# Modulo Scheduling of Loops in Control-Intensive Non-Numeric Programs

Daniel M. Lavery  Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois, Urbana-Champaign, IL 61801
lavery@crhc.uiuc.edu, hwu@crhc.uiuc.edu

## Abstract

*Much of the previous work on modulo scheduling has targeted numeric programs, in which, often, the majority of the loops are well-behaved loop-counter-based loops without early exits. In control-intensive non-numeric programs, the loops frequently have characteristics that make it more difficult to effectively apply modulo scheduling. These characteristics include multiple control flow paths, loops that are not based on a loop counter, and multiple exits. In these loops, the presence of unimportant paths with high resource usage or long dependence chains can penalize the important paths. A path that contains a hazard such as another nested loop can prohibit modulo scheduling of the loop. Control dependences can severely restrict the overlap of the blocks within and across iterations.*

*This paper describes a set of methods that allow effective modulo scheduling of loops with multiple exits. The techniques include removal of control dependences to enable speculation, extensions to modulo variable expansion, and a new epilogue generation scheme. These methods can be used with superblock and hyperblock techniques to allow modulo scheduling of the selected paths of loops with arbitrary control flow. A case study is presented to show how these methods, combined with superblock techniques, enable modulo scheduling to be effectively applied to control-intensive non-numeric programs. Performance results for several SPEC CINT92 benchmarks and Unix utility programs are reported and demonstrate the applicability of modulo scheduling to this class of programs.*

## 1. Introduction

The scheduling of instructions in loops is of great interest because many programs spend the majority of their execution time in loops. It is often necessary for the scheduler to overlap successive iterations of a loop in order to find sufficient instruction-level parallelism (ILP) to effectively utilize the resources of high-performance processors.

Software pipelining [18, 6, 1, 15], is a loop scheduling scheme that allows motion of instructions from one iteration to another and maintains the overlap of loop iterations throughout the execution of the loop. A description of the various approaches to software pipelining is given in [17]. This paper focuses on a class of software pipelining methods called *modulo scheduling* [16].

Modulo scheduling simplifies the generation of overlapped schedules by initiating iterations at a constant rate and by requiring all iterations of the loop to have a common schedule. The constant interval between the start of successive iterations is called the *initiation interval* (II). The initial candidate II is chosen as the maximum of two lower bounds. The resource-constrained lower bound on the II (ResMII) [16] is equal to the number of cycles that the most heavily used resource is used by a single iteration. The worst-case constraint among all the cycles in the dependence graph determines the recurrence-constrained lower bound on the II (RecMII) [16].

Most of the previous work on modulo scheduling has targeted numeric programs, in which, often, the majority of the loops are well-behaved "DO" loops (loop-counter-based loops) without early exits. All of the more extensive performance evaluations of modulo scheduling techniques have been for such loops. There seems to exist a perception that modulo scheduling is primarily applicable only to numeric programs.

In control-intensive non-numeric programs, the loops frequently have characteristics that make it more difficult to apply modulo scheduling and to obtain significant speedup. These characteristics include multiple control flow paths, loops that are not based on a loop counter, and multiple exits. Several techniques have been developed to allow modulo scheduling of loops with intra-iteration control flow such as hierarchical reduction [11], predicated execution [5], and reverse if-conversion [21]. The above work has assumed that all of the paths through the loop body are included for scheduling. Including all of the paths can be detrimental to overall loop performance. The presence of unimportant paths with high resource usage or long dependence chains can result in a schedule that penalizes the important paths. An infrequent path that contains a hazard, such as another nested loop or a function call, can prohibit modulo scheduling of the loop.

Previous work has also been done on modulo schedul-

ing of loops that are not based on a loop counter [20, 19]. The key difficulty with this type of loop is that it may take many cycles to determine whether or not to start the next iteration, limiting the overlap of the iterations. This difficulty is overcome by speculatively initiating the next iteration. The work in [20] also mentions a source-to-source transformation to convert a loop with multiple exits into a single-exit loop. The resulting loop contains multiple paths of control and is dealt with using one of the methods for modulo scheduling of loops with intra-iteration control flow referred to above. However this method adds extra instructions and delays the early exits until the end of the loop body. More work is needed to evaluate the performance of this approach, especially for architectures without predicated execution.

This paper describes a new set of methods that allow effective modulo scheduling of loops with multiple paths of control and multiple exits. We use superblock [10] (and in the future hyperblock [14]) techniques to exclude the unimportant and detrimental paths from the loop. Loops with multiple exits often occur naturally in control-intensive programs and the beneficial exclusion of paths via the formation of superblock and hyperblock loops creates many more of them. Thus, an effective method for handling multiple exits is essential.

Rather than transform the loop into a single exit loop, the proposed methods modulo schedule the loop "as is", with the multiple exits present. A new code generation scheme is described which creates correct epilogues for the early exits. Speculation is used to increase both the overlap of the basic blocks within each iteration and the overlap of successive iterations. We extend modulo variable expansion to allow the speculation of instructions that write to variables that are live at the loop exits. Altogether, the methods described in this paper allow effective modulo scheduling of the selected paths of loops with arbitrary control flow.

This paper reports speedup results for several SPEC CINT92 benchmarks and Unix utilities. These are the first reported performance results for modulo scheduling on control-intensive non-numeric programs, and they demonstrate the applicability of modulo scheduling to this class of programs and validate the correctness of the proposed methods.

The paper is organized as follows: Section 2 describes the methods developed and presents a case study to show how these methods, when combined with superblock techniques, enable modulo scheduling to be effectively applied to control-intensive loops. Section 3 reports the performance results, and Section 4 provides a summary and directions for future work.

## 2. Modulo Scheduling of Control-Intensive Loops

A detailed example is used to illustrate the difficulties caused by control-intensive loops and the benefits of the techniques developed. The loop chosen for this case study is one of the frequently executed loops in *lex*, the lexical analyzer generator. The source code for the loop is shown in Figure 1.

```
for (i = n; i >= 0; i--) {
      j = state[i];
S1:  if (count == *j++) {
            for (k = 0; k < count; k++)
                  if (!temp[*j++]) break;
            if (k >= count)
                  return (i);
      }
}
```

**Figure 1. Source code for example loop from *lex*.**

Loops in general purpose non-numeric programs frequently have complex control flow, and this is evident in the example loop. The outer loop contains an if-statement, an inner loop, and an early exit via a return statement. The inner loop contains an if-statement and an early exit via a break statement.

Obviously, this loop contains a number of hazards for modulo scheduling. Modulo scheduling would ordinarily target the inner loop. However, profile information indicates that the inner loop is infrequently invoked and usually has few iterations. The condition for the if-statement S1 evaluates to false more than 90% of the time. Figure 2a shows a simplified version of the control flow graph for the loop. Block X contains the code to load **state**[**i**] and ***j** and do the comparison for statement S1. Block Y consists of the post-increment of the pointer **j** and all the code in the body of the if-statement S1. The control flow within block Y has been omitted for clarity. Block Z contains the code to update **i** and to test the exit condition.

