April 1991

UILU-ENG-91-2219
CRHC-91-12

Center for Reliable and High-Performance Computing

USING PROFILE INFORMATION TO ASSIST CLASSIC CODE OPTIMIZATIONS

Pohua P. Chang
Scott A. Mahlke
Wen-mei W. Hwu

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY none	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE none	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-91-2219 CRHC-91-12	5. MONITORING ORGANIZATION REPORT NUMBER(S) none
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION NSF, NCR, NASA, AMD
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) NSF: 1800 G. Street, Washington, DC 20552 NCR: Personal Computer Div.-Clemson 1150 Anderson dr., Liberty, SC 29657
8a. NAME OF FUNDING/SPONSORING ORGANIZATION same as 7a. 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NSF: MIP-8809478 NASA: Nag 1-613
8c. ADDRESS (City, State, and ZIP Code) same as 7b.	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO. | WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Using Profile Information to Assist Classic Code Optimizations

12. PERSONAL AUTHOR(S)
Chang, Pohua P., Mahlke, Scott A., and Hwu, Wen-mei W.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM ________ TO ________	1991 April	46

16. SUPPLEMENTARY NOTATION
none

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
code optimization, profile-based code optimization, C, profiler, compiler

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

An execution profiler can be integrated into an optimizing compiler to provide the compiler with run-time information about user programs. For each code optimization, the profile information along with information from static loop analysis is used to reduce the execution time of the most frequently executed program regions. This paper describes how the profile information helps several classic code optimizations to identify more optimization opportunities. A profiler and several profile-based classic code optimizations have been implemented in our prototype C compiler. This paper describes our implementation and presents reasons as to why these optimizations are effective. Evaluation has been done with realistic C application programs. Experimental results clearly show the importance of profile-based classic code optimizations.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

7b. NASA Langley Research Center, Hampton, VA 23665
Advanced Micro Devices, 5900 East Ben White Blvd., Austin, TX 78741

Using Profile Information to Assist Classic Code Optimizations

Pohua P. Chang, Scott A. Mahlke, and Wen-mei W. Hwu

Center for Reliable and High-performance Computing
University of Illinois, Urbana-Champaign
hwu@crhc.uiuc.edu

Abstract

An execution profiler can be integrated into an optimizing compiler to provide the compiler with run-time information about user programs. For each code optimization, the profile information along with information from static loop analysis is used to reduce the execution time of the most frequently executed program regions. This paper describes how the profile information helps several classic code optimizations to identify more optimization opportunities. A profiler and several profile-based classic code optimizations have been implemented in our prototype C compiler. This paper describes our implementation and presents reasons as to why these optimizations are effective. Evaluation has been done with realistic C application programs. Experimental results clearly show the importance of profile-based classic code optimizations.

Key Words: code optimization, profile-based code optimization, C, profiler, compiler

This research has been supported by the National Science Foundation (NSF) under Grant MIP-8809478, Dr. Lee Hoevel at NCR, the AMD 29K Advanced Processor Development Division, the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS).

1 Introduction

A program consists of a finite set of instructions that may be partitioned into n disjoint sets, denoted by S_i, $i = 1..n$. Let $count(S_i)$ denote the execution count of S_i, and $time(S_i)$ denote the time that is needed to execute S_i.[1] The execution time of the program is $\sum_{i=1}^{n}(count(S_i) * time(S_i))$. The major purpose of code optimizations is to reduce the execution time. Classic code optimizations can be classified into two major categories. The first category reduces $time(S_i)$ without affecting $time(S_j)$ for all $j \neq i$. For example, dead code elimination, common subexpression elimination, and copy propagation belong to this category [Allen 71] [Aho 86]. The second category reduces $time(S_i)$ of a frequently executed S_i with the side effect of possibly increasing the execution time of other sets of instructions. Code optimizations that are in this category are effective only if the execution count of the optimized set of instructions is much larger than the sum of the execution counts of the sets whose execution times are increased. For example, loop invariant code removal decreases the execution time of a loop body at the cost of increasing that of the loop's preheader. The compiler expects that the body executes many more times than the loop preheader.

Static loop analysis can identify loop structures. However, static analyses that estimate $count(S_i)$ have several shortcomings.

1. The outcome of a conditional (if) statement is not always predictable at compile time.

2. The iteration count of a loop is not always predictable at compile time.

[1] Depending on the input, $count(S_i)$ and $time(S_i)$ may vary from run to run.

3. The invocation count of a (recursive) function is not always predictable at compile time.

Because conditional statements and loops can be nested, and function calls can be recursive in C, the results of static analysis may be inaccurate because the prediction errors are amplified in a nested structure. For example, a loop with a large iteration count, nested within a conditional statement will not contribute to the execution time if the condition for its evaluation is never true. Optimizing such a loop may degrade the overall program performance if the optimizations increase the execution time of other sets of instructions.

Classic code optimizations use other static analysis methods, such as live-variable analysis, reaching definitions, and definition-use chain, to ensure the correctness of code transformations [Aho 86].[2] There are often instances where a value will be destroyed on an infrequently executed path, which exists to handle rare events, such as error handling.

Profiling is better at detecting the most frequently executed and the most time consuming program regions than static loop analysis, because profiling gives exact $count(S_i)$. Profiling is the process of selecting a set of inputs for a benchmark program, executing the program with these inputs, and recording the dynamic behavior of the program. Profiling has been a widely used method for hand-tuning algorithms and programs. Many standard profiling tools are widely available on UNIX systems.

The motivation to integrate a profiler into a C compiler is to guide the code optimizations using profile information. We refer to this scheme as *profile-based code optimization.* In this

[2]In this paper, we assume that the reader is familiar with the static analysis methods.

paper, we present our method for using profile information to assist classic code optimizations. The idea is to transform the control flow graph according to the profile information so that the optimizations are not hindered by rare conditions. Because profile-based code optimizations demand very little work from the user (i.e. selecting a set of input data), they can be applied to very large application programs. One can argue that automatic profile-based code optimizations may be less effective than algorithm and program tuning by hand. Even if that is true, much of the tedious work that is involved in writing more efficient code can be eliminated from the hand-tuning process by profile-based code optimizations. The programmers can concentrate on more intellectual work, such as algorithm tuning.

1.1 Contribution of this paper

The intended audience of this paper is optimizing compiler designers, and production software developers. Compiler designers can reproduce the techniques that are described in this paper. Production software developers can evaluate the cost-effectiveness of profile-based code optimizations for improving product performance.

The contribution of this paper is a description of our experience with the generation and use of profile information in an optimizing C compiler. The prototype profiler that we have constructed is robust and tested with large C programs. We have modified many classic code optimizations to use profile information. The experimental data show that these code optimizations can substantially speedup realistic non-numeric C application programs. We also provide insight into why these code optimizations are effective.[3]

1.2 Organization of this paper

The rest of this paper is divided into 5 sections. Section 2 reviews some profile-based code optimizations reported in previous studies. Section 3 briefly discusses how a profiler is integrated into our C compiler and where the profile-based classic code optimizations are implemented in our prototype compiler. Section 4 describes the changes that need to be made to classic code optimizations in order to use profile information, and provides insights as to why these optimizations are effective. Section 5 presents some experimental data that demonstrate the importance of profile-based classic code optimizations. Section 6 contains our concluding remarks.

2 Related Studies

Using profile information to hand-tune algorithms and programs has become a common practice for serious program developers. Several UNIX profilers are available, such as prof/gprof[Graham 82, Graham 83] and tcov[AT&T 79][4]. The prof output shows the execution time and the invocation count of each function. The gprof output not only shows the execution time and the invocation count of each function, but also shows the effect of called functions in the profile of each caller. The tcov output is an annotated listing of the

[3]It should be noted that profile-based code optimizations are not alternatives to conventional optimizations, but are meant to be applied in addition to conventional optimizations.

[4]Tcov is available only on Sun-3 and Sun-4 systems

source program. The execution count of each straight-line segment of C statements is reported. When there are multiple profile files, these profiling tools show the sum of the profile files. These profiling tools allow programmers to identify the most important functions and the most frequently executed regions in the functions.

