Profile-Assisted Instruction Scheduling

William Y. Chen Scott A. Mahlke Nancy J. Warter Sadun Anik Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois, Urbana-Champaign, IL

Abstract

Instruction schedulers for superscalar and VLIW processors must expose sufficient instruction-level parallelism to the hardware in order to achieve high performance. Traditional compiler instruction scheduling techniques typically take into account the constraints imposed by all execution scenarios in the program. However, there are additional opportunities to increase instruction-level parallelism for the frequent execution scenarios at the expense of the less frequent ones. Profile information identifies these important execution scenarios in a program. In this paper, two major categories of profile information are studied: control-flow and memory-dependence. Profile-assisted code scheduling techniques have been incorporated into the IMPACT-I compiler. These techniques are acyclic global scheduling and software pipelining. This paper describes the scheduling algorithms, highlights the modifications required to use profile information, and explains the hardware and compiler support for dealing with hazards that arise from aggressive use of profile information. The effectiveness of these profile-based scheduling techniques is evaluated for a range of superscalar and VLIW processors.

1 Introduction

Instruction scheduling typically takes into account the constraints imposed by all possible execution scenarios of the program. The constraints are determined using static analysis methods, such as live-variable analysis, reaching definitions, loop detection, and data-dependence analysis [1]. These static analysis methods do not distinguish between frequent and infrequent execution scenarios. Although this approach ensures that instructions are scheduled correctly, it may prevent the compiler from achieving a better schedule for frequent scenarios because of the constraints from infrequent scenarios.

There are two run-time variables that determine execution behavior: conditional branches and memory accesses. If these variables can be predicted, the compiler can determine the frequent scenarios and aggressively schedule them. For instance, if the compiler can predict which path of a conditional branch is usually taken, it can aggressively schedule along that path ignoring constraints from the other path. Likewise, if it can predict that two memory accesses usually do not conflict, it can assume that they do not conflict and

move one access above the other.

In order to aggressively schedule the frequently executed scenarios, the infrequently executed scenarios could be penalized. For example, an instruction may be moved above a branch from the more frequently executed path. Thus, the infrequently executed scenario will execute an instruction unnecessarily. Because one path is optimized at the cost of another, inaccurate predictions may actually *reduce* the programs performance. While the compiler can use control flow and memory dependence analysis to estimate some run-time behavior, these techniques currently cannot be used to accurately estimate which execution scenarios to aggressively schedule. Profiling is a technique for instrumenting programs in order to collect run-time information. There are two types of profiling that can be used to predict execution scenarios: control-flow and memory-dependence. Control-flow profiling collects information about the relative frequency of execution paths. Memory-dependence profiling summarizes the frequency of address conflicts between two memory references.

In this paper we discuss how to collect profile information and how to modify instruction scheduling to use control-flow and memory-dependence profiling. In addition to explaining the usefulness of each form of profiling, we address the scheduling hazards due to mispredicting branches and memory dependences and discuss software and hardware solutions for handling such hazards. To demonstrate the effectiveness of control-flow and memory-dependence profiling for instruction scheduling, we have modified the acyclic global code scheduler and the software pipeline scheduler in the IMPACT-I compiler to take advantage of profile information. These profile-assisted scheduling techniques and their relative performance benefits are presented. We conclude with a discussion of additional uses of profiling information for assisting other compiler transformations.

2 Profile Collection

Profile information is collected through probes inserted within the program. Figure 1 shows the phases of profile collection in the compilation process. Probes are inserted into the program intermediate representa-

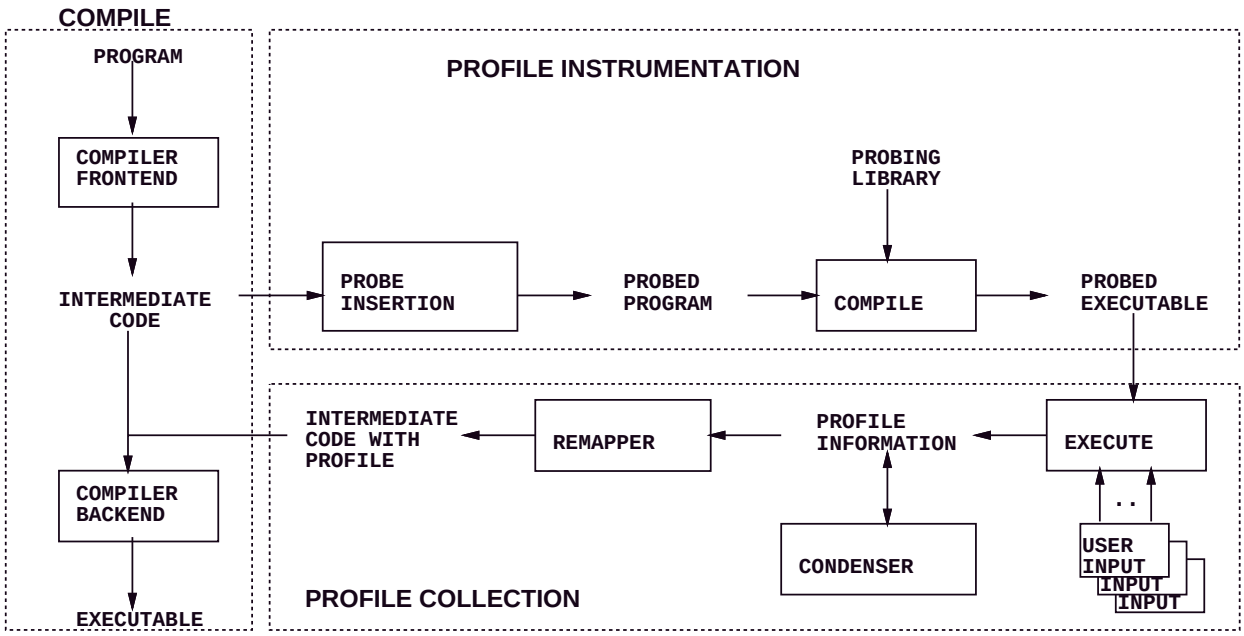

Figure 1: Profile collection phases.

tion. Linking the probed program with the probing library forms the probed executable which when executed outputs the profile information. The profile information is collected for user specified inputs through multiple runs of the probed executable. Finally, the compiler takes the profile information collected and uses it to perform aggressive instruction scheduling.

In this section, we describe the instrumentation for control-flow and memory-dependence profiling. For control-flow profiling, the instrumentation provides the compiler with the relative frequency of alternative execution paths. For memory dependence profiling, the instrumentation summarizes the frequency of address matching between two memory references.

2.1 Control-Flow Profiling

For the purpose of instruction scheduling, control-flow profiling provides the following information averaged over all the profile runs: the execution frequency of each basic block and the branch taken frequency of two-way and multi-way branches. These are maintained in a structure known as a *weighted flow graph.* A weighted flow graph is a quadruplet $\{V, E, count, arc\_count\}$, where each node in V is a basic block, each

arc in E is a 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.

Control-flow profiling is setup automatically by the compiler. Using the weighted flow graph of the program, the code generator annotates the program to keep track of the execution frequency of each basic block and of the condition of each branch. For different runs of the same program, a condenser summarizes and averages the profile information. Lastly, the condensed profile information is mapped back into the weighted flow graph by a remapper.

The following steps are taken by the code generator to produce a probed executable for profile collection. A unique function id (*fnid*) is set upon entry into the function, and resets after each function call. A unique basic block id (*bbid*) is also assigned to each basic block within a function. Within each basic block, a call to the probing function is inserted. The probing function also identifies between conditional branches, unconditional branches, and register jumps by the branch type (*br*). In the case of a conditional branch, the branch condition (*cond*) is passed into the probing function. For register jumps, the value switched on (*cond*) is used by the probing function. When the basic block is executed, the probing function uses the unique *fnid-bbid* pair to increment the basic block execution frequency and the unique *fnid-br* pair along with *cond* to determine the branching frequency. The output of the code generator is then linked in with the probing function to generate a probed executable of the program.