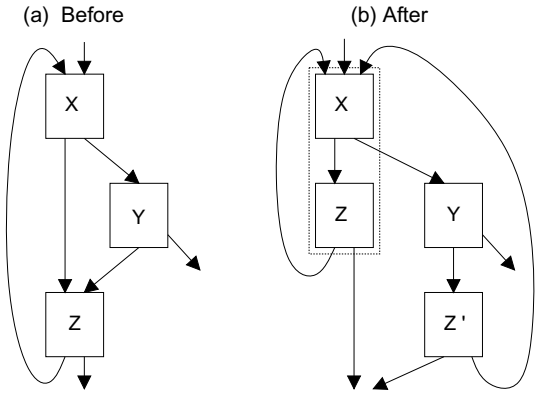


**Figure 2. Superblock formation for example loop.**

The detrimental path containing the inner loop can be excluded from the loop via superblock formation. Effective superblock formation can be done using profile information [3] and/or static analysis of the structure and hazards in the program [9]. A superblock loop consisting of the most frequent path through the outer loop (blocks X and Z) is formed as shown in Figure 2b. The path through block

Y has been excluded via tail duplication of block Z. A superblock loop consists of a single path through a loop, with a single entrance and one or more exits. The loop consisting of blocks X and Z now appears to be an inner loop with multiple exits and can be targeted for modulo scheduling. For a detailed description of superblock formation, see [10, 3].

It has been shown that superblock optimization and acyclic scheduling techniques provide substantial speedup [10]. In general, the ability of superblocks (and similarly hyperblocks) to exclude undesirable paths of execution can provide the following benefits for modulo scheduling:

- Decrease ResMII by excluding unimportant paths with high resource usage.
- Decrease RecMII by excluding unimportant paths that contribute to long dependence cycles.
- Increase the number of loops that can be modulo scheduled by excluding paths containing hazards such as nested loops and function calls.

Although the modulo scheduling methods developed in this paper are described using superblock examples, they are equally applicable to hyperblock code.

Figure 3 shows the assembly code for the example superblock loop. Each instruction is numbered for later reference. Block X in the control flow graph consists of instructions 1 through 3. Instructions 4 through 6 are in block Z. The assembly code shown is that produced by the IMPACT compiler after classic optimizations have been applied. The elements of the array **state** are four bytes in size. The registers shown are virtual registers. Register allocation is done after modulo scheduling.

| Inst. | Assembly | Register Contents |
|---|---|---|
| 1 | L1: r12 = MEM(r34+r8) | r8 = state |
| 2 | r13 = MEM(r12+0) | r34 = i*4 |
| 3 | beq (r6 r13) L2 | r12 = j |
| 4 | r4 = r4 - 1 | r13 = *j |
| 5 | r34 = r34 - 4 | r6 = count |
| 6 | ble (0 r4) L1 | r4 = i |

**Figure 3. Assembly code for superblock loop.**

Control exits the superblock loop if instruction 3 is taken, or if instruction 6 is not taken. In this paper, the exit associated with the fall-through path of the loop back branch is termed the *final exit*. Any other exits from a superblock loop are via taken branches and are termed *early exits*.

The virtual registers r34, r4, and r12 are live out when the early exit to **L2** (block Y) is taken. The values in r34 and r4 are decremented in block Z′. The value in r12 is incremented in block Y. No virtual registers are live out when the loop exits via the final exit (instruction 6).

Loops with complex control flow occur frequently in general purpose non-numeric programs. Table 1 shows statistics on the percentage of dynamic instructions that are in single basic block loops (*Basic Block*) and multiple exit superblock loops (*Superblock*) for the SPEC CINT92 benchmarks and several Unix utility programs. The column labeled *Total* is the sum of the other two columns. The time not spent in these two types of loops is spent in the excluded paths of inner and outer loops and in acyclic code.

**Table 1. Percentage of dynamic instructions in single basic block and superblock loops.**

| Benchmark | Basic Block | Superblock | Total |
|---|---|---|---|
| 008.espresso | 5.6 | 57.8 | 63.4 |
| 022.li | 0.7 | 21.4 | 22.1 |
| 023.eqntott | 1.8 | 70.5 | 72.3 |
| 026.compress | 0.6 | 49.8 | 50.4 |
| 072.sc | 4.4 | 34.6 | 39.0 |
| 085.gcc | 14.1 | 28.5 | 42.6 |
| cmp | 0.0 | 94.5 | 94.5 |
| eqn | 2.6 | 20.9 | 23.5 |
| lex | 2.0 | 86.2 | 88.2 |
| tbl | 17.4 | 9.6 | 27 |
| wc | 0.0 | 64.7 | 64.7 |
| yacc | 3.2 | 45.5 | 48.7 |

For all the programs except *gcc* and *tbl*, little or no time is spent in single basic block loops. For all the programs except *tbl*, more time (usually much more) is spent in multiple exit superblock loops than in single basic block loops. From this table, it is clear that modulo scheduling must be able to effectively handle loops with control flow to be applicable to these programs. The remainder of this section describes how the proposed techniques overcome the control dependences and register anti-dependences associated with loops that have multiple exits and live-out virtual registers. A code generation scheme for loops with multiple exits is also presented.

## 2.1. Overcoming control dependence using speculative code motion

Control dependences are a major impediment to the exploitation of ILP in the loops of general-purpose non-numeric programs. Cross-iteration control dependences restrict the overlap of loop iterations by delaying the start of subsequent iterations until all the branches from the current iteration have been executed. Frequently the branches are dependent on earlier computations in the loop body and cannot be executed until late in the iteration, severely limiting any overlap.

Intra-iteration control dependences combined with cross-iteration data dependences create recurrences which limit the throughput of the modulo scheduled loop. They also increase the length of the critical paths through a single iteration, resulting in a longer schedule for each iteration, an important consideration for short trip count loops.

As described in [20, 19], the cross-iteration control dependences from the loop back branch to the instructions

in the next iteration can be relaxed, allowing speculative code motion and overlap of the iterations. For loops with multiple exits, this concept must be extended to the early exit branches. It is often necessary to remove the cross-iteration control dependences from an early exit branch to the instructions in subsequent iterations to achieve the desired level of overlap. It is also often necessary to remove intra-iteration control dependences to allow overlap of the blocks within an iteration and to achieve good performance for short trip count loops.

Removal of either type of control dependence is not quite so simple however. We currently assume that stores and branches are not speculatively executed. The reordering of branches will be the subject of future work. In order to speculatively execute loads and other instructions that can cause exceptions, either the processor architecture must contain support for speculative execution [4, 13] or the compiler must be able to prove via program analysis that the speculatively executed instruction will not except [2]. In this paper, we assume an instruction set architecture that contains silent (non-trapping) versions of the instructions that can cause exceptions [4]. Furthermore, instructions that write to virtual registers that are live at the loop exits require special attention when removing control dependences. This issue will be discussed in Section 2.2. We now show the effect of control dependences on the example superblock loop.