Recent studies of profile-based code optimizations have provided solutions to specific architectural problems. The accuracy of branch prediction is important to the performance of pipelined processors that use the squashing branch scheme. It has been shown that profile-based branch prediction (at compile time) performs as well as the best hardware schemes[McFarling 86, Hwu 89.2]. Trace scheduling is a popular global microcode compaction technique[Fisher 81]. For trace scheduling to be effective, the compiler must be able to identify frequently executed sequences of basic blocks. The trace scheduling algorithm operates on one sequence of basic blocks at a time. It has been shown that profiling is an effective method to identify frequently executed sequences of basic blocks in a flow graph[Ellis 86, Chang 88]. Instruction placement is a code optimization that arranges the basic blocks of a flow graph in a particular linear order to maximize the sequential locality and to reduce the number of executed branch instructions. It has been shown that profiling is an effective method to guide instruction placement[Hwu 89.1, Pettis 90]. A C compiler can implement a multiway branch (a *switch* statement) as a sequence of branch instructions or as a hash table lookup jump[Chang 89]. If most occurrences are satisfied by few case conditions, then it is better to implement a sequence of branch instructions, starting from the most likely case to the least likely case. Otherwise, it is better to implement a hash table lookup jump. Profile information can help a register allocator to identify the most frequently accessed variables[Wall 86, Wall 88]. Function inline expansion eliminates the overhead of function calls and enlarges the scope of global code optimizations. Using profile information, the compiler can identify the most frequently invoked calls and determine the best expansion sequence[Hwu 89.3].

An optimized counter-based execution profiler that measures the average execution times and their variance (with a runtime overhead less than 5% in practice) has been described in [Sarkar 89.1]. Extensions from this work to program partitioning and scheduling for multiprocessors have been described in [Sarkar 89.2].

3 Design Overview

Figure 1 shows the major components of our prototype C compiler. *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 analysis 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 compiler front-end also performs local code optimizations to eliminate redundant computations within basic blocks. Our prototype compiler generates code for several existing processor architectures: MIPS R2000, SPARC, i860, and AMD29k. Each code generator performs the following tasks: (1) machine-dependent code optimizations, (e.g., constant preloading, instruction selection),

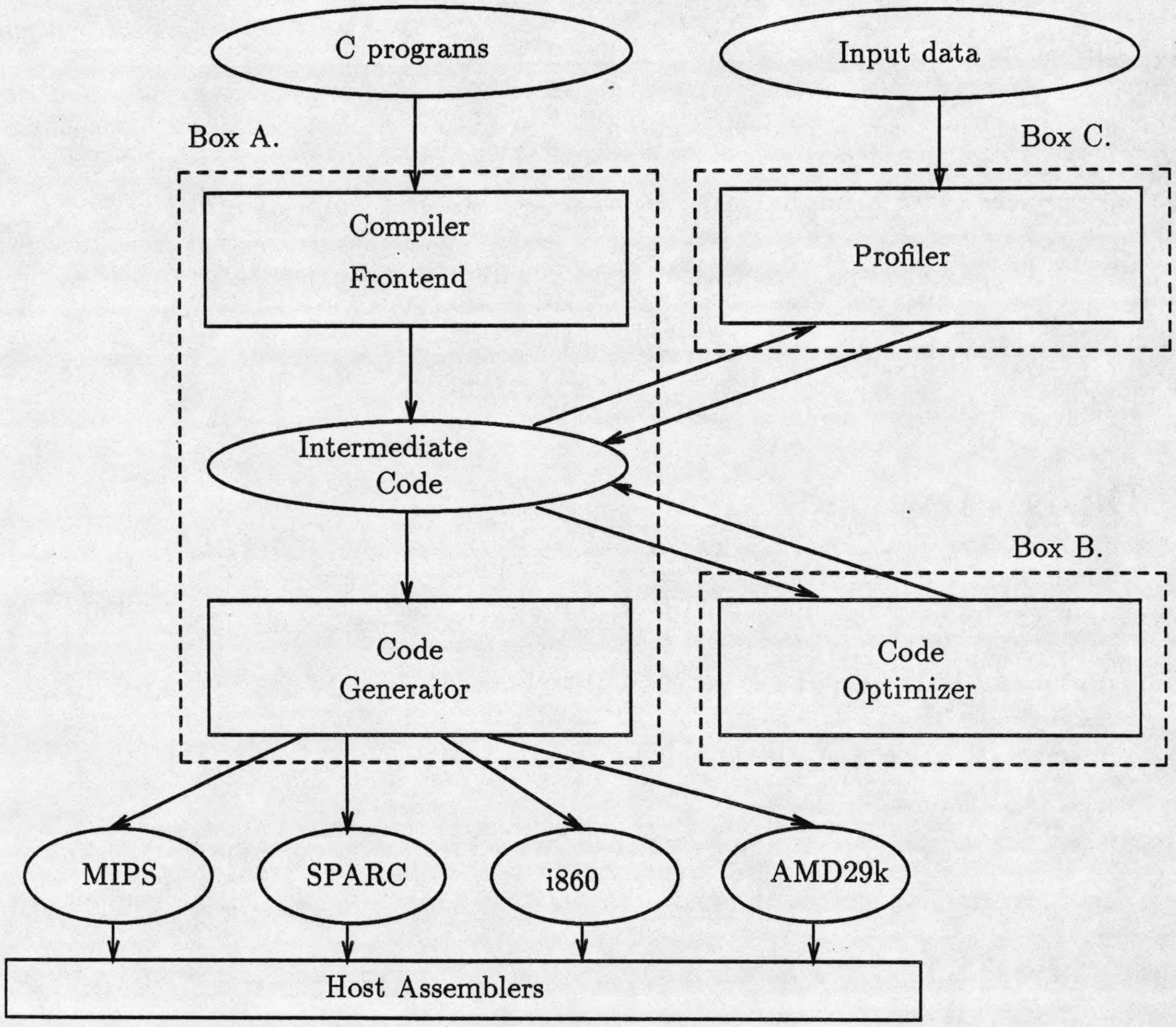

Figure 1: A block diagram of our prototype C compiler.

(2) register allocation, and (3) assembly/machine code generation. To improve the output code quality, we have added a global code optimizer as shown in *Box B* of Figure 1. In *Box B*, we have installed many classic global code optimizations. Table 1 lists the local and global code optimizations that we have implemented in our prototype compiler. In sum, *Box A* corresponds to a basic working C compiler, and *Box A* and *Box B* together form an optimizing C compiler.

local	*global*
constant propagation	constant propagation
copy propagation	copy propagation
common subexpression elimination	common subexpression elimination
redundant load elimination	redundant load elimination
redundant store elimination	redundant store elimination
constant folding	loop unrolling
strength reduction	loop invariant code removal
constant combining	loop induction strength reduction
operation folding	loop induction elimination
dead code removal	dead code removal
code reordering	global variable migration

Table 1: Classic code optimizations.

In order to have profile-based code optimizations, a new *Box C* has been added to our prototype compiler. The input to *Box C* 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 to produce a profile file. After we have obtained all profile files, they are summarized. The summarized profile information is then integrated into the intermediate code. The global code optimizations in *Box B* are modified to use the profile information. In a later subsection, we provide more detailed descriptions of the profiling procedure.

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*).

3.1 Program representation

In optimizing compilers, a function is typically represented by a flow graph[Aho 86], where each node is a basic block and each arc is a potential control flow path between two basic blocks. Because classic code optimizations have been developed based on the flow graph data structure[5], we extend the flow graph data structure to contain profile information. We define a *weighted flow graph* as a quadruplet $\{V, E, count, arc\_count\}$, where each node in V is a basic block, each arc in E is a potential control flow path between two basic blocks,

[5]Algorithms for finding dominators, detecting loops, computing live-variable information, and other dataflow analysis have been developed on the flow graph data structure[Aho 86].

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 multi-way branch instruction (*switch* statement in C), or (4) an arithmetic instruction. For simplicity, we assume that a jump-subroutine instruction is an arithmetic instruction because it does not change the control flow within the function where the jump-subroutine instruction is defined.[6] Except 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 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.

[6]An exception is when a longjmp() is invoked by the callee of a jump-subroutine instruction and the control does not return to the jump-subroutine instruction. Another exception is when the callee of a jump-subroutine instruction is exit(). Because the above cases are rare events in most C application programs, their effects on code optimization decisions can be considered as noise and be neglected.

3.2 Profiler implementation

We are interested in collecting the following information with the profiler in *Box C* of Figure 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 multi-way 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 shows the major components of the profiler that appears in *Box C* of Figure 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.

(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. 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 profile data is mapped into the original intermediate code. Because we have not changed the structure of the program, it is straight-forward 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.

We have found the need for an automatic consistency check program because a program may have tens of thousands of basic blocks. It is not possible to check manually that the profile data has been correctly projected onto the intermediate form. The projection may fail if that the original intermediate code is changed after profiling. A simple check is that the sum of the invocation count of all control paths into a basic block equals the execution count of that basic block. Likewise, the sum of the invocation count of all control paths out from a basic block should equal the execution count of that basic block. Some basic blocks may fail this check due to longjmp() and exit() calls, and cause warning messages to be displayed on the screen. Because the errors due to longjmp() and exit() calls are small, the user can ignore those warning messages.

