop.
  3. $dest(x)$ is not the same as $dest(y)$.
  4. $dest(x)$ is not in any $in(bb(j))$, where $bb(j)$ is a block which can be immediately reached when the loop is exited.
  5. $dest(x)$ and $dest(y)$ are basic induction variables in the same family.
  6. $inc(dest(x)) = inc(dest(y))$.
  7. All operations in the loop, $op(j)$, which use $dest(x)$ as a source operand are move, add, subtract, load, store, comparison, or branch operations, and have only $src1(j) = dest(x)$.
  8. All operations in the loop, $op(j)$, with $src1(j) = dest(x)$, have variables that are not modified in the loop as $src2(j)$.
  9. For each basic block with a definition of $dest(x)$, there are no uses of $dest(x)$ between the leading definition of either $dest(x)$ or $dest(y)$ and the last definition of either $dest(x)$ or $dest(y)$.
  10. If $dest(y)$ can be eliminated and replaced with $dest(x)$, then there are more uses of $dest(y)$ in the loop than $dest(x)$.

- Actions for type 4

  1. Delete all increment operations in the loop, $op(j)$, with $dest(j) = dest(x)$.
  2. For all operations in the loop, $op(j)$, with $src1(j) = dest(x)$, let $old\_src2 = src2(j)$, (if $op(j)$ is a move, $old\_src2 = 0$), and perform the following:
     (a) Change $src1(j)$ to $dest(y)$.
     (b) If $op(j)$ is a move, add, load, or store, insert a new subtract operation, $op(a)$, after the last operation in the preheader of the loop, with $dest(a) = temp\_reg1$, $src1(a) = dest(x)$, and $src2(a) = dest(y)$; otherwise, insert a new subtract operation, $op(a)$, after the last operation in the preheader of the loop, with $dest(a) = temp\_reg1$, $src1(a) = dest(y)$, and $src2(a) = dest(x)$.
     (c) Insert a new add operation, $op(b)$, after the last operation in the preheader of the loop, with $dest(b) = temp\_reg2$, $src1(b) = old\_src2$, and $src2(b) = dest(a)$.
     (d) If $op(j)$ is a move operation, change $op(j)$ to an add operation with $src2(j) = dest(b)$, ($dest(j)$ and $src1(j)$ remain unchanged); otherwise, change $src2(j)$ to $dest(b)$.

- Predicate functions for type 5

1. $dest(x)$ is a basic induction variable in the loop.
2. $dest(y)$ is a basic induction variable in the loop.
3. $dest(x)$ is not the same as $dest(y)$.
4. $dest(x)$ is not in any $in(bb(j))$, where $bb(j)$ is a block which can be immediately reached when the loop is exited.
5. $dest(x)$ and $dest(y)$ are basic induction variables in the same family.
6. $inc(dest(x))$ and $inc(dest(y))$ are both integers.
7. $inc(dest(x))$ evenly divides $inc(dest(y))$, e.g., $inc(dest(y))$ % $inc(dest(x)) = 0$.
8. All operations in the loop, $op(j)$, which use $dest(x)$ as a source operand are comparison or branch operations, and have only $src1(j) = dest(x)$.
9. All operations in the loop, $op(j)$, with $src1(j) = dest(x)$, have variables that are not modified in the loop as $src2(j)$.
10. For each basic block with a definition of $dest(x)$, there are no uses of $dest(x)$ between the leading definition of either $dest(x)$ or $dest(y)$ and the last definition of either $dest(x)$ or $dest(y)$.
11. If $dest(y)$ can be eliminated and replaced with $dest(x)$, then there are more uses of $dest(y)$ in the loop than $dest(x)$.

- Actions for type 5

1. Delete all increment operations in the loop, $op(j)$, with $dest(j) = dest(x)$.
2. $ratio = inc(dest(y))/inc(dest(x))$.
3. For all operations in the loop, $op(j)$, with $src1(j) = dest(x)$, let $old\_src2 = src2(j)$, and perform the following:
   (a) Change $src1(j)$ to $dest(y)$.
   (b) Insert a new subtract operation, $op(a)$, after the last operation in the preheader of the loop, with $dest(a) = temp\_reg1$, $src1(a) = old\_src2$, and $src2(a) = dest(x)$.
   (c) Insert a new multiply operation, $op(b)$, after the last operation in the preheader of the loop, with $dest(b) = temp\_reg2$, $src1(b) = dest(a)$, and $src2(b) = ratio$.
   (d) Insert a new add operation, $op(c)$, after the last operation in the preheader of the loop, with $dest(c) = temp\_reg$, $src1(c) = dest(b)$, and $src2(c) = dest(y)$.
   (e) Change $src2(j)$ to $dest(c)$.

(f) If $ratio$ is negative, then change the opcode of $op(j)$ to the inverse compare or branch condition; (a negative ratio means that both operators of a comparison have been multiplied by a negative number, therefore the comparison must be inverted, e.g., $a < b$ and $-a \geq -b$).