Figure 4a shows the dependence graph. Each node is numbered with the ID (from Figure 3) of the instruction it represents. The branch nodes are shaded. The data and control dependences are shown with solid and dashed lines respectively. Some of the transitive dependences are not shown. None of the register anti-dependences are shown, assuming that they can be removed. Removal of anti-dependences is discussed in Section 2.2.

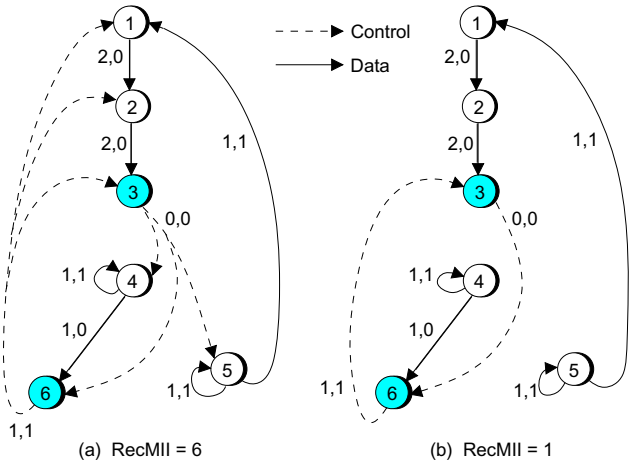


**Figure 4. Dependence graph for example loop.**

Each arc is labeled with two numbers. The first is the minimum delay in cycles required between the start of the two instructions. The second number is the distance, which is the number of iterations between the two dependent instructions. Arcs with a distance of zero are intra-iteration dependences and those with a distance greater than zero are cross-iteration dependences. The instruction set assumed is similar to HP's PA-RISC 1.1 but has no branch delay slots. Except for the branches, the delays shown are those of the PA7100. It is assumed that the instructions in the fall-through path of a branch can potentially be executed in the same cycle as the branch and that instructions in the taken path are executed the cycle following the branch.

There are several non-trivial recurrences apparent in the graph. The longest recurrence circuit runs through instructions 1, 2, 3, 4, 6, and back to 1. It has a total delay of six and spans one iteration, resulting in a RecMII of six. If the loop is scheduled using this dependence graph, there is no overlap of the iterations. The cross-iteration control dependences from the loop back branch to the instructions in the next iteration (except instruction 3) can be removed, allowing speculative code motion and overlap of the iterations. However, there are still limiting control dependences present. The recurrence circuit consisting of instructions 1, 2, 3, and 5 limits the RecMII to five. To break this recurrence, the intra-iteration control dependence between instructions 3 and 5 must be removed, enabling speculative execution of instruction 5. The control dependence from instruction 3 to instruction 4 must also be removed to break the remaining limiting recurrence. Figure 4b shows the dependence graph after all of the limiting control dependences have been removed, reducing the RecMII to one.

An instruction can legally be moved during modulo scheduling from above to below a branch if the branch is not data dependent on the instruction. For example, instruction 5 could legally be scheduled after instruction 6. When an instruction is moved from above to below a branch, it is automatically moved into both paths of the branch during the generation of epilogues which follows the actual modulo scheduling process. In Sections 2.2 and 2.4, it is shown that special attention must be paid to this type of code motion for correct code generation in multiple exit loops.

Assuming a 4-issue processor that can execute one branch per cycle, the ResMII for the example loop is two. The RecMII was one, resulting in an II of two. This is a speedup of three over modulo scheduling using the dependence graph of Figure 4a.

## 2.2. Overcoming anti-dependence using modulo variable expansion

Thus far, nothing has been said about anti-dependences and the constraints imposed by the virtual registers that are live out of the loop exits. In its original form, an instruction **I** that writes a virtual register **V** that is live out of an exit branch **B** cannot be moved from below to above **B**, because it overwrites the value in **V** that is used when exit **B** is taken. This constraint on upward code motion is exactly the same as if **V** was one of the operands of **B** (i.e. it is an anti-dependence constraint) but is represented differently in many compilers. Instead of adding an explicit anti-dependence arc, many compilers, including IMPACT,

overload the control dependence arc to represent both the control dependence and the anti-dependence.

There are several examples of anti-dependence in the case study loop. Instruction 1 uses r34 which is later defined by instruction 5. Virtual register r34 is live out when the branch to **L2** (instruction 3) is taken, so there is an anti-dependence between instruction 3 and instruction 5.

Anti-dependences can be removed by renaming. Modulo variable expansion [11, 19] unrolls the kernel and renames the successive lifetimes corresponding to the same loop-variant so that they no longer overlap in time. This allows register anti-dependences to be removed before scheduling, knowing that modulo variable expansion will correct the overlap of lifetimes that the lack of these dependences allows. The modulo variable expansion algorithm, as originally described [11], allows the removal of cross-iteration anti-dependences. However, intra-iteration anti-dependences can also be removed if the lifetime analysis and the renaming algorithms are extended to include lifetimes that cross iterations. It is assumed that this can be done in [19]. In this paper, we describe the changes needed.

Figure 5 illustrates the relaxation of a cross-iteration anti-dependence using modulo variable expansion, as described in [11]. Three iterations of an abstract loop body containing a definition and a use of a virtual register r1 are shown. There is an intra-iteration flow dependence (marked with an f) and a cross-iteration anti-dependence (marked with an a). The cycle in which each instruction is issued is shown in square brackets to the right of the abstract instruction, assuming the delay for the flow dependence is two and the anti-dependence is zero. In its original form as shown on the left, the minimum II that can be achieved is two. Using modulo variable expansion, the anti-dependence can be removed prior to scheduling, reducing the II to one. Two virtual registers are now used as shown on the right.

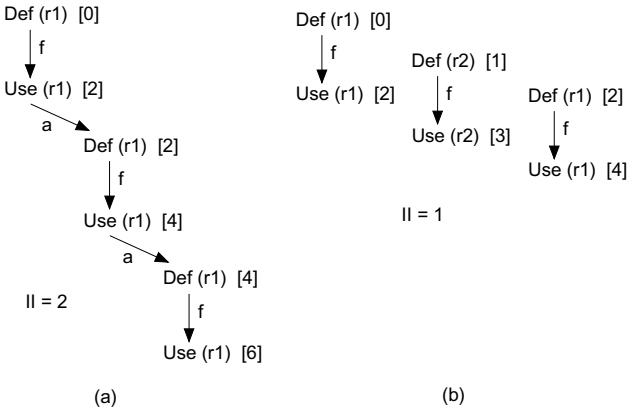


**Figure 5. Relaxation of cross-iteration anti-dependence.**

Figure 6 shows the relaxation of an intra-iteration anti-dependence. In this case, the use appears before the definition in the original iteration, and the lifetime of r1 now crosses the iterations. Removal of the intra-iteration anti-dependence prior to scheduling allows the definition to be moved above the use within the iteration as shown on the right. As in the previous case, two registers are used and the II is reduced from 2 to 1.