4 Code Optimization Algorithms

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.* We will illustrate this concept us-

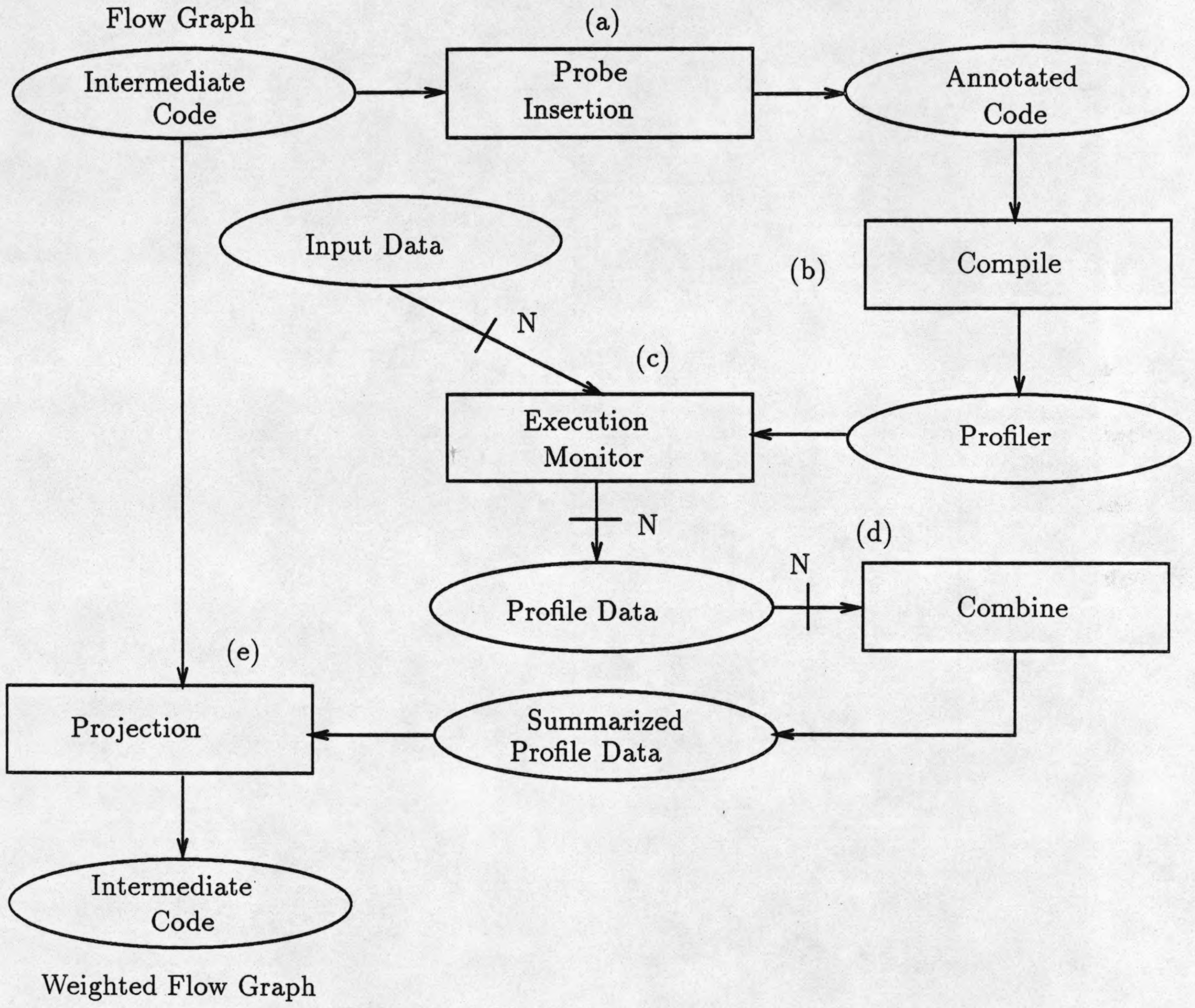

Figure 2: A block diagram of the profiler.

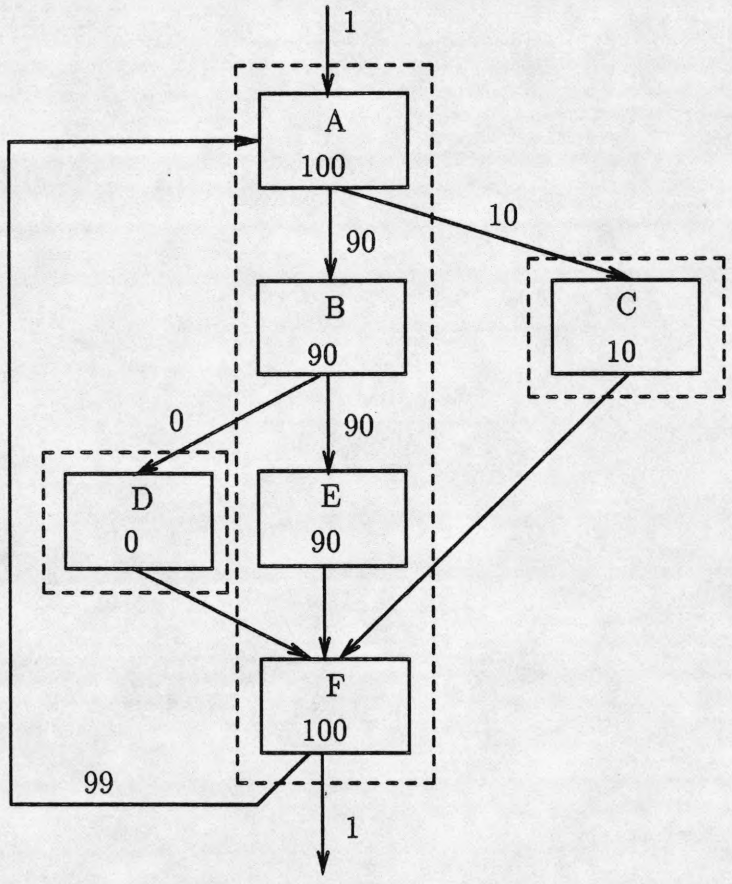

Figure 3: A weighted flow graph.

ing an example. Figure 3 shows a weighted flow graph which represents a loop program. The *count* of basic blocks $\{A, B, C, D, E, F\}$ are $\{100, 90, 10, 0, 90, 100\}$, respectively. The *arc_count* of $\{A \rightarrow B, A \rightarrow C, B \rightarrow D, B \rightarrow E, C \rightarrow F, D \rightarrow F, E \rightarrow F, F \rightarrow A\}$ are $\{90, 10, 0, 90, 10, 0, 90, 99\}$, 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 non-loop based classic code optimizations are conservative and do not perform transformations that may increase the execution time of any basic block. The formulation of loop based classic code optimizations consider the entire loop body as a whole and do not consider the case where some basic blocks in the loop body are rarely executed because of a very biased if statement. In the rest of this section, we describe several profile-based code optimizations that make more aggressive decisions and explore more optimization opportunities.

We propose the use of a simple data structure, called a super-block, to represent a frequently executed path. A super-block has the following features. (1) It is a linear sequence of basic blocks $B(i), i = 1..n$, 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.

