Enhanced Modulo Scheduling for Loops with Conditional Branches

Nancy J. Warter Grant E. Haab *
Krishna Subramanian

Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801

John W. Bockhaus

Hewlett-Packard
Systems Technology Division
Fort Collins, CO 80525

Abstract

Loops with conditional branches have multiple execution paths which are difficult to software pipeline. The modulo scheduling technique for software pipelining addresses this problem by converting loops with conditional branches into straight-line code before scheduling. In this paper we present an Enhanced Modulo Scheduling (EMS) technique that can achieve a lower minimum Initiation Interval than modulo scheduling techniques that rely on either Hierarchical Reduction or If-conversion with Predicated Execution. These three modulo scheduling techniques have been implemented in a prototype compiler. We show that for existing architectures which support one branch per cycle, EMS performs approximately 18% better than Hierarchical Reduction. We also show that If-conversion with Predicated Execution outperforms EMS assuming one branch per cycle. However, with hardware support for multiple branches per cycle, EMS should perform as well as or better than If-conversion with Predicated Execution.

1 Introduction

Software pipelining has been shown to be an effective technique for scheduling loop intensive programs on VLIW and superscalar processors [1] [2] [3] [4]. The principle behind software pipelining is to overlap or pipeline different iterations of the loop body in order to expose sufficient operation-level parallelism to exploit the underlying parallel hardware. The resulting software pipeline schedule consists of a prologue, a kernel, and an epilogue. The prologue initiates the first p iterations. After the first $p * II$ cycles, where II is the *Initiation Interval*, a steady state is reached. In the steady state or kernel, one iteration is completed every II cycles. After the kernel finishes execution, the epilogue completes the last p iterations. For loops with large trip counts, most of the execution time is spent executing the kernel. Thus, the goal of software pipelining techniques is to find the smallest possible kernel, or equivalently, II.

*Supported by a Fannie and John Hertz Foundation Graduate Fellowship.

The two basic approaches to software pipelining differ in the way that the operations are scheduled and resource constraints are applied. The first approach is based on global code compaction which schedules an operation as early as possible and enforces resource constraints after a steady state is found. Techniques which utilize this approach are Perfect Pipelining [5], Enhanced Pipeline Scheduling [6], and GURPR* [7]. The second approach, Modulo Scheduling, uses the resource and recurrence constraints to determine a tight lower bound on II and delays operations in order to resolve resource conflicts [8] [9] [10] [11]. For loops without conditional branches and cross-iteration dependences, Rau and Glaeser have proven that Modulo Scheduling will yield an optimal schedule for certain resource constraints [9]. Furthermore, for loops without conditional branches, Modulo Scheduling should perform better than the global compaction approaches, which may misschedule the resources. Jones and Allan empirically showed that Modulo Scheduling performs better than Enhanced Pipeline Scheduling for single basic block loops [12].

These two approaches also differ in the way that loops with conditional branches are handled. Loops with conditional branches are difficult to software pipeline because there are multiple paths of execution to schedule. The global code compaction approach uses Fisher's code motion rules [13] to guarantee that operations are scheduled properly in the presence of conditional branches. The Modulo Scheduling approach avoids the need for such operations by transforming the loop into straight-line code before scheduling.

Two techniques, Hierarchical Reduction [3] and If-conversion with Predicated Execution [14] [15], have been proposed to convert loops with conditional branches into straight-line code. Hierarchical Reduction collapses conditional constructs (e.g. if-then-else) into pseudo-operations by list scheduling both paths of the conditional construct and merging them into one path by taking the union of the resource usages along each path [3] [10]. Hierarchical Reduction does not assume special hardware support. Thus, after modulo scheduling, the code is regenerated by expanding the pseudo-operations and copying other op-

erations scheduled during the same cycle to both paths of execution. If-conversion removes conditional branches by computing a condition for the execution of each operation [16] [17]. Predicated Execution is an architecture feature that supports conditional execution by providing predicate registers that hold the condition for execution of the operations. A predicate register is specified for each operation, and predicate define operations are used to set the predicate registers based on the appropriate condition [14] [15].

The Modulo Scheduling technique first determines a tight lower bound on II based on the resource usages and cross-iteration dependence cycles. In this paper, *minimum* II refers to this lower bound. In the presence of conditional branches, the minimum II due to resource usages is determined by the most heavily used resource along any execution path. Both Hierarchical Reduction and If-conversion with Predicated Execution place restrictions on the operation scheduling that may prevent Modulo Scheduling from achieving this II. If-conversion with Predicated Execution schedules operations along all execution paths together. Thus, the minimum II for Predicated Execution is constrained by the sum of the resource usages of all loop operations rather than those along the most constrained execution path. Although Hierarchical Reduction takes the union of the resource usages along both paths of a conditional construct, it restricts the code schedule by first list scheduling the operations along both paths. This creates pseudo-operations with complicated resource usage patterns. These pseudo-operations are more likely to have resource conflicts which prevent finding a schedule for the minimum II.

In this paper we present an Enhanced Modulo Scheduling (EMS) technique that schedules loops with conditional branches in such a way that the minimum II can be obtained. Essentially, EMS uses If-conversion with no additional hardware support for conditional execution. Thus, like Hierarchical Reduction, the code must be regenerated by inserting conditional branches after modulo scheduling. EMS combines the benefits of both previous techniques, since If-conversion eliminates the need for prescheduling conditional constructs, and regeneration eliminates the need to sum the resource constraints from all execution paths. Also, like Modulo Scheduling with Hierarchical Reduction [3], EMS does not require special hardware support. Therefore, it can be used on existing processors which do not provide hardware support for predicated execution.