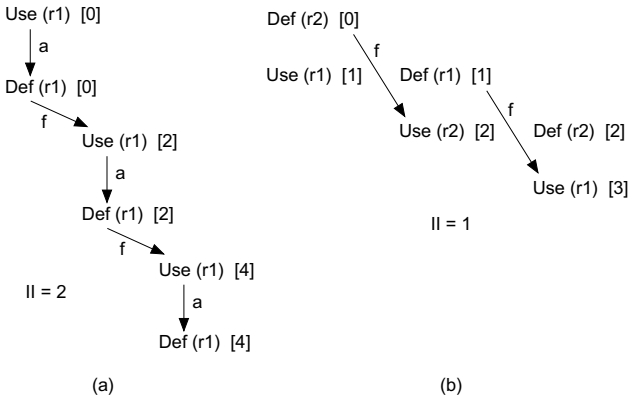


**Figure 6. Relaxation of intra-iteration anti-dependence.**

The lifetime of a virtual register extends from its first definition to its last use. The lifetime of a loop-variant virtual register **V** from a definition **D** to a use **U** is computed using the following equation, assuming that the lifetime starts when **D** is issued and ends when **U** is issued.

$$Lifetime(\mathbf{V}) = Issue(\mathbf{U}) - Issue(\mathbf{D}) + II * Dist(\mathbf{V}) \quad (1)$$

Issue(**D**) and Issue(**U**) are the issue time of the instances of **D** and **U** from the same original iteration [1]. Dist(**V**) is the number of iterations separating **D** and the instance of **U** that uses the value defined by **D** in the original loop.

Note that in equation 1, use **U** could be a branch for which **V** is live out. For correct renaming, the lifetime analysis must be extended to include such uses. There is an additional consideration for live out virtual registers. Instruction **D** can be moved downward across the branch **B**. If such code motion occurs, the definition is moved into both paths of the branch during epilogue generation, **V** is no longer live-out, and Lifetime(**V**) as computed by equation 1 becomes less than or equal to 0. Thus, the lifetime of **V** is computed for all the uses except those associated with the exits that **D** has been moved down across.

Figure 7 shows the execution of two iterations of the case study loop after modulo scheduling. The first iteration starts at time 0 and its instructions are denoted with the subscript 1. The second iteration starts at time 2 and its instructions are denoted with subscript 2. The second iteration's instructions are also shaded to further distinguish the two iterations. The lifetimes of all the virtual registers written in the loop are shown to the right of the execution record. Each virtual register's lifetime begins with its

[1] Modulo scheduling generates a schedule for a single iteration of the original loop. It is this schedule that we are working with when analyzing the lifetimes for modulo variable expansion.

definition in the first iteration. Each of the subsequent tic marks denotes either an explicit use of the virtual register as a source operand, or a branch for which the register is live-out. The lifetime extends until the last use of the register.

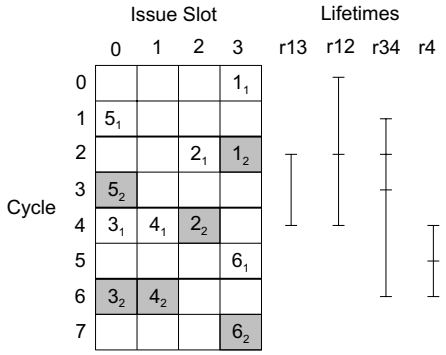


**Figure 7. Execution record and lifetimes for two iterations.**

The lifetime of r13 is entirely contained within one iteration. It is defined by instruction 2 and used by instruction 3. $Issue(2)$ is 2, $Issue(3)$ is 4, $Dist(r13)$ is 0, and II is 2. Using equation 1, the length of the lifetime is 2. The lifetime of r34 crosses iterations. It is defined by instruction 5, used by instruction 1 and 5 of the next iteration, and live out of instruction 3 of the next iteration. $Issue(5)$ is 1, $Issue(3)$ is 4, and $Dist(r34)$ is 1. Using equation 1, the total length of the lifetime is 5.

The definitions of **V** are renamed by cycling through the set of virtual registers assigned for **V**. Each use of **V** is renamed by first finding the iteration that contains the corresponding definition of **V** (the current iteration if Distance(**V**) is zero or the previous iteration if Distance(**V**) is one) and using the same virtual register name as the definition from that iteration.

The longest lifetime, that of r34, is 5 cycles so the loop must be unrolled three times for modulo variable expansion. Figure 8 shows the unrolled kernel of the modulo scheduled loop after modulo variable expansion. The instructions have been renumbered. When renaming, one of the names used is the original virtual register name. The set of registers used for r34 is r34, r342, and r343. The set of registers used for r12 is r12, r122, and r123. The instructions have been put into sequential order, as would be done when generating code for a superscalar processor. The target and fall-through path for the first two copies of the loop back branch (instructions 6 and 12) have been reversed in preparation for epilogue generation. Block **L3** is the original fall through path of the loop.

| Inst. | | Assembly | Cycle |
|---|---|---|---|
| 1 | L1: | beq (r6 r13) L2 | 0 |
| 2 | | r4 = r4 - 1 | 0 |
| 3 | | r13 = MEM(r123+0) | 0 |
| 4 | | r12 = MEM(r343+r8) | 0 |
| 5 | | r34 = r343 - 4 | 1 |
| 6 | | bgt (0 r4) L3 | 1 |
| 7 | | beq (r6 r13) L2 | 2 |
| 8 | | r4 = r4 - 1 | 2 |
| 9 | | r13 = MEM(r12+0) | 2 |
| 10 | | r122 = MEM(r34+r8) | 2 |
| 11 | | r342 = r34 - 4 | 3 |
| 12 | | bgt (0 r4) L3 | 3 |
| 13 | | beq (r6 r13) L2 | 4 |
| 14 | | r4 = r4 - 1 | 4 |
| 15 | | r13 = MEM(r122+0) | 4 |
| 16 | | r123 = MEM(r342+r8) | 4 |
| 17 | | r343 = r342 - 4 | 5 |
| 18 | | ble (0 r4) L1 | 5 |
| | L3: | | |

**Figure 8. Unrolled kernel for superscalar processor.**

## 2.3. Review of a code generation scheme for single exit loops

In this subsection, we review an existing code generation scheme for single exit loops in preparation for introducing a modified scheme for multiple exit loops. For a complete discussion of this and other possible code schemes for single exit loops, see [19]. We use the abstract code representation of [19] to reduce the complexity of the examples. Figure 9a shows a single iteration of a generic single-exit loop after modulo scheduling. Each square represents the code from one stage, or II cycles, of a single iteration of the original source loop. The number of stages is called the *stage count*.