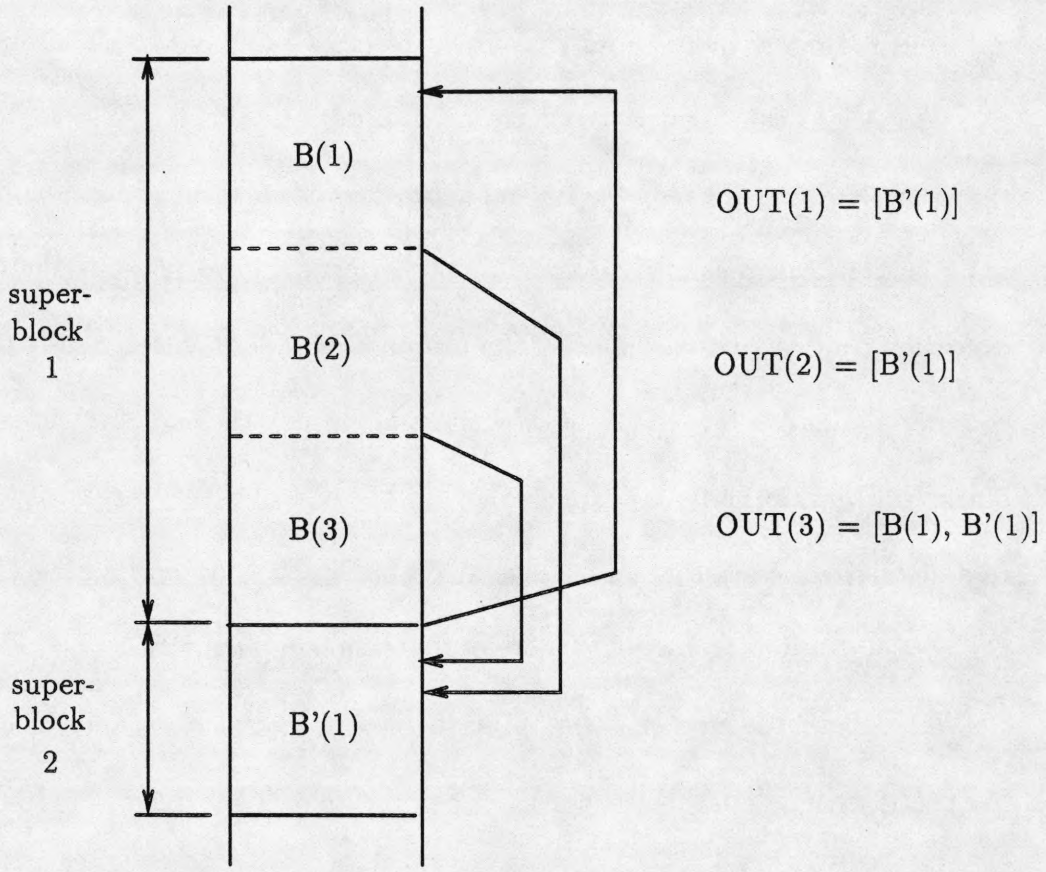

Figure 4: Two super-blocks.

Figure 4 shows two super-blocks. The top super-block contains three basic blocks $B(1)$, $B(2)$, and $B(3)$. Because the control may leave the super-block from $B(1)$ and $B(2)$ to $B'(1)$, $OUT(1)$ and $OUT(2)$ are both $[B'(1)]$. In the case of a multi-way branch, the OUT set may contain many basic blocks. The last basic block of a super-block is a special case where the fall through path is also in the OUT set. For example, $OUT(3)$ contains both $B(1)$ and $B'(1)$.

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. 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. Trace selection was first used in trace scheduling[Fisher 81, Ellis 86]. An experimental study of several trace selection algorithms was reported in [Chang 88]. For completeness, the outline of a trace selection algorithm is shown in Figure 5. The $best\_predecessor\_of(node)$ ($best\_successor\_of(node)$) function returns the most probable source (destination) basic block of *node*, if the source (destination) basic block has not yet been marked. The growth of a trace is stopped when the most probable source (destination) basic block of the first (last) node has been marked.

Figure 3 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

```
algorithm trace_selection(a weighted flow graph G) begin
    mark all nodes in G unvisited;
    while (there are unvisited nodes) begin
        seed = the node with the largest execution count
               among all unvisited nodes;
        mark seed visited;
        /* grow the trace forward */
        current = seed;
        loop
            s = best_successor_of(current);
            if (s=0) exit loop;
            add s to the trace;
            mark s visited;
            current = s;
        end_loop
        /* grow the trace backward */
        current = seed;
        loop
            s = best_predecessor_of(current);
            if (s=0) exit loop;
            add s to the trace;
            mark s visited;
            current = s;
        end_loop
    end_while
end_algorithm
```

Figure 5: A trace-selection algorithm.

```
algorithm tail_duplication(a trace B(1..n)) begin
    Let B(i) be the first basic block that
      is an entry point to the trace, except for i=1;
    for (k=i..n) begin
        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;
    end_for
end_algorithm
```

Figure 6: The tail-duplication algorithm.

converted into a super-block by duplicating the tail part of the trace, in order to ensure that the program control can only enter at the top basic block. The tail duplication algorithm is shown in Figure 6. Using the example in Figure 3, 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 7.[7] More code transformations can be applied after tail duplication to eliminate jump instructions. For example, the F' super-block in Figure 7 could be duplicated and each copy be combined with the C and D super-blocks to form two larger super-blocks.

In order to control the amount of code duplication, we add a basic block to a trace only if the execution count of that basic block is more than some threshold value (e.g., 100). After

[7]Note that the profile information has to be scaled accordingly. Scaling the profile information will destroy the accuracy. Fortunately, code optimizations after forming super-blocks only need approximate profile information. In order to take measurements from the weighted flow graph data structure, the transformed program needs to be profiled.

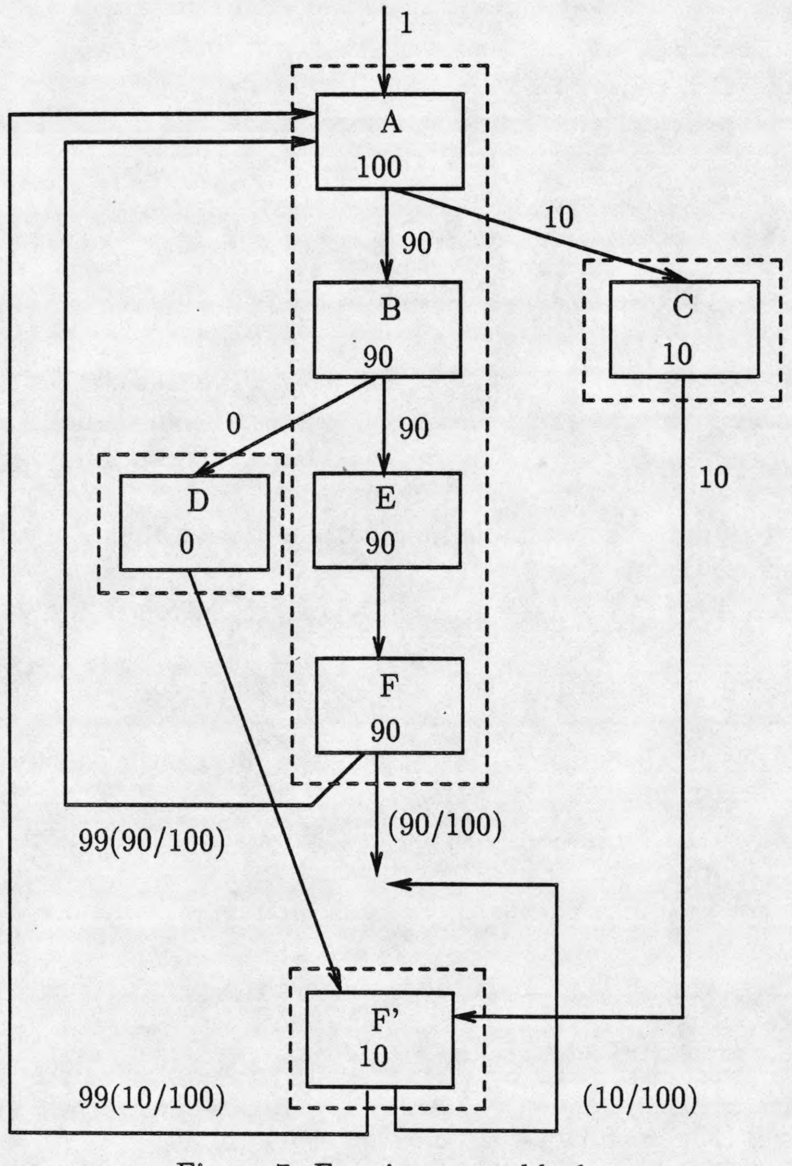

Figure 7: Forming super-blocks.

forming super-blocks, we only optimize super-blocks whose execution counts are higher than the threshold value.

4.3 Formulation of code optimizations

Table 2 shows a list of classic code optimizations that we have extended to use profile information. The formulation of these classic code optimizations can be found in [Allen 71] [Aho 86]. In the rest of this section, we will describe the modifications that need to be applied to each of these classic code optimizations. In Table 2, the *scope* column describes the scopes of these code optimizations. The non-loop 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 back-edge from its last node to its first node.

name	*scope*
constant propagation	super-block
copy propagation	super-block
constant combining	super-block
common subexpression elimination	super-block
redundant store elimination	super-block
redundant load elimination	super-block
dead code removal	super-block
loop invariant code removal	super-block loop
loop induction variable elimination	super-block loop
global variable migration	super-block loop

Table 2: Super-block code optimizations.

A code optimization is a triplet, $\{search, precondition, action\}$, where *search* is a function that identifies sets of instructions within the scope of the code optimization that may be eligible for this code optimization, *precondition* is a predicate function that returns true if and only if a set of instructions that has been identified by the *search* function satisfies the constraints (preconditions) of the code optimization and produces a speedup of the overall program performance, and *action* is a transformation function that maps a set of instructions that has passed through both the *search* and *precondition* functions to another set of instructions that is more efficient.

