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

SUMMARY

This paper describes the design and implementation of an optimizing compiler that automatically generates profile information to assist classic code optimizations. This compiler contains two new components, an execution profiler and a profile-based code optimizer, which are not commonly found in traditional optimizing compilers. The execution profiler inserts probes into the input program, executes the input program for several inputs, accumulates profile information, and supplies this information to the optimizer. The profile-based code optimizer uses the profile information to expose new optimization opportunities that are not visible to traditional global optimization methods. Experimental results show that the profile-based code optimizer significantly improves the performance of production C programs that have already been optimized by a high-quality global code optimizer.

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

INTRODUCTION

The major objective of code optimizations is to reduce the execution time. Some classic code optimizations, such as dead code elimination, common subexpression elimination, and copy propagation, reduce the execution time by removing redundant computation. Other code optimizations, such as loop invariant code removal and loop induction variable elimination, reduce the execution time by moving instructions from frequently executed program regions to infrequently executed program regions. This paper describes an optimizing compiler that accurately identifies frequently executed program paths and optimizes them.

Static analysis, such as loop detection [1], can estimate execution counts, but the estimates are imprecise: outcome of conditional statements, loop iteration counts, and recursion depth are rarely predictable using static techniques. For example, a loop nested within a conditional statement does 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 it increases the execution time of other parts of the program.

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[1].[1] These static analysis methods do not distinguish between frequently and infrequently executed program paths. However, there are often instances where a value is destroyed on an infrequently executed path, which exists to handle rare events. As a result, one cannot apply optimizations to the frequently executed paths unless the infrequently executed paths are systematically excluded from the analysis. This requires an accurate estimate of the program run-time behavior.

Profiling is the process of selecting a set of inputs for a program, executing the program with these inputs, and recording the run-time behavior of the program. By carefully selecting inputs, one can derive accurate estimate of program run-time behavior with profiling. The motivation to integrate a profiler into a C compiler is to guide the code optimizations with profile information. We refer to this scheme as *profile-based code optimization.* In this paper, we present a new 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 less work from the user than hand-tuning of a program does, profile-based code optimizations can be applied to very large application

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

programs. With profile-based code optimizations, much of the tedious work can be eliminated from the hand-tuning process. The programmers can concentrate on more intellectual work, such as algorithm tuning.

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. 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.[2]

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.

RELATED STUDIES

Using profile information to hand-tune algorithms and programs has become a common practice for serious program developers. Several UNIX [3] profilers are available, such as *prof/gprof*[2] [3] and *tcov*[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 source program. The execution count of each straight-line segment of C

[2]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.

[3]UNIX is a Trademark of AT&T.

statements is reported. 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[5] [6]. Trace scheduling is a popular global microcode compaction technique[7]. For trace scheduling to be effective, the compiler must be able to identify frequently executed sequences of basic blocks. It has been shown that profiling is an effective method to identify frequently executed sequences of basic blocks in a flow graph[8] [9]. 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[10] [11]. A C compiler can implement a multiway branch, i.e., a *switch* statement in C, as a sequence of branch instructions or as a hash table lookup jump. 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[12].

Profile information can help a register allocator to identify the frequently accessed variables[13] [14]. 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[15]. A counter-based execution profiler that measures the average execution times and their variance can be optimized to achieve a run-time overhead less than 5% [16]. The estimated execution times can be used to guide program partitioning and scheduling for multiprocessors [17].