An example of probe insertion is shown in Figure 2. The C source and corresponding assembly code are given in Figure 2a. Figure 2b shows the instrumented assembly code. When entering the function, *fnid* is set to a unique number, 2 in this case. Then for each basic block within the function, a *bbid* is assigned at the beginning. Depending on the type of the branch instruction, the appropriate *cond* and *br* values are set. Finally, a call to the probing function (the probing function is presented in Figure 3) is inserted at the end of the basic block before the branch. When the probing function is invoked, the data structure for keeping profile weights is indexed by *fnid* and *bbid* to obtain the correct entry. The basic block counter, *bb_count*, is incremented by one to take into account that the flow of control has entered this basic block. If the basic

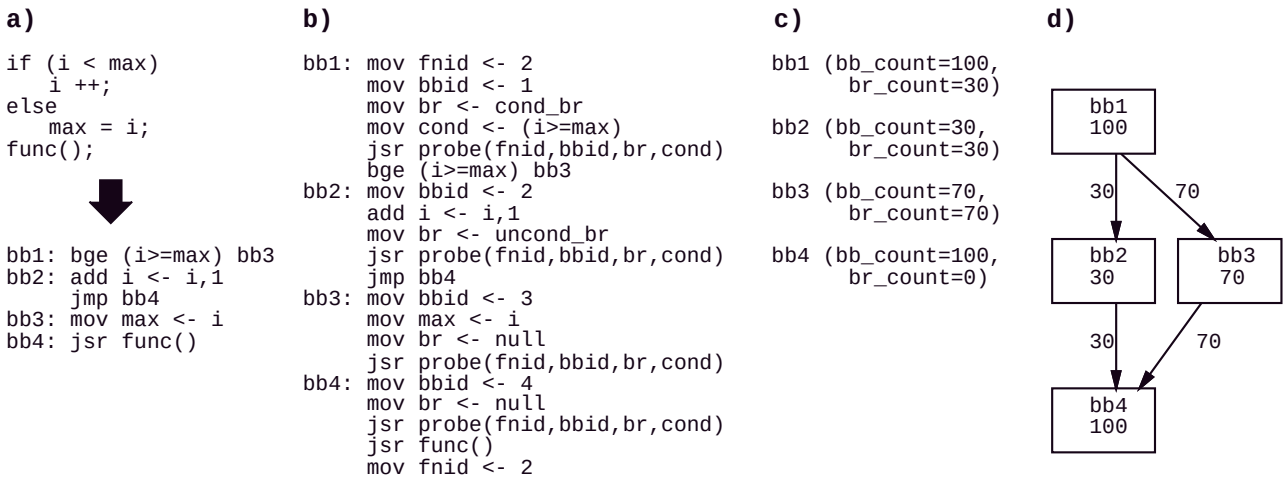

Figure 2: Insertion of probes for control-flow profiling. a) The original source and assembly code. b) Assembly code segment with probes inserted. c) Profile output indicating basic block execution frequency and branch taken frequency for all basic blocks. d) The weighted flow graph with the profile information.

block ends in a branch instruction, *br* is used to update the branch taken count, *br_count*. In the case of a conditional branch, *br_count* is updated only when the branch is taken (*cond* equals to 1).

The user is required to execute and supply the instrumented program with the appropriate input files. The probed program is executed once for each input. Each execution produces a summary of the profile result for that particular run. Figure 2c shows a possible profile statistic. Individual profile results are combined into a single average file which can be mapped into the original intermediate code using the unique *fnid* and *bbid*. The final weighted flow graph is shown in Figure 2d. The number shown within each box is the basic block execution weight, and the number on the arc is the number of times that particular path of control has been taken.

2.2 Memory-Dependence Profiling

The basic concept behind memory-dependence profiling is to collect the frequency of address matches between pairs of loads and stores. Memory dependence analysis is used to answer the question of whether two references are independent of one another. In the case of a definite dependence, the addresses between the reference pair always match. In the case of a definite independence, the addresses between the reference pair

```
Probe(fnid,bbid,br,cond) {
   increment bb_count indexed by fnid and bbid
   if (br is a conditional branch)
     if (branch is taken (cond==1))
       increment br_count indexed by fnid and bbid
   if (br is a unconditional branch)
       increment br_count indexed by fnid and bbid
   if (br is a register jump)
       increment HASH(fnid,bbid,cond)
 }
```

Figure 3: Probing function.

never match. For memory dependence profiling, the cases for definite dependence and definite independence are disregarded since their address match frequency is known. Memory dependence profiling concentrates on the *ambiguous* reference pairs for which memory dependence analysis cannot provide a conclusive results.

Because memory-dependence profiling is much more expensive than control-flow profiling in terms of data storage and execution overhead, memory-dependence profiling is performed for only the most important execution paths identified by control-flow profiling. Based on the control-flow profile information, the program is divided into several regions. Instruction scheduling is considered only within each region. The compiler first marks those ambiguous store/load pairs which an aggressive instruction scheduler would reorder to improve performance. [1] Only the conflict status of these relevant store/load pairs are recorded during dependence profiling. After dependence profiling, the compiler utilizes the conflict frequency and a conflict threshold value to make the final instruction scheduling decisions.

The subroutine *check_address*, shown in Figure 4, is created to assist dependence profiling. An example of its use is shown in Figure 5. The inputs are the load id (*lid*), store id (*sid*), load address (*laddr*), and store address (*saddr*). A table containing the mapping between the *lid/sid* and the load/store type is statically constructed before the memory-dependence profiling phase. This mapping table is read into the database by the probed executable during profiling. The probing function uses the *lid* and the *sid* to find the access

[1] In our implementation, a pseudo instruction scheduling phase is used to determine the benefit of reordering ambiguous store/load pairs. To accomplish this, all memory dependences between ambiguous store/load pairs are removed during the pseudo instruction scheduling phase.

```
check_address(lid,sid,laddr,saddr) {
   if (laddr == saddr)
      increment conflict counter indexed by lid and sid
      return
   if (laddr < saddr)
      ld_access_size = access size of load(lid)
      if ((laddr + ld_access_size) > saddr)
         increment conflict counter indexed by lid and sid
         return
   if (saddr < laddr)
      st_access_size = access size of store(sid)
      if ((saddr + st_access_size) > laddr)
         increment conflict counter indexed by lid and sid
         return
   return
}
```

Figure 4: Check address function.

types of the memory instructions. It is then used to determine whether the two memory accesses overlap and to keep the access conflict statistics.

During probe insertion, for each of the ambiguous store/load pairs that the compiler determined to be beneficial to reorder, a *check_address* subroutine call is placed in the code sequence right after the load (Figure 5c). The address for the store is calculated before the store instruction and preserved up to the point of *check_address*. The load id and store id along with the memory addresses for the pair are passed into the probing function. The computed store address is immediately reset afterward to prevent false matches later on.

When the probed program is executed, the probing function keeps count of then address overlaps for all the store/load pairs of interest. A possible profile output is shown in Figure 5d. Similar to control-flow profiling, results from different runs are averaged. However, the profile result is not remapped into the intermediate code. The profile information is instead read into a separate data structure to facilitate code reordering during instruction scheduling.