4. USING PROFILE INFORMATION TO FURTHER CLASSICAL OPTIMIZATIONS

Classical optimization techniques use static loop analysis to identify important sections of a program. However, there are three deficiencies with static analysis: the outcome of an *if* statement is not always predictable at compile time, the iteration count of a loop is not always predictable at compile time, and the invocation count of a function is not always predictable at compile time. Many of these constructs can be nested within each other, often yielding large prediction errors for static analysis. Profiling more accurately identifies the most important program regions. Profile information has been used for many program optimizations, including branch prediction [15], function inline expansion [16], intelligent code placement [14], intelligent generation of switch statements [6], and register allocation [21]. It is also possible to modify classical code optimizations to account for available profile information to make more aggressive optimization decisions [9]. In this chapter, the use of profile information in classical local, global, and loop optimizations is discussed.

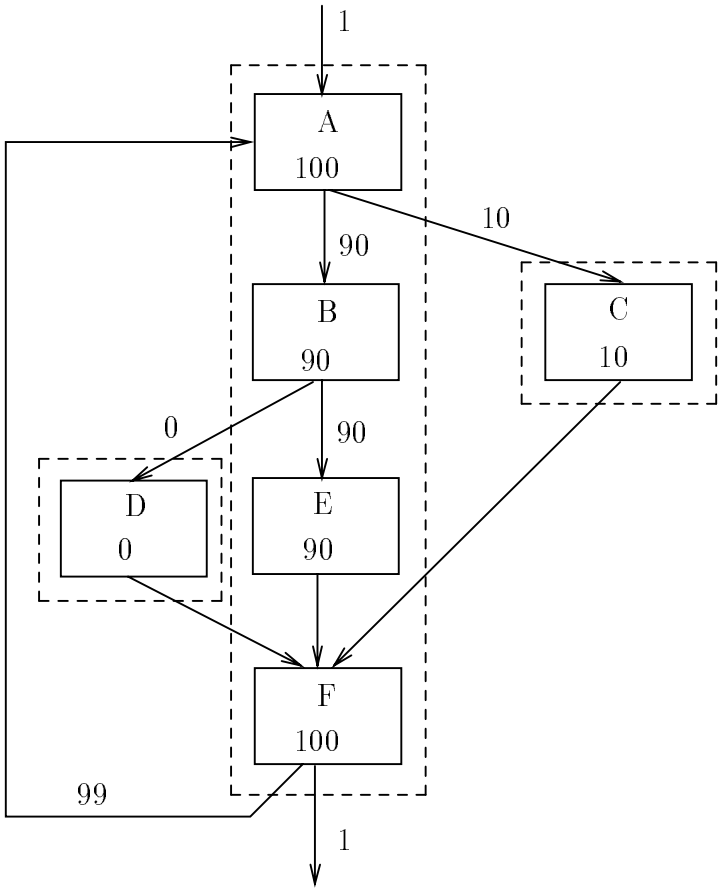

Figure 4.1: An example weighted control flow graph.

4.1 Optimizing Frequently Executed Paths

All profile-based code optimizations that will be presented in this section explore a single concept: *optimizing the most frequently executed paths*. This concept is illustrated using an example. Figure 4.1 shows a weighted control flow graph which represents a loop program. The *count* of basic blocks $\{A, B, C, D, E, F\}$ is $\{100, 90, 10, 0, 90, 100\}$, respectively. The *arc_count* of $\{A \rightarrow B, A \rightarrow C, B \rightarrow D, B \rightarrow E, C \rightarrow F, D \rightarrow F, E \rightarrow F, F \rightarrow A\}$ is $\{90, 10, 0, 90, 10, 0, 90, 99\}$, respectively. Clearly, the most frequently executed path in this example is the basic block sequence $< A, B, E, F >$. Because basic blocks in

this sequence are executed many more times than basic blocks C and D, the code optimizer can apply transformations that reduce the execution time of the $< A, B, E, F >$ sequence, but increase the execution time of basic blocks C and D. The formulation of nonloop-based classic code optimizations is conservative and does not perform transformations that may increase the execution time of any basic block. The formulation of loop-based classic code optimizations considers the entire loop body as a whole and does not consider the case in which some basic blocks in the loop body are rarely executed because of a very biased if statement. In the rest of this chapter, several profile-based code optimizations that make more aggressive decisions and explore more optimization opportunities are described.

A simple data structure, called a super-block, is used to represent a frequently executed path. A super-block has the following features: (1) It is a linear sequence of basic blocks $B(i), i = 1 \cdots n$, where $n \geq 1$. (2) It can be entered only from $B(1)$. (3) The program control may leave the super-block from any basic block. The set of all basic blocks that may be reached when control leaves the super-block from basic block $B(i)$ is denoted by $OUT(i)$. (4) When a super-block is executed, it is very likely that all basic blocks in that super-block are executed.

4.2 Forming Super-blocks

The formation of super-blocks is a two-step procedure: (1) trace selection and (2) tail duplication. Trace selection identifies basic blocks that tend to execute in a sequence and groups them into a trace [13] [12]. The definition of a trace is the same as the definition of a super-block, except that the program control is not restricted to enter at the first basic block. An experimental study of several trace selection algorithms was reported

in [5]. Figure 4.1 shows the result of trace selection. Each dotted-line box represents a trace. There are three traces: $\{A, B, E, F\}$, $\{C\}$, and $\{D\}$.