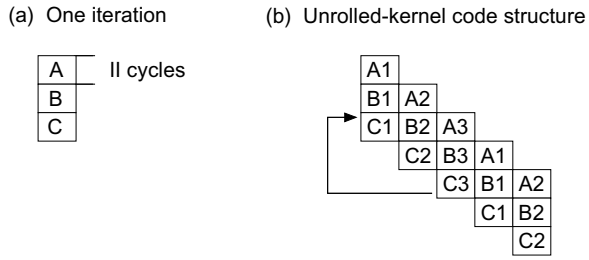


**Figure 9. Abstract representation of iterations.**

Figure 9b shows the code structure for the modulo scheduled loop after kernel unrolling and generation of the prologue and epilogue. The iterations progress from left to right with each one starting one stage after the previous one. The back-edge arrow from row 6 to row 3 identifies the start and end of the unrolled kernel (the degree of unrolling is unrelated to the stage count). The squares before the kernel represent the prologue and the squares afterward represent the epilogue. Each square is now also given a number to specify which version of the code is being used.

Each version uses different names for the registers to avoid overwriting live values. The code structure in Figure 9b is simplistic and does not allow an arbitrary number of iterations to be correctly executed [19], but it illustrates some of the basic concepts and prepares the reader for the more complex (and correct) code schemes described later in this section.

In this paper, *loop back branch* refers to the loop back branch in the original loop body. There are multiple copies of this branch after modulo scheduling because of the kernel unrolling and prologue generation. For all the copies except the one that becomes the loop back branch of the kernel, the target and fall-through path are reversed, as was shown in Figure 8, so that the loop is exited when the branch is taken rather than when it falls through. All the exits associated with the copies of the loop back branch are called *final exits*. All other exits are *early exits*.

The chain of dependences leading up to the loop back branch determines the earliest stage in which the loop back branch can be placed. The stage in which the loop back branch is scheduled determines the number of iterations that are speculatively initiated. Assume the stages of an iteration are numbered such that stage A corresponds to 0, stage B corresponds to 1, and so on. Using the terminology of [19], if the loop back branch is scheduled in stage $\theta$, then there are $\theta$ speculatively executed stages for each iteration after the first. For example, in Figure 9b, if the loop back branch is scheduled in stage B, then stage A of every iteration after the first is executed speculatively. In this paper, the *last* iteration refers to the last iteration that would have been executed in the original non-pipelined loop. When the exit from the last iteration is taken, any speculative iterations are aborted.

Figure 10 shows the structure of the code that is generated for each of the possible stages in which the loop back branch could be placed for a three-stage schedule. In Figure 10a, b, and c, the loop back branch is scheduled at the end of stage A, B, and C respectively.

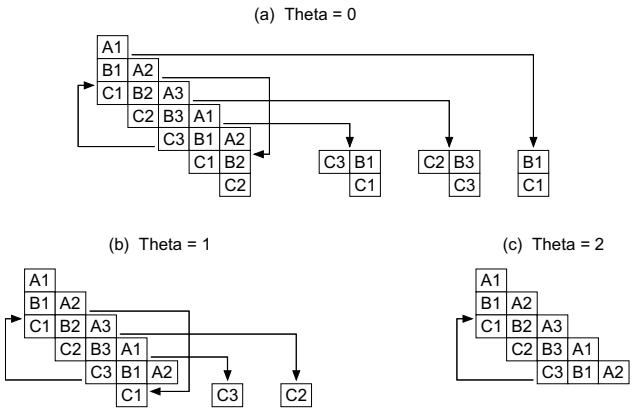


**Figure 10. Code generation scheme for single exit loops.**

The arrows (except the back-edge) represent control transfers from the prologue and kernel to the epilogues shown. Because the final exits are scheduled at the end of the stage, the arrows originate very close to the bottom of each row of squares. Each epilogue contains the code to complete the non-speculative iterations that are in progress at the time the exit is taken. Although it is not explicitly shown, at the end of each epilogue there exists code to move any live out values to the registers in which the code outside the loop expects to find them and to jump to the original target block of the exit.

By comparing Figure 10a and b, one can see how the structure of the generated code changes when the loop back branch is scheduled at the end of stage B instead of stage A. Because the loop back branch is executed one stage later, there are fewer stages left to execute in the epilogues for the last iteration and its predecessors. Thus, the epilogues all have one less row. The one speculative iteration that is in progress when the loop exits is aborted and so there is one less column in each epilogue. There is one less exit from the prologue, so one of the epilogues has disappeared altogether. In general, when the loop back branch is placed in stage $\theta$ instead of stage 0, the $\theta$ rightmost columns of each epilogue are removed, corresponding to the $\theta$ aborted speculative iterations [19]. The resulting epilogues have $\mathbf{SC} - \theta - 1$ rows where **SC** is the stage count.

In Figure 10c, the loop back branch is scheduled at the end of the last stage. Thus, the last iteration and its predecessors are complete when the loop exits. The epilogues consist only of the code needed to move the live values and are not shown.

## 2.4. A code generation scheme for multiple exit loops

Figure 11 illustrates the changes to the code generation scheme for multiple exit loops. The figure assumes a loop with two exits, where both the early exit and the loop back branch are scheduled in the same stage. In Figure 11a, b, and c, the branches are scheduled in stage A, B, and C respectively. There are now more exits from the modulo scheduled loop and thus more epilogues. The arrows associated with the early exits originate very close to the top of each row and have dashed lines to distinguish them from the final exits.

There are two key differences between a final exit and an early exit. First, the final exit is scheduled at the very end of the stage, but an early exit branch can be in the middle of a stage. Thus, for a final exit, the epilogue starts at the beginning of the stage following the one containing the final exit branch. For an early exit, the remainder of the row containing the exit branch in the kernel must be examined for copying to the epilogue. For all the iterations before the last one, the remainder of that iteration's stage in the row containing the exit branch is copied to the epilogue. The treatment of the last iteration will be discussed shortly. In Figure 11, small letters are used to denote a partial stage resulting from an exit branch from the middle of a stage.

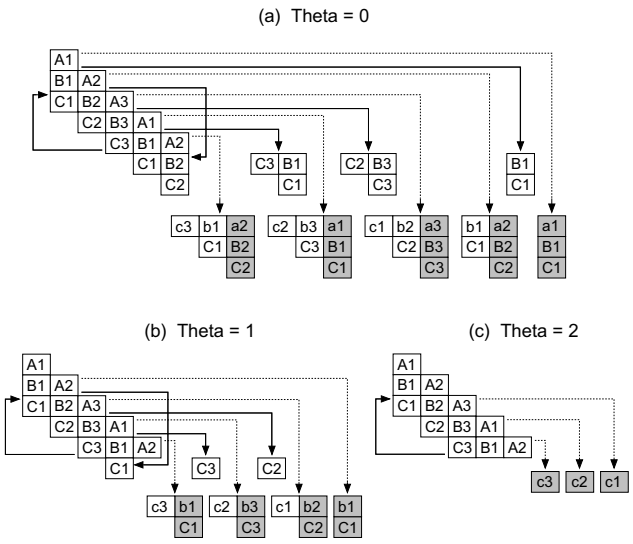


**Figure 11. Code generation scheme for multiple exit loops.**

The second key difference is that the loop back branch is always the last instruction in the original loop body, but an early exit branch is somewhere in the middle of the original loop body. When a final exit is taken, the last iteration is always fully executed. The remaining stages for the last iteration are copied to the epilogue in their entirety. However after an early exit is taken, only the instructions from the last iteration that appeared before the exit branch in the original loop body should be executed.