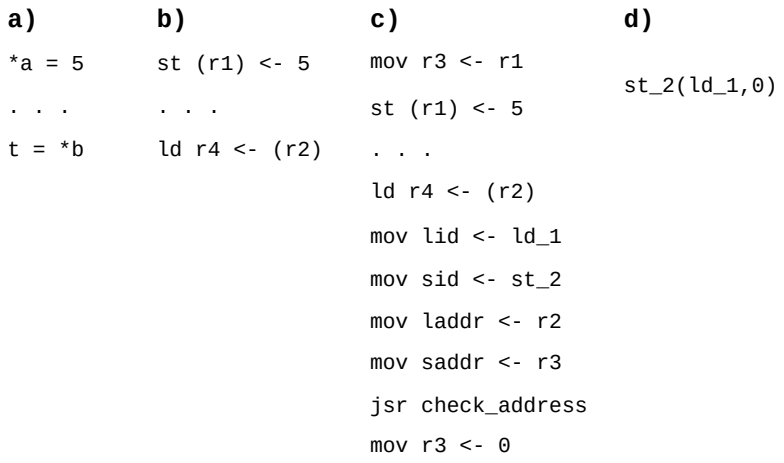

Figure 5: Example insertion of probes for memory-dependence profiling. a) The original source code. b) Corresponding assembly code. c) Assembly code with probes inserted. d) Profile output indicating address overlap frequency for the store/load pair. In this case, a zero means the addresses did not overlap for the given user input.

3 Scheduling with Control-flow Profile Information

Control flow profile information can be used to assist two basic components of global instruction scheduling techniques. The first component is inter-basic block code motion. Inter-basic block code motion is used to overlap and reorder instructions among groups of connected basic blocks. By examining large groups of basic blocks, global instruction schedulers increase the opportunity for achieving a compact schedule. The second component is loop iteration overlap. Loop iteration overlap refers to overlaying the execution of consecutive iterations of a loop during global scheduling. Overlapping loop iterations provides global schedulers with an additional dimension (across iterations) to schedule.

3.1 Inter-basic block code motion

Control flow profile information can be used to effectively guide code motion among basic blocks. The possible types of code motion associated with instruction scheduling are illustrated in Figure 6. Code motion is broken down based on the direction of the motion, upward or downward, and the nature of the control flow graph, split or join point. A particular movement of an instruction, A, consists of removing A from its source basic block (src) and inserting a copy of A into one or more destination basic blocks (dst).

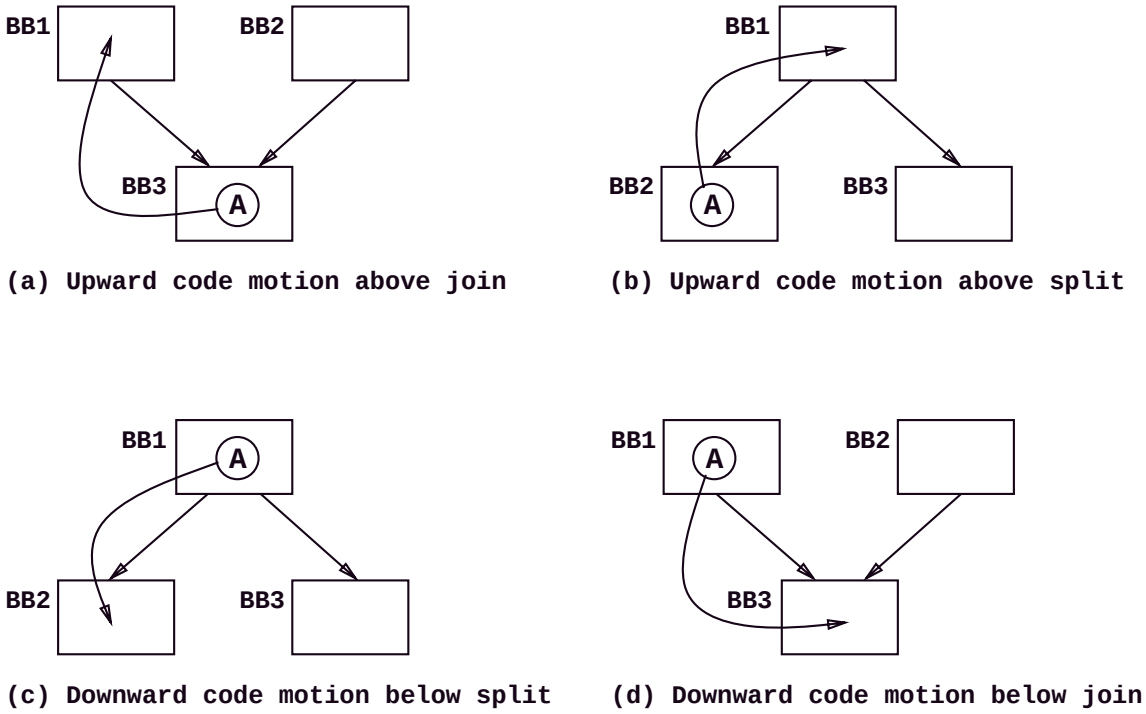

Figure 6: Inter-basic block code motion.

For example, the code motion illustrated in Figure 6a has a src of BB3 and one dst, BB1.

Guiding code motion with control-flow profile information. The scheduling objective of moving instructions among basic blocks is to achieve a tighter schedule and thereby reduce the execution time of the program. This objective can be expressed with the following two equations.

$$\triangle_{bb} = cycles_{after} - cycles_{before} \tag{1}$$

$$net\_cycles = \triangle_{src} + \sum_{j} \triangle_{dstj} \tag{2}$$

The performance effect that a particular code motion has on a basic block is the difference between the number of cycles to execute the basic block before and after scheduling (Equation 1). The overall effect on performance is then computed by summing the individual effects of all basic blocks impacted by the code motion (Equation 2). Many global schedulers utilize a sequence of code motions to achieve performance

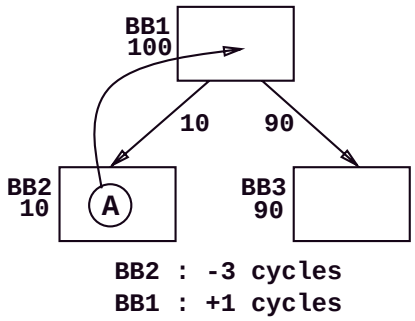

Figure 7: Example of an undesirable upward code motion.

improvements. Equation 2 can be generalized to reflect a sequence of i code motions as shown below.

$$net\_cycles = \sum_{i}(\triangle_{srci} + \sum_{j} \triangle_{dstij}) \tag{3}$$

The problem with this approach is that undesirable scheduling decisions are likely to be made since the relative execution frequencies of src and dst are not known. For example, consider the code motion in Figure 7. In this example, the scheduler attempts to achieve a more compact schedule by moving instruction A from BB2 to BB1. As a result, the execution time of BB2 is reduced by 3 cycles while increasing the execution time of BB1 by 1 cycle. Using Equation 3, an overall reduction in net execution time is predicted. However, assuming the program execution is as specified by the basic block and arc weights in Figure 7, a net increase of 70 cycles is observed by performing the code motion.

Control-flow profile information can be used to greatly improve code motion heuristics. Control-flow profile information identifies to the scheduler likely preceding and succeeding basic blocks. Code motion is then favored among these blocks, since a likely performance gain is expected. Also, code motions which increase the performance of a frequently executed basic block while reducing an infrequently executed basic block may be avoided. Equation 1 augmented with basic block weights obtained from control-flow profiling is given below.

$$\triangle_{bb} = (cycles_{after} - cycles_{before}) \times weight_{bb} \tag{4}$$

Using Equation 4 in the previous example (Figure 7), the scheduler can easily predict a performance loss by moving instruction A into BB1, and therefore would choose not to perform this code motion.

The four types of code motion illustrated in Figure 6 cannot be freely performed by an instruction scheduler. Each requires some additional compiler [2][3] and/or architectural support [4][5][6][7] to handle hazards associated with the code movement. The hazards associated with each form of code motion and some available solutions are briefly discussed in the remainder of this section.

Hazards with upward code motion above a join. Upward code motion above a join point introduces execution paths which no longer execute the moved instruction. In Figure 6a, moving instruction A from BB3 to BB1 removes A from the BB2-BB3 execution path. Therefore, to correctly update execution along all paths of execution, the moved instruction is copied to all incoming basic blocks of a join. In this manner, the instruction is executed along all paths of execution into the join both before and after code motion.