Let $\{op(j) \mid j = 1..m, m \geq 1\}$ denote an ordered set of instructions in a super-block, such that $op(x)$ precedes $op(y)$ if $x < y$. Except for the global variable migration, the profile-based classic code optimizations that are listed in Table 2 optimize one instruction or a pair of instructions at a time. For optimizing one instruction at a time, the *search* function is a simple $for(i = 1..m)$ loop that sequences through each instruction in the super-block. When optimizing a pair of instructions at a time, the *search* function is a simple nested loop of the form $for(i = 1..m-1) for(j = i+1..m)$. In the following paragraphs, we focus on the *precondition* and the *action* functions of the profile-based classic code optimizations.

For an instruction $op(i)$, we denote the set of variables (in register or in memory) that $op(i)$ modifies by $dest(i)$.[8] We denote the set of variables that $op(i)$ requires as source operands by $src(i)$. We denote the operation code of $op(i)$ by f_i. Therefore, $op(i)$ refers to the operation $dest(i) \leftarrow f_i(src(i))$.

[8]In this paper, we assume that there can be at most one element in $dest(i)$ of any instruction $op(i)$.

4.4 Operation combining

Constant propagation, copy propagation, and constant combining are special cases of operation combining. Operation combining operates on a pair of instructions at a time. Let $op(x)$ and $op(y)$ be a pair of instructions in a super-block, where $op(x)$ precedes $op(y)$. The *precondition* function of operation combining consists of the following boolean predicates that must all be satisfied.

1. $dest(x)$ is a subset of $src(y)$.
2. The variables in $dest(x)$ are not modified by $\{op(k), k = x + 1..y - 1\}$.
3. The variables in $src(x)$ are not modified by $\{op(j), j = x..y - 1\}$.
4. There is a simple transformation of $op(y)$ to $dest(y) \leftarrow f'_y(src(x) \cup (src(y) - dest(x)))$.

The *action* function of operation combining replaces $op(y)$ by $dest(y) \leftarrow f'_y(src(x) \cup (src(y) - dest(x)))$.

The purpose of operation combining is to eliminate the flow dependence between $op(x)$ and $op(y)$. When $op(y)$ no longer uses the value that is produced by $op(x)$, it is possible that $op(x)$ will become dead code. Duplicating the tail part of a trace to form a super-block eliminates control paths that enter the trace in the middle. For many $op(x)$ and $op(y)$ instruction pairs, $op(x)$ becomes the sole producer of one or more source operands of $op(y)$. Therefore, profile-based operation combining can find more opportunities for optimization than traditional operation combining.

Constant propagation is a special case of operation combining, where $op(x)$ is of the form $dest(x) \leftarrow K$, where K is a constant. Copy propagation is another special case of operation combining, where $op(x)$ is a simple register move instruction. Constant combining is another special case, where both $op(x)$ and $op(y)$ have constant source operands and $op(y)$ can be formulated by combining the constant source operands. For example, when $op(x)$ is $r1 \leftarrow r0 + 10$ and $op(y)$ is $r2 \leftarrow memory(r1 + 4)$, $op(y)$ can be transformed to $r2 \leftarrow memory(r0 + 14)$. In its general form, constant combining can also optimize multiply, shift, and divide instructions. As another example, a compare instruction and a branch instruction may be combined into a compare-and-branch instruction for some processor architectures.

4.5 Common subexpression elimination

Common subexpression elimination operates on a pair of instructions at a time. Let $op(x)$ and $op(y)$ be a pair of instructions in a super-block, where $op(x)$ precedes $op(y)$. The *precondition* function of common subexpression elimination consists of the following boolean predicates that must all be satisfied.

1. f_x is the same as f_y.

2. $src(x)$ is the same as $src(y)$.

3. The variables in $src(x)$ are not modified by $\{op(j), j = x..y-1\}$.

4. The variables in $dest(x)$ are not modified by $\{op(k), k = x+1..y-1\}$.

The *action* function of common subexpression elimination transforms $op(y)$ into $dest(y) \leftarrow dest(x)$.[9]

The purpose of common subexpression elimination is to remove redundant instructions. More common subexpression elimination opportunities are available when control paths that enter in the middle of traces are eliminated. For many $op(x)$ and $op(y)$ instruction pairs, $op(x)$ becomes the only producer of $dest(x)$ when the program control is at $op(y)$.

When $src(x)$ contains memory variables (e.g., $op(x)$ is a memory load instruction), eliminating control paths that enter in the middle of traces also helps the compiler to prove that $src(x)$ is not modified between $op(x)$ and $op(y)$. Redundant load elimination and redundant store elimination are special cases of common expression elimination, in which the operands are memory variables.

Figure 8 shows a simple example of super-block common subexpression elimination. The original program is shown in Figure 8(a). After trace selection and tail duplication, the program is shown in Figure 8(b). Because of tail duplication, opC cannot be reached from opB; therefore, common subexpression elimination can be applied to opA and opC.

4.6 Dead code removal

Dead code removal operates on one instruction at a time. Let $op(x)$ be an instruction in a super-block. The traditional formulation of the *precondition* function of dead code removal

[9]Because some cases of common subexpression elimination and operation combining undo the work of each other, there must be some tie-breaking rules between these special cases in the actual implementation.

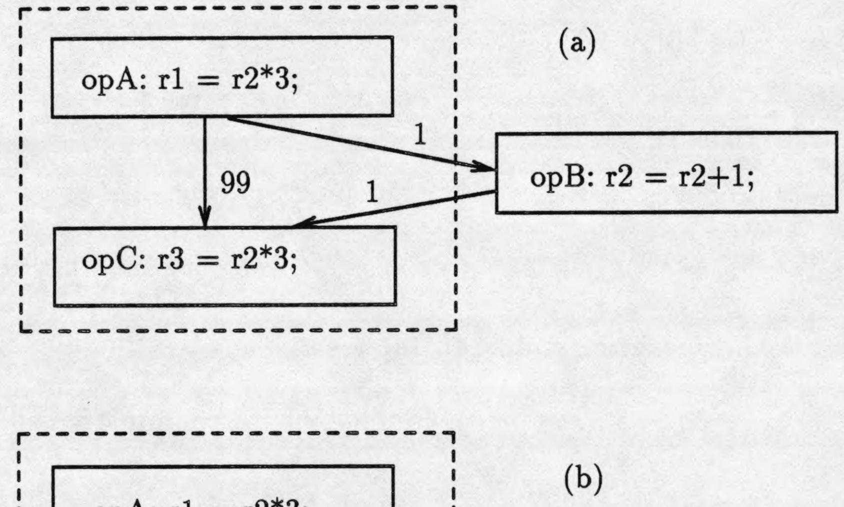

Figure 8: An example of common subexpression elimination.

is simply that the values of $dest(x)$ will not be used later in execution. Then, $op(x)$ can be eliminated.

In addition to the traditional formulation, we propose an extension to dead code removal. In the extension, the *precondition* function consists of the following sequential steps.

1. If the super-block where $op(x)$ is defined is a super-block loop, return false.

2. If $op(x)$ is a branch instruction, return false.

3. If $dest(x)$ is used before redefined in the super-block, return false.

4. Find an integer y, such that $op(y)$ is the first instruction that modifies $dest(x)$ and $x < y$. If $dest(x)$ is not redefined in the super-block, set y to $m + 1$, where $op(m)$ is the last instruction in the super-block.

5. Find an integer z, such that $op(z)$ is the last branch instruction in $\{op(k), k = x + 1..y - 1\}$. If there is no branch instruction in $\{op(k), k = x + 1..y - 1\}$, return true. If $src(x)$ is modified by an instruction in $\{op(j), j = x + 1..z\}$, return false.

6. Return true.

If the *precondition* function returns true, we proceed to the *action* part of dead code removal. The *action* function consists of the following steps.

1. For every branch instruction in $\{op(i), i = x + 1..y - 1\}$, if $dest(x)$ is live[10] when $op(i)$

[10] A variable is live if its value will be used before redefined. An algorithm for computing live variables can be found in [Aho 86].

is taken, copy *op*(x) to a place between *op*(i) and every possible target super-block of *op*(i) when *op*(i) is taken.

2. If y is $m+1$ and the super-block where *op*(x) is defined has a fall-thru path (because the last instruction in the super-block is not a branch or is a conditional branch), copy *op*(x) to become the last instruction of the super-block.

3. Eliminate the original *op*(x) from the super-block.

Like operation combining and common subexpression elimination, tail duplication is a major reason why more dead code elimination opportunities exist. Another reason is that we allow an instruction to be eliminated from a super-block by copying it to all control flow paths that exit from the middle of the super-block. This code motion is beneficial because the program control rarely exits from the middle of a super-block.

Figure 9 shows a simple example of dead code removal. 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 9(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 9(b).