In this paper we present the EMS algorithm and compare the performance of the three Modulo Scheduling techniques: EMS, Hierarchical Reduction, and Predicated Execution. In addition, we compare the performance of the Modulo Scheduling techniques against a global code compaction technique, GURPR* [7]. We also discuss the fixed-II limitation [18] of these techniques and some of the possible solutions currently being explored. Although the methods discussed in this paper can be used for both VLIW and superscalar processors, we will use VLIW terminology to clarify the discussion. Thus, an instruction refers to a very long instruction word which contains multiple operations.

2 Enhanced Modulo Scheduling

The EMS algorithm consists of five steps:

1. Apply If-conversion to convert the loop body into straight-line predicated code.
2. Generate the data dependence graph.
3. Modulo schedule the loop body.
4. Generate the software pipeline stages using modulo variable expansion to rename overlapping register lifetimes.
5. Regenerate the explicit control structure of the code by inserting conditional branch operations.

While the EMS algorithm can handle more complicated control flow graphs[1], due to space limitations, the algorithm presented in this paper applies to loops with structured control flow graphs. The more general EMS algorithm is presented in [19].

2.1 If-conversion

Before modulo scheduling can be performed, the code must be converted into straight-line code by removing conditional branches and thus control dependences. If-conversion is a technique to convert control dependences into data dependences by computing a condition for the execution of each operation [16] [17]. To support If-conversion, EMS uses an internal predicated representation similar to the one used in the compiler for the Cydra 5 processor, which has explicit hardware support for Predicated Execution [14] [15]. Additional features have been added to the representation to account for the fact that there is no explicit hardware support [19].

The RK algorithm, developed by Park and Schlansker, is used to perform If-conversion using predicates [20]. Conditional branches are replaced by predicate define operations, and the basic blocks are assigned the appropriate predicates. Each basic block is assigned only one predicate, which is defined for all the operations in the block. Each predicate has both a true and a false form. For the conditional branch in a simple if-then-else construct, the fall-through basic block is assigned the false predicate and the branch-target basic block is assigned the true predicate. The predicate define operation sets the true (false) predicate and clears the false (true) predicate if the branch condition is true (false).

Figure 1 shows an example loop control flow graph and Figure 2 shows how the RK algorithm would predicate this

[1] This is another benefit of If-conversion over Hierarchical Reduction, the application of which is limited to if-then-else constructs.

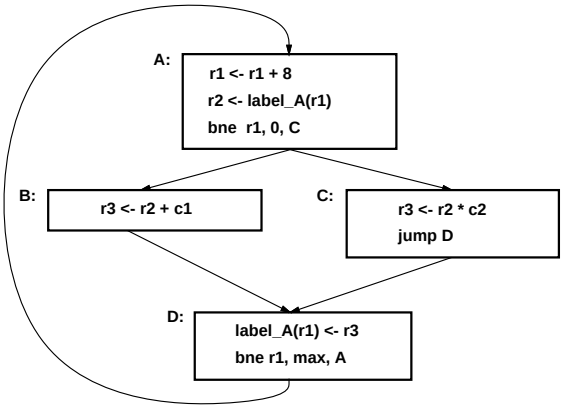

Figure 1: Example loop before If-conversion.

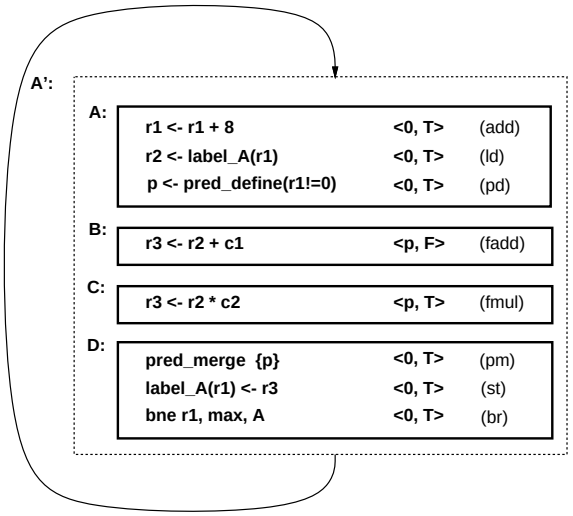

Figure 2: Example loop after If-conversion.

graph. Each node in Figure 1 represents a basic block. In Figure 2, predicate define operations have been inserted into the basic blocks, and the basic blocks have been assigned the correct predicates. A predicate has an *id* and *type*, represented as $< id,\ type >$. For example, basic block **B** has a predicate with $id = \mathbf{p}$ and $type = \mathbf{F}$ (false) and basic block **C** has a predicate $< \mathbf{p}, \mathbf{T} >$. The predicate define operation in basic block **A** will set (clear) the predicate $< \mathbf{p}, \mathbf{T} >$ ($< \mathbf{p}, \mathbf{F} >$) if **r1 != 0** and clear (set) the predicate $< \mathbf{p}, \mathbf{T} >$ ($< \mathbf{p}, \mathbf{F} >$) if **r1 == 0**. Note that basic blocks **A** and **D** have the default predicate $< \mathbf{0}, \mathbf{T} >$, which is always set. After predication, all the basic blocks can be merged into one basic block, $\mathbf{A}'$.