After trace selection, each trace is converted into a super-block by duplicating the tail part of the trace, to ensure that the program control can enter only at the top basic block. An algorithm for performing tail duplication is given in Figure 4.2. Using the example

```
algorithm tail_duplication(a trace B(1..n))
{
    Let B(i) be the first basic block that
      is an entry point to the trace, except for i=1;
    for (k=i..n) {
        create a trace that contains a copy of B(k);
        place the trace at the end of the function;
        redirect all control flows to B(k), except
          the ones from B(k-1), to the new trace;
    }
}
```

Figure 4.2: An algorithm to perform tail duplication.

in Figure 4.1, we see that there are two control paths that enter the $\{A, B, E, F\}$ trace at basic block F. Therefore, we duplicate the tail part of the $\{A, B, E, F\}$ trace starting at basic block F. Each duplicated basic block forms a new super-block that is appended to the end of the function. The result is shown in Figure 4.3. More code transformations can be applied after tail duplication to eliminate jump instructions. For example, the F' super-block in Figure 4.3 could be duplicated and each copy be combined with the C and D super-blocks to form two larger super-blocks.

To control the amount of code duplication, a basic block is added to a trace only if the execution count of that basic block is more than some threshold value, e.g., 100. After forming super-blocks, we optimize only those super-blocks whose execution counts are higher than the threshold value.

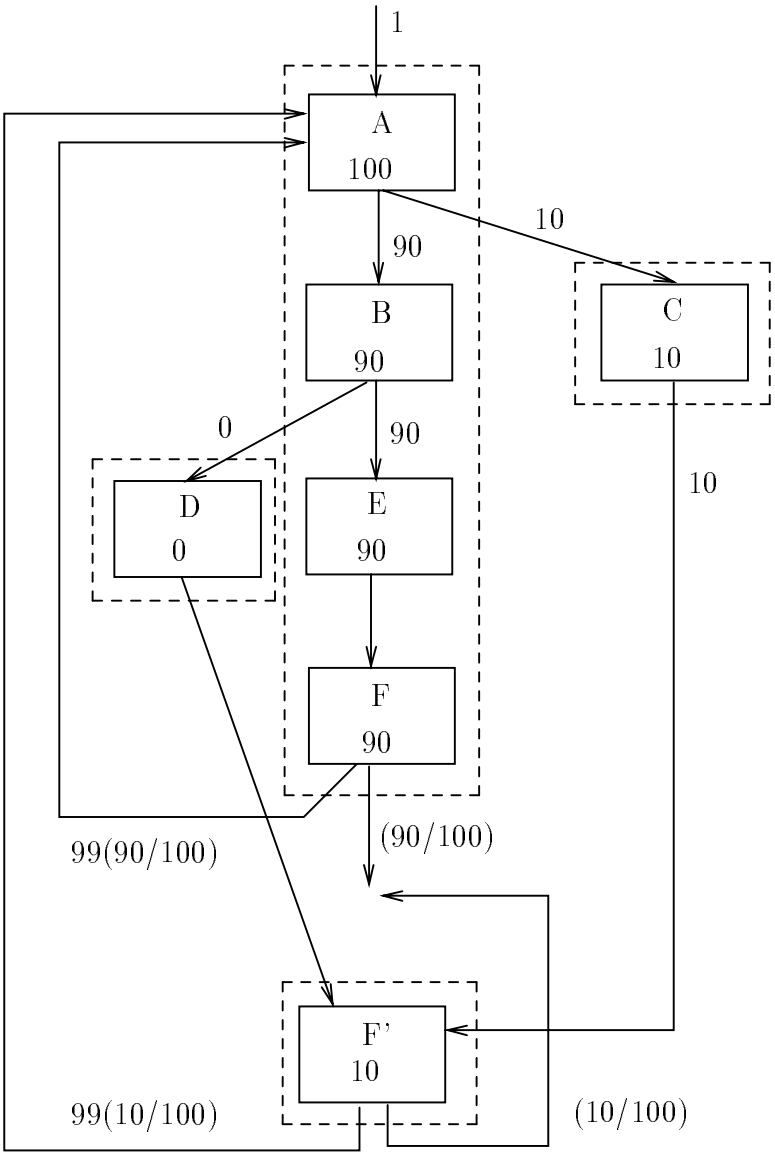

Figure 4.3: An example weighted control flow graph after tail duplication.

Table 4.1: Super-block code optimizations.

name	*scope*
constant propagation	super-block
forward copy propagation	super-block
backward copy propagation	super-block
memory copy propagation	super-block
arithmetic common subexpression elimination	super-block
redundant load elimination	super-block
redundant store elimination	super-block
constant combining	super-block
arithmetic operation folding	super-block
branch operation folding	super-block
operation cancellation	super-block
dead code removal	super-block
loop invariant code removal	super-block loop
loop induction variable elimination	super-block loop
loop global variable migration	super-block loop

4.3 Super-block Optimization

Table 4.1 shows a list of classic code optimizations that have been extended to use profile information. The nonloop-based code optimizations work on a single super-block at a time. The loop-based code optimizations work on a single super-block loop at a time. A super-block loop is a super-block that has a frequently taken *backedge* from its last node to its first node. The local, global, and loop optimizations discussed in the previous chapter can be extended in a straightforward manner to optimize super-blocks and super-block loops. Therefore, the super-block optimizations will not be formulated; however, a series of examples illustrating the effectiveness of super-block optimizations will be given.