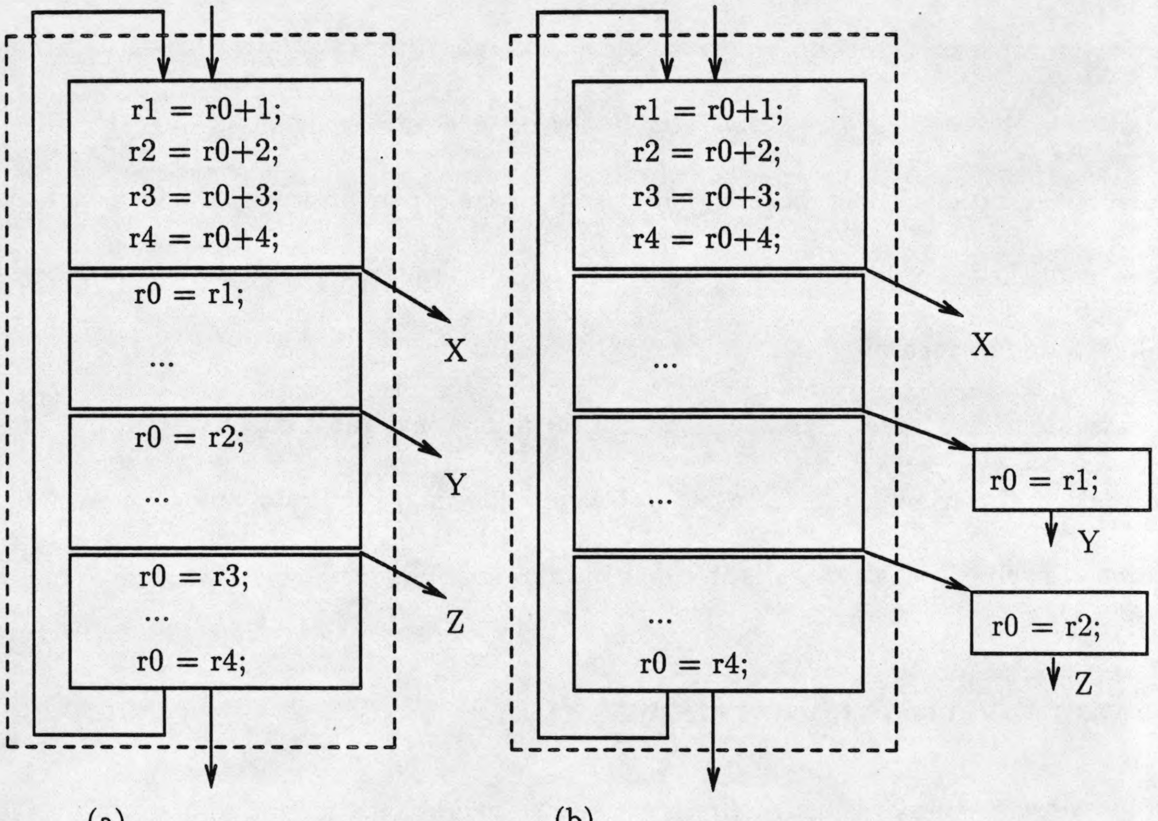

Figure 9: An example of dead code removal.

4.7 Loop optimizations

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 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. Traditional loop optimizations are modified to process super-block loops correctly.

We have identified three important loop optimizations that most effectively utilize profile information: invariant code removal, global variable migration and induction variable elimination. Each optimization is discussed in a following subsection.

4.8 Loop invariant code removal

Invariant code removal moves instructions, whose source operands do not change within the loop, to a preheader block. Instructions of this type are then executed only once each time the loop is invoked, rather than on every iteration.

The *precondition* function for invariant code removal consists of the following boolean predicates that must all be satisfied.

1. $src(x)$ is not modified in the super-block.

2. *op*(x) is the only instruction which modifies *dest*(x) in the super-block.

3. *op*(x) must precede all instructions which use *dest*(x) in the super-block.

4. *op*(x) must precede every exit point of the super-block in which *dest*(x) is live.

5. If *op*(x) is preceded by a conditional branch, it must not possibly cause an exception.

The *action* function of invariant code removal is moving *op*(x) to the end of the preheader block of the super-block loop.

In the *precondition* function, predicate 5 returns true if *op*(x) is executed on every iteration of the super-block loop. An instruction that is not executed on every iteration may not be moved to the preheader if it can possibly cause an exception. Memory instructions, floating point instructions, and integer divide are the most common instructions which cannot be removed unless they are executed on every iteration.

Predicate 6 is dependent on two optimization components: memory disambiguation and interprocedural analysis. Currently our prototype C compiler performs memory disambiguation, but no interprocedural analysis. Thus, if *op*(x) is a memory instruction, predicate 6 will return false if there are any subroutine calls in 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 invariant code removal is best illustrated by an example. Figure 10 shows a simple example of super-block loop invariant code removal. In Figure 10(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

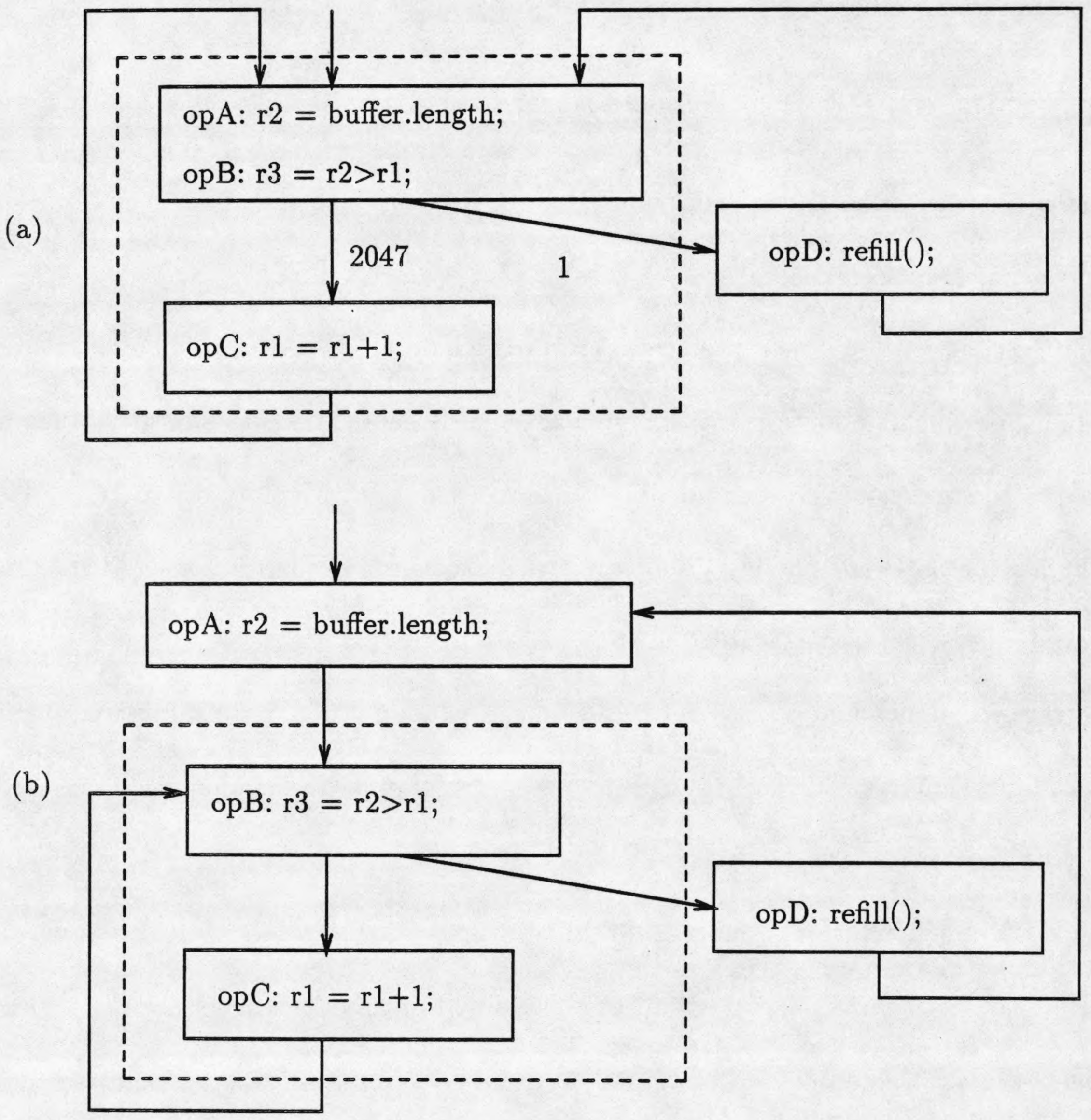

Figure 10: An example of loop invariant code removal.

(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 10(b).