To find the tightest software pipeline schedule, all operations should be available during modulo scheduling. Adding operations after scheduling may unnecessarily increase the size of the kernel (II). Thus, operations required for code regeneration should be explicit.

During code regeneration, predicate define operations are replaced by the corresponding conditional branches which create two paths. Separate paths are only needed for the lifetime of the predicate. Once the predicate is no longer live (i.e., there are no further uses of the predicate), its paths can be merged. Although correct code can be generated without merging, this causes unnecessary code expansion. Alternatively, this information can be determined during code regeneration by performing live-variable analysis [21] and inserting a jump operation once the predicate is no longer live. However, this prevents scheduling the jump operation during modulo scheduling. Thus, a predicate merge operation is used that has the scheduling attributes of a jump operation. During predication, a predicate merge operation is inserted at the beginning of a basic block if the basic block has multiple predecessors, and it post-dominates [21] all of its predecessors. A predicate merge operation is inserted at the beginning of basic block **D** since **D** post-dominates both **B** and **C**. The predicates being merged are those of the post-dominated basic blocks. In the example, the predicate merge operation in **D** merges predicate **p**.

2.2 Dependence Graph Generation

Only operations which have a control path between them can be dependent on one another. In the original loop body, operations along different control paths are in different basic blocks with no path of control connecting them. After If-conversion, the loop body is reduced to one basic block. Thus, predicates need to be used to determine whether there is a control path between two operations in one iteration. There is always a control path between two operations from different iterations. A *Predicate Graph* is used to determine if there is a control path between two operations within one iteration [22].

After determining that there is a control path between two operations, the dependence relation (flow, anti, output) is determined [21]. The dependences arising from the predicate operations are determined by the following rules: There is a flow dependence between predicate define operations and the operations that are assigned that predicate. There is an output dependence between the predicate define operation and predicate merge operation for a given predicate. There is an anti dependence between operations assigned a predicate and the respective predicate merge.

Figure 3 shows the dependence graph for the example in Figure 2. The arcs are labeled with the tuple $< type,\ distance,\ latency >$. The type is either flow (f), anti (a), or output (o). The distance is the number of iterations the dependence spans. The latency is the minimum number of cycles needed to satisfy the dependence (with respect to issue times). For this example we assume that the `fadd` and `fmul` have a two cycle latency, and `ld` operations have a three cycle latency. The remaining operations have a single cycle latency. Note that there is no output dependence between the `fadd` and `fmul` since there is no

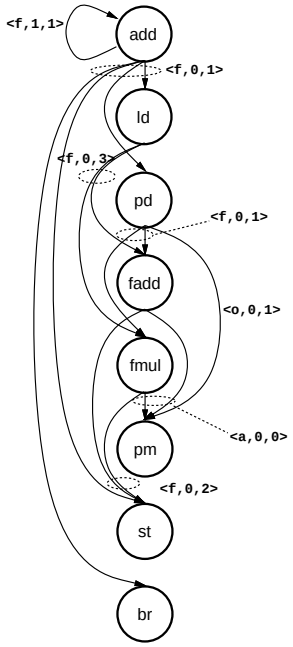

Figure 3: Simple example loop dependence graph.

control path between predicates $< \mathbf{p}, \mathbf{F} >$ and $< \mathbf{p}, \mathbf{T} >$. Also note that the predicate merge operation (`pm`) is output dependent on `pd` and anti dependent on `fadd` and `fmul`. Cross-iteration anti dependences are not shown in Figure 3 since they are removed by register renaming during modulo variable expansion.

2.3 Modulo Scheduling

In a modulo-scheduled software pipeline, a loop iteration is initiated every II cycles, where II is the *Initiation Interval* [9] [3] [11] [23]. The II is constrained by the most heavily utilized resource and the worst-case recurrence for the loop. These constraints each form a lower bound for II. The minimum II is the maximum of these lower bounds. In the EMS approach, the minimum II due to resource constraints is determined by the most heavily utilized resource along any execution path. If an execution path p uses a resource r for c_{pr} cycles and there are n_r copies of this resource, then the minimum II due to resource constraints, RII, is

$$RII = \max_{p \in P} \left(\max_{r \in R} \left\lceil \frac{c_{pr}}{n_r} \right\rceil \right),$$

where P is the set of all execution paths and R is the set of all resources.

Cross-iteration dependences can cause recurrences that force a maximum latency for the operations on the recurrence path or dependence cycle. If a dependence edge, e, in a cycle has latency l_e and connects operations that are d_e iterations apart, then the minimum II due to dependence cycles, CII, is

$$CII = \max_{c \in C} \left\lceil \frac{\sum_{e \in E_c} l_e}{\sum_{e \in E_c} d_e} \right\rceil,$$

where C is the set of all dependence cycles and E_c is the set of edges in dependence cycle c.