Figure 4.4 shows a simple example of super-block common subexpression elimination. The original program is shown in Figure 4.4(a). After trace selection and tail duplication,

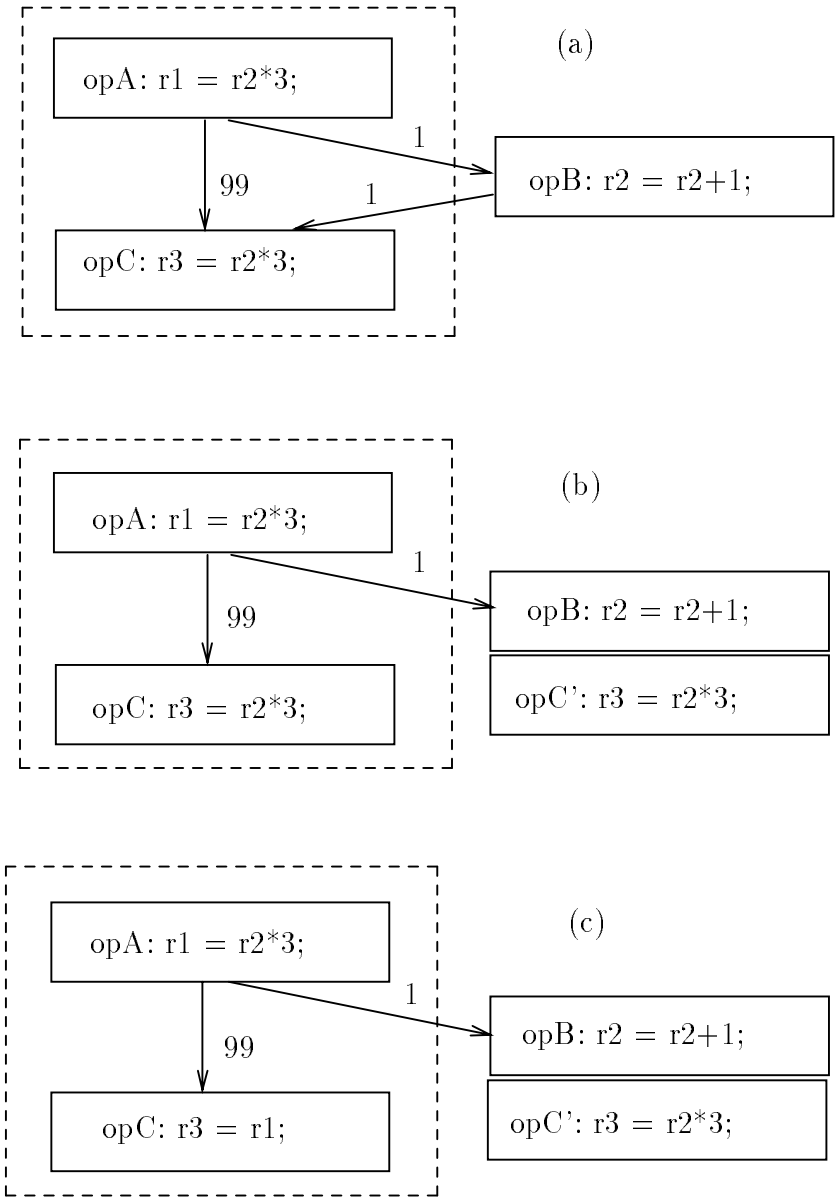

Figure 4.4: An example of super-block common subexpression elimination. (a) Original program segment. (b) Program segment after super-block formation. (c) Program segment after common subexpression elimination.

the equivalent program is shown in Figure 4.4(b). Because of tail duplication, opC cannot be reached from opB; therefore, common subexpression elimination can be applied to opA and opC. The resultant code segment after optimization is shown in Figure 4.4(c).

The definition of dead code can be extended for super-blocks. Traditional dead code removal (global dead code removal) removes operations whose value will never be used in the future. For super-blocks, an operation whose value is not used in the super-block and is not live at the end of the super-block can also be considered as dead code. However, the operation is not deleted but rather copied to all control flow paths that exit in the middle of the super-block in which its value may be used. In this manner, an operation is eliminated from the super-block. This extension is effective because program control rarely exits from the middle of a super-block. An example illustrating dead code removal in a super-block is given in Figure 4.5. The program is a simple loop that has been unrolled four times. The loop index variable (r0) has been expanded into four registers (r1, r2, r3, r4) that can be computed in parallel. If the loop index variable is live after the loop execution, then it is necessary to update the value of r0 in each iteration, as shown in Figure 4.5(a). According to the definition of super-block dead code removal, these update instructions (e.g., r0=r1, r0=r2, and r0=r3) become dead code, since their uses are replaced by r1, r2, r3, and r4. These update instructions can be moved out from the super-block, as shown in Figure 4.5(b).