Hazards with upward code motion above a split. Upward code motion above a control split point introduces two difficult problems. These problems arise because code motion above a split involves executing an instruction before it is certain it should have executed. This is known as *speculative execution.* All side effects of the instruction must be properly handled when the instruction is unnecessarily executed.

The first hazard is that a moved instruction may destroy a value used along an alternative path of execution. For example in Figure 6b, instruction A may overwrite a value which is live (referenced before redefined) in BB3. This hazard may be overcome with either compile-time renaming or hardware renaming. With compile-time renaming, the scheduler modifies the destination of the speculative instruction to a new variable, and substitutes any uses of the destination with the new variable. With hardware renaming, alternative destination registers (shadow registers) are provided to buffer the results of speculative instructions [4]. In both cases the hazard is avoided by providing the speculative instruction with a new destination, so the contents of the original destination are not destroyed.

The second hazard is that a speculative instruction may cause an exception which should not have occurred in the original program. For example in Figure 6b, instruction A may be a load instruction and the conditional branch at the end of BB1 tests whether the address of the load is zero or not. The speculative load instruction is executed regardless of the branch condition, therefore, whenever the branch results in a jump to BB3, the load will produce an illegal address signal. However, for these cases, the load was not

supposed to be executed, so any exceptions raised by A should be ignored. Improper exception hazards may be handled with a variety of methods which vary in complexity and effectiveness.

The simplest solution to overcome signalling improper exceptions is to limit candidates for speculative execution to those instructions which may never result in an exception. A second solution is to expand a processor's instruction set to include non-excepting or silent versions for all instructions which normally except [8] [5] [7]. When the scheduler generates speculative instructions, it converts the opcode of the speculative instruction to its silent counterpart. With additional compiler and architectural support, exceptions for speculative instructions may be delayed until it is confirmed the speculative instruction should have executed [4] [6]. In this manner, no exceptions in the original program are hidden with speculative execution.

Hazards with downward code motion below a split. Downward code motion below a split introduces execution paths which no longer execute the moved instruction. This is the same hazard which code motion above a join causes. In Figure 6c, moving instruction A from BB1 to BB2 removes A from the BB1-BB3 execution path. To correctly update all execution paths, the moved instruction is copied to all outgoing basic blocks of a split.

Hazards with downward code motion below a join. Downward code motion below a join introduces another difficult problem. Again, the problem arises because the moved instruction is executed along more paths of control than it was before scheduling. In Figure 6d, instruction A is originally executed when BB1 is entered. However, after code motion, A is executed when either BB1 or BB2 are entered. The side effects of the moved instruction must be properly handled when the instruction is unnecessarily executed. The methods of dealing with downward code motion below a join are completely different from those for dealing with upward code motion above a split.

To eliminate the unnecessary execution of the moved instruction, the scheduler applies code restructuring [2]. The restructuring involved introduces a new basic block for each block which enters the join. The join point is then adjusted to after the moved instruction, and instructions prior to the moved instruction are copied into each of the new basic blocks. In this manner, the moved instruction executes on the same control

paths before and after code motion. Alternatively, the hazard may be overcome with architectural support for predicated execution [9]. With predicated execution, instructions are executed based on the value of a boolean input flag. When the boolean value is true, the instruction is executed normally. When the boolean value is false, the instruction is converted into a no_op instruction. Therefore, the scheduler may associate a predicate with each moved instruction and place an assertion to that predicate in the original block of the instruction if one does not exist. Again, the moved instruction executes on the same control paths before and after code motion.

3.2 Overlapping iterations

Another form of code motion is to move instructions across loop iteration boundaries and thus, overlap the execution of different iterations. Originally, the steady-state of a loop's execution is determined by the critical path through one iteration. The steady-state can be reduced if delays along the critical path are hidden by executing instructions from different iterations. Software pipelining is a systematic approach for overlapping loop bodies to reduce steady-state execution time.

One drawback to some software pipelining techniques is that the length of the steady-state schedule is fixed for all paths through the loop [10][11][12][13]. This can particularly limit the performance if the most frequently executed path is much shorter than other paths through the loop. Control-flow profiling information can be used to determine which paths to include in the software pipelined loop body. Figure 8a shows the weighted control flow graph of a simple loop. The loop body consists of four instructions, where instruction A is a conditional branch.[2] Since instruction C is only executed 10% of the time, only the path $\{A,\ B,\ D\}$ is software pipelined.

When control paths are excluded from the software pipeline, a mispredicted branch can break the software pipeline. The simplest approach to handling this hazard is to empty the software pipeline, execute the code along the taken path, and refill the pipeline. Figure 8b shows the software pipelined loop using this hazard resolution technique. This is a software hazard resolution technique since the compiler generates the necessary

[2] To keep the example simple, the loop back branch and its handling are ignored.

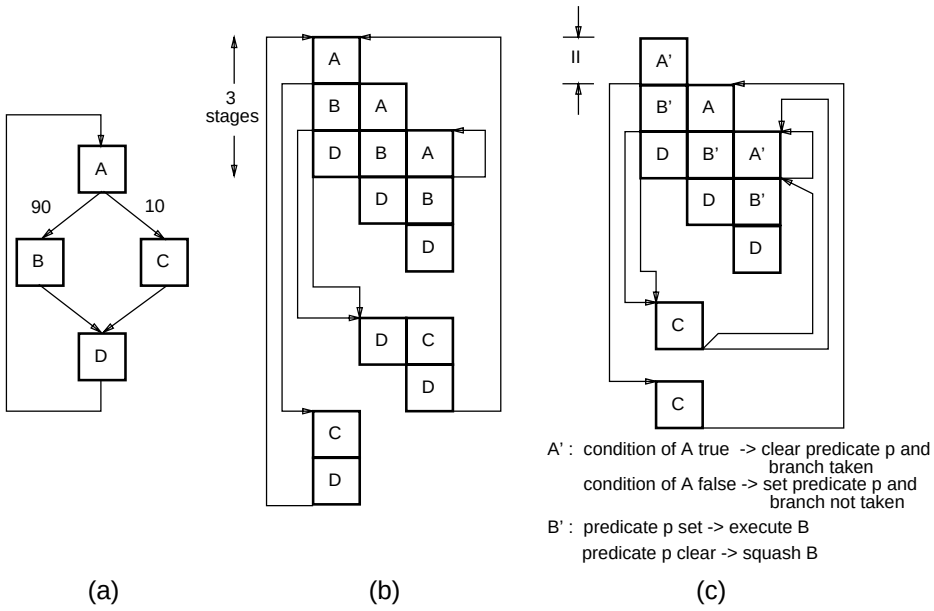

Figure 8: Using control-flow profile for software pipelining. a) Weighted control-flow graph. b) Software hazard resolution. c) Hardware hazard resolution.

code to empty the pipeline and to execute the code along the taken path. Note that this example is overly simplified to illustrate the order in which instructions are executed when a branch misprediction hazard occurs. Thus, explicit branch instructions other than A are not shown, but, their corresponding control flow arcs are shown.[3]

In this simple example, there are only two stages in the software pipeline until the steady-state execution is reached. Typically, the number of stages is higher (in [15] the average number of stages is five). Thus, it can be costly to recover from a mispredicted branch using a software resolution technique.

It would be ideal if the execution could jump out of the pipeline, execute the taken path code, and jump back into the pipeline. In order to do so, the instructions in the pipeline that are along the not taken path of the branch need to be squashed. Squashing implies that the instructions are fetched but not executed. Predicated hardware support can be used to squash instructions in the software pipeline [9][12]. Figure 8c shows the software pipeline schedule assuming predicated hardware support. In this example, A' is assumed to be a conditional branch instruction that also asserts a predicate p. Instruction B' executes

[3] When the loop back branch is considered, code is also required to handle early exits from the software pipeline [14].