Once the minimum II has been determined, operations are scheduled using the basic algorithm outlined by Rau and Glaeser [9] with improvements from the Cydra 5 compiler implementation [23]. Before any operations are scheduled, the loop-back branch is fixed in the last cycle of the schedule (assuming no branch delay slots). The remaining operations are scheduled according to their priority and dependence constraints, such that all higher-priority operations are scheduled before lower-priority operations. The scheduling priorities are assigned such that the most constrained operations have the highest priority and the least constrained have the lowest priority. Thus, the operations in a dependence cycle have the highest priority. Operations that have predecessors and successors which are in a dependence cycle have the next highest priority. The remaining operations have the lowest priority. Operations involved in dependence cycles and operations constrained from above and below by operations in dependence cycles are scheduled using heuristics developed for the Cydra 5 compiler [23]. Once an operation is ready to schedule, it is scheduled at the earliest start time[2] unless there is a resource conflict. If there is a resource conflict, it is scheduled in the next available slot as discussed below.

If a schedule cannot be found for the minimum II, II is incremented and the scheduling process is repeated. This iterative scheduling process proceeds until the II reaches a predetermined upper limit, at which time the loop is considered to be unfit for software pipelining.

During scheduling, resource conflicts are identified using a modulo resource reservation table [3]. Figure 4 shows the modulo resource reservation table after the operations in the example of Figure 2 have been scheduled. For this example, the processor has two uniform functional units with the exception that only FU2 has a branch unit. That is, any operation except a branch operation can be executed in either functional unit. The minimum II due to resources, RII, is the maximum of the uniform resource constraints and the branch constraint. That is, $RII = \max(\lceil \frac{7}{2} \rceil, \lceil \frac{3}{1} \rceil) = 4$. Since the only recurrence is for the `add` operation, which has a self-flow dependence with unit distance and unit latency, $CII = 1$. Thus, $II = \max(RII, CII) = 4$. There are II rows in the modulo resource reservation table and a column for each functional unit. In order to schedule an operation that uses a

[2] When scheduling forward (backwards), i.e. after all predecessors (successors) have been scheduled, an operation is scheduled at the earliest (latest) start time. For brevity, we use earliest start time throughout this paper.

Cycle Mod II	FU1 (no branch)	FU2 (w/ branch)
0	add **s = 0**	fadd <p,F> / fmul <p,T> **s = 4**
1	ld **s = 1**	pd **s = 1**
2	st **s = 6**	pm **s = 6**
3		br **s = 3**

Figure 4: Modulo resource reservation table of example loop. **s** is the start time of the operation in cycles.

functional unit i at time t_j, there must not be a resource conflict at row t_j mod II and column FUi.

There are three possible states for each slot in the reservation table: empty, no-conflict, and full. A slot is empty if no resources have been scheduled in that slot. A no-conflict slot occurs when there is no control path between the operation being scheduled and operations that have already reserved the slot. For instance, operations from one iteration that are from different paths of a conditional branch can be scheduled in the same slot. In Figure 4, the `fadd` and `fmul` operations do not conflict since they are from the same iteration (i.e., they have the same start time) and they have complimentary predicates. A slot is full with respect to the operation being scheduled if there is a control path between the operation and the operations that have already reserved the slot.

To find the tightest schedule, first determine if there are any no-conflict slots in the modulo resource reservation table for this operation. If there are, select the earliest available slot with respect to the earliest start time for that operation. Otherwise, schedule the operations in the earliest available empty slot. Note in Figure 4 that operation `pm` is ready to be scheduled in cycle 5. However, there is a resource conflict, and thus, it must be delayed until cycle 6. This delaying of operations to satisfy resource conflicts is the essence of Modulo Scheduling.

EMS has more scheduling freedom than other Modulo Scheduling techniques. Hierarchical Reduction will list schedule the if-construct to form a pseudo-operation. For example, in Figure 1, the conditional branch in basic block **A** and the operations in basic blocks **B** and **C** are list scheduled to form a pseudo-operation that is then modulo scheduled with the other operations in the loop. Pseudo nodes tend to have more complicated resource patterns than individual operations causing more resource conflicts. Since EMS does not preschedule the operations before modulo scheduling, there are fewer resource conflicts.

With Predicated Execution, all operations in the loop are fetched, and those with their predicates set complete execution. Thus, whereas both Hierarchical Reduction and EMS can schedule two operations from different control paths in the same slot, Predicated Execution only allows one operation per slot. For this reason, `fadd` and `fmul` cannot be scheduled in the same slot as they are in Figure 4. Compared to EMS, Predicated Execution has a larger number of operations along the critical path[3].

2.4 Software Pipeline Generation with Modulo Variable Expansion

At this point, the steady state or kernel of the software pipeline has been scheduled. It consists of one II. Before generating the rest of the software pipeline, we have to determine if the kernel needs to be unrolled to avoid overlapping register lifetimes. Since one loop iteration can span multiple II's, the lifetime of a register can overlap itself. To guarantee that a value in a register is not overwritten, the loop body must be unrolled enough times to satisfy the longest register lifetime. Then the overlapping registers are renamed. This optimization is called modulo variable expansion [3]. The lifetime of a predicate variable may also overlap itself. Although these variables do not map to physical registers, they are also renamed in order to regenerate the code properly[19].

After renaming, the kernel has been unrolled u times. Next, the stages of the prologue and epilogue are generated, where each stage has II cycles. The number of stages in the prologue (and epilogue), p, is $\lceil \frac{latest\ issue\ time}{II} \rceil - 1$, where the latest issue time is defined by the modulo schedule over all operations in the loop.

