Using Profile Information to Assist Advanced Compiler Optimization and Scheduling

William Y. Chen, Scott A. Mahlke, Nancy J. Warter, Richard E. Hank,
Roger A. Bringmann, Sadun Anik, and Wen-mei W. Hwu

Abstract

Compilers for superscalar and VLIW processors must expose sufficient instruction-level parallelism in order to achieve high performance. Compile-time code transformations which expose instruction-level parallelism typically take into account the constraints imposed by all execution scenarios in the program. However, there are additional opportunities to increase instruction-level parallelism along the frequent execution scenario at the expense of the less frequent execution sequences. Profile information identifies these important execution sequences in a program. In this paper, two major categories of profile information are studied: control-flow and memory-dependence. Profile-based transformations have been incorporated into the IMPACT compiler. These transformations include global optimization, acyclic global scheduling, and software pipelining. The effectiveness of these profile-based techniques is evaluated for a range of superscalar and VLIW processors.

1 Introduction

Compile-time code transformations such as code optimization and scheduling typically take into account the constraints imposed by all possible execution scenarios of the program. This is usually achieved using static analysis methods, such as live-variable analysis, reaching definitions, define-use chains, and data-dependence analysis [1]. These static analysis methods do not distinguish between frequent and infrequent execution scenarios. Although this conservative approach ensures the correctness of the transformations, it may overlook opportunities to optimize for frequent scenarios because of the constraints from infrequent scenarios. This problem can be addressed by using profile information, which estimates the frequency of a scenario, to assist the compiler in code transformations.

Using profile information to assist compile-time code transformations has received increasing attention from both the research and the product development communities. In the

area of handling conditional branches in pipelined processors, it has been shown that profile-based branch prediction at compile time performs as well as the best hardware schemes [2] [3]. In the area of global code scheduling, trace scheduling is a popular global microcode compaction technique [4]. For trace scheduling to be effective, the compiler must be able to identify the frequently executed sequences of basic blocks in a flow graph. It has been shown that profiling is an effective way to do this [5] [6]. Instruction placement is a code optimization which 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 way to guide instruction placement [7] [8].

Profile information can help a register allocator to identify the frequently accessed variables [9]. 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 [10]. In the area of classic code optimization, by eliminating the constraints imposed on frequent execution paths by infrequent paths, control flow profiling has been shown to improve the performance of classic global and loop optimizations [11].

In this paper, we present methods to obtain and use profile information to improve the performance of compile-time code transformations that expose instruction-level parallelism (ILP). The objective is to improve the performance of superscalar and VLIW processors by helping the compiler to expose and exploit more ILP. In particular, this paper presents methods that take advantage of two major categories of profile information: control-flow and memory-dependence.

In control-flow profiling, the instrumentation provides the compiler with the relative frequency of alternative execution paths. This paper presents methods that allow ILP-enhancing code optimizations, acyclic global scheduling, and software pipelining to focus on frequent program execution paths. By systematically eliminating the constraints imposed by the infrequent execution paths, the overall program performance can be improved by better parallelization and scheduling of the frequent paths.

In memory-dependence profiling, the instrumentation summarizes the frequency of address matching (conflicts) between two memory references. This information allows the compiler to optimize and/or reorder a pair of memory references in the presence of inconclusive memory-dependence analysis. When the run-time instrumentation indicates that two memory references almost never have the same address, the compiler can perform code reordering and optimization assuming that there is never a conflict. After performing the code reordering, the compiler also generates repair code to correct the execution state when a conflict does occur. Special hardware support is used to detect the occurrence of conflicts and invoke the repair code at run-time.

To demonstrate the effectiveness of control-flow profiling and memory-dependence profiling, we have modified the optimizer, the acyclic global code scheduler, and the software pipeliner in the IMPACT compiler to take advantage of the profile information. This paper describes the modified optimization and scheduling methods, explains the usefulness of profile information for each one, and quantifies the benefit of using profile information for these code transformations.

2 Using Control-Flow Profiling

Compilers usually represent the control flow within a function by a flow graph [1]. The nodes in a flow graph represent basic blocks in the function and the arcs represent possible control-flow transfers between two basic blocks. With control-flow profiling, the flow graph is augmented with weights. A weight is associated with each node to represent the number of times each basic block is executed. A weight is associated with each arc to represent the number of times each control transfer is taken. A flow graph augmented with node and arc weights is referred to as a weighted flow graph.

This section discusses the advantages of control-flow profile information for global optimization and global scheduling using the *superblock* structure. Superblock optimization and scheduling are most effective for exposing and exploiting ILP in general purpose applications. The use of control-flow profile information for software pipelining is also presented. Software pipelining is an effective technique for scheduling loops in numerical applications for VLIW and superscalar processors.

2.1 The Superblock

An effective structure to utilize control-flow profile information for compiler optimization and scheduling is the *superblock* [12] [11]. A superblock is a sequence of instructions in which control may only enter at the top, but may leave at one or more exit points. Equivalently, a superblock is a linear sequence of basic blocks in which control may only enter at the first basic block. Superblocks occur in functions naturally; however, few superblocks are typically found, and the size of natural superblocks is often small.

A compiler can form more and larger superblocks by utilizing a weighted flow graph. Superblock formation consists of two steps. First, traces are identified within the function. A trace is a set of basic blocks which are likely to execute in sequence [4]. Traces are selected by identifying a seed node in the weighted flow graph and growing the trace forward/backward to likely preceding/succeeding nodes until either there is no likely predecessor/successor or until the likely predecessor/successor has already been placed into a trace [6]. Each basic block is a member of exactly one trace. An example of trace selection applied to a weighted flow graph is shown in Figure 1a. In this example, three traces are identified in the code segment consisting of the following basic blocks: (A,B,E,F), (C), and (D).