Super-block loop optimizations can identify more optimization opportunities than traditional loop optimizations that must account for all possible execution paths within a loop. Super-block loop optimizations reduce the execution time of the most likely path of execution through a loop. In traditional loop optimizations, a potential optimization may be inhibited by a rare event such as a function call to handle a machine failure in a device driver program, or a function call to refill a large character buffer in text

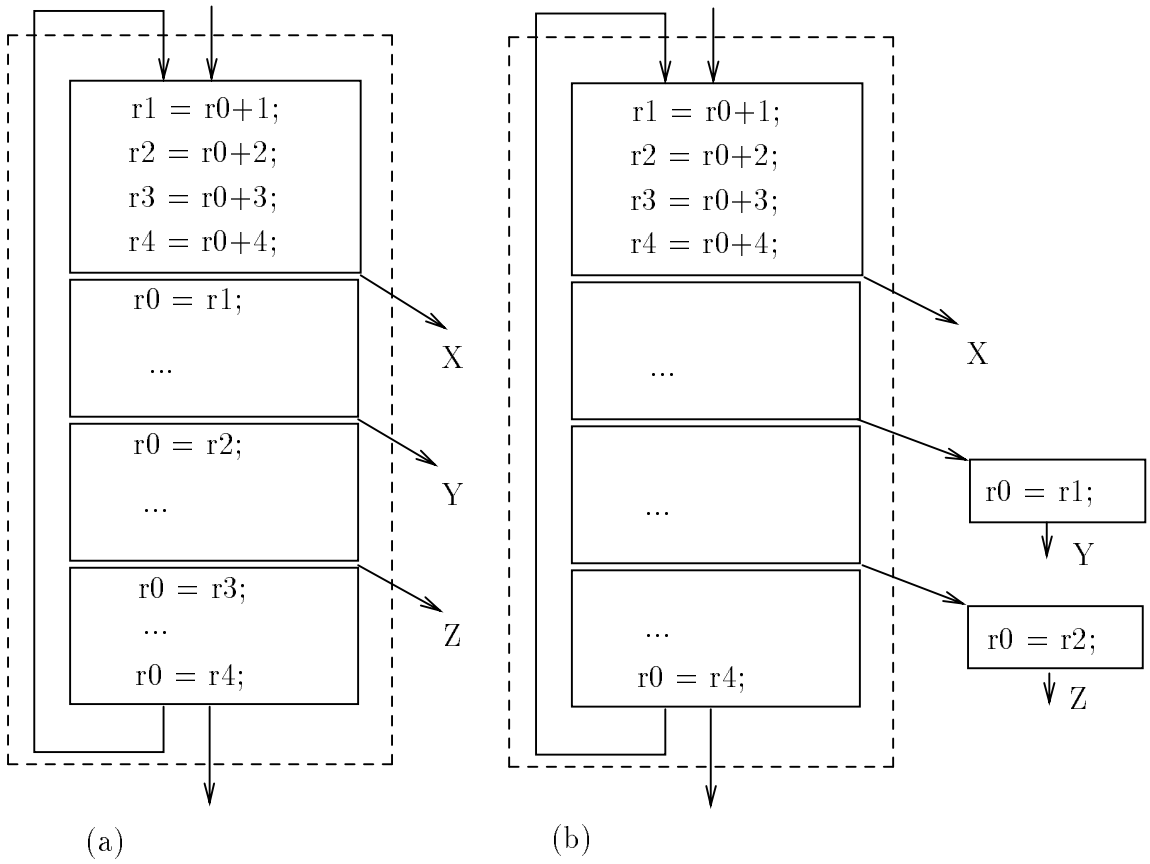

Figure 4.5: An example of super-block dead code removal. (a) Original program segment. (b) Program segment after dead code removal.

processing programs. In super-block loop optimizations, function calls that are not in the super-block loop do not affect the optimization of the super-block loop.

The increased optimization opportunities created by limiting the search space to within a super-block (versus the entire loop body) for loop invariant code removal is illustrated by the example in Figure 4.6. In Figure 4.6(a), opA is not loop invariant (in the traditional sense) because its source operand is a memory variable, and opD is a function call that may modify any memory variable (assuming that the compiler does not perform interprocedural memory disambiguation). On the other hand, opA is invariant in the super-block loop. The result of super-block loop invariant code removal is shown in Figure 4.6(b).

Figure 4.7 shows an example of super-block loop global variable migration. The memory variable x[r0] cannot be migrated to a register in traditional global variable migration, because r0 is not loop invariant in the entire loop. On the other hand, r0 is loop invariant in the super-block loop, and x[r0] can be migrated to a register by super-block global variable migration. The result is shown in Figure 4.7(b). Extra instructions (opX and opY) are added to the super-block loop boundary points to ensure correctness of execution.