After all stages are created, the loop back branch is removed from all but the last stage in the kernel. Allowing early exits from the loop requires special epilogues for each stage in the prologue and kernel, which increases the code generation complexity and code expansion considerably. With only one exit from the loop, the software pipelined loop must execute $p + k * u$ times, where k is an integer greater than or equal to one. A non-software pipelined version of the loop is required to execute the remaining number of iterations. If the loop trip count is greater than $p + u$, the remaining number of iterations is $(trip\ count - p)$ mod u. If the trip count is less than $p + u$, only the non-software pipelined loop is executed. If the trip count is known to be less than $p+u$ at compile time, the software pipeline is not generated.

2.5 Code Regeneration

Once each stage of the pipeline has been generated and the registers have been renamed, the basic block representation must be regenerated by replacing the predicate define operations with conditional branches.

[3]EMS does require additional jump instructions that are not needed for Predicate Execution. However, Predicate Execution may require additional operations such as predicate clears that are not needed for EMS.

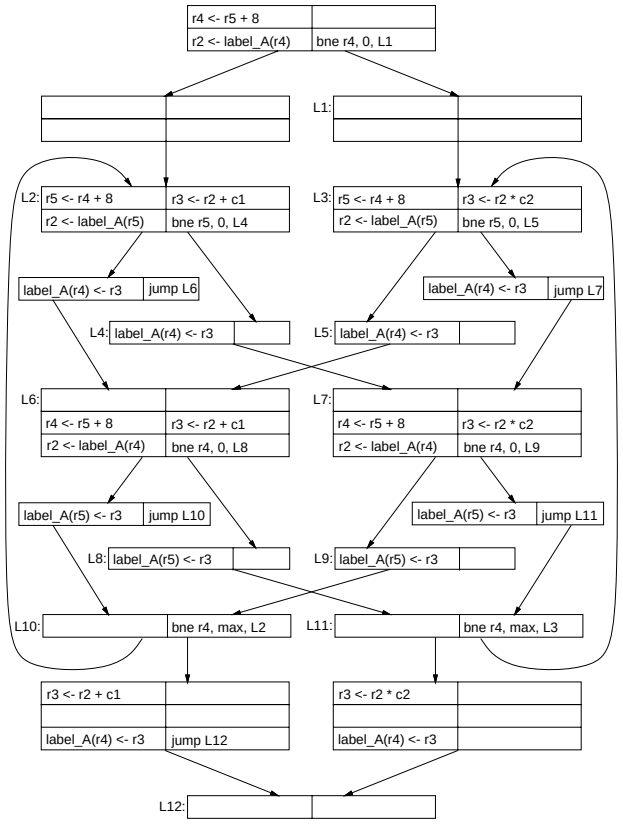

Figure 5: The software pipelined loop after regeneration.

The regeneration of the software pipeline for the example loop is shown in Figure 5. Each node is a basic block, and each row in a node is a VLIW instruction. Note that the kernel must be unrolled two times since the longest register lifetime modulo *II* is two. Register **r1** is the only register that must be renamed. For a VLIW processor without interlocking, the empty operation slots are filled with no-op's. For a VLIW processor with interlocking, instructions which consist of only empty slots are deleted. For partially full instructions, the empty operation slots are filled with no-op's[4].

3 Experimental Results

3.1 Compiler Support

The three Modulo Scheduling techniques, referred to as EMS, Hierarchical Reduction, and Predicated Execution, have been implemented in the IMPACT C compiler. In order to compare Modulo Scheduling against another *fixed-II* software pipelining technique, GURPR* [7] has also been implemented in the IMPACT compiler [24]. Like EMS and Hierarchical Reduction, GURPR* does not require additional hardware support.

In the GURPR* algorithm, the loop body is compacted and pipelined assuming a minimum *II* determined by the inter-body dependence distance. From this intermediate pipeline representation, *II* is determined as the shortest interval that contains all operations in the loop. Once this interval is determined, it may contain multiple copies of an operation. Any redundant operations are deleted so that exactly one iteration is completed within one *II*. In our implementation of GURPR* no global code compaction is performed. We found that techniques such as trace scheduling and code percolation tend to increase both the longest path and the resource conflicts, thereby increasing *II*. It is possible that some heuristics could be applied to these code compaction techniques to improve the performance of GURPR*.

For each technique, scheduling is applied to the appropriate loops after performing classical code optimizations [21] and after translation into the target machine assembly code but before register allocation. In our current implementation, we apply software pipelining to inner loops that do not have function calls or early exits from the loop (e.g., return statements).

3.2 Machine Model

The machine model for these experiments is a VLIW processor with no interlocking. There are uniform resource constraints with the exception that only one branch can be issued per cycle. Other than the branch operation, we use the instruction set and operation latencies of the Intel i860[5]. Most integer operations take 1 cycle except for the integer load which takes 2 cycles. The integer multiply and divide and the floating point divide are implemented using approximation algorithms [25]. The floating point load, ALU, and single-precision multiply take 3 cycles, and the double precision multiply takes 4 cycles. For the branch operation we assume that the compare and branch are performed in 1 cycle. Thus, there are no branch delay slots. There are four basic kinds of compare and branch operations: equal, not equal, greater than, and greater than or equal. For each kind there are three types: integer, single-precision floating point, and double-precision floating point. There are also signed and unsigned versions of the integer greater than and greater than or equal operations. In total, there are 14 types of branch and compare operations.

Likewise, for the Predicated Execution model, there are 14 predicate define operations and a predicate clear operation to reset the predicates. These operations have one cycle latency. The architecture support for Predicated Execution is similar to the model used in the Cydra 5 [15]. There is a predicate register file and hardware to prevent the write back of results for operations whose predicate is

[4] For a superscalar processor, the empty slots can be ignored.