B if predicate p is asserted, otherwise B is squashed. When the condition of A is false, the predicate p is asserted and instruction B executes. When the condition of A is true, the predicate p is not asserted and control branches to execute instruction C. After C executes, control branches back to the instruction following the mispredicted branch. Since the predicate p is not asserted, instruction B will not execute but other instructions scheduled with B' will. An algorithm for generating the software pipeline using hardware hazard resolution is presented in Section 5.2.

4 Scheduling with Memory-dependence Profile Information

The freedom of global instruction scheduling is limited by memory dependences. 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. Due to the practical limitations of current memory dependence analysis techniques, 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.

Memory-dependence profiling can be used 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 reference pair is reordered as if there were no dependence and conflict detection and repair code are added to maintain correct program execution. An important benefit of memory-dependence profiling is that it minimizes the negative impact of the added repair code. Using the profile information, the invocation frequency of the correction code is kept low, therefore reduces the repair cost.

Repair code is invoked when reordered load and store addresses overlap. Detection of overlapping addresses can be performed through software [16] or through hardware support. An example of software detection is to insert explicit address comparison operations within the instruction stream to detect when

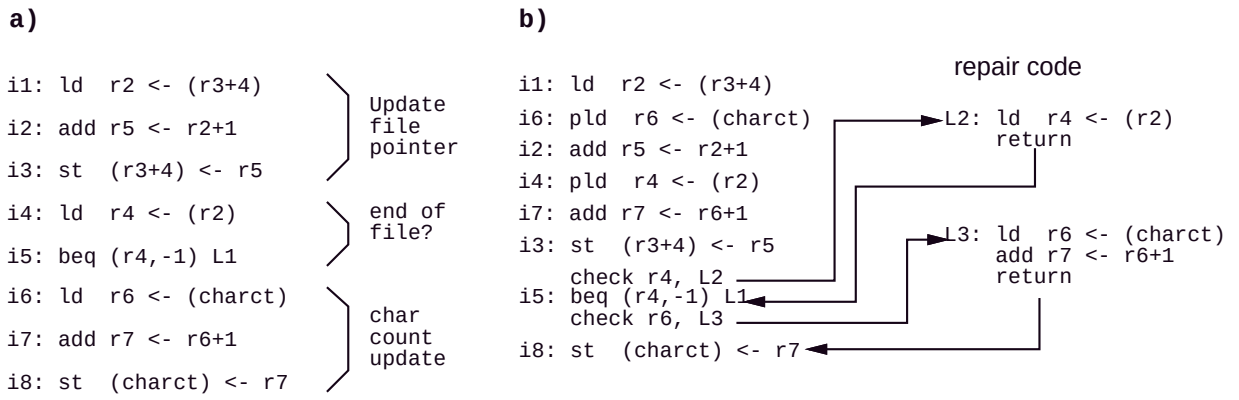

Figure 9: An example code segment taken from UNIX utility, *wc*, using MCB. a) Original code segment. b) Code re-ordering using MCB support with repair code.

a load and store address are the same. Software comparison of addresses assumes that only memory instructions of the same type can be reordered. An advantage of the software detection is that the compare instruction can be freely scheduled into an available resource slot. The disadvantage of the software detection is that extra resource slot is needed for the compare instruction, and that multiple compare instructions are necessary if a load is scheduled above multiple ambiguous stores.

An architectural support, referred to as the *memory conflict buffer* (MCB), eliminates the comparison overhead of the software method. The major components of the MCB are a set of address registers to store the addresses of the loads which have bypassed ambiguous stores (preloads), compare units to match the store addresses with the saved load addresses, and a number of status bits to keep track of the occurrence of address overlap. If an address overlap occurs, a predetermined status bit is set. This signals the need to invoke the repair code to re-execute the load instruction and the instructions which depend on it.

The status bit is examined by a new conditional branch opcode, called a *check* instruction. When a check instruction is executed, the status bit specified by the instruction is examined. If the status bit is set, the processor branches to the repair code. 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.

The use of MCB for store/load movement is shown in Figure 9. In the original code segment, *i5* and *i6* cannot be disambiguated with *i4*. Using the MCB, the resulting code segment is shown in Figure 9b. The

check instructions are placed in the instruction sequence to invoke the repair code when the load and store addresses do match.

5 Uses of Profiling in IMPACT

Profiling is used extensively in the IMPACT-I compiler. The IMPACT-I compiler is a prototype optimizing compiler designed to generate efficient code for superscalar and VLIW processors. In this section we present an acyclic scheduling technique, *superblock scheduling* that uses control-flow and memory-dependence profiling. We also present a cyclic scheduling technique, *modulo scheduling* that uses control-flow profiling. While memory-dependence profiling can be used for cyclic scheduling, it has not been incorporated into the modulo scheduling technique yet.

5.1 Acyclic scheduling

Profile-based acyclic scheduling in the IMPACT-I compiler is based on an efficient structure referred to as the *superblock* [17]. 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. Control-flow profile information is utilized directly to form superblocks.

Superblock formation consists of two major steps. First, traces are identified within the function. A trace is a contiguous set of basic blocks which are likely to execute in sequence [2]. Traces are selected by identifying a seed basic block and growing the trace forward (backward) to likely preceding (succeeding) basic blocks until either there is no likely predecessor (successor) or until the likely predecessor (successor) has already been placed into a trace [18]. Each basic block is a member of exactly one trace. An example of trace selection applied to a group of basic blocks is shown in Figure 10a. In this example, three traces are identified in the code segment consisting of the following basic blocks: (1,2,5,6), (3), and (4).

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

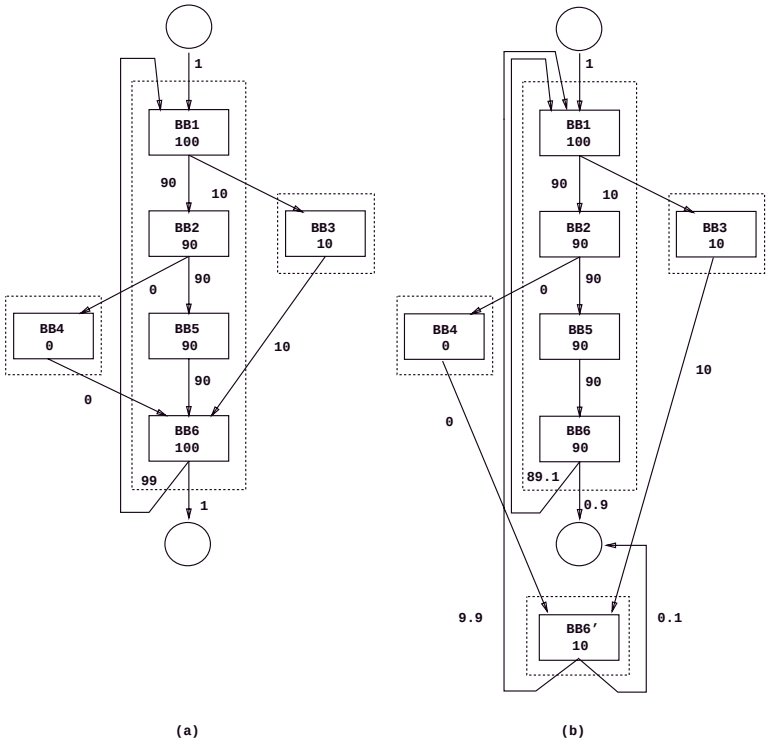

Figure 10: An example of superblock formation, (a) trace selection, (b) tail duplication.

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 10b. The trace consisting of basic blocks (1,2,5,6) contains two control entry points to basic block 6. Tail duplication replicates basic block 6 (basic block 6′) and adjusts the control transfers from basic blocks 3 and 4 to basic block 6′. After removing all side entrances, trace (1,2,5,6) becomes a superblock.

With superblock scheduling, control-flow profile information is utilized to identify likely execution paths in the code. List scheduling is then applied to the resultant superblocks. By scheduling the superblock as a single unit, a more compact schedule can be achieved along the execution path included in the superblock. Since superblock scheduling can result in both upward and downward code motion, hazards associated with each type of code motion must be handled. Upward code motion hazards are handled with compile time

renaming and non-excepting opcodes for all excepting opcodes in the instruction set. Downward code motion hazards are handled with compile-time code restructuring. The reader is referred to Section 3.1 for more details regarding handling hazards for inter-basic block code motion.