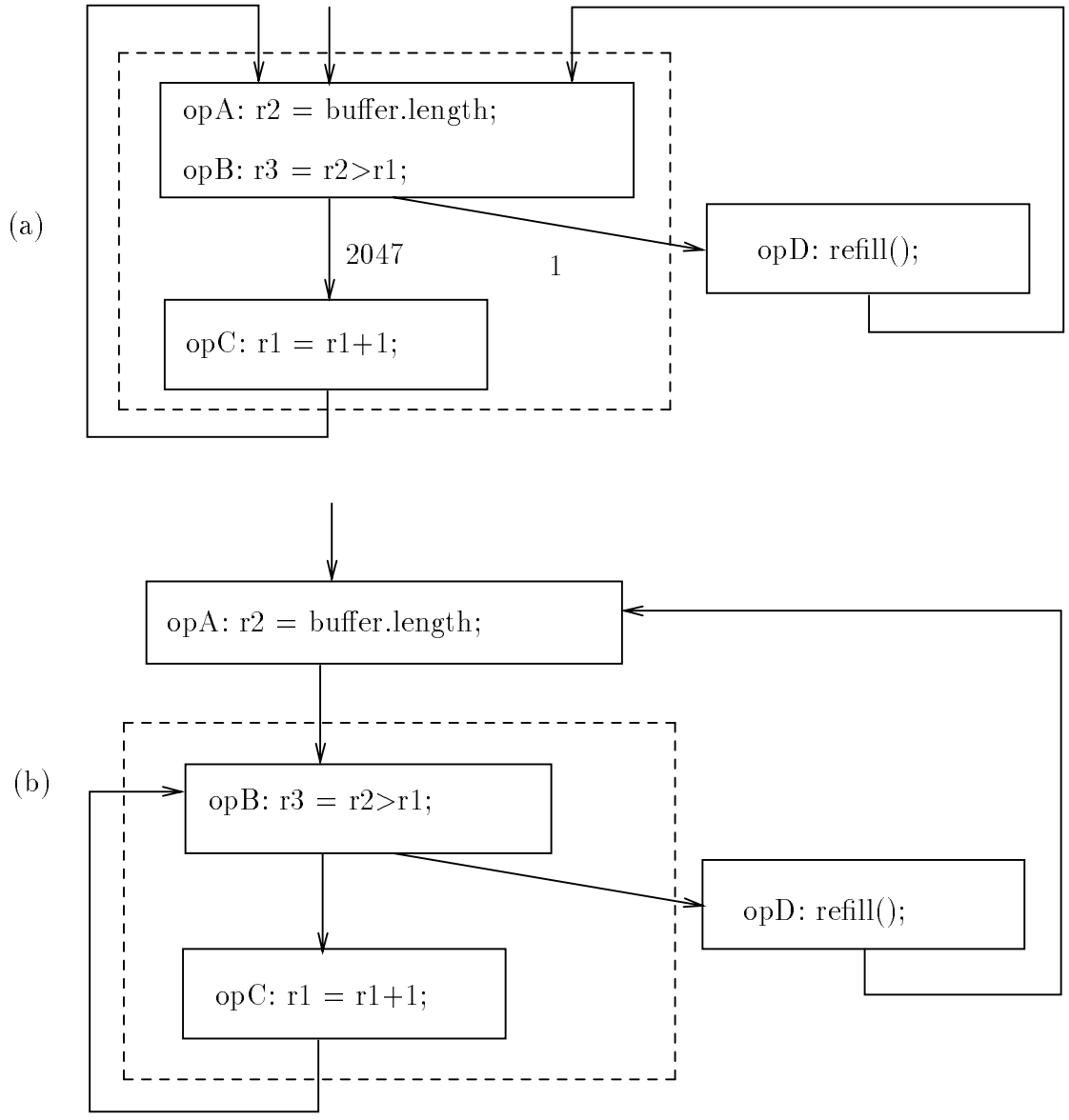

Figure 4.6: An example of super-block loop invariant code removal. (a) Original program segment. (b) Program segment after loop invariant code removal.