[5] We assume that the load and floating point pipelines are automatically advanced.

cleared in the predicate register file. Each operation in a VLIW instruction has a predicate register specifier.

The base processor for these experiments is a RISC processor with an infinite register file and ideal cache. For the predicated execution model, the predicate register file size is unlimited. The base schedule is a basic block schedule. The experiments were performed using machines with instruction widths or issue rates of 2, 4, and 8.

3.3 Benchmarks

The focus of this study is to analyze the relative performance of conditional branch handling techniques. To run our experiments, we collected a set of 26 loops with conditional branches from the Perfect benchmarks. All conditional constructs are structured and non-nested. Of the 26 loops considered, 18 have one conditional construct, 7 have two constructs, and 1 has three constructs. Only *DOALL* loops (loops without cross-iteration memory dependencies) were included in the test suite.

3.4 Results

Performance

Given equivalent resource constraints, EMS should perform better than both Predicated Execution and Hierarchical Reduction since Predicated Execution needs to fetch operations from every execution path through the loop and Hierarchical Reduction has pseudo-operations with complicated resource usage patterns. Figure 6 shows the speedup of the four techniques for issue 2, 4, and 8, where each point is the harmonic mean of the speedup of the individual loops. EMS performs 18%, 17%, and 19% better than Hierarchical Reduction for issue rates 2, 4, and 8, respectively. Due to the one branch per cycle restriction for EMS, Predicated Execution performs 2%, 6% and 27% better than EMS for issue rates 2, 4, and 8, respectively.

The GURPR* technique may have redundant operations that are removed after II has been determined. Thus, the resource constraints are not entirely known while the loops are being overlapped. As shown in Figure 6, GURPR* tends to have a larger II than the Modulo Scheduling techniques which apply the exact resource constraints.

Figure 7 shows the arithmetic mean of RII and the achieved II for the four techniques. Since loops without cross-iteration memory dependences were used, the minimum II is always RII[6]. The graph shows the number of cycles per II, and thus, the size of the bar is inversely proportional to the performance of the technique. Figure 7 illustrates why EMS performs better than both Hierarchical Reduction and GURPR* and not as well as Predicated Execution.

[6]An optimization, *induction variable reversal* [26] is performed to transform induction variable recurrences into self-recurrences. Thus, CII is one for the loops used in this paper.

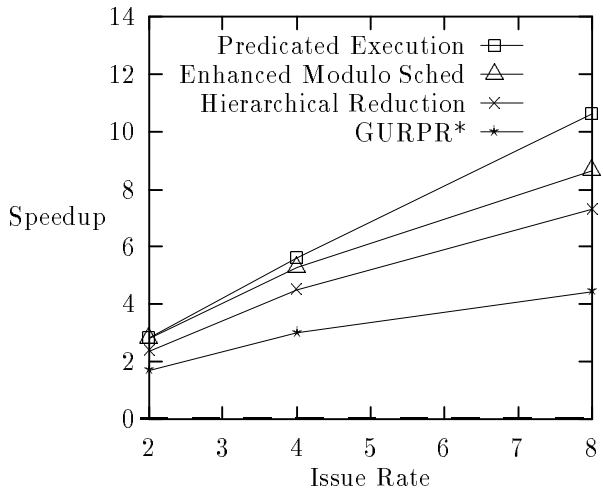

Figure 6: Speedup of the three Modulo Scheduling techniques and GURPR*.

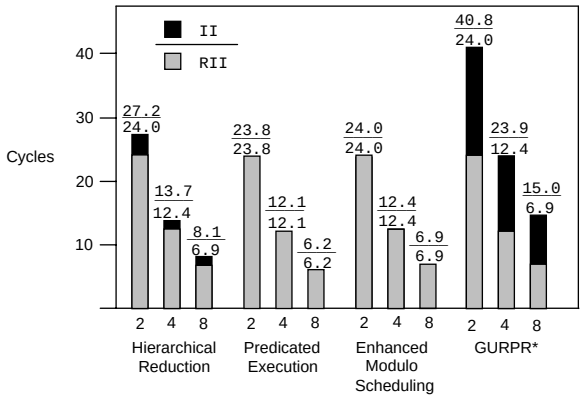

Figure 7: Average minimum II (RII) versus achieved II.

For EMS, RII is determined by the most heavily utilized resource along any execution path. This minimum II may not be achieved if operations from different execution paths have start times from different II's, which prevent them from using the same resource. In the worst case, the resource usages are summed as in Predicated Execution. Our results indicate that EMS almost always achieves its minimum II. Although the precision in Figure 7 does not show the differences, the average II is 0.1% and 0.3% percent larger than the the average RII for issue rates 2 and 4, respectively.

For Hierarchical Reduction, RII is determined after the operations in the conditional construct have been list scheduled and the pseudo-operations have been formed by taking the union of the resource usages along both paths. The union of the resource usages after list scheduling is always greater than or equal to the most constrained re-

source along either path. Figure 7 shows that, on average, *RII* is larger for Hierarchical Reduction than for EMS. In addition to having a larger average *RII*, Hierarchical Reduction often cannot achieve this minimum due to the complex resource usage patterns of the pseudo-operations.