Memory-dependence profile information is utilized in superblock scheduling to achieve a more compact schedule for each superblock. With conflict information available for store/load pairs, dependences among store/load pairs in the same superblock with infrequent conflicts are removed. Therefore, the superblock scheduler may freely re-order these memory accesses to obtain a tighter schedule. Hazards associated with re-ordering store/load pairs are handled using the MCB architectural support along with the compiler generated conflict correction code is discussed in Section 4.

The superblock scheduling algorithm used in the IMPACT-I compiler is shown in Figures 11 – 13. The algorithm is broken down into 3 parts. An initialization step (Figure 12) is first performed. Initialization includes dependence graph construction in which the appropriate store/load dependences are omitted based on the memory-dependence profile information. Also, a check instruction is inserted into the superblock for each load which may possibly be converted into a preload. The second part of the algorithm (Figure 11) performs list scheduling and deals with scheduling hazards as they occur. Check instructions are removed during scheduling when their corresponding load was not converted into a preload. Note that when dealing with downward code motion hazards, a test is performed to ensure that the the superblock to which a new instruction must be inserted has not been scheduled. This test is necessary for VLIW processors with strict timing requirements; however, it is not necessary for superscalar processors. The final part of the algorithm (Figure 13) generates the necessary conflict correction code for all preloads. Correcting a conflict involves re-executing the preload instruction and all subsequent instructions before the check which directly or indirectly use the result of the preload.

Experimental methodology. To study the effectiveness of 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 representative control-flow and memory-dependence profile information. An input different from those used for profiling is

```
schedule(superblock) {
    schedule_initialization(superblock)
    issue_time = 0
    while (unscheduled set of instructions is not empty) {
        issue_time += 1
        active_set = set of unscheduled instructions that are ready
        sort active set according to instruction priority
        for each instruction in active_set, I {
            if (I cannot be scheduled at issue_time)
                continue
            /* checks for loads not scheduled as preloads are unnecessary */
            if ((I is a check) and (corresponding load not scheduled as a preload)) {
                delete I from set of unscheduled instructions
                continue
            }
            mark required resources of I busy in resource_usage_map
            delete I from set of unscheduled instructions
            I→issue_time = issue_time
            /* check for control-flow hazards associated with an upward code motion */
            br_moved_above = I→prev_br - {set of scheduled branches}
            if (br_moved_above not empty) {
                mark I as speculative
                for each branch in br_moved_above, BR {
                    if (dest(I) live when BR is taken) {
                        old_dest(I) = dest(I)
                        rename dest(I)
                        substitute use of old_dest(I) in superblock with dest(I)
                        if (old_dest(I) live along any unscheduled BR in superblock)
                            insert a new copy instruction J (old_dest(I) = dest(I)) into set of unscheduled instructions
                        break
                    }
                }
            }
            /* check for control-flow hazards associated with an downward code motion */
            br_move_below = I→post_br - {set of unscheduled branches}
            for each branch in br_move_below, BR {
                if (dest(I) not live when BR is taken)
                    continue
                if ((target superblock of BR has single predecessor) or (target superblock of BR is already scheduled))
                    insert a copy of I into target superblock of BR
                else {
                    create a new superblock, new_sb
                    insert jump instruction to target of BR into new_sb
                    modify target of BR to new_sb
                    insert a copy of I into new_sb
                }
            }
            /* check for memory-dependence hazards associated with an upward code motion */
            st_move_above = I→prev_st - {set of scheduled stores}
            if (st_move_above not empty)
                mark I as a preload
        }
    }
    insert_conflict_correction_code(superblock)
}
```

Figure 11: Superblock scheduling algorithm.

```
schedule_initialization(superblock) {
    construct dependence_graph, G
    for each memory flow dependence in G, dep {
        if (conflict_frequency(dep) < threshold)
            remove dep from G
    }
    clear resource_usage_map
    for each instruction in superblock, I {
        I→priority = computed priority of I
        I→prev_br = all branches lexically before I in superblock
        I→post_br = all branches lexically after I in superblock
        if (I is a load) {
            I→prev_st = all ambiguous stores lexically before I in superblock
            I→post_st = all ambigous stores lexically after I in superblock
            if (I→prev_st not empty) {
                insert a check instruction for I immediately after I
                add a memory flow dependence from all elements of I→prev_st to the check
                add a memory anti dependence from the check to all elements of I→post_st
                add a control dependence from all elements of I→prev_br to the check
            }
        }
        add I to set of unscheduled instructions
    }
}
```

Figure 12: Initialization algorithm for superblock scheduling.

```
insert_conflict_correction_code(superblock) {
    for each preload instruction in superblock, I {
        flow_dep = list of instructions flow dependent on I scheduled prior to the check of I
        create new superblock, new_sb
        insert a copy of I into new_sb, convert I to normal load
        insert a copy of all instructions in flow_dep into new_sb
        insert a jump instruction whose target is the instruction following the check of I into new_sb
        modify target of check of I to new_sb
    }
}
```

Figure 13: Correction code algorithm routine for superblock scheduling.

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
qsort	136	quick sort
tbl	2817	format tables for troff
wc	120	word count
xlisp	7747	lisp interpreter
yacc	2303	parser generator

Table 1: Non-numeric benchmarks.

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.

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 [19]. The processor is assumed to have in-order execution with register interlocking and deterministic instruction latencies (Table 2). The processor contains 64 integer registers and 32 floating point registers. Non-excepting versions of all instructions excluding store instructions and architectural support for the memory conflict buffer [20] are further assumed in the processor.

Experimental results. The performance of superblock scheduling is compared for 4-issue and 8-issue superscalar processors. The issue width is the maximum number of instructions the processor can fetch and issue per cycle. No limitation is placed on the combination of instructions that may be issued in the same cycle. For each 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 configuration. The base configuration used is

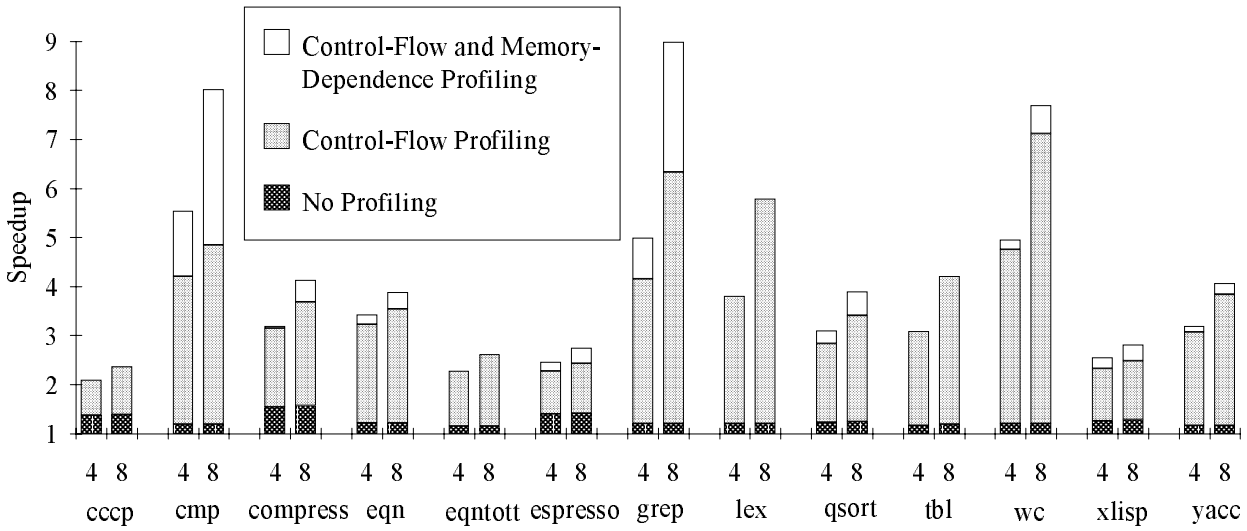

Figure 14: Performance comparison of acyclic scheduling with varying levels of profile information.

an 1-issue processor with conventional basic block code scheduling.