Second, to create superblocks from traces, control entry points into the middle of a trace must be eliminated. Side entrances can be eliminated by duplicating a set of the basic blocks in the trace. This set is the union of all blocks which are side entry points and those blocks within the trace to which control may subsequently transfer. The control transfers into the side of the trace are then moved to the corresponding duplicated basic block. This process of converting traces to superblocks is referred to as *tail duplication*. An example of tail duplication is shown in Figure 1b. The trace consisting of basic blocks (A,B,E,F) contains two control entry points to basic block F. Tail duplication replicates basic block F (basic block F′) and adjusts the control transfers from basic blocks C and D to basic block F′. After removing all side entrances, trace (A,B,E,F) becomes a superblock.

The use of superblocks in optimization and scheduling for superscalar and VLIW processors is discussed in the next two sections.

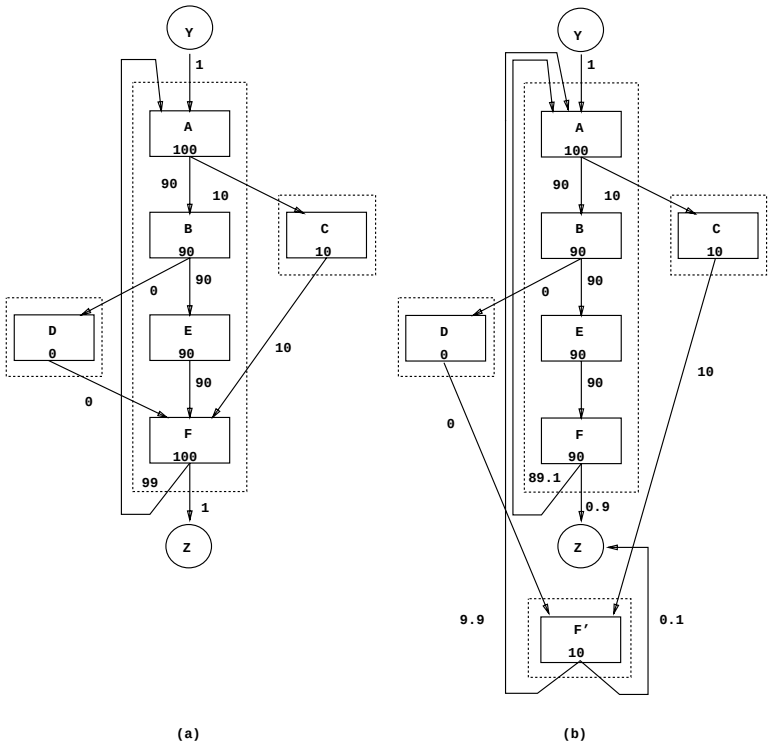

Figure 1: An example of superblock formation, (a) trace selection, (b) tail duplication.

2.1.1 Superblock Optimizations

In order to utilize the parallel hardware in superscalar and VLIW processors, sufficient ILP must be exposed by the code optimizer. A large number of compiler optimizations can be applied to superblocks to increase ILP. These superblock ILP optimizations are divided into two categories: superblock-enlarging optimizations and superblock dependence-removing optimizations. The purpose of superblock-enlarging optimizations is to increase the size of the most frequently executed superblocks so that the scheduler has a larger number of instructions to manipulate. The scheduler is more likely to find independent instructions to schedule at every cycle in a superblock when there are more instructions to choose from. An important feature of superblock enlarging optimizations is that only the most frequently executed parts of a program are enlarged. This selective enlarging strategy keeps the overall code expansion under control.

The three superblock enlarging optimizations that are currently being utilized are branch target expansion, loop peeling, and loop unrolling. Branch target expansion appends a copy of the target superblock to a superblock which ends with a likely-taken control transfer. Superblock loop peeling and loop unrolling replicate the body of superblock loops. A superblock loop is a superblock that ends with a likely control transfer to itself. Superblock loops are unrolled or peeled based on both the weight of the superblock and the expected number of times the superblock loop will iterate.

The purpose of superblock dependence-removing optimizations is to eliminate data dependences between instructions in frequently executed superblocks. These optimizations directly increase the amount of ILP available to the code scheduler. As a side effect, some

of these optimizations require that additional instructions be inserted at superblock entry and exit points. However, by applying these optimizations only to frequently executed superblocks, the code expansion is again regulated.

Six superblock dependence-removing optimizations are currently utilized: register renaming, induction variable expansion, accumulator variable expansion, operation migration, operation combining, and tree height reduction. Register renaming is a common technique to remove anti and output dependences between instructions [13]. Induction variable expansion and accumulator variable expansion remove flow, anti, and output dependences between definitions of induction and accumulator variables in unrolled superblock loop bodies [14]. Temporary induction and accumulator variables are created to remove the dependences. Operation migration moves an instruction from an important superblock to a less important superblock when the instruction's result is not directly used in its home superblock [14]. Operation combining eliminates flow dependences between pairs of instructions each of which has a constant source operand [15]. Tree height reduction exposes ILP in the computation of arithmetic expressions [16]. The effectiveness of superblock optimizations for superscalar and VLIW processors is experimentally evaluated later in this section.

2.1.2 Superblock Scheduling

To efficiently schedule code for superscalar and VLIW processors, it is necessary to schedule instructions among many basic blocks. Therefore instructions must be moved both above and below conditional branches. Superblocks provide an effective framework to schedule instructions along the most important paths of execution. Superblock scheduling consists of two steps: dependence graph construction followed by list scheduling.

The dependence graph contains four types of dependences: flow, anti, output, and control. Control dependences are initially inserted between an instruction and all preceding branches in a superblock. Based on the scheduling model and underlying processor support for speculative execution, these control dependences are later removed to allow more code motion flexibility. List scheduling using the instruction latencies and resource constraints is then applied to the superblock to determine the final schedule.