Assume the basic blocks in a superblock are assigned numerical IDs sequentially from zero to $\mathbf{SC}-1$. Define the *home block* for an instruction to be the basic block in which the instruction resides in the original loop body. For an early exit, an instruction from the remaining stages of the last iteration is copied to the epilogue only if the ID of its home block is less than or equal to the home block ID of the exit branch. In Figure 11, shaded squares are used to denote the stages for which the home block is checked before copying the instruction.

In Figure 11, the epilogues for the final exits are all the same as in Figure 10. The epilogues for the early exits always have one more row and usually have one more column than the corresponding final exit from the same stage. The extra row consists of the remainder of the row in the kernel in which the early exit branch resides. Thus, all the squares in the extra row are marked with small letters. The extra column corresponds to the last stage of the oldest iteration in progress at the time the exit is taken. For a final exit, this iteration is just finished when the exit is taken. For an early exit, part of the last stage remains and is completed in the epilogue. The epilogues for the early exits from the prologue do not have an extra column because none of the iterations have started execution of the last stage. The rightmost column of each early exit epilogue is shaded. This column corresponds to the last iteration. For the last iteration, only the instructions that appeared before the exit branch in the original loop body are copied to the epilogue.

Figure 12 shows the algorithm for generating an epilogue for an exit branch. The algorithm starts with the instructions following the exit branch and copies rows of instructions from the unrolled kernel to the epilogue, wrapping around the kernel until the last row of the epilogue is complete. Squares are not copied if they correspond to instructions from iterations after the last or to instructions from the last iteration that appeared after the exit branch in the original loop body. The algorithm as shown assumes a processor that does not have branch delay slots. The following paragraphs describe the data structures and concepts needed to understand the algorithm.

The unrolled kernel is divided into sections of II cycles each called *kernel rows*. There are **kmin** rows where **kmin** is the degree of unrolling of the kernel. Each row contains a linked list of the instructions contained in that row. The data structure for each instruction contains a pointer to an information structure which contains among other items: the stage in which the instruction is scheduled, the instruction's home block ID, and the row of the kernel that contains the instruction.

There are $\mathbf{SC}-\theta$ rows in each epilogue (numbered zero to $\mathbf{SC}-\theta-1$) where $\theta$ is the stage in which the exit branch is scheduled. Row zero is the partial row and is empty for a final exit (the linked list for each kernel row ends with a final exit). In row **epi_row** of the epilogue, the last iteration is executing in stage $\mathbf{epi\_row}+\theta$. Instructions from stages less than $\mathbf{epi\_row}+\theta$ must be from iterations after the last, and thus are not copied.

For simplicity, the algorithm shown generates correct epilogues for exits from the kernel, but not for exits from the prologue. In practice, the algorithm contains additional code to map an exit in the prologue to the corresponding exit in the kernel. The prologue is generated in a similar manner to the epilogues, by copying selected instructions from the rows of the unrolled kernel. Mapping a prologue exit to a corresponding kernel exit facilitates the copying of rows for the epilogue. Also in practice, if the epilogue is for an exit from the prologue, the algorithm does not copy an instruction that is from a later stage than the stage that the very first iteration is executing. Such instructions correspond to the non-existent iterations prior to the first one.

We now apply the code generation scheme to the example loop. The schedule for a single iteration of the example loop contains 3 stages. Stage A consists of instructions 1 and 5 from the original loop (see Figure 7). Stage B contains instruction 2. Instructions 3 (early exit), 4, and 6 (final exit) are in stage C. The code scheme in Figure 11c is similar to what would be generated for the example loop. Because of the dependence structure of the loop, there is no opportunity for downward code motion across the early exit branch. Thus, when the early exit branch is taken there are no remaining instructions from the last iteration that appeared before the exit branch in the original loop body

```
Algorithm gen_epi(exit branch) {

  create epilogue block
  sc = stage count
  theta = exit->info->stage
  row = exit->info->row
  exit_home_block = exit->info->home_block

  /* Generate all rows of epilogue */
  for (epi_row = 0 to sc - theta - 1) do {

    /* Determine where to start copying */
    if (epi_row == 0)
      /* Partial row.  If exit is a final exit,
         exit->next_op is NULL and no instructions
         will be copied to the partial row. */
      oper = exit->next_op
    else
      /* Full row */
      oper = kernel[row]->first_op

    /* Generate one full or partial row */
    while (oper != NULL) {

      oper_stage = oper->info->stage
      oper_home_block = oper->info->home_block

      /* Copy instruction if it is from an
         iteration previous to the last iteration,
         or if it is from the last iteration and
         appears before the exit branch in the
         original loop body */
      if ( (oper_stage > epi_row + theta) or
           (oper_stage == epi_row + theta and
            oper_home_block <= exit_home_block) ) {
        new_op = copy_operation(oper)
        insert_op_after(epilogue->last_op, new_op)
      }
      oper = oper->next_op
    }
    /* Rotate through the rows of the kernel */
    row = (row + 1) mod kmin
  }

  /* insert moves at end of epilogue for variants
     that are live out of exit */
  insert_moves_for_live_variants(epilogue, exit)

  /* Last exit branch falls through to epilogue */
  if (exit is not last exit in unrolled kernel) {
    jump = create jump to target of exit branch
    insert_op_after(epilogue->last_op, jump)
    make epilogue block the target of exit branch
  }
}
```

**Figure 12. Epilogue generation algorithm.**

and the shaded epilogues are empty.

### 2.5. Insertion of moves for live-out values

As mentioned earlier, code must be appended to the end of each epilogue to move the values that are live-out of the corresponding exit into the register in which the code outside the loop expects to find them. For single exit superblock loops, the value used outside the loop must have been defined in the last iteration. Thus, for each final exit, the instructions from the last iteration are examined in the corresponding epilogue and in the kernel. If the value produced by the instruction is live-out and the destination register is not the one expected outside the loop, a move instruction is inserted at the end of the epilogue.

For multiple exit loops, the procedure is the same for the final exits. However, for the early exits there is an additional consideration. The live out value could be defined in the last iteration by one of the instructions that preceded the exit branch in the original loop body, or it could be defined in the second-to-last iteration by one of the instructions that followed the exit branch in the original loop body. Thus, the last iteration is examined for instructions that originally resided in the same or earlier home block as the early exit branch and the second-to-last iteration is examined for instructions that originally resided in a later home block than the exit.

In the example loop, when the early exit (instruction 3 from Figure 3) is taken, the live out values of r34 and r4 are from the second-to-last iteration and the live out value of r12 is from the last iteration. There are no values live out of the final exit. Figure 13 shows the code generated for the example loop using the multiple epilogue code scheme of Figure 11c. The instructions have again been renumbered. The moves for the live out values (instructions 25, 27, 28, and 30) are also shown.