4.9 Global variable migration

Global variable migration moves frequently accessed memory variables (globally declared scalar variables, array elements, or structure elements) into registers for the duration of the loop. Loads and stores to these variables within the loop are replaced by register accesses. A load instruction is inserted in the preheader of the loop to initialize the register, and a store is placed at each loop exit to update memory after the execution of the loop.

The *precondition* function for global variable migration consists of the following boolean predicates that must all be satisfied. For this explanation if $op(x)$ is a memory access, let $address(x)$ denote the memory address of the access.

1. $op(x)$ is a load or store instruction.
2. $address(x)$ is invariant in the super-block loop.
3. If $op(x)$ is preceded by a conditional branch, it must not possibly cause an exception.
4. The compiler must be able to detect, in the super-block loop, all memory accesses whose addresses can equal $address(x)$ at run-time, and these addresses must be invariant in the super-block loop.

The *action* function of global variable migration consists of three steps.

1. A new load instruction $op(a)$, with $src(a) = address(x)$ and $dest(a) = temp\_reg$, is inserted after the last instruction of the preheader of the super-block loop.

2. A store instruction $op(b)$, with $dest(b) = address(x)$ and $src(b) = temp\_reg$, is inserted as the first instruction of each block that can be immediately reached when the super-block loop is exited.[11]

3. All loads in the super-block loop with $src(i) = address(x)$ are converted to register move instructions with $src(i) = temp\_reg$, and all stores with $dest(i) = address(x)$ are converted to register move instructions with $dest(i) = temp\_reg$. The unnecessary copies are removed by later applications of copy propagation and dead code removal.

Figure 11 shows a simple example of super-block 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 11(b). Extra instructions (opX and opY) are added to the super-block loop boundary points to ensure correctness of execution.

4.10 Loop induction variable elimination

Induction variables are variables in a loop incremented by a constant amount each time the loop iterates. Induction variable elimination replaces the uses of an induction variable by

[11]If a basic block that is immediately reached from a control flow exit of the super-block loop can be reached from multiple basic blocks, a new basic block needs to be created to bridge the super-block loop and the originally reached basic block.

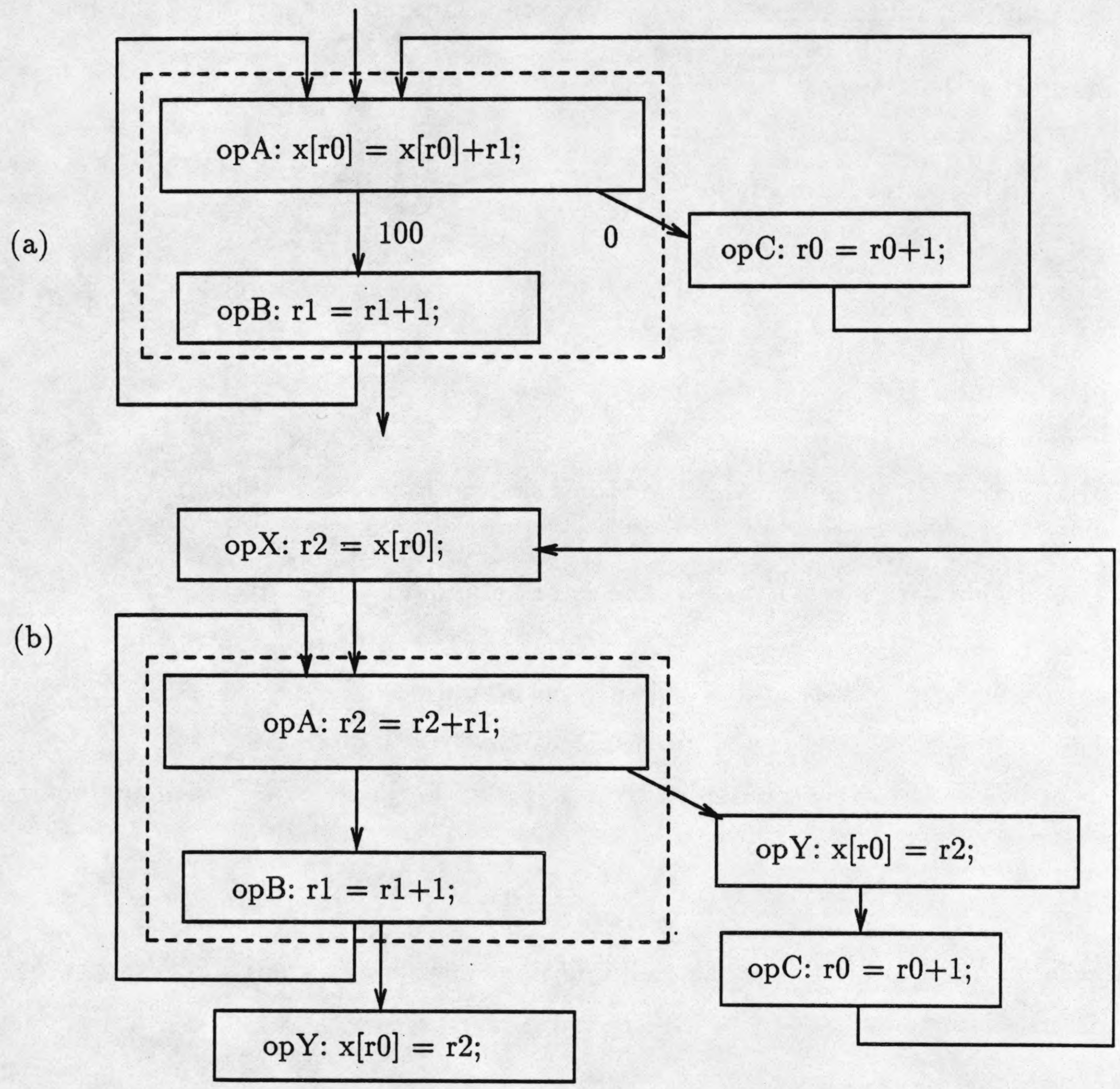

Figure 11: An example of super-block global variable migration.

another induction variable, thereby eliminating the need to increment the variable on each iteration of the loop. If the induction variable eliminated is then needed after the loop is exited, its value can be derived from one of the remaining induction variables.

The *precondition* function for induction variable elimination consists of the following boolean predicates that must all be satisfied.

1. $op(x)$ is an inductive instruction of the form $x \leftarrow x + K1$.
2. $op(x)$ is the only instruction which modifies $dest(x)$ in the super-block.
3. $op(y)$ is an inductive operation of the form $y \leftarrow y + K2$.
4. $op(y)$ is the only instruction which modifies $dest(y)$ in the super-block.
5. $op(x)$ and $op(y)$ are incremented by the same value ($K1 = K2$).[12]
6. There are no branch instructions between $op(x)$ and $op(y)$.
7. All uses of $dest(x)$ can be modified to $dest(y)$ in the super-block (without incurring time penalty).[13]

The *action* function of induction variable elimination consists of 4 steps.

1. $op(x)$ is deleted.

[12]The restriction of predicate 5 ($K1 = K2$) can be removed in some special uses of $dest(x)$, however these special uses are too complex to be discussed in this paper.

[13]For example, if we know that $dest(x) = dest(y) + 5$ because of different initial values, then a (branch if not equal) $bne(dest(x), 0)$ instruction is converted to a $bne(dest(y), -5)$ instruction. For some machines, $bne(dest(y), -5)$ needs to be broken down to a compare instruction plus a branch instruction; then, the optimization may degrade performance.

2. A subtraction instruction $op(m)$, $temp\_reg1 \leftarrow dest(x) - dest(y)$, is inserted after the last instruction in the preheader of the super-block loop.

3. For each instruction $op(a)$ which uses $dest(x)$, let $other\_src(a)$ denote the src operand of $op(a)$, which is not $dest(x)$. A subtraction instruction $op(n)$, $temp\_reg2 \leftarrow other\_src(a) - dest(m)$, is inserted after the last instruction in the preheader. The source operands of $op(a)$ are then changed from $dest(x)$ and $other\_src(a)$ to $dest(y)$ and $dest(n)$, respectively.

4. An addition instruction $op(o)$, $dest(x) \leftarrow dest(y) + dest(m)$, is inserted as the first instruction of each block that can be immediately reached when the super-block loop is exited in which $dest(x)$ is live in.

It should be noted that step 3 of the *action* function may increase the execution time of $op(a)$ by changing a source operand from an integer constant to a register. For example, a branch-if-greater-than-zero instruction becomes a compare instruction and a branch instruction if the constant zero source operand is converted to a register. Precondition predicate 7 prevents the code optimizer from making a wrong optimization decision. In traditional loop induction elimination, we check the entire loop body for violations of precondition predicate 7. In super-block loop induction elimination, we check only the super-block and therefore find more optimization opportunities.