DESIGN OVERVIEW

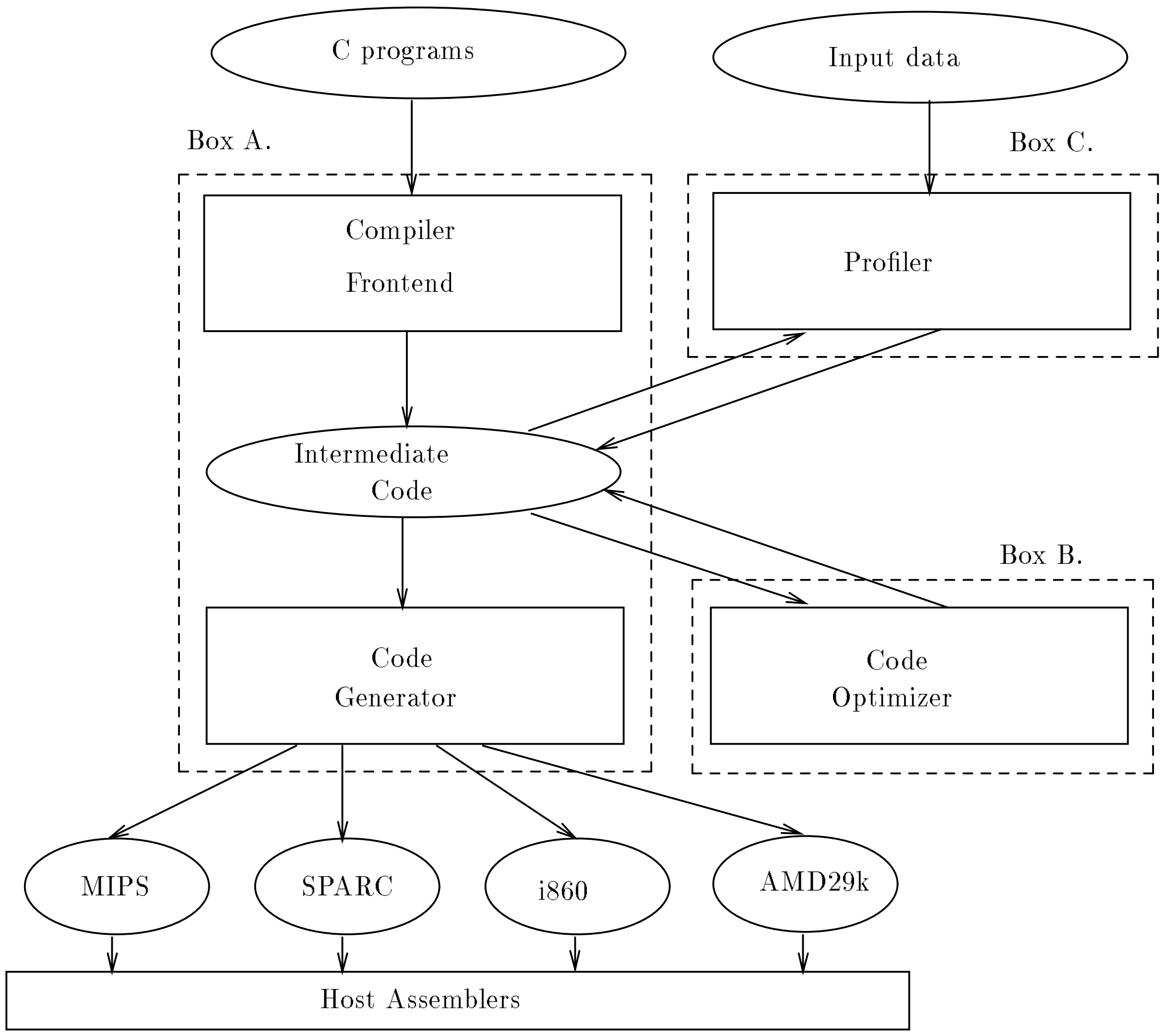

Figure 1: A block diagram of our prototype C compiler.

Figure 1 shows the major components of our prototype C compiler. *Box A* contains the compiler front-end and the code generator. *Box B* is the global code optimizer that operates on the intermediate form. Table 1 lists the local and global code optimizations that we have implemented in our prototype compiler. In order to have profile-based code optimizations, we have added a new

Box C to the prototype compiler. The profile information is then integrated into the intermediate code. Some code optimizations in *Box B* are modified to use the profile information. These code optimizations form a separate pass that is performed after the classic global code optimizations. Our prototype compiler generates code for several existing processor architectures: MIPS R2000, SPARC, Intel i860, and AMD29k.

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.

Program representation Our intermediate code has the following properties: (1) The operation codes are very close to those of the host machines, 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.

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. Because classic code optimizations have been developed based on the flow graph data structure[4], 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, $count(v)$ 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, 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.[5] 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.

Profiler implementation We are interested in collecting the following information with the profiler.

1. The number of times a program has been profiled.
2. The invocation count of each function.
3. The execution count of each basic block.
4. For each 2-way conditional branch instruction, the number of times it has been taken.

[4]Algorithms for finding dominators, detecting loops, computing live-variable information, and other dataflow analysis have been developed on the flow graph data structure[1].

[5]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(). However, these exceptions do not affect the correctness of code optimizations based on flow graphs.

5. For each multi-way branch instruction, the number of times each case has been taken.

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

Automatic profiling is supported by four tools: a probe insertion program, an execution monitor, a program to combine several profile files into a summarized profile file, and a program that maps the summarized profile data into a flow graph to generate a weighted flow graph data structure. All that a user has to do to perform profiling is to supply input files. The compiler automatically performs the entire profiling procedure in five steps:

(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. 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 this problem.

(d) Step (c) is repeated once for each additional input. All profile files are combined into a profile file by summing the counts and keeping a counter that indicates the number of profile files combined. From the above information, the average execution counts can be derived.

(e) Finally, the average profile data is mapped into the original intermediate code using the assigned function and basic block identifiers.

CODE OPTIMIZATION ALGORITHMS

Optimizing frequently executed paths All profile-based code optimizations presented in this section explore a single concept: *optimizing the most frequently executed paths.* We illustrate this concept using an example. Figure 2 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 $< A, B, E, F >$. Traditionally, 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.