The blocks labeled **Pro** and **L1** are the prologue and the unrolled kernel respectively. The blocks labeled **LE1, LE3** and **LE5** are epilogues. The block immediately following the kernel is the epilogue reached by falling through the loop back branch. Block **L3** is the original fall through path of the loop. Label **L2** is the start of block Y. The epilogues for the final exits (instructions 11, 17, and 23) are all empty because no code was moved downward across the loop back branch and there are no virtual registers live out of the final exits. Rather than branching to empty epilogues, the final exits branch directly to **L3**. The exception is the loop back branch, which falls through into its epilogue and then jumps to **L3**.

The early exits (instructions 6, 12, and 18) all require moves for one or more of the live virtual registers, so all branch to epilogues. As mentioned at the end of 2.2, when renaming, one of the names used is the original virtual register name. Thus, if the live out value is already in the correct register, a move is not necessary. This is the case for r34 in epilogue **LE1** and r12 in epilogue **LE5**. A jump is placed at the end of each early exit epilogue to transfer control to Block Y.

| Inst. | | Assembly | Cycle |
|---|---|---|---|
| 1 | Pro: | r122 = MEM(r34+r8) | 0 |
| 2 | | r342 = r34 - 4 | 1 |
| 3 | | r13 = MEM(r122+0) | 2 |
| 4 | | r123 = MEM(r342+r8) | 2 |
| 5 | | r343 = r342 - 4 | 3 |
| 6 | L1: | beq (r6 r13) LE1 | 0 |
| 7 | | r4 = r4 - 1 | 0 |
| 8 | | r13 = MEM(r123+0) | 0 |
| 9 | | r12 = MEM(r343+r8) | 0 |
| 10 | | r34 = r343 - 4 | 1 |
| 11 | | bgt (0 r4) L3 | 1 |
| 12 | | beq (r6 r13) LE3 | 2 |
| 13 | | r4 = r4 - 1 | 2 |
| 14 | | r13 = MEM(r12+0) | 2 |
| 15 | | r122 = MEM(r34+r8) | 2 |
| 16 | | r342 = r34 - 4 | 3 |
| 17 | | bgt (0 r4) L3 | 3 |
| 18 | | beq (r6 r13) LE5 | 4 |
| 19 | | r4 = r4 - 1 | 4 |
| 20 | | r13 = MEM(r122+0) | 4 |
| 21 | | r123 = MEM(r342+r8) | 4 |
| 22 | | r343 = r342 - 4 | 5 |
| 23 | | ble (0 r4) L1 | 5 |
| 24 | | jump L3 | 0 |
| 25 | LE1: | r12 = r122 | 0 |
| 26 | | jump L2 | 0 |
| 27 | LE3: | r12 = r123 | 0 |
| 28 | | r34 = r342 | 0 |
| 29 | | jump L2 | 0 |
| 30 | LE5: | r34 = r343 | 0 |
| 31 | | jump L2 | 0 |
| | L3: | | |

**Figure 13. Final assembly code for the example loop.**

The virtual registers are renamed during modulo variable expansion such that the uses of a live-in virtual register in the first iteration refer to the original virtual register name. Thus, no moves are required for live-in values. For example, virtual register r34 is live-in and the first iteration in the prologue uses r34 (instructions 1 and 2) rather than one of the renamed versions (r342 and r343).

## 3. Experimental Results

In this section, we report experimental results on the applicability of modulo scheduling to control-intensive non-numeric programs. The results were obtained using the IMPACT compiler. Inter-procedural alias analysis [7] and data dependence analysis are done in the front end and memory dependence arcs are passed to the back end, giving the optimizer and the schedulers accurate dependence information. In addition to the classic optimizations, optimizations are performed in the back end to increase the ILP of the code [10].

Modulo scheduling is done before prepass acyclic scheduling and global register allocation. The modulo scheduler is an implementation of Rau's Iterative Modulo Scheduling [16]. It uses a machine description system [8] to get information on instruction latencies and resource requirements. The modulo scheduler has been used to pipeline loops for high issue rate versions of the PA-RISC (in this paper) and SPARC architectures. Loops are eligible for modulo scheduling if they are inner loops (outer loops may become inner loops after superblock formation), are single basic block or superblock loops, and do not contain function calls on the included path (function calls may be excluded from the loop by superblock formation, enabling modulo scheduling).

The target processors for these experiments are multiple issue processors with issue rates between 4 and 8 with varying resource constraints. Table 2 shows the functional unit mix for each processor. All processors are assumed to have 32 integer registers and 32 double-precision floating-point registers. The latencies used are those of the HP PA7100 processor.

**Table 2. Processor characteristics.**

| Name | Number of | | | | |
|---|---|---|---|---|---|
| | Issue Slots | Integer ALUs | Memory Ports | Branch Units | FP ALUs |
| Base | 1 | 1 | 1 | 1 | 1 |
| A | 4 | 2 | 2 | 1 | 1 |
| B | 4 | 4 | 4 | 1 | 4 |
| C | 4 | 4 | 4 | 4 | 4 |
| D | 8 | 8 | 8 | 8 | 8 |

All speedups are reported over the single-issue base processor. For the base processor, ILP optimizations and modulo scheduling are not applied. For the multiple issue processors, code is generated three ways, once without modulo scheduling, once with modulo scheduling of only the single basic block loops, and once with modulo scheduling of superblock loops using the techniques described in this paper. All the code that is not software pipelined is scheduled using acyclic superblock scheduling [10].

None of the loops are unrolled before acyclic scheduling or modulo scheduling. In general, prior unrolling improves the performance of both acyclicly scheduled and modulo scheduled loops [12]. However, there are complex interactions between unrolling, optimization, and scheduling, which add another variable when we are trying to focus on the effect of modulo scheduling. The purpose of the paper is primarily to describe an effective method for modulo scheduling loops in control-intensive non-numeric programs. The results in this paper are used to demonstrate the applicability of modulo scheduling to this class of programs and to validate the correctness of the method. The effects of unrolling prior to scheduling and performance comparisons of modulo scheduling and acyclic scheduling of unrolled loops in control-intensive non-numeric programs are the subject of future work as described in Section 4.