Note that *RII* for EMS is larger than *RII* for Predicated Execution. This is because the one branch per cycle constraint often limits the minimum *II* for EMS (especially for issue-8 machines). If the underlying architecture supports multiple branches per cycle, EMS should perform as well as or better than Predicated Execution since EMS almost always achieves its minimum *II*.

While GURPR* does not use *RII* to determine the minimum *II* during the pipelining stage, it does represent a lower bound for the achieved *II*. EMS and GURPR* have the same minimum *II* since neither has hardware support and neither creates pseudo-operations that may increase *RII*. However, due to the insertion of cycles when a resource conflict arises, GURPR* rarely achieves this minimum *II*.

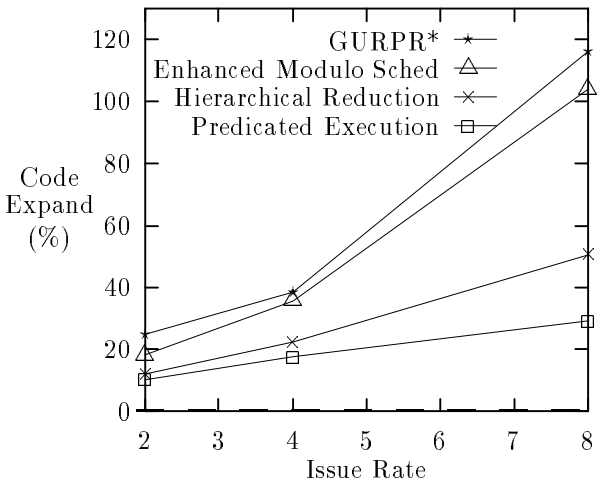

Figure 8: Code Expansion of the three Modulo Scheduling techniques and GURPR*.

	#FP-Ops	*#Branches*	*#Ops*
Avg.	1.85	1.35	52.62
`padec`	6	1	166
`wcont`	1	1	15

Table 1: Benchmark loop characteristics before software pipelining.

Code Expansion

The disadvantage of EMS and other techniques that require explicit conditional branches is that there may be multiple copies of an operation, each on a different control path. Furthermore, after software pipelining, a conditional construct can overlap itself. If it overlaps itself n times, there can be order 2^n times code expansion. Figure 8 shows the arithmetic mean of the code expansion of the four techniques. The code expansion is determined by the number of instructions in the software pipeline (after code regeneration for EMS, Hierarchical Reduction, and GURPR*) divided by the number of instructions in the basic block schedule. The code expansion due to the additional non-software pipelined loop is not included. However, this will add the same number of instructions for all techniques. Also, for Predicated Execution, the code expansion due to the predicate register specifiers has not been included since it is implementation dependent.

As expected, the code expansion for EMS is larger than for Predicated Execution due to code regeneration and larger than for Hierarchical Reduction since forming a tighter schedule will overlap more conditional constructs. The code expansion for EMS is 52%, 60%, and 105% larger than for Hierarchical Reduction and is 75%, 103%, and 257% larger than for Predicated Execution for issue 2, 4, and 8 respectively. If the underlying architecture supported multiple branches per cycle, the code expansion for EMS would be even larger.

Even though GURPR* does not perform as well as the other techniques, it has the largest code expansion. This is because the original *II* is equal to one for loops without cross-iteration memory dependences and after induction variable reversal. Thus, many iterations are overlapped during the pipelining phase before the final *II* is determined [24]. This increases the overlap of the conditional constructs resulting in larger code expansion.

Tables 1 and 2 are provided to add insight into how the loop characteristics and scheduling parameters affect code expansion. In these tables, data for the loops are presented which correspond to the worst case (`padec`) and best case (`wcont`) code expansion (with respect to EMS) as well as for the average over all loops. In Table 1, *#FP-Ops* is the number of floating point operations in the longest path of a conditional construct, *#Branches* refers to the number of conditional branches within the loop body (not including the loop-back branch), and *#Ops* refers to the number of operations in the loop. In Table 2, four scheduling parameters are presented: the number of stages in the prologue, the number of times the kernel is unrolled, the number of VLIW instructions (assuming no interlocking), and *II*.

With hardware support for predicated execution, the number of VLIW instructions assuming no interlock is $(2 * Stages + Unroll) * II$. Without hardware support, the number of VLIW instructions cannot be directly calculated since it depends on the degree of overlapping of the conditional constructs. The code expansion for the techniques that use regeneration can be determined by comparing their scheduling parameters against those of Predicated Execution. For example, considering EMS for the worst case loop (`padec`), although *Stages* and *Unroll* are identical and *II* is similar, *VLIW* for EMS is 31% larger than *VLIW* for Predicated Execution.

Technique	Param.	Avg.	padec	wcont
Hierarch. Reduction	*Stages*	4.50	7	3
	Unroll	5.76	8	4
	VLIW	301.88	1202	47
	II	13.73	41	4
Predicated Execution	*Stages*	5.46	6	3
	Unroll	6.88	8	4
	VLIW	235.62	840	40
	II	12.08	42	4
Enhanced Modulo Scheduling	*Stages*	5.34	6	3
	Unroll	6.50	8	4
	VLIW	416.77	1100	47
	II	12.42	41	4
GURPR*	*Stages*	3.27	4	3
	Unroll	4.27	5	4
	VLIW	397.69	1360	57
	II	23.85	88	5

Table 2: Scheduling parameters for the four software pipelining techniques.