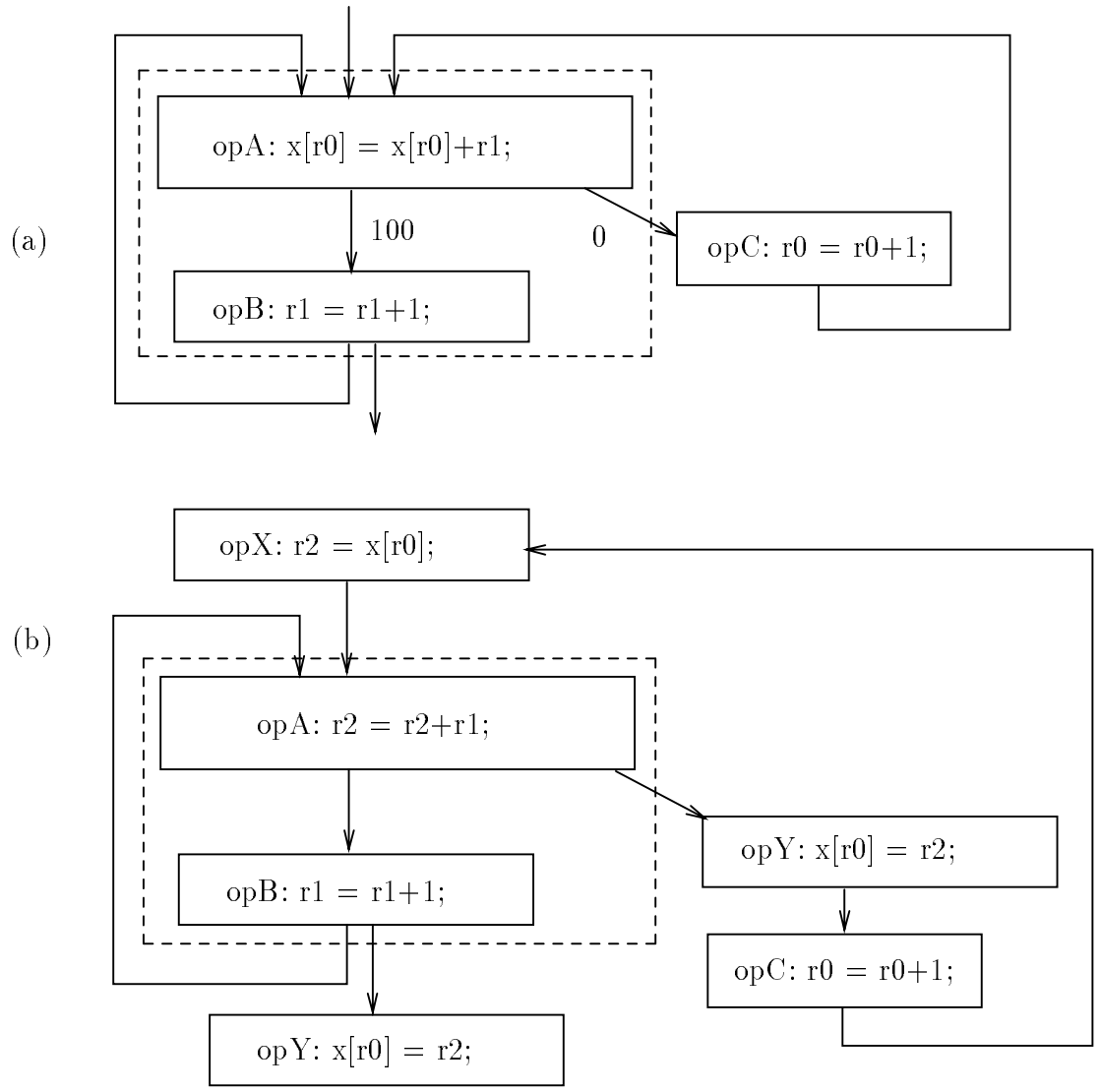

Figure 4.7: An example of super-block loop global variable migration. (a) Original program segment. (b) Program segment after loop global variable migration.

5. PERFORMANCE EVALUATION

The performance of the global optimizer has been evaluated on a DEC 3100 workstation using the code generator for the MIPS R2000 processor [10]. For all levels of optimization, the machine-dependent optimizations performed in conjunction with code generation are included. In addition, the following optimizations performed outside the global optimizer are included in all measurements: function inline expansion [16], intelligent code placement [14], and intelligent generation of switch statements [6]. Table 5.1 shows the benchmark programs that have been chosen. Three of the programs, *eqntott*, *espresso*, and *xlisp*, are from the SPEC benchmark set [18], and the others are commonly used application programs. The *size* column gives the length of each benchmark measured in numbers of lines of code. To profile every benchmark a set of input data has been selected for each program. Table 5.2 gives a brief description of the characteristics of the selected input data sets.

The performance of the global optimizer without any use of profile information will be compared against two existing compilers. The first compiler is the host C compiler of the MIPS R2000 processor, the MIPS C compiler (release 2.1) using its highest degree of optimization (-O4). The second compiler is the GNU C compiler (release 1.37.1) using

Table 5.1: Benchmarks.

name	*size*	*description*
cccp	4787	GNU C preprocessor
cmp	141	compare files
compress	1514	compress files
eqn	2569	typeset mathematical formulas for troff
eqntott	3461	Boolean minimization
espresso	6722	Boolean minimization
grep	464	string search
lex	3316	lexical analysis program generator
mpla	38970	pla generator
tbl	2817	format tables for troff
wc	120	word count
xlisp	7747	Lisp interpreter
yacc	2303	parsing program generator

Table 5.2: Input data for profiling.

name	*input*	*description*
cccp	20	C source files (100 - 5000 lines)
cmp	20	similar / different files
compress	20	C source files (100 - 5000 lines)
eqn	20	troff files (100 - 4000 lines)
eqntott	5	Boolean equations
espresso	20	Boolean functions (original espresso benchmarks)
grep	20	C source files (100 - 5000 lines) with various search strings
lex	5	lexers for C, Lisp, Pascal, awk, and pic
mpla	20	Boolean functions minimized by espresso
tbl	20	troff files (100 - 4000) lines
wc	20	C source files (100 - 5000) lines
xlisp	5	Gabriel benchmarks
yacc	10	grammars for C, Pascal, pic, eqn, awk, etc.

Table 5.3: Speedup comparison of classical optimizations.

name	*MIPS-O4*	*GNU-O*	*local*	*all*
cccp	1.0	0.99	1.04	1.08
cmp	1.0	0.99	1.00	1.05
compress	1.0	0.96	0.96	1.02
eqn	1.0	0.98	0.95	1.08
eqntott	1.0	0.78	0.64	1.04
espresso	1.0	0.89	0.91	1.02
grep	1.0	0.84	0.90	1.03
lex	1.0	0.98	0.97	1.01
mpla	1.0	0.92	0.89	1.06
tbl	1.0	0.95	0.96	1.02
wc	1.0	0.90	1.01	1.04
xlisp	1.0	0.86	1.03	1.13
yacc	1.0	0.91	0.88	1.01
avg.	1.0	0.92	0.93	1.05
s.d.	-	0.06	0.10	0.03

its highest degree of optimization (-O) [19]. Table 5.3 shows the speedup obtained for two levels of optimization, local optimization only and all levels of classical optimization (e.g., local, global, and loop). Note that in this comparison, local optimization includes global dead code removal for internal reasons. The MIPS compiler is used as the basis for all comparisons.