The code scheduler moves instructions in a superblock both above and below branches to achieve an efficient schedule. A non-control (not a branch or subroutine call) instruction, J, can always be moved below a branch, BR, provided a copy of J is placed at the target of BR if the location J writes to is live when BR is taken. Equivalently, if $dest(J)$ is in $live\_out(BR)$, then to move J below BR a copy of J must be placed at $target(BR)$. A non-control instruction, J, can be moved above a branch, BR, provided the following two restrictions are met:
Restriction 1 - $dest(J)$ is not in $live\_out(BR)$.
Restriction 2 - J will never cause an exception that may terminate program execution.
The first restriction can be eliminated with sufficient compiler renaming support. However, the second restriction is more difficult to eliminate. For conventional processors, memory load, memory store, integer divide, and all floating-point instructions may not be moved above branches within the superblock unless the compiler can prove that the instruction will never cause an exception.

Architectural support in the form of non-trapping versions of instructions which normally trap can be utilized to remove Restriction 2 [17] [12]. When the scheduler moves a possible

Benchmark	Size (Lines)	Benchmark Description
cccp	4787	GNU C preprocessor
cmp	141	compare files
compress	1514	compress files
eqn	2569	format math formulas for troff
eqntott	3461	boolean equation minimization
espresso	6722	truth table minimization
grep	464	string search
lex	3316	lexical analyzer generator
tbl	2817	format tables for troff
wc	120	word count
xlisp	7747	lisp interpreter
yacc	2303	parser generator

Table 1: Benchmarks.

trapping instruction above a branch in a superblock, it converts the instruction to its non-trapping counterpart. Instructions which depend on the possible trapping instruction can also be moved above the branch. When an exception condition exists for a non-trapping instruction, a garbage value is written into the destination of the instruction. For programs which would never have trapped when scheduled using conventional techniques, this garbage value does not affect the correctness of the program because the results of the instructions moved above the branch are not used when the branch is taken. However, for all other programs (e.g. undebugged code, or programs which rely on traps during normal operation), errors which would have caused a trap may now cause an exception at a later trapping instruction, or may cause an incorrect result. Superblock scheduling with non-trapping support is referred to as *general code percolation*. As part of our future work, promising techniques which allow the code scheduling flexibility of general percolation without ignoring exceptions are being investigated.

2.1.3 Experimental Results

Superblock optimization and scheduling have been implemented in the IMPACT compiler. The IMPACT compiler is a prototype optimizing compiler designed to generate efficient code for superscalar and VLIW processors. To study the effectiveness of superblock optimization and superblock scheduling, execution-driven simulation is performed for a range of superscalar and VLIW processors. The benchmarks used in this study are described in Table 1. Each benchmark is profiled on a variety of inputs to obtain fair control-flow profile information. An input different from those used for profiling is then used to perform the simulation for all the experiments reported in this work.

The basic processor used in this study is a RISC processor which has an instruction set similar to the MIPS R2000 [18]. The processor is assumed to have register interlocking and deterministic instruction latencies (Table 2). The processor contains 64 integer registers and 32 floating point registers. Furthermore, architectural support for code scheduling using general percolation is assumed [12].

The performance of superblock optimizations and scheduling is compared for superscalar/VLIW processors with issue rates 2, 4, and 8. The issue rate is the maximum number

Instruction Class	Latency	Instruction Class	Latency
integer ALU	1	FP ALU	3
integer multiply	3	FP conversion	3
integer divide	10	FP multiply	3
branch	2	FP divide	10
memory load	2	memory store	1

Table 2: Instruction latencies.

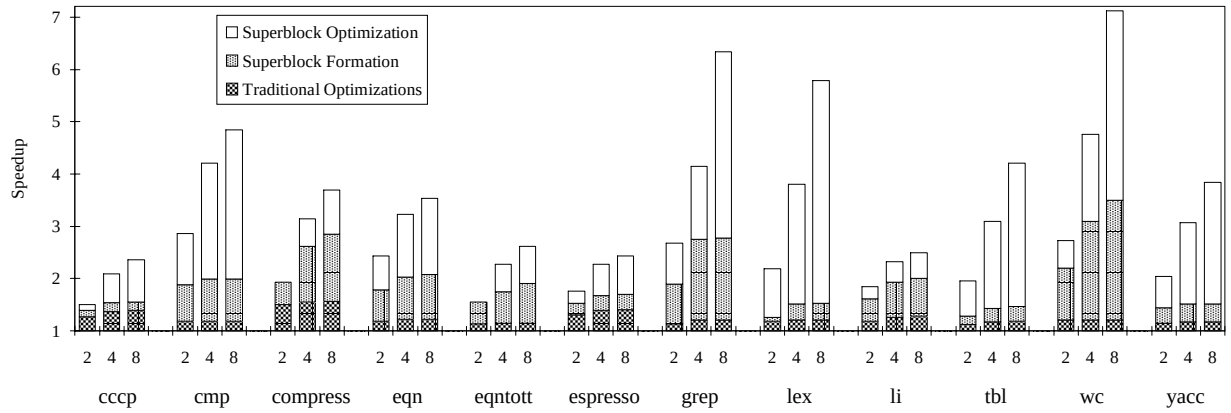

Figure 2: Performance improvement resulting from superblock optimization and superblock scheduling.

of instructions the processor can fetch and issue per cycle. No limitation has been placed on the combination of instructions that can be issued in the same cycle. For each machine configuration, the program execution time, assuming a 100% cache hit rate, is reported as a speedup relative to the program execution time for the base machine configuration. The base machine configuration is an issue-1 processor with traditional optimization support and basic-block code scheduling. Traditional optimizations include all conventional local, global, and loop optimizations that do not utilize control-flow profile information [1]. Furthermore, the following optimizations which utilize control-flow profile information are also included as traditional: function inline expansion, register allocation, instruction placement, and branch prediction.