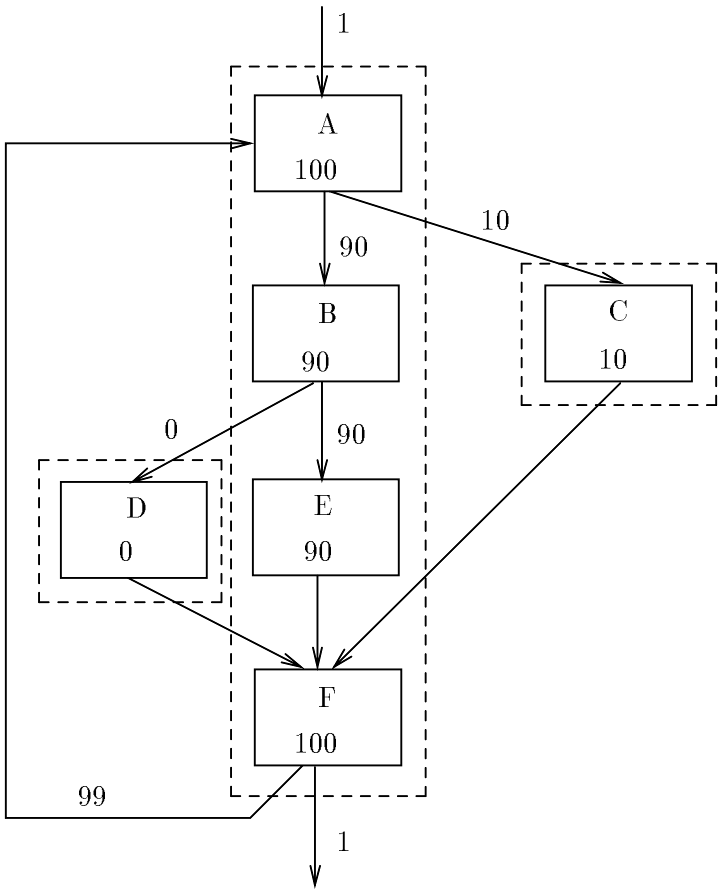

Figure 2: A weighted flow graph.

We propose the use of a simple data structure, called a super-block, to represent a frequently executed path. A super-block is a linear sequence of basic blocks that can be reached only from the first block in the sequence. The program control may leave the super-block from any basic block. When execution reaches a super-block, it is very likely that all basic blocks in that super-block are executed.

The basic blocks in a super-block do not have to be consecutive in the code. However, our implementation restructures the code so that as far as the optimizer is concerned, all blocks in a super-block are always consecutive.

Forming super-blocks The formation of super-blocks is a two step procedure: trace selection and tail duplication. Trace selection identifies basic blocks that tend to execute in 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[7] [8]. An experimental study of several trace selection algorithms was reported in [9]. The outline of a trace selection algorithm is shown in Figure 3. The $best\_predecessor\_of(node)$ function returns the most probable source basic block of $node$, if the source basic block has not yet been marked. The growth of a trace is stopped when the most probable source basic block of the $current$ node has been marked. The $best\_successor\_of(node)$ function is defined symmetrically.