The performance of the profile-based code optimizations will be compared against the performance of all global optimizations using no profile information (*all* column of Table 5.3). Table 5.4 shows the speedup obtained by profile-based code optimization over classical optimization. As can be seen, significant speedups are possible as a result of profile-based code optimizations.

A side-effect of profile-based code optimization is the resulting code expansion. As shown in Table 5.5, this is not that large. It is also possible to limit code expansion to some maximum value during optimization.

Table 5.4: Speedup comparison of profile-based optimizations.

name	*classical*	*profile*
cccp	1.0	1.04
cmp	1.0	1.42
compress	1.0	1.11
eqn	1.0	1.25
eqntott	1.0	1.16
espresso	1.0	1.03
grep	1.0	1.21
lex	1.0	1.01
mpla	1.0	1.18
tbl	1.0	1.03
wc	1.0	1.32
xlisp	1.0	1.16
yacc	1.0	1.08
avg.	1.0	1.15
s.d.	-	0.12

Table 5.5: Code expansion due to profile-based optimizations.

name	*classical*	*profile*
cccp	1.0	1.03
cmp	1.0	1.11
compress	1.0	1.01
eqn	1.0	1.10
eqntott	1.0	1.00
espresso	1.0	1.07
grep	1.0	1.09
lex	1.0	1.08
mpla	1.0	1.13
tbl	1.0	1.06
wc	1.0	1.01
xlisp	1.0	1.20
yacc	1.0	1.09
avg.	1.0	1.07
s.d.	-	0.06

6. CONCLUSIONS

In this thesis the detailed design of a machine-independent global code optimizer is presented. Each optimization is formulated as a rule consisting of a set of predicate functions and a set of actions. Optimization opportunities are identified by all predicate functions within a rule returning true. The code transformation is then carried out by performing the set of actions. The performance of the global optimizer has been evaluated against a leading commercial compiler, the MIPS C compiler. Results show that the code quality produced by the IMPACT-I C compiler is comparable to that of the MIPS compiler

The use of profile information to further the effectiveness of classical optimizations has also been examined. Profile information effectively identifies the most important regions within a program. More aggressive optimizations can then be made on the critical regions of a program. Experimental results show that significant performance improvements can be obtained from profile-based classical code optimizations. Future work in the area of compile-time optimization using profile information is encouraging. Many parallel processing optimization techniques can benefit greatly from the use of profile information.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley Publishing Company, 1986.

[2] F. E. Allen and J. Cocke, "A catalogue of optimizing transformations," in R. Rustin, Ed., *Design and Optimization of Compilers*. Englewood Cliffs, NJ: Prentice-Hall, 1972.

[3] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Journal of the ACM*, vol. 19, no. 3, March 1976.

[4] M. Auslander and M. Hopkins, "An overview of the PL.8 compiler," *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, June 1982.

[5] P. P. Chang and W. W. Hwu, "Trace selection for compiling large C application programs to microcode," *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitectures*, November 1988, pp. 21-29.

[6] P. P. Chang and W. W. Hwu, "Control flow optimization for supercomputer scalar processing," *Proceedings of the 1989 International Conference on Supercomputing*, June 1989.

[7] P. P. Chang and W. W. Hwu, "The Lcode language and its enviroment," Internal Report, April 1991.

[8] P. Chang, "Compiler support for multiple instruction issue architectures," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[9] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," Center for Reliable and High-performance Computing Report, University of Illinois, Urbana, IL, April 1991.

[10] W. Y. Chen, "An optimizing compiler code generator: a platform for RISC performance analysis," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[11] F. C. Chow, "A portable machine-independent global optimizer – design and measurements," Ph.D. dissertation, Department of Electrical Engineering, Stanford University, December 1983.

[12] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*. Cambridge, MA: The MIT Press, 1986.

[13] J. A. Fisher, "Trace scheduling: a technique for global microcode compaction," *IEEE Transactions on Computers*, vol. C-30, no. 7, pp. 478-490 July 1981.

[14] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," *Proceedings of the 16th Annual International Symposium on Computer Architecture*, May, 1989.

[15] W. W. Hwu, T. M. Conte, and P. P. Chang, "Comparing software and hardware schemes for reducing the cost of branches," *Proceedings of the 16th Annual International Symposium on Computer Architecture*, May 1989.

[16] W. W. Hwu and P. P. Chang, "Inline function expansion for compiling realistic C programs," *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, June 1989.

[17] G. Kildall, "A unified approach to global program optimization," *Proceedings of the 1st ACM Symposium on the Principles of Programming Languages*, pp. 194-206, 1973.

[18] SPEC Benchmark Suite Release 1.0, September 1989.

[19] R. M. Stallman, *Internals of GNU CC*, 1988.

[20] A. S. Tanenbaum, H. V. Staveren, E. G. Keizer, and J. W. Stevenson, "A practical tool kit for making portable compilers," *Communications of the ACM*, vo1. 26, no. 9, pp. 654-660, September 1983.

[21] D. W. Wall, "Global register allocation at link time," *Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction*, June 1986.