4 Future Work

In this paper we have introduced the EMS technique and shown how EMS can produce a tighter schedule than Modulo Scheduling with either Hierarchical Reduction or Predicated Execution. However, more work is required to complete the analysis of the three Modulo Scheduling techniques. We are currently expanding the regeneration algorithm to handle architectures which support multiple branches per cycle. Furthermore, we are analyzing the performance impact of code expansion using a non-ideal instruction cache. Finally, we are studying the effect of a finite register file using the first-fit allocation with conflict-ordering register allocation technique proposed by Rau et. al. [27].

There is a limitation of the fixed-*II* techniques that variable-*II* techniques [6] do not encounter [18]. Fixed-*II* techniques create gaps in the schedule for execution paths not constrained by the minimum *II*. We are currently investigating several solutions to this problem. One possible solution is to perform global code compaction after modulo scheduling. This will also remove gaps formed after unrolling for modulo variable expansion such as those created by removing the loop-back branch from all but the last stage in the kernel. Another possible solution is to use profiling information to identify the most frequently executed path, which is then software pipelined [28]. This solution is particularly beneficial for loops in which the most frequently executed path is much shorter than the other paths. In the loops we have studied, this is often the case since the longer paths often correspond to exception handling code or code which calculates a boundary condition.

5 Conclusion

In this paper we have presented a new technique for modulo scheduling that converts loops with conditional branches into straight line code by applying If-conversion assuming no conditional execution hardware support. This technique, Enhanced Modulo Scheduling (EMS), combines the benefits of the previous Modulo Scheduling techniques, If-conversion with Predicated Execution and Hierarchical Reduction, to remove the limitations of each technique. If-conversion eliminates the need for prescheduling the paths of the conditional construct, which is essential to Hierarchical Reduction. Code regeneration eliminates the need to sum the resource usages from all execution paths as required in If-conversion with Predicated Execution.

All three Modulo Scheduling techniques and GURPR* have been implemented in a prototype compiler. For existing architectures which support one branch per cycle, EMS performs approximately 18% better than Modulo Scheduling with Hierarchical Reduction. For future architectures, EMS with hardware support for multiple branches per cycle should perform as well as or slightly better than Modulo Scheduling with Predicated Execution. Given the approximately equivalent performance of these two techniques, future designers can determine the more cost-effective hardware implementation.

Acknowledgements

The authors would like to thank Wen-mei Hwu, Paul Ryan, Jeff Baxter, and all members of the IMPACT research group for their comments and suggestions. The authors would also like to acknowledge Scott Mahlke, David Lin, and William Chen for their contributions of compiler support for predicated execution. Special thanks to the anonymous referees whose comments and suggestions helped to improve the quality of this paper significantly.

This research has been supported by the Joint Services Engineering Programs (JSEP) under Contract N00014-90-J-1270, Dr. Lee Hoevel at NCR, the AMD 29K Advanced Processor Development Division, Matsushita Electric Industrial Corporation Ltd., Hewlett-Packard, 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. Charlesworth, "An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family," in *IEEE Computer*, September 1981.

[2] R. Touzeau, "A Fortran compiler for the FPS-164 Scientific Computer," in *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, June 1984.

[3] 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.

[4] R. L. Lee, A. Kwok, and F. Briggs, "The floating point performance of a superscalar SPARC processor," in *Proceedings of the 4th International Conference on Architecture Support for Programming Languages and Operating Systems*, pp. 28–37, April 1989.

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

[6] K. Ebcioglu and T. Nakatani, "A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture," in *Second Workshop on Languages and Compilers for Parallel Computing*, August 1989.

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

[8] J. H. Patel and E. S. Davidson, "Improving the throughput of a pipeline by insertion of delays," in *Proceedings of the 3rd International Symposium on Computer Architecture*, pp. 159–164, 1976.

[9] 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.

[10] M. Lam, *A Systolic Array Optimizing Compiler*. PhD thesis, Carnegie Mellon University, Pittsburg, PA, 1987.

[11] C. Eisenbeis, "Optimization of horizontal microcode generation for loop structures," in *International Conference on Supercomputing*, pp. 453–465, July 1988.

[12] R. B. Jones and V. H. Allan, "Software pipelining: An evaluation of Enhanced Pipelining," in *Proceedings of the 24th International Workshop on Microprogramming and Microarchitecture*, pp. 82–92, November 1991.

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

[14] 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.

[15] 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.

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

[17] 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.

[18] F. Gasperoni, "Compilation techniques for VLIW architectures," Tech. Rep. 66741, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY 10598, August 1989.

[19] N. J. Warter and W. W. Hwu, "Enhanced modulo scheduling," Tech. Rep. CRHC-92-11, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, November 1992.

[20] J. C. H. Park and M. Schlansker, "On Predicated Execution," Tech. Rep. HPL-91-58, Hewlett Packard Software Systems Laboratory, May 1991.

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

[22] D. C. Lin, "Compiler support for predicated execution in superscalar processors," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1992.

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

[24] J. W. Bockhaus, "An implementation of GURPR*: A software pipelining algorithm," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1992.

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

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

[27] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker, "Register allocation for software pipelined loops," in *Proceedings of the ACM SIGPLAN 92 Conference on Programming Language Design and Implementation*, pp. 283–299, June 1992.

[28] W. Y. Chen, S. A. Mahlke, N. J. Warter, and W. W. Hwu, "Using profile information to assist advanced compiler optimization and scheduling," in *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, August 1992.