Code Generation Schema for Modulo Scheduled Loops

B. Ramakrishna Rau, Michael S. Schlansker, P. P. Tirumalai
Hewlett Packard Laboratories, Palo Alto, California

Abstract

Software pipelining is an important instruction scheduling technique for efficiently overlapping successive iterations of loops and executing them in parallel. Modulo scheduling is one approach for generating such schedules. This paper addresses an issue which has received little attention thus far, but which is non-trivial in its complexity: the task of generating correct, high-performance code once the modulo schedule has been generated, taking into account the nature of the loop and the register allocation strategy that will be used. This issue is studied both with and without hardware features that are specifically aimed at supporting modulo scheduling.

Keywords: code generation, modulo scheduling, software pipelining, VLIW processors, superscalar processors

1 Introduction

1.1 Software pipelining

Software pipelining is a loop scheduling technique which yields highly optimized loop schedules. Algorithms for achieving software pipelining fall into two broad classes:

- modulo scheduling, in which all iterations of the loop have a common schedule [16] and,
- algorithms in which the loop is continuously unrolled and scheduled until a situation is reached allowing the schedule to wrap back on itself without draining the pipelines [15].

Although, to the best of our knowledge, there have been no published measurements on this issue, it is the authors' belief that the second class of software pipelining algorithms can cause unacceptably large code size expansion. Consequently, our interest is in modulo scheduling. In general, this is an NP-complete problem and subsequent work has focused on various heuristic strategies for performing modulo scheduling, e.g., [24, 9, 11, 10, 12] and the as yet unpublished heuristics in the Cydra 5 compiler [5]. Modulo scheduling of loops with early exits is described by Tirumalai, et al. [22]. Modulo scheduling is applicable to RISC, CISC, superscalar, superpipelined, and VLIW processors, and is useful whenever a processor implementation has instruction-level parallelism either by virtue of having pipelined operations or by allowing multiple operations to be issued per cycle.

This paper describes code generation alternatives for modulo scheduled loops, both on processors such as the Cydra 5 [19] which have hardware support for modulo