The performance improvement due to superblock formation and scheduling, and then the further improvement due to superblock optimization is shown in Figure 2. Superblock formation and optimization are applied in addition to the traditional optimizations. From the figure, it can be seen that superblock techniques substantially increase the performance of superscalar/VLIW processors. With only traditional optimization and scheduling support, little performance improvement over the base processor is achieved. With superblock formation, significant performance improvements are observed for all benchmarks due to the increased number of instructions the scheduler examines at one time. For example, with an issue-4 processor, superblock formation and scheduling provide an average performance gain of 58% over traditional compiler optimization and scheduling. Superblock optimization applied in addition to superblock formation provides larger performance improvements. With an issue-4 processor, an average improvement of 160% over traditional compiler optimization

and scheduling is observed.

The importance of superblock optimization and scheduling is most apparent for higher-issue-rate processors. For these processors, a compiler must expose a large amount of ILP to fully utilize the existing resources. With only traditional compiler support, little performance gain is observed by increasing the issue rate beyond two. However, with superblock optimization and scheduling an increased level of ILP is exposed to take better advantage of the higher issue rates.

2.2 Assisting Software Pipelining

Software pipelining is a compile-time scheduling technique that exploits VLIW and superscalar processors by overlapping the execution of loop iterations. When overlapping loops with conditional constructs (e.g. if-then-else constructs) there is the potential for a large amount of code expansion. Every time two conditional constructs overlap, the number of execution paths double. If a software pipeline schedule overlaps n copies of a conditional construct, each from a different iteration of the loop, then there is a 2^n code expansion. Almost all of the existing techniques have this code expansion [19] [20] [21] [22]. One technique, modulo scheduling with if-conversion, does not incur any code expansion [23] [24]. This method requires architectural support for predicated execution [25]. An alternative modulo scheduling technique uses hierarchical reduction [21]. This technique can be modified such that a conditional construct from two different iterations will not overlap. Thus, the only code expansion comes from duplicating code scheduled with the conditional construct and from overlapping different conditional constructs of the loop body. A loop body with b conditional constructs may have up to 2^b code expansion due to the overlapping of these conditional constructs. While this technique will have relatively low code expansion and does not require special architectural support, it will not improve the performance as much as hierarchical reduction without this restriction or as much as if-conversion.

Control-flow profiling information can be used to improve the performance of this modified hierarchical reduction based modulo scheduling technique. In general, software pipelining is a good method for scheduling numeric loops because the loops often do not have dependence cycles and thus can be fully overlapped. Even with dependence cycles, the loops can be partially overlapped. In numeric codes, branches are often used to detect boundary or exception conditions where the branch is infrequently executed. In these cases, the infrequently executed path typically takes longer to execute. Profile information can be used to remove this longer execution path from the pipelined loop. In this section, we review modulo scheduling with hierarchical reduction and discuss how the technique can be modified to reduce the code expansion. Then we show how profiling can be used to improve the performance of this modified hierarchical reduction technique and compare the results with if-conversion.

2.2.1 Modulo Scheduling with Modified Hierarchical Reduction

In modulo scheduling, the interval at which loop iterations are initiated, the iteration interval (II), is fixed for every iteration of the loop [23]. The algorithm determines the lower bound on II and then tries to find a schedule for this II. The algorithm first tries to schedule operations involved in dependence cycles and then list schedules the remaining operations.

If no schedule can be found, the II is incremented and the process is repeated until a schedule is found or the II reaches a predefined limit. The minimum II is determined by the resource and dependence cycle constraints. In this paper, we focus on loops without dependence cycles.

Without dependence cycles, modulo scheduling consists of three basic steps. First the data dependence graph is constructed. Since there are no dependence cycles, the graph will only have intra-iteration dependencies. Second, a lower bound for II is determined by the most heavily utilized resource for the loop. If an iteration uses a resource r for c_r cycles and there are n_r copies of this resource, then the minimum II due to resource constraints, RII, is

$$RII = \max_{r \in R} \left\lceil \frac{c_r}{n_r} \right\rceil ,$$

where R is the set of all resources. The last step is to list schedule the loop body using a modulo resource reservation table [21]. The modulo resource reservation table has a row for every cycle in the iteration interval and a column for every function unit. An operation can be scheduled at time t_i if the necessary functional unit is free in row $t_i \ mod \ II$ in the modulo resource reservation table.

In order to modulo schedule loops with conditional branches, the control dependences must be converted into data dependences. With architectural support for predicated execution, if-conversion can be used to convert control dependences into data dependences [24]. Without hardware support to explicitly convert control dependences into data dependences, hierarchical reduction implicitly converts control dependences by encapsulating the operations of a control construct (e.g., an *if-statement*) within a pseudo operation (pseudo-op) [21]. To form the pseudo-op, both paths of the control statement are list scheduled including the conditional branch. The resource pattern of the pseudo-op is the union of the resource pattern of each path. Any data dependences between an operation within the pseudo-op and another operation outside the pseudo-op is migrated up to the pseudo-op with the associated delays adjusted according to the pseudo-op's list schedule. If there are nested control statements, the control statements are hierarchically converted into pseudo-op's. Once the conditional statements are converted into pseudo-op's, the new data dependence graph can be modulo scheduled. To avoid exponential code expansion, a restriction is added during modulo scheduling that requires each pseudo-op to be scheduled within one II. Table 3 summarizes the characteristics of modulo scheduling using the modified hierarchical reduction technique and using if-conversion.

2.2.2 Profile-based Optimization

Since the additional constraint for hierarchical reduction requires that the pseudo-op be scheduled within one II, the minimum II must be as long as the longest path through a conditional construct. This will degrade the performance of hierarchical reduction as compared to if-conversion. However, as stated earlier, numerical programs often have predictable branches where the least frequently executed path is longer than the frequently executed path. Profile information can be used to remove this infrequently executed path. Figure 3 shows an example of the profile optimization for hierarchical reduction assuming an issue-2 VLIW processor where no limitation is placed on the combination of operations issued. A, B, C, D, E, and F are operations, where A is a conditional branch and D is a jump operation.