Figure 2 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, in order to ensure that the program control can only enter at the top basic block. The tail duplication algorithm is shown in Figure 4. Using

```
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 3: 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 4: The tail-duplication algorithm.

the example in Figure 2, 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 5.[6] More code transformations are applied after tail duplication to eliminate jump instructions. For example, the F' super-block in Figure 5 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 exclude all basic blocks whose execution count is below a threshold value, e.g., 100 per run, from the trace selection process. They are also excluded from profile-based code optimization to control the increase in compile time.

Formulation of code optimizations Table 2 shows a list of classic code optimizations that we have extended to use profile information. The original formulation of these classic code optimizations can be found in [1] [19]. In Table 2, the *scope* column describes the extended 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. The optimizer first applies live-variable analysis to detect variables that are live across super-block boundaries, and then optimizes one super-block at a time. For each super-block, the profile-based code optimizations are applied one or more times, up to a limit or when no more opportunities can be detected.

In the following discussion, each code optimization consists of a *precondition* function and an *action* function. The precondition function is used to detect optimization opportunities and

[6]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.

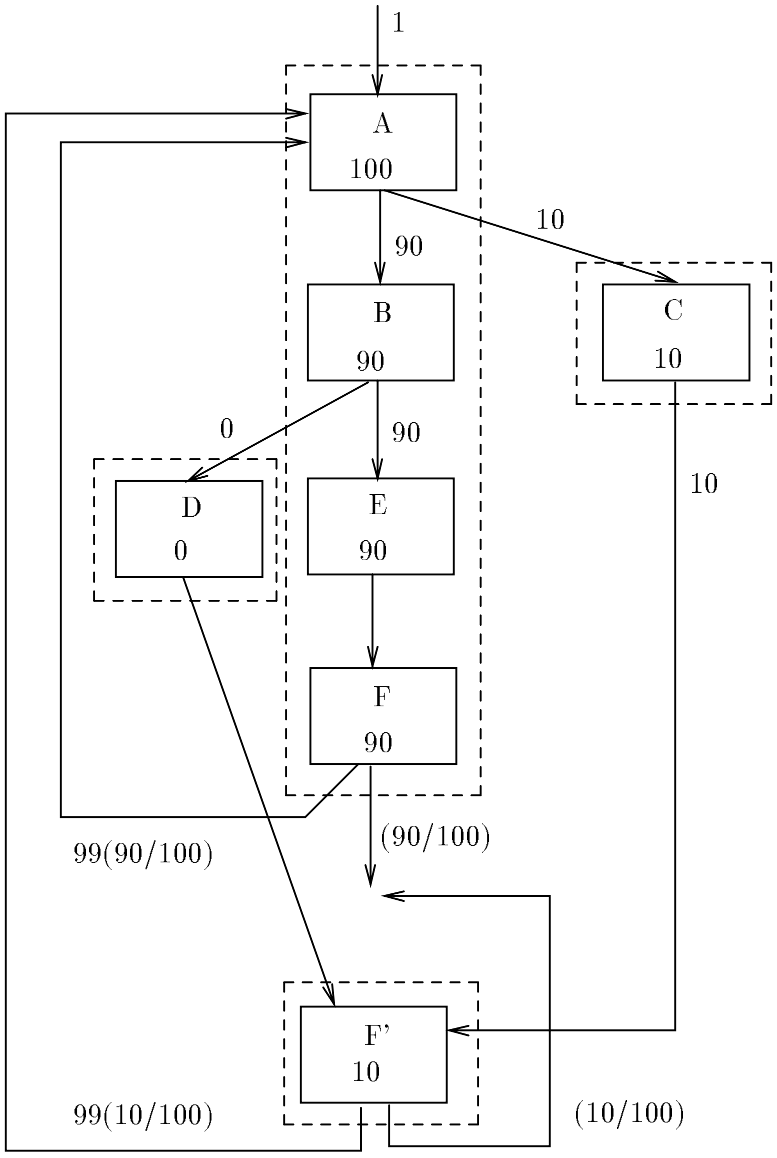

Figure 5: Forming super-blocks.

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.

to ensure that the transformation improves overall program performance. The *action* function performs the actual code transformation. To apply a code optimization, the optimizer identifies sets of instructions that may be eligible for the optimization. The precondition function is then invoked to make an optimization decision for each set. With the approval from the precondition function, the action function transforms the eligible sets into their more efficient equivalents.

We denote the set of variables that an instruction $op(i)$ modifies by $dest(i)$.[7] 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 $dest(i) \leftarrow f_i(src(i))$.