4.11 Extension of super-block loop optimizations

In order to further relax the conditions for invariant code removal and global variable migration, the compiler can unroll the super-block loop body once. The first super-block serves as the first iteration of the super-block loop for each invocation, while the duplicate is used for iterations 2 and above. The compiler is then able to optimize the duplicate super-block loop knowing each instruction in the super-block has been executed at least once. For example, instructions that are invariant, but conditionally executed due to a preceding branch instruction, can be removed from the duplicate super-block loop. For invariant code removal, precondition predicates 3, 4, and 5 are always true for the duplicate super-block. For global variable migration, precondition predicate 3 is always true for the duplicate super-block.

In the actual implementation of the super-block loop optimization, a preheader is created for the original super-block loop before optimization. After optimization, those super-blocks with further optimization opportunities are unrolled once, and the original super-block, now serving as the first iteration of the loop, additionally serves as the preheader block of the duplicate super-block. Further restrictions can be placed on those loops that are unrolled to reduce the code expansion. However as we will discuss in the next section, we have found this increase to be small.

5 Experimentation

Table 3 shows the characteristics of the benchmark programs. The *size* column indicates the sizes of the benchmark programs measured in numbers of lines of C code. The *description*

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 3: Benchmarks.

column briefly describes the benchmark programs.

For each benchmark program, we have selected a number of input data for profiling. Table 4 shows the characteristics of the input data sets. The *input* column indicates the number of inputs that are used for each benchmark program. The *description* column briefly describes the input data. For each benchmark program, we have selected one additional input and used that input to measure the performance. The execution time of the benchmark programs that are annotated with probes for collecting profile information is from 25 to 35 times slower than that of the original benchmark programs. It should be noted that our profiler implementation is only a prototype and has not been tuned for performance.

Table 5 shows the output code quality of our prototype compiler. We compare the output code quality against that of the MIPS C compiler (release 2.1, -O4) and the GNU C compiler (release 1.37.1, -O), on a DEC3100 workstation which uses a MIPS-R2000 processor. The

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	ditroff 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 (original espresso benchmarks)
tbl	20	ditroff 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 4: Input data for profiling.

name	*global*	*profile*	*local*	*MIPS.O4*	*GNU.O*
cccp	1.0	1.04	0.96	0.93	0.92
cmp	1.0	1.42	0.95	0.96	0.95
compress	1.0	1.11	0.95	0.98	0.94
eqn	1.0	1.25	0.88	0.92	0.91
eqntott	1.0	1.16	0.62	0.96	0.75
espresso	1.0	1.03	0.89	0.98	0.87
grep	1.0	1.21	0.88	0.97	0.81
lex	1.0	1.01	0.96	0.99	0.96
mpla	1.0	1.18	0.84	0.95	0.87
tbl	1.0	1.03	0.94	0.98	0.93
wc	1.0	1.32	0.97	0.96	0.87
xlisp	1.0	1.16	0.91	0.88	0.76
yacc	1.0	1.08	0.87	1.00	0.90
avg.	1.0	1.15	0.89	0.96	0.88
s.d.	-	0.12	0.09	0.03	0.07

Table 5: Speed for each individual benchmark.

numbers that are shown in Table 5 are the speedups over the execution times of globally optimized codes that are produced by our prototype compiler. The *profile* column shows the speedup that is achieved by applying the set of profile-based code optimizations that have been discussed in the previous section on each globally optimized code. The *local* column shows the negative speedup that is due to using only local but no global code optimizations. The *MIPS.O4* column shows the speedup that is achieved by the MIPS C compiler over our global code optimizations. The *GNU.O* column shows the speedup that is achieved by the GNU C compiler over our global code optimizations. The numbers in the *MIPS.O4* and *GNU.O* columns show that our prototype global code optimizations performs slightly better than the two production compilers for all benchmark programs.

Comparing the performance improvement from local code optimization to global code optimization, to that from global code optimization to profile-based super-block code optimization, we clearly see the importance of these super-block code optimizations.

The sizes of the executable programs directly affect the cost of maintaining these programs in a computer system in terms of disk space. In order to control the code expansion due to tail-duplication, basic blocks are added to become a part of a trace only if their execution counts exceed a predefined constant threshold. For these experiments we use an execution count threshold of 100. Table 6 shows how code optimizations affect the sizes of the benchmark programs. The *profile* column shows the sizes of profile-based code optimized programs relative to the sizes of globally optimized programs. The *local* column shows the sizes of locally optimized programs relative to that of globally optimized programs. In

name	*global*	*profile*	*local*
cccp	1.0	1.03	0.98
cmp	1.0	1.11	1.00
compress	1.0	1.01	1.00
eqn	1.0	1.10	1.00
eqntott	1.0	1.00	1.00
espresso	1.0	1.07	1.01
grep	1.0	1.09	1.01
lex	1.0	1.08	1.02
mpla	1.0	1.13	1.01
tbl	1.0	1.06	1.00
wc	1.0	1.01	0.99
xlisp	1.0	1.20	1.01
yacc	1.0	1.09	1.00
avg.	1.0	1.07	1.00
s.d.	-	0.06	0.01

Table 6: Ratios of code expansion.

Table 6, we show that our prototype compiler has effectively controlled the code expansion due to forming super-blocks.

The cost of implementing the profile-based classic code optimizations is modest. The global code optimizer in our prototype compiler consists of approximately 32,000 lines of C code. The profile-based classic code optimizer consists of approximately 11,000 lines of C code. The profiler has about 2,000 lines of C code and a few assembly language subroutines.

6 Conclusions

We have shown how an execution profiler can be integrated into an optimizing compiler to provide the compiler with run-time information about user programs. For each code optimization, the profile information along with information from static loop analysis is used to reduce the execution time of the most frequently executed program regions. We have described our implementation techniques and presented the formulation of profile-based classic code optimizations. We have identified two major reasons why these code optimizations are effective: (1) eliminating control flows into the middle sections of a trace, and (2) optimizing the most frequently executed path in a loop. Evaluation has been done with realistic C application programs. Experimental results have shown that significant performance improvement can be obtained from profile-based classic code optimizations.

References

[Aho 86] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986.

[Allen 71] F. Allen and J. Cocke, "A Catalogue of Optimizing Transformations", pp.1-30 of [Rustin 72], 1972.

[AT&T 79] AT&T Bell Laboratories, *UNIX Programmer's Manual*, Murray Hill, N.J., January 1979.

[Chang 88] 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, pp.21-29, San Diego, California, November 1988.

[Chang 89] P. P. Chang and W. W. Hwu, "Control Flow Optimization for Supercomputer Scalar Processing", Proceedings, 1989 International Conference on Supercomputing, Crete, Greece, June 1989.

[Ellis 86] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, The MIT Press, 1986.

[Ferrari 83] D. Ferrari, G. Serazzi, and A. Zeigner, *Measurement and Tuning of Computer Systems*, Prentice-Hall, Inc., Englewood Cliffs, N.J. 07632, 1983.

[Fisher 81] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction", IEEE Transactions on Computers, Vol.C-30, No.7, July 1981.

[Graham 82] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: A Call Graph Execution Profiler", Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices, Vol.17, No.6, pp.120-126, June 1982.

[Graham 83] S. L. Graham, P. B. Kessler, and M. K. McKusick, "An Execution Profiler for Modular Programs", Software-Practice and Experience, Vol.13, John Wiley & Sons, Ltd., New York, 1983.

[Hwu 89.1] W. W. Hwu and P. P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler", Proceedings, 16th Annual International Symposium on Computer Architecture, Jerusalem, Israel, June 1989.

[Hwu 89.2] 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, Jerusalem, Israel, May 1989.

[Hwu 89.3] W. W. Hwu and P. P. Chang, "Inline Function Expansion for Compiling Realistic C Programs", Proceedings, ACM SIGPLAN'89 Conference on Programming Language Design and Implementation, Portland, Oregon, June 1989.

[McFarling 86] S. McFarling and J. L. Hennessy, "Reducing the Cost of Branches", The 13th International Symposium on Computer Architecture Conference Proceedings, pp.396-403, Tokyo, Japan, June 1986.

[Pettis 90] K. Pettis and R. C. Hansen, "Profile Guided Code Positioning", Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, pp.16-27, June 1990.

[Rustin 72] R. Rustin (Editor), *Design and Optimization of Compilers*, Prentice-Hall, Englewood Cliffs, N.J., 1972.

[Sarkar 89.1] V. Sarkar, "Determining Average Program Execution Times and Their Variance", Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, June 1989.

[Sarkar 89.2] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989.

[Wall 86] D. W. Wall, "Global Register Allocation at Link Time", Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction, June 1986.

[Wall 88] D. W. Wall, "Register Window vs. Register Allocation", Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, June 1988.