Feature	Hierarchical Reduction	If-Conversion
resource constraints	union of resources along along both paths of conditional construct	sum of resources along both paths of conditional construct
additional constraints	conditional construct scheduled within one II	none
code expansion	due to the duplicate code scheduled with conditional constructs and the overlap of different conditional constructs	none
hardware support	none	predicated execution

Table 3: Characteristics of modified hierarchical reduction and if-conversion.

In Figure 3d and e, the operations are scheduled as VLIW instructions.

The basic profile optimization algorithm is as follows. First, generate a weighted control-flow graph as shown in Figure 3a. Remove basic blocks for infrequently executed paths such as path C-D. Second, form a pseudo-node, (A-B), from the corresponding frequently executed path B. Generate the data dependency graph for the remaining loop operations. Figure 3b shows the data dependence graph where the dependence distance is zero and the arcs are labeled with the type of dependence (flow) and the operation latencies. Third, modulo schedule the operations in the dependence graph, (A-B), E, F, using the modulo resource reservation table as shown in Figure 3c. Note, that operation F is available at cycle 4 but since there are no functional units available it is delayed to cycle 5. Fourth, duplicate operations scheduled with the most frequently executed paths (e.g., B) and schedule them with the corresponding infrequently executed operations (e.g., C and D). Correct the jump operation of the infrequently executed basic block (e.g. D) to jump to the instruction after the last instruction in the frequently executed path (e.g., B). As shown in Figure 3d, the next instruction following operation B is the first instruction in the loop. The resulting loop kernel [1] is shown in Figure 3d. The kernel execution is shown in Figure 3e, where the numbers indicate which iteration the operation is from. When branch A3 is taken, C3, D3, and the last operation of iteration one, F1, are executed but operation B3 is not.

A similar technique can be used for if-conversion [27]. The if-conversion profile-based optimization preserves the zero-code expansion by using predicates to eliminate the need for copying operations to the infrequently executed paths.

2.2.3 Experimental Results

Two versions of modulo scheduling have been implemented in the IMPACT compiler, one using the modified hierarchical reduction technique and the other using if-conversion. To evaluate the effectiveness of using control-flow profiling to improve the performance of both techniques, an execution-driven simulation is performed for a range of superscalar and VLIW

[1] A software pipeline has a prologue, kernel, and epilogue. The prologue and epilogue fill and empty the software pipeline respectively. The kernel is the steady state of the loop execution.

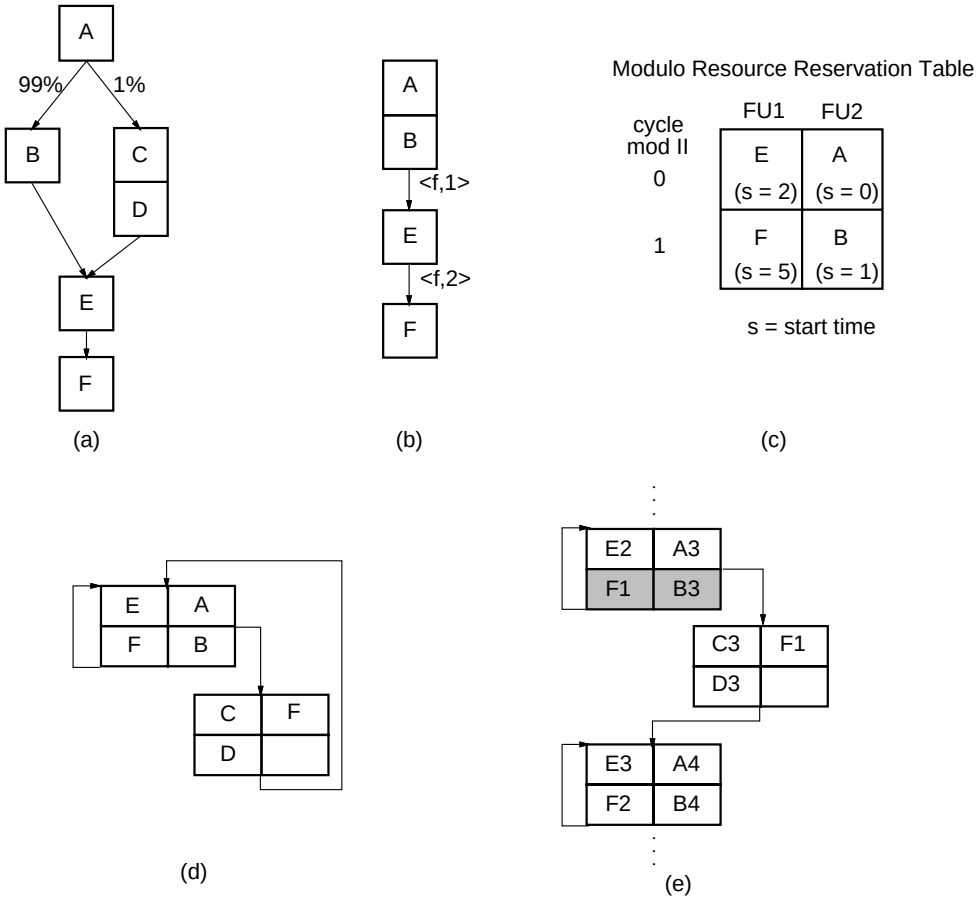

Figure 3: Modulo scheduling with modified hierarchical reduction using profile-based optimization, (a) weighted control-flow graph, (b) data dependence graph, (c) modulo schedule of A-B-E-F, (d) kernel schedule, (e) kernel execution when A is taken.

processors. The benchmarks used in this study are 25 loops selected from the Perfect Suite [28]. These loops have no cross-iteration dependences and have at least one conditional construct. The profiling optimization is applied for branches where one path is taken at least 80% of the time. The loops are assumed to execute a large number of times and thus only the steady-state (kernel) execution is measured for the modulo scheduled loops.