The performance improvement observed with control-flow and memory-dependence profile information available to the scheduler is shown in Figure 14.[4] Basic block scheduling provides little speedup for both the 4-issue and 8-issue processor. With control-flow profile information available, large speedup improvements result for all benchmarks. The superblock structure provides the scheduler with many more opportunities to extract instruction-level parallelism. Furthermore, by directly exposing important execution paths to the scheduler with superblocks, intelligent inter-basic block code motion decisions are made.

With memory-dependence profile information also available to the scheduler, additional performance improvement is observed. The largest improvements were observed for *cmp* and *grep* for an 8-issue processor. The additional freedom to re-order ambiguous store/load pairs enabled the scheduler to obtain a much tighter schedule in the critical sections of these programs. For those benchmarks in which no measurable performance was observed, *cccp*, *eqntott*, *lex*, and *tbl*, few stores were found in the frequently executed code

[4] Note that in this study super-linear speedups are possible. This is because both the issue width of the processor and the scheduling scheme are varied.

sections. Therefore, little opportunity is available in these benchmarks to improve performance by reordering store/load pairs.

5.2 Cyclic scheduling

Modulo Scheduling is a software pipelining technique that schedules loops whose body consists of a simple basic block [10]. In order to schedule loops with conditional constructs, the conditional constructs must be converted into straight-line code. If-conversion can be used to convert conditional constructs into straight-line code [21][12][15]. In this paper we discuss modulo scheduling with if-conversion assuming predicated hardware support [9][12]. As discussed in Section 3.2, predicated hardware support allows for efficient hazard resolution. When the branch is taken (mispredicted), the taken path code is executed and control branches back to the instruction following the branch. The predicates ensure that the not taken path code is squashed.

Control-flow profiling information is used to modulo schedule the most frequently executed paths in the loop. After the most frequently executed paths have been selected, if-conversion is applied to these paths. For conditional branches where both successor basic blocks are included in the modulo schedule, the branch is converted to a predicate assert operation and operations along both paths are predicated. For conditional branches where only one successor is in the modulo schedule, the branch is converted into a branch and predicate assert operation and only the not taken path (the path in the modulo schedule) is predicated.

In modulo scheduling, the interval at which loop iterations are initiated, the iteration interval (II), is fixed for every iteration of the loop [10]. The scheduler determines the lower bound on II and then tries to find a schedule for this II. The scheduler 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.

Figures 15 - 19 present the algorithms used to modulo scheduled profiled loops without dependence cycles. These algorithms are discussed at a fairly high level of abstraction. For more detailed information, refer to [22]. In these algorithms, we refer to the elements to schedule as operations before scheduling and

```
modulo_schedule(profile_loop) {
    construct dependence graph, G
    determine minimum II due to resource constraints
    while (schedule_II(profile_loop, II) not found)
        increment II
    /* use MVE to rename registers that span more than one II (N = number of times to unroll loop) */
    N = MVE(profile_loop)
    kernel = create_kernel(profile_loop, N)
    rename registers in kernel according to MVE
    /* determine number of stages in prologue and epilogue */
    S = [latest_issue_time/II] - 1
    softpipe = create_prologue_epilogue(kernel, S)
    for instruction in softpipe, I {
        if I has a branch and assert operation and not loop back branch {
            schedule taken path code
            branch back to instruction following I
        }
    }
    generate remainder loop to execute the remainder of (loop_bound - S)/N iterations
    append softpipe to remainder loop
}
```

Figure 15: Modulo scheduling algorithm without dependence cycles.

instructions after scheduling, where an instruction consists of all the operations scheduled at the same cycle.

The basic modulo scheduling algorithm is presented in Figure 15. The first step in this algorithm is to find an II that can be scheduled. The routine schedule_II in Figure 16 attempts to find a schedule for a given II. Note that if an operation cannot be scheduled in the cycle that it is ready, it is delayed until it can be scheduled. If it is delayed more than II cycles, it cannot be scheduled and schedule_II fails.

After scheduling II, the software pipeline can be constructed. Each stage in the software pipeline is II cycles long. The software pipeline consists of three parts, the kernel, the prologue, and the epilogue. The kernel represents the steady-state execution. Since some register lifetimes may span more than one II, the kernel is unrolled to satisfy the longest lifetime. Modulo variable expansion (MVE) is used to rename registers with lifetimes that span more than one II [11]. The algorithm to generate the kernel is presented in Figure 17. The prologue and epilogue are used to fill and empty the software pipeline. The algorithm to generate the prologue and epilogue is shown in Figure 18. The prologue and epilogue are created in a fashion that ensures that registers are aligned properly at the prologue-kernel boundary and at the kernel-epilogue boundary.

Note that the algorithm in Figure 18 only creates one epilogue which is executed at the end of the kernel

```
schedule_II(profile_loop, II) {
    clear resource_usage_map
    for each operation in profile_loop, op {
        op→priority = computed priority of op
        add op to set of unscheduled operations
    }
        sort unscheduled set according to operation priority
    issue_time = 0
    while (unscheduled set of operations is not empty) {
        issue_time = issue_time + 1
        active_set = set of unscheduled operations that are ready
        for each operation in active_set, op {
            schedule op at earliest available resource
            if(op cannot be scheduled within II cycles)
                return schedule not found
            mark required resources of op busy in modulo II resource_usage_map
            delete op from set of unscheduled operations
            op→issue_time = issue_time
    }
}
```

Figure 16: List scheduling algorithm to schedule II.

```
create_kernel(profile_loop, N) {
    for operation in profile_loop, op {
        /* fill II instructions in II_schedule */
        place op in instruction of II_schedule at op- > issue_time mod II
    }
    for N times {
        for instruction in II_schedule, I {
            for operation in I, op {
                copy_operation(op)
            }
        }
    }
}
```

Figure 17: Algorithm to create kernel code of software pipeline.

code. To ensure that the loop executes the correct number of times, a remainder loop is prepended to the software pipeline schedule as shown at the end of the algorithm in Figure 15.

The only difference between the modulo scheduling algorithms with and without profiling is that the taken path of the branch must be copied every time the branch and assert operation is copied. Figure 19 shows the copy operation algorithm. After the software pipeline has been constructed, the taken paths of each branch and assert operation are scheduled taking into account the dependence and resource constraints of the pipelined schedule. Once the taken path is scheduled, a branch operation is inserted to branch to the instruction following the instruction containing the branch and assert operation.

During scheduling, the constraints of the excluded paths are ignored. This is possible since 1) speculative execution is not allowed, and 2) control branches back to the instruction following the branch and assert operation. Since speculative execution is not allowed, control dependent operations will not be moved above the branch and assert operation. Disallowing speculative execution will not affect the size of II, but may lengthen the number of stages in the pipeline. Since control branches back to the instruction following the branch and assert operation, operations moved from above to below the branch during scheduling will always be executed. Furthermore, since the paths merge at the instruction after the branch, the branch operation prevents operations from being moved above the merge point. Operations below the merge point that are control dependent on the branch will be predicated.

Experimental methodology. To evaluate the effectiveness of using control-flow profiling for cyclic code, we applied modulo scheduling with if-conversion to a range of superscalar and VLIW processors with predicated hardware support. The benchmarks used in this study are 25 loops selected from the Perfect Suite [23]. These loops have no cross-iteration dependences and have at least one conditional construct. The loops are assumed to execute a large number of times and thus only the steady-state (kernel) execution is calculated for the modulo scheduled loops.