The execution times of the whole programs are calculated using scheduler cycle counts for each basic block and profile information. A 100% cache hit rate is assumed. The benchmarks are profiled after all transformations to insure accuracy. The profiling is done by instrumenting the target (virtual) processor's assembly code and then emulating it

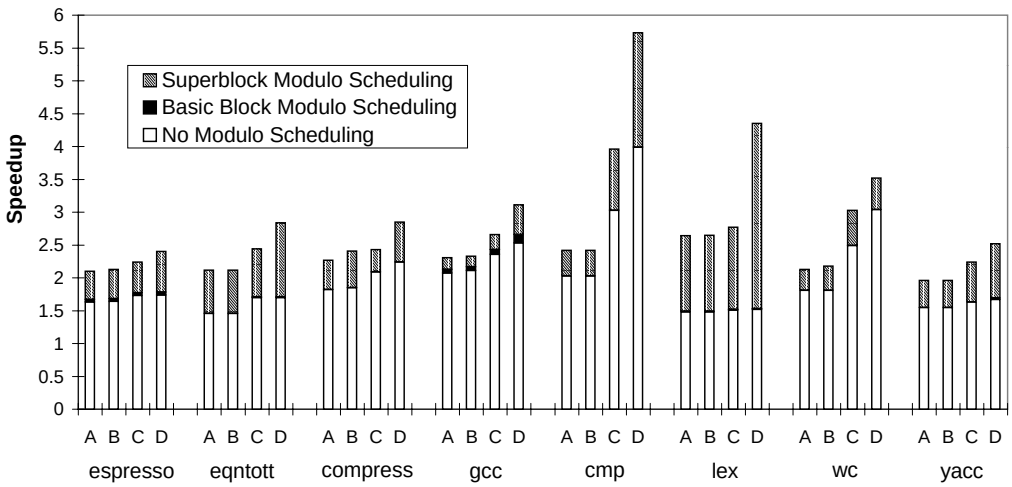


**Figure 14. Speedup over single issue processor with and without modulo scheduling.**

on an HP Series 700 workstation. This execution produces benchmark output which is used to verify the correctness of the target processor's assembly code.

The benchmarks chosen for the experiments are the four SPEC CINT92 and four Unix programs from Table 1 (espresso, eqntott, compress, gcc, cmp, lex, wc, and yacc) that spend the most time in basic block and superblock loops, loops to which we apply modulo scheduling. For all of the chosen programs, over 40% of the dynamic instructions were in such loops. A total of 305 loops were modulo scheduled.

Figure 14 shows the speedup results. The white part of the bars show the speedup over the base processor when acyclic scheduling is applied to all of the code. For *espresso, eqntott, lex,* and *yacc,* the performance is flat as resources are increased. Without overlapping the iterations, the ILP that can be exploited is limited. The black part of the bars show the slightly increased performance when modulo scheduling is applied to the single basic block loops. For all of the benchmarks except *gcc*, less than 6% of the dynamic instructions are in basic block loops. Thus only a slight performance improvement can be expected. The benchmark *gcc* spends about half as much time (14%) in single basic block loops as it does in superblock loops and shows speedups of about 5%.

The cross-hatched part of the bars show the increased performance when superblock modulo scheduling is applied to the eligible loops. Modulo scheduling almost doubles the performance of *lex* for the 4-issue processor and almost triples performance for the 8-issue processor. As was shown in the case study, there is very limited ILP within a single iteration of the loops in that program. Modulo scheduling provides good speedup across all the benchmarks and processors. In particular, speedups of 25% or more are obtained across all the processors for *espresso, eqntott, compress, lex,* and *yacc*. For the most aggressive processor, performance is improved by 30% or more for all the benchmarks except *gcc* and *wc*.

With superblock modulo scheduling, the performance of *espresso, eqntott, lex,* and *yacc* is no longer flat as the processor resources are increased. More ILP is being exploited by overlapping the loop iterations. The results clearly show that modulo scheduling, using the techniques described in this paper, is applicable to control-intensive, non-numeric programs.

## 4. Conclusion

This paper has described a set of methods that allow effective modulo scheduling of loops with multiple exits. These methods can be used to allow modulo scheduling of the selected paths of loops with arbitrary control flow. A case study was presented to show how these methods enable modulo scheduling to be effectively applied to control-intensive non-numeric programs. Performance results for several SPEC CINT92 benchmarks and Unix utility programs demonstrated that modulo scheduling can significantly accelerate loops in this class of programs.

Previous work has shown that unrolling prior to modulo scheduling improves performance for numeric programs [12]. Unrolling enables additional optimization and an effective II that is not an integer. For acyclic scheduling, unrolling is done to allow both optimization and overlap of iterations. For modulo scheduling, unrolling is done to optimize the effective ResMII and RecMII. Much research needs to be done to study the effect of unrolling prior to modulo scheduling for non-numeric programs and to understand the amount of unrolling necessary to achieve the minimum II possible for a given loop. This will be the next step in our effort to apply modulo scheduling to control-intensive non-numeric programs. One result of the next step will be the ability compare modulo scheduling and global acyclic scheduling of unrolled loops within a common framework.

## Acknowledgments

The research for this paper has benefited from conversations with Mike Schlansker and Bob Rau at HP Labs. Thanks to Bob Rau, Scott Mahlke, and Grant Haab for providing feedback on a very early version of this paper and to Brian Deitrich, John Gyllenhaal, and the anonymous referees for their suggestions on the submitted version. The authors would also like to thank Jurgen Mihm, whose work inspired some of our thoughts on modulo variable expansion, and Nancy Warter-Perez, Noubar Partamian, and the past and present members of the IMPACT research group for providing the underlying technology on which the modulo scheduler is built.

This research has been supported by the National Science Foundation (NSF) under grant MIP-9308013, Intel Corporation, Advanced Micro Devices, Hewlett-Packard, SUN Microsystems, NCR, and the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS).

## References

[1] A. Aiken and A. Nicolau. A realistic resource-constrained software pipelining algorithm. In A. Nicolau, D. Galernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 274–290. Pitman/The MIT Press, London, 1991.

[2] R. A. Bringmann. *Compiler-Controlled Speculation*. PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[3] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21(12):1301–1321, December 1991.

[4] P. P. Chang, N. Warter, S. A. Mahlke, W. Y. Chen, and W. W. Hwu. Three architectural models for compiler-controlled speculative execution. *IEEE Transactions on Computers*, 44(4):481–494, April 1995.

[5] 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*, pages 26–38, April 1989.

[6] K. Ebcioglu and T. Nakatani. A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture. In *Languages and Compilers for Parallel Computing*, pages 213–229, 1989.

[7] D. M. Gallagher. *Memory Disambiguation to Facilitate Instruction-Level Parallelism Compilation*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[8] J. C. Gyllenhaal. A machine description language for compilation. Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[9] R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W. W. Hwu. Superblock formation using static program analysis. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, December 1993.

[10] 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 technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.

[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*, pages 318–328, June 1988.

[12] D. M. Lavery and W. W. Hwu. Unrolling-based optimizations for modulo scheduling. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 327–337, Nov. 1995.

[13] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling: A model for compiler-controlled speculative execution. *Transactions on Computer Systems*, 11(4), November 1993.

[14] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, December 1992.

[15] M. Rajagopalan and V. H. Allan. Efficient scheduling of fine grain parallelism in loops. In *Proceedings of the 26th International Symposium on Microarchitecture*, pages 2–11, December 1993.

[16] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 63–74, December 1994.

[17] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *The Journal of Supercomputing*, 7(1):9–50, January 1993.

[18] 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*, pages 183–198, October 1981.

[19] 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*, pages 158–169, December 1992.

[20] P. Tirumalai, M. Lee, and M. Schlansker. Parallelization of loops with exits on pipelined architectures. In *Supercomputing*, November 1990.

[21] 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*, pages 170–179, December 1992.