The basic processor used in this study is a RISC processor which has an instruction set similar to the Intel i860 [29]. Intel i860 instruction latencies are used. The processor has an infinite number of registers and an ideal cache.

The performance of the two modulo scheduling techniques, modified hierarchical reduction and if-conversion, with and without the profiling optimization are compared for superscalar/VLIW processors with issue rates 2, 4, and 8[2]. No limitation is placed on the combination of instructions that can be issued in the same cycle. For if-conversion, architectural support for predicated execution is assumed [25]. For each machine configuration, the loop execution time is reported as a speedup relative to the loop execution time for the base machine configuration. The base machine configuration is an issue-1 processor with

[2]Induction variable reversal was not applied to these loops before modulo scheduling [30]

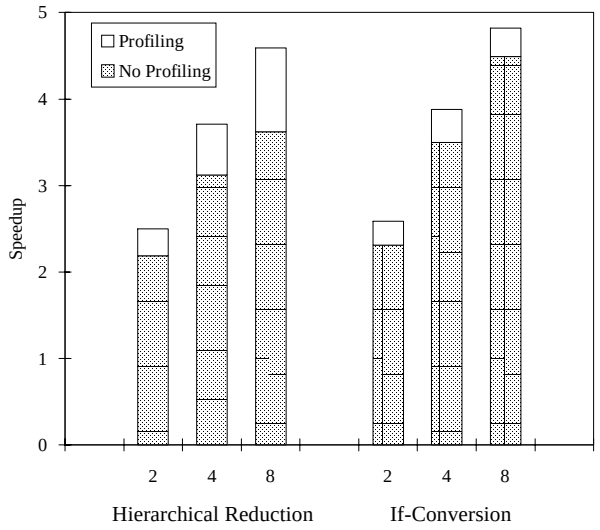

Figure 4: Performance improvement for modulo scheduling using profile optimization.

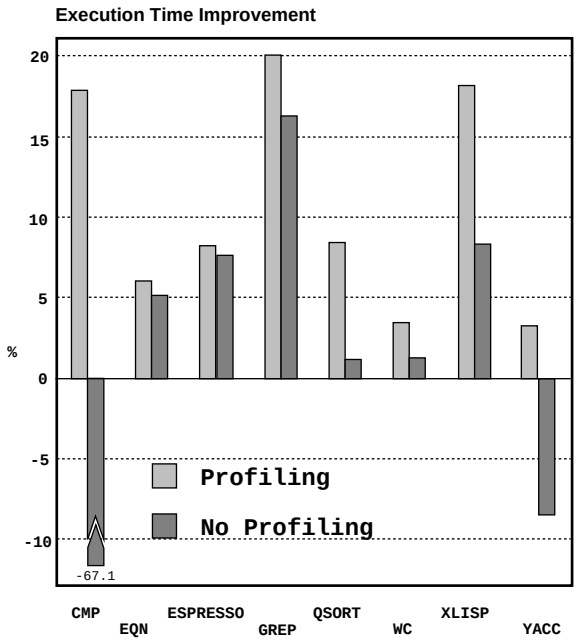

Figure 5: Performance comparison with and without memory-dependence profiling.

traditional local and global optimization support and basic-block code scheduling. The base machine does not support predicated execution.

The benefit of using control-flow profile information to improve the performance of the modified hierarchical reduction technique is shown in Figure 4. The speedups are calculated using the harmonic mean. Overall, profiling improves the performance of the hierarchical reduction scheme by approximately 14% for the issue-2 machine, 19% for the issue-4 machine, and 27% for the issue-8 machine. Without the profiling optimization, if-conversion on average performs approximately 12% better than hierarchical reduction for an issue-4 machine and approximately 24% better for an issue-8 machine. With profiling, the performance of if-conversion over hierarchical reduction is reduced to approximately 5% for both issue-4 and issue-8 machines. Also note that hierarchical reduction with profiling performs better than if-conversion without profiling.

It is interesting to note the affect of the profile optimization for each technique. For hierarchical reduction, limiting the scheduling of the conditional construct to one II limits the number of iterations that can be overlapped. This constrains the speedup of this technique as the issue rate increases. Since the least frequently executed path is often the longest path in these loops, using profiling information reduces the size of the pseudo-node that must be scheduled within one II and thus allows for more overlap. Thus, for hierarchical reduction, the improvement due to profiling increases as the issue rate increases. For if-conversion, the resource constraints due to always fetching operations from both paths of the condition construct limits the performance. Thus, using profiling reduces the resource constraints. This particularly benefits the lower issue rates which incur more resource conflicts.

In this section we have shown how profiling can be used to improve the performance of modulo scheduling with either hierarchical reduction or if-conversion. Hierarchical reduction

with profiling has good performance and low code expansion without the need for additional hardware support. For no code expansion and the best performance, both architectural support for predicated execution and profiling should be used.

3 Using Memory-Dependence Profiling

As shown in Section 2, superblock optimizations substantially increase the performance of superscalar/VLIW processors. However, many dependences between pairs of memory references still remain in the superblocks. These dependences restrict the ability of the scheduler to move loads upward past stores. Because loads often occur on critical paths in the program, the loss of these code reordering opportunities can limit the effectiveness of compile-time code scheduling. Because of the practical limitations of current memory-dependence-analysis techniques, dependence arcs between memory references are added conservatively. Many independent pairs of memory references are marked as dependent because the dependence analyzer cannot conclusively determine that the two references always have different addresses. Also, two dependent memory references may actually have the same address only occasionally during execution.