The base processor used in this study is a RISC processor which has an instruction set similar to the Intel i860 [24]. Intel i860 instruction latencies are used. The processor has an unlimited number of registers and an ideal cache.

```
create_prologue_epilogue(kernel, S) {
    S_unroll = S mod N
    S_peel = remainder of S/N
    /* create prologue */
    issue_time = 0
    for S_peel times {
        for instructions starting at (N - S_peel) stage in kernel, I {
            for operation in I, op {
                if(op→issue_time ≤ issue_time)
                    copy_operation(op)
            }
            issue_time = issue_time + 1
        }
    }
    for S_unroll times {
        for instruction in kernel, I {
            for operation in I, op {
                if(op→issue_time ≤ issue_time)
                    copy_operation(op)
            }
            issue_time = issue_time + 1
        }
    }
    prepend prologue to beginning of kernel
    /* create epilogue */
    issue_time = (S + N)*II
    for S_unroll times {
        for instruction in kernel, I {
            for operation in I, op {
                if(op→issue_time > issue_time - (S + N)*II)
                    copy_operation(op)
            }
            issue_time = issue_time + 1
        }
    }
    for S_peel times {
        for instructions starting at beginning of kernel, I {
            for operation in I, op {
                if(op→issue_time > issue_time - (S + N)*II)
                    copy_operation(op)
            }
            issue_time = issue_time + 1
        }
    }
    append epilogue to kernel
}
```

Figure 18: Algorithm to create prologue and epilogue code of software pipeline.

```
copy_operation(op) {
    copy op
    if (op is a branch and assert operation) {
        copy taken path of op
        change branch destination of op to point to copy of taken path
    }
}
```

Figure 19: Algorithm to copy operations.

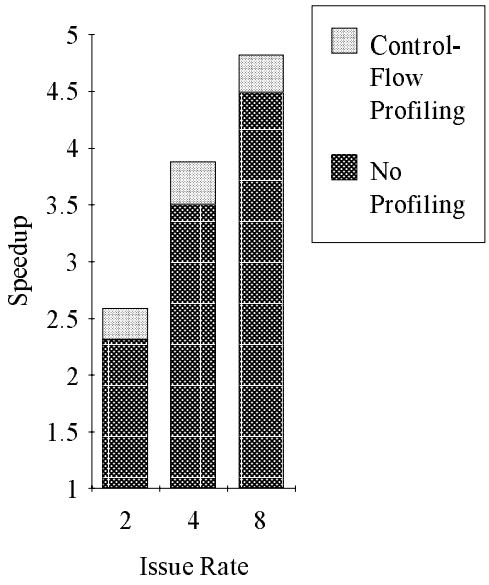

Figure 20: Performance improvement for modulo scheduling using profile optimization.

Experimental results. The benefit of control-flow profiling for modulo scheduling with predicated hardware support is illustrated by comparing the speedup with and without profiling information for superscalar/VLIW processors with issue rates 2, 4, and 8.[5] No limitation is placed on the combination of instructions that can be issued in the same cycle. 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 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 modulo scheduling with predicated hardware support is shown in Figure 20. The speedups are calculated using the harmonic mean. Overall, profiling improves the performance by approximately 12% for the issue-2 machine, 11% for the issue-4 machine, and 7% for the issue-8 machine.

It is interesting to note the effect of control-flow profiling. By eliminating paths before modulo scheduling, the scheduling constraints along excluded paths can be ignored and a tighter II can be found. There are two

[5] Induction variable reversal was not applied to these loops before modulo scheduling [25]

types of scheduling constraints, dependences and resources. With predicated hardware support, reducing the resource constraints can be particularly important since all instructions along all control paths are fetched. Thus, the resource constraint is determined by the most heavily used resource along *all* paths. By eliminating some of the less frequently executed control flow paths, the minimum II is reduced. As shown in Figure 20, this particularly benefits the lower issue rates which incur more resource conflicts.

One final point to note is that while this technique has been presented as a way to exclude infrequently executed paths, it can also be used to exclude paths which contain software pipeline preventing code. For instance, loops with subroutine calls are often not software pipelined. However, if the subroutine is only called only some execution paths, they can be excluded and the remaining paths can be efficiently scheduled.

6 Related Work

Profile information provides valuable data about the dynamic behavior of a program. Control-flow profile information has been used to assist compile-time code transformations both by 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 [26] [27]. In the area of global code scheduling, trace scheduling is a popular global microcode compaction technique [2]. 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 control-flow profiling is an effective way to do this [3] [18].

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 control-flow profiling is an effective way to guide instruction placement [28] [29].

Control-flow profile information can help in identifying the frequently accessed variables [30] in a program resulting in increased efficiency for register allocation. Function inline expansion eliminates the overhead of function calls and enlarges the scope of global code optimizations. Control-flow profiling can be extended

to measure the frequency of subroutine calls in order to determine the best expansion sequence [31]. 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 [32].

7 Conclusion and Future Work

This paper extends the use of control-flow profiling information to acyclic global scheduling and software pipelining. By systematically eliminating the constraints imposed by the infrequent execution paths on the frequently executed paths, the overall program performance can be improved through better scheduling. In acyclic instruction scheduling, profile information provides the compiler the means to predict the performance implications of code movement across basic blocks. As described in Section 3.1, this allows the compiler to decrease the overall program execution time by utilizing scheduling techniques which improve the execution time of certain paths in the program at the cost of others. Without profile information, such scheduling techniques cannot be used because of the possibility of optimizing the performance of a infrequently executed path at the expanse of a frequently executed path.

In cyclic instruction scheduling, the final instruction schedule is constrained by the resource requirements of the loop body. In this case profile information can be used to eliminate the infrequently executed basic blocks from the schedule. Software pipelining only the frequently executed path in a loop body results in a tighter schedule improving the overall program performance.

The use of memory-dependence profiling in the IMPACT compiler allowed speculative movement of memory load operations above stores in an efficient way. The profile information lets the compiler perform this optimization for load-store pairs which are unlikely to conflict. This minimizes the performance penalty introduced by the repair code which is invoked in the case of a conflict.

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. Some of

the 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 access and reuse patterns and which can be used to estimate which references actually need to be prefetched and when.

The techniques presented in this paper and the current research in compiler optimizations improve the performance of a program by predicting its runtime behavior at compile time. For optimizations where performance of one execution scenario of a program is improved at the cost of other scenarios, the accuracy of the compiler predictions translate into program execution performance. Profile information provides the compiler with an efficient means of predicting the program behavior.

References

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

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

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

[4] M. D. Smith, M. S. Lam, and M. A. Horowitz, "Boosting beyond static scheduling in a superscalar processor," in *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 344–354, May 1990.

[5] 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.

[6] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling for superscalar and VLIW processors," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 238–247, October 1992.

[7] D. Weaver, *SPARC-V9 Architecture Specification*. SPARC International Inc., 1992.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt, "Overlapped loop support in the Cydra 5," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 26–38, April 1989.

[13] B. Su and J. Wang, "GURPR*: A new global software pipelining algorithm," in *Proceedings of the 24th International Conference on Microarchitecture*, pp. 212–216, November 1991.

[14] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai, "Code generation schema for modulo scheduled loops," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 158–169, December 1992.

[15] N. J. Warter, G. E. Haab, K. Subramanian, and J. W. Bockhaus, "Enhanced modulo scheduling for loops with conditional branches," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 170–179, December 1992.

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

[17] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An effective structure for VLIW and superscalar compilation," *Journal of Supercomputing*, February 1993.

[18] 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.

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

[20] 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.

[21] 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.

[22] N. J. Warter and W. W. Hwu, "Enhanced modulo scheduling," Tech. Rep. in preparation, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, 1993.

[23] M. Berry and *et. al*, "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.

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

[25] N. J. Warter, D. M. Lavery, and W. W. Hwu, "The benefit of predicated execution for software pipelining," in *Proceedings of the 26rd Hawaii International Conference on System Sciences*, pp. 497–506, January 1993.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.