Local optimizations extended to superblocks There are several local code optimizations that can be extended in a straightforward manner to super-blocks. [8] These local optimizations include constant propagation, copy propagation, constant combining, common subexpression elimination, redundant load elimination, and redundant store elimination [1] [19].

[7]In this paper, we assume that there can be at most one element in $dest(i)$ of any instruction $op(i)$.
[8]The details of the required extensions can be found in a technical report [18].

Traditionally, local optimization cannot be applied across basic blocks and global code optimization must consider each possible execution path equally. However, there are often instances where an optimization opportunity is inhibited by an infrequently executed path. As a result, one cannot apply optimizations to the frequently executed paths unless the infrequently executed paths are systematically excluded from the analysis. Forming superblocks with tail duplication achieves this effect. Therefore, profile-based code optimizations can find more opportunities than traditional code optimizations.

To illustrate why local code optimizations are more effective when they are applied to superblocks, consider the case of common subexpression elimination shown in Figure 6. The original program is shown in Figure 6(a). After trace selection and tail duplication, the program is shown in Figure 6(b). Because of tail duplication, opC cannot be reached from opB; therefore, common subexpression elimination can be applied to opA and opC.

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 is that if the values of $dest(x)$ will not be used later in execution, $op(x)$ can be eliminated. To take full advantage of profile information, we propose an extension to dead code removal. In the extension, the precondition function consists of the following boolean predicates.

1. The super-block where $op(x)$ is defined is not a super-block loop.

2. $Op(x)$ is not a branch instruction.

3. $Dest(x)$ is not used before redefined in the super-block.

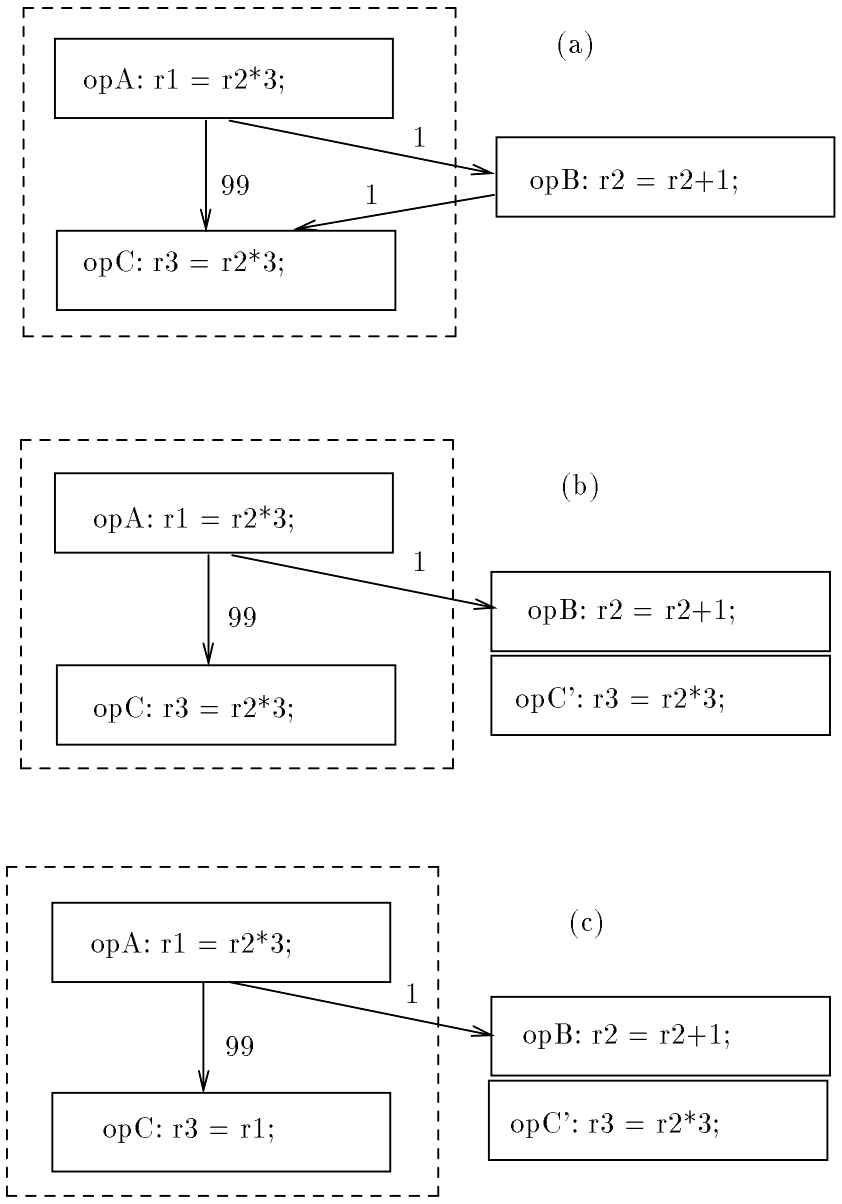

Figure 6: An example of super-block common subexpression elimination. (a) Original program segment. (b) Program segment after super-block formation. (c) Program segment after common subexpression elimination.

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. Find an integer z, such that $op(z)$ is the last branch instruction in $\{op(k), k = x + 1..y - 1\}$. Either there is no branch instruction in $\{op(k), k = x + 1..y - 1\}$ or $src(x)$ is not modified by an instruction in $\{op(j), j = x + 1..z\}$.

The action function of dead code removal consists of the following steps.

1. For every branch instruction in $\{op(i), i = x+1..y-1\}$, if $dest(x)$ is live[9] when $op(i)$ 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 an unconditional branch, copy $op(x)$ to become the last instruction of the super-block.

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

Dead code elimination is like common subexpression elimination in that tail duplication is a major source of opportunities to apply it. A special feature of our dead code elimination is that it can eliminate an instruction from a super-block by copying it to some 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 7 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

[9]A variable is live if its value will be used before redefined. An algorithm for computing live variables can be found in [1].

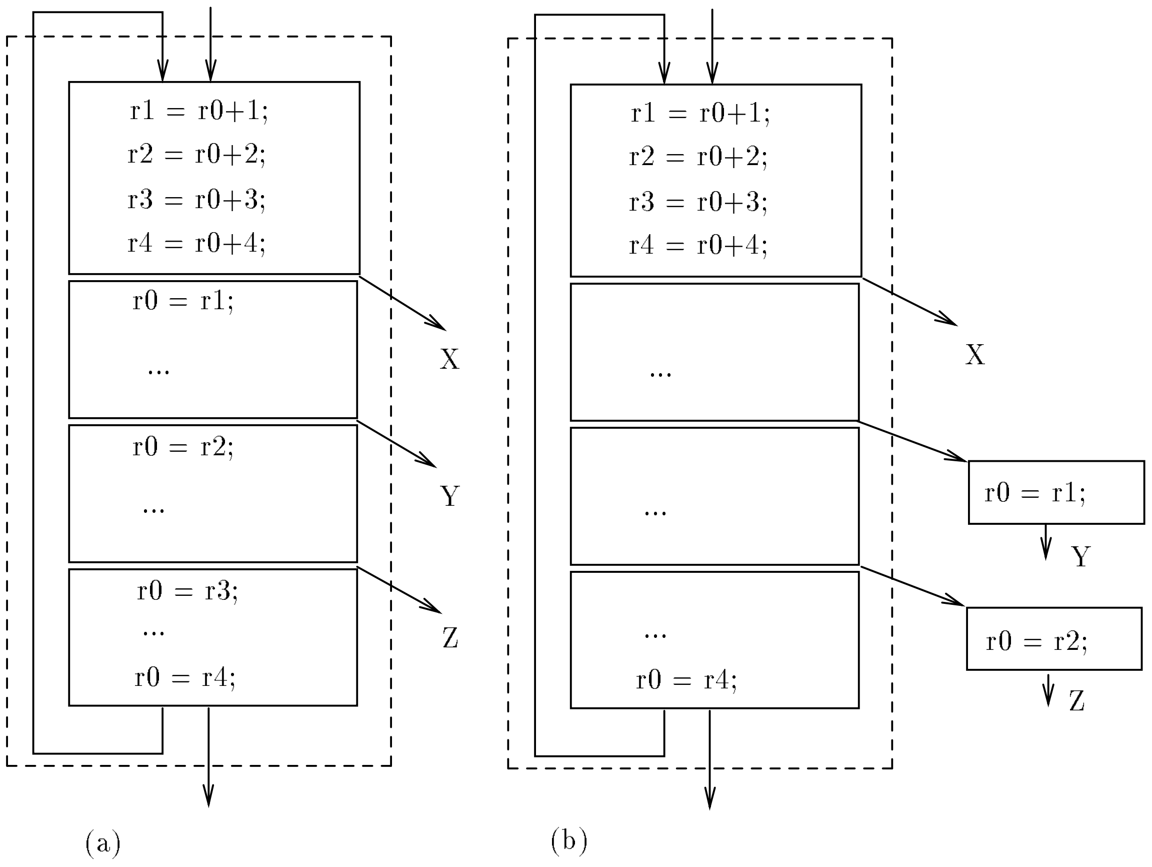

Figure 7: An example of super-block dead code removal. (a) Original program segment. (b) Program segment after dead code removal.

(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 7(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 7(b).

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 hardware 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.

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.

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 in the super-block, 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 in every iteration.

Predicates 1 and 2 depends 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 2 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 for invariant code removal is best illustrated by an example. Figure 8 shows a simple example of super-block loop invariant code removal. In Figure 8(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. 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 8(b).

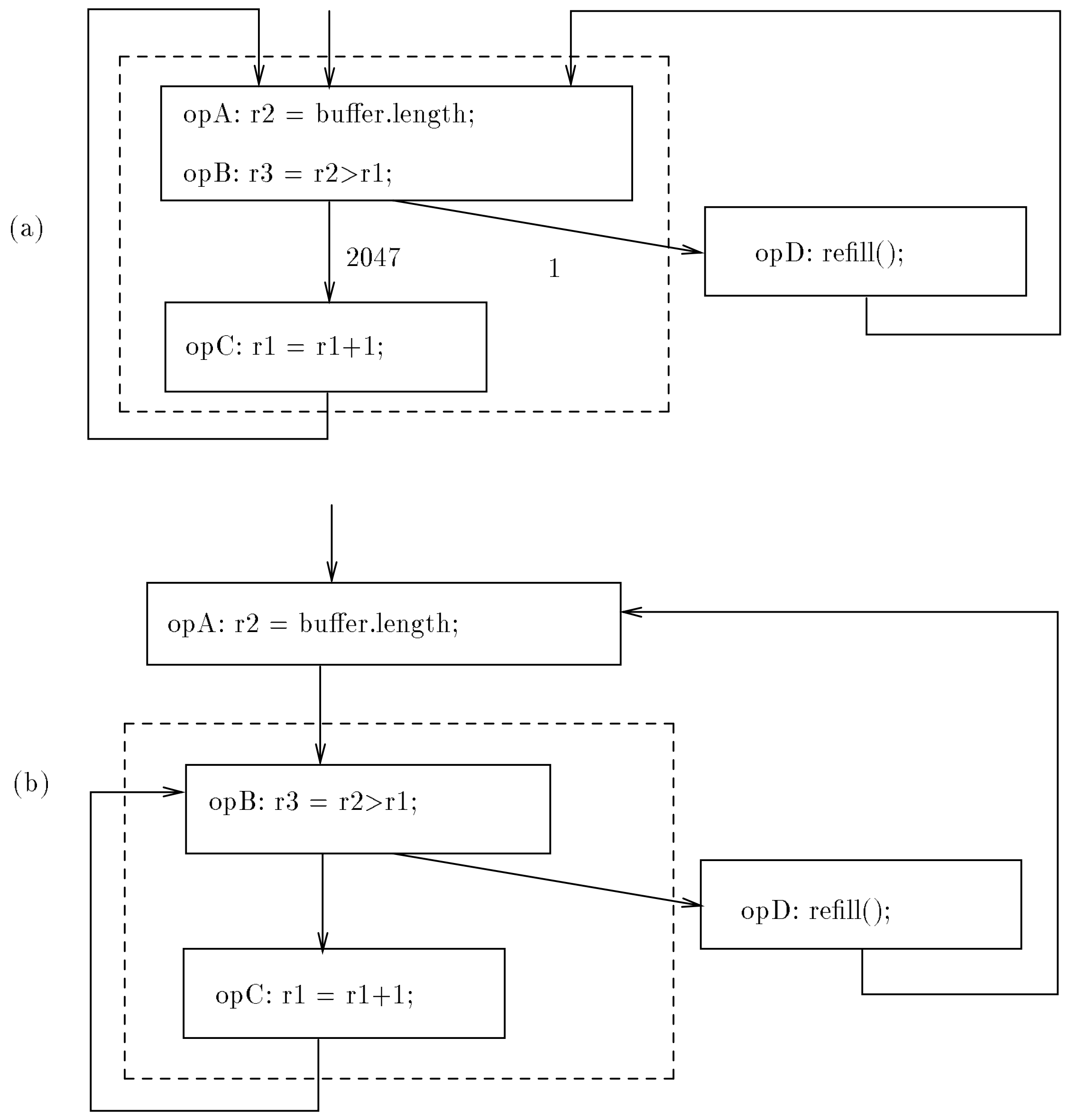

Figure 8: An example of super-block loop invariant code removal. (a) Original program segment. (b) Program segment after loop invariant code removal.

Global variable migration Global variable migration moves frequently accessed memory variables, such as 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. 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.[10]

[10]If a basic block that is immediately reached from a control flow exit of the super-block loop can be reached from

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 9 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 9(b). Extra instructions (opX and opY) are added to the super-block loop boundary points to ensure correctness of execution.

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 another induction variable, thereby eliminating the need to increment the variable on each iteration of the loop. If the induction variable eliminated is 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 $dest(x) \leftarrow dest(x) + K1$.

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 $dest(y) \leftarrow dest(y) + K2$.

multiple basic blocks, a new basic block needs to be created to bridge the super-block loop and the originally reached basic block.

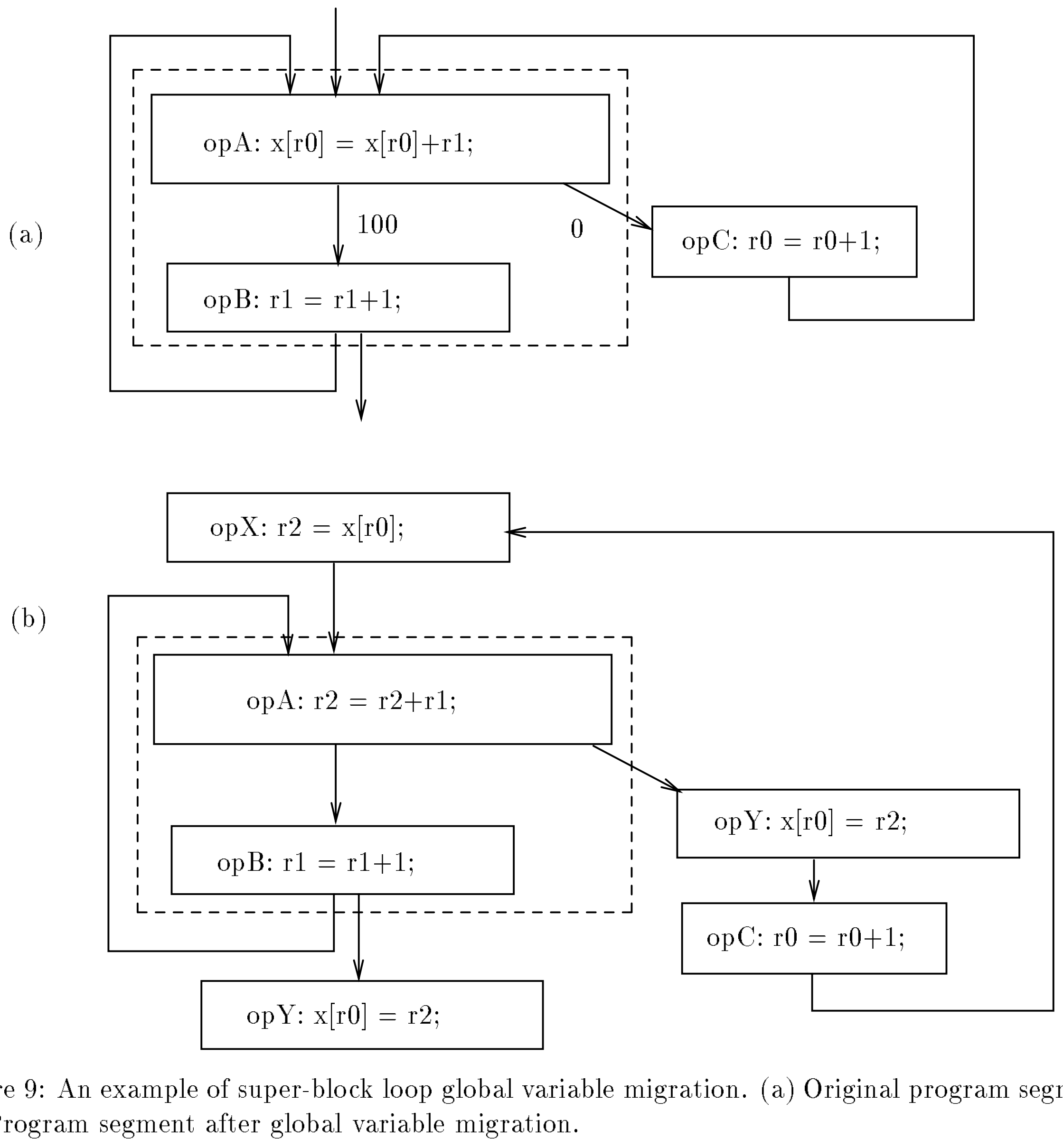

Figure 9: An example of super-block loop global variable migration. (a) Original program segment. (b) Program segment after global variable migration.

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, i.e., $K1 = K2$.[11]

6. There are no branch instructions between $op(x)$ and $op(y)$.

7. For each operation op(j) in which src(j) contains dest(x), either $j = x$ or all elements of src(j) except dest(x) are loop invariant.

8. All uses of $dest(x)$ can be modified to $dest(y)$ in the super-block without incurring time penalty.[12]

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

1. $op(x)$ is deleted.

2. A subtraction instruction $op(m)$, $dest(m) \leftarrow dest(x) - dest(y)$, 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 $op(n)$, $dest(n) \leftarrow other\_src(a) - dest(m)$, 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.

[11]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.

[12]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.

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. Predicate 8 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 predicates. In super-block loop induction elimination, we check only the super-block and therefore find more optimization opportunities.

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. With this extension, precondition predicates 3, 4, and 5 for invariant code removal and predicate 3 for global variable migration can be eliminated. The implementation of our C compiler includes this extension.

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* column briefly describes the benchmark programs.

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.

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.

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 collected 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.

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

Table 5: DEC3100 execution speed for each individual benchmark.

Table 5 shows the output code quality of our prototype compiler. We compare the output code speed 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 numbers that are shown in Table 5 are the speedups over the actual execution times of globally optimized code

name	*global*	*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

Table 6: Ratios of code expansion.

produced by our prototype compiler. The *profile* column shows the speedup that is achieved by applying profile-based code optimizations in addition to global code optimizations. Note that the input data used to measure the performance of profile-based code optimizations is different from those used to gather the profile information.

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. Table 5 clearly shows 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 into 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. In 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 conventional 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 is implemented with about 2,000 lines of C code and a few assembly language subroutines.

CONCLUSIONS

We have shown how an execution profiler can be integrated into an optimizing compiler to provide the compiler with run-time information about input programs. We have described our design and implementation 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. Experimental results have shown that profile-based classic code optimizations significantly improve the performance of production C programs.

Acknowledgements

The authors would like to thank Nancy Warter, Andy Glew, William Chen and all members of the IMPACT research group for their support, comments and suggestions. We would like to acknowledge the anonymous referees whose comments have helped us to improve the quality of this paper significantly. 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).

References

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

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

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

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

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

[15] 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.

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

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

[18] P. P. Chang, S. A. Mahlke, W. W. Hwu, *Using Profile Information to Assist Classic Code Optimizations*, Technical Report, Center for Reliable and High-Performance Computing, CRHC-91-12, University of Illinois, Urbana-Champaign, 1991.

[19] F. Allen and J. Cocke, "A Catalogue of Optimizing Transformations", pp.1-30 of [20], 1972.

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