The dependences can be removed between these memory references during compilation, and more aggressive code reordering can be done [31]. *Memory-dependence profiling* uses run-time information to estimate how often pairs of memory references access the same location. There are many cases in which the dependence analyzer indicates a dependence between two memory references, but the profile information reports that the references rarely or never have the same address. In these cases, the references are reordered as if there were no dependence and repair code provided by the compiler is added to maintain correct program execution. Special hardware support called the *memory conflict buffer* (MCB) can be used to reduce the overhead by detecting when the reordered dependent memory references may cause incorrect program execution and invoking the repair code [32].

If a memory store precedes a memory load and may access the same location as the load (i.e., the dependence analysis indicates a dependence but cannot conclusively determine that the pair of references always accesses the same address), the store is referred to as an *ambiguous store* with respect to that load. The pair of references is called an *ambiguous store/load pair*. When a load is moved above an ambiguous store, the load becomes a *preload*. The situation in which a preload and an ambiguous store have the same address is referred to as a *conflict* between the pair of references. In the scheduling model presented in this section, instructions that use the preloaded data can also be moved above the ambiguous store. When a conflict occurs, any computations involving the preload destination register must be retried.

3.1 Memory-Dependence Profiling

The execution of the repair code when a ambiguous store/load pair conflicts adds run-time overhead. If the overhead due to executing the repair code is greater than benefit of reordering the pair, then an overall decrease in program performance will occur. Thus, it is important to reorder a pair of memory references only when the repair code is predicted to be infrequently invoked. The information obtained from memory-dependence profiling is

used by the compiler to decide when it is beneficial to reorder an ambiguous store/load pair.

A simple way to implement memory-dependence profiling would be to compare the address of each ambiguous store against all subsequent load addresses. However, this would require too much storage and execution time for the profiler. To solve this problem, the program is instrumented to measure conflicts only for the ambiguous store/load pairs that the scheduler would reorder if there were no dependence between them. In our current implementation, memory-dependence profiling is done as follows. First, the code is scheduled with all memory-dependence arcs removed between ambiguous store/load pairs. Second, probes are then inserted to compare the only the addresses referenced by the reordered pairs. Repair code is inserted for the reordered pairs so that the program will run correctly during profiling. Last, the program is executed. The repair code invocation count for each reordered ambiguous store/load pair is collected and mapped back into the superblock data structure.

The compiler utilizes the conflict frequency and a conflict threshold value to make store/load pair reordering decisions. Starting with the original code sequence with dependence arcs present for all ambiguous store/load pairs, code generation is done as follows. First, the dependence arc is removed between any ambiguous store/load pair meeting the conflict threshold criteria. Second, the code scheduler reorders the store/load pairs and may also move instructions dependent upon the preloads above the ambiguous stores. Third, the repair code is generated and a *check* instruction (part of the MCB support which will be described in detail later) is placed immediately after the first ambiguous store which the preload is moved above. This instruction is responsible for invoking the repair code when conflicts occur. The code scheduler is not allowed to schedule ambiguous stores and their corresponding check instructions in the same cycle. Fourth, virtual register renaming is performed to preserve all source operands used in the repair code that are overwritten by the instructions moved above the stores.

An important benefit of memory-dependence profiling is that it minimizes the negative impact of the added repair code on the instruction cache performance. Using the profile information, the invocation frequency of the correction code can be kept low, therefore reduce cache interference. Furthermore, by placing all of the repair code at the end of the function, the compiler can reduce wasted fetches of this code into the instruction cache.

3.2 Overview of Memory Conflict Buffer

This section contains an overview of the MCB design. The reader is referred to a technical report [32] for more details and the implementation considerations of the MCB. The MCB hardware supports code reordering by (1) detecting the conflicts, and (2) invoking a repair code supplied by the compiler to restore the correctness of the program execution.

The major components of the MCB hardware consist of the following: a set of address registers to store the preload addresses, compare units to match the store addresses with the preload addresses, and a conflict vector having number of bits equal to the number of general purpose registers to keep track of the occurrence of conflicts. When a preload is executed, its virtual address is saved in an address register. When a store instruction is executed, its virtual address is compared against all valid preload addresses in the address register file. If a match occurs in an address register, the bit in the conflict vector corresponding to the preloaded register is set. This signals the need to reload this register from memory and to

re-execute the instructions which depend on it.

The conflict bits are examined by a new conditional branch opcode, called a *check* instruction. When a check instruction is executed, the conflict bit specified by the instruction is examined. If the conflict bit is set, the processor branches to the repair code. The repair code re-executes the preload and the instructions which depend on it. A branch instruction at the end of the repair code brings the execution back to the instruction immediately after the check. Normal execution resumes from this point.

3.3 Experimental Results

In this section, to illustrate the usefulness of memory-dependence profiling, eight non-numeric programs (cmp, eqn, espresso, grep, qsort, wc, xlisp, and yacc) are scheduled with and without reordering of ambiguous store/load pairs. For each program, the effects of reordering the pairs with and without memory-dependence profile information is studied. The base architecture used in this study is an issue-4 processor with an instruction set similar to the MIPS R2000. No limitation is placed on the combination of instructions that can be issued in the same cycle. The processor is assumed to have register interlocking and deterministic instruction latencies (Table 2). The processor contains 64 integer registers and 32 floating point registers. Superblock optimization and scheduling with general percolation support is assumed. MCB support is provided when the ambiguous store/load pairs are reordered.

Figure 5 shows the performance comparison between an issue-4 machine with MCB support and the base architecture. The lighter bar shows the percentage performance gain over the base architecture achieved by reordering the ambiguous store/load pairs and utilizing the memory-dependence profiling information. The darker bar presents the results when no memory-dependence profiling information is used. Using the information gathered with memory-dependence profiling, scheduling with the MCB support obtains an average 10.8% performance increase over the base architecture. Without profiling, however, scheduling with the MCB support can lead to performance loss due to high invocation frequency of the repair code [32]. Overall, using memory-dependence profiling improves the performance for each program tested.

4 Conclusion and Future Directions

Profiling can be a powerful tool to assist the compiler in making optimization decisions. We have developed a system for exposing and exploiting ILP by utilizing profile information. The potential of profiling to assist compilation has been demonstrated with a series of experiments. Control-flow profile information can be used to form superblocks, assist superblock optimizations, perform global code scheduling, and aid software pipelining. Additional ILP can be obtained by using memory-dependence profiling and conflict detection hardware to reorder dependent memory references.

There are many other optimizations within the compilation process which can benefit from profile information. Examples include compiler-assisted data prefetching and data locality optimizations. The possible benefits of profiling for these optimizations are discussed below.

Some of the traditional problems associated with compiler-assisted data prefetching are increased instruction count, memory bandwidth, and data cache pollution. Due to cache mapping conflicts, it is a difficult task to determine at compile-time which accesses actually need to be prefetched and when to prefetch them. Prefetching data that is already in the cache unnecessarily adds a prefetch instruction and associated address calculation instructions to the code and wastes memory bandwidth. Prefetching data too early can displace useful data in the cache. A memory-access profiler can gather information about data reuse patterns which can be used to estimate which references actually need to be prefetched and when.

Data locality optimizations include data layout and loop distribution. Based on profile information about the frequency of different data referencing patterns within the cache, data structures can be arranged in memory so as to minimize conflicts. Loop distribution can be used to separate the accesses of data structures that can conflict in the cache. Cost functions based on the frequency of cache block replacement due to conflicting mapping of data structures and the data reuse pattern can be used to aid in deciding how to perform loop distribution.

Currently, our understanding of memory-access profiling is very limited. We are investigating the advantages of using the information gathered by memory-access profiling. Issues worth investigating are the type of information that is obtainable during program execution, how to collect it cheaply, how to map it back to the compiler, and what additional optimizations can benefit from this information. In the future, we hope to incorporate some of these ideas into the IMPACT compiler system.

References

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

[2] S. McFarling and J. Hennessy, "Reducing the cost of branches," in *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 396–403, June 1986.

[3] W. W. Hwu, T. M. Conte, and P. P. Chang, "Comparing software and hardware schemes for reducing the cost of branches," in *Proceedings of the 16th International Symposium on Computer Architecture*, pp. 224–233, May 1989.

[4] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. c-30, pp. 478–490, July 1981.

[5] J. Ellis, *Bulldog: A Compiler for VLIW Architectures*. Cambridge, MA: The MIT Press, 1985.

[6] P. P. Chang and W. W. Hwu, "Trace selection for compiling large C application programs to microcode," in *Proceedings of the 21st International Workshop on Microprogramming and Microarchitecture*, pp. 188–198, November 1988.

[7] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proceedings of the 16th International Symposium on Computer Architecture*, pp. 242–251, May 1989.

[8] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pp. 16–27, June 1990.

[9] D. W. Wall, "Global register allocation at link time," in *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, pp. 264–275, June 1986.

[10] W. W. Hwu and P. P. Chang, "Inline function expansion for compiling realistic C programs," in *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pp. 246–257, June 1989.

[11] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," *Software Practice and Experience*, vol. 21, pp. 1301–1321, December 1991.

[12] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 266–275, May 1991.

[13] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, pp. 207–218, January 1981.

[14] S. A. Mahlke, W. Y. Chen, J. C. Gyllenhaal, W. W. Hwu, P. P. Chang, and T. Kiyohara, "Compiler code transformations for superscalar-based high-performance systems," Proceeding of Supercomputing '92, Nov, 1992.

[15] T. Nakatani and K. Ebcioglu, "Combining as a compilation technique for VLIW architectures," in *Proceedings of the 22nd International Workshop on Microprogramming and Microarchitecture*, pp. 43–55, September 1989.

[16] D. J. Kuck, *The Structure of Computers and Computations*. New York, NY: John Wiley and Sons, 1978.

[17] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," in *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 180–192, April 1987.

[18] G. Kane, *MIPS R2000 RISC Architecture*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

[19] A. Aiken and A. Nicolau, "Optimal loop parallelization," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 308–317, June 1988.

[20] K. Ebcioglu, "A compilation technique for software pipelining of loops with conditional jumps," in *Micro 20*, pp. 69–79, December 1987.

[21] M. S. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 318–328, June 1988.

[22] B. Su and J. Wang, "Gurpr*: A new global software pipelining algorithm," in *Micro 24*, pp. 212–216, November 1991.

[23] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, pp. 183–198, October 1981.

[24] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pp. 177–189, January 1983.

[25] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, pp. 12–35, January 1989.

[26] R. Towle, *Control and Data Dependence for Program Transformations*. PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1976.

[27] N. J. Warter, D. M. Lavery, and W. W. Hwu, "Using profile information to assist modulo scheduling," tech. rep., Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, May 1992.

[28] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orzag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin, "The PERFECT club benchmarks: Effective performance evaluation of supercomputers," Tech. Rep. CSRD-827, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, May 1989.

[29] Intel, *i860 64-Bit Microprocessor*. Santa Clara, CA, 1989.

[30] N. J. Warter, D. M. Lavery, and W. W. Hwu, "The benefit of Predicated Execution for software pipelining," in *Proceedings of the 23rd Hawaii International Conference on System Sci ences*, to appear January 1993.

[31] A. Nicolau, "Run-time disambiguation: coping with statically unpredictable dependencies," *IEEE Transactions on Computers*, vol. 38, pp. 663–678, May 1989.

[32] W. Y. Chen, S. A. Mahlke, W. W. Hwu, and T. Kiyohara, "Assisting compile-time code reordering with the memory conflict buffer," tech. rep., Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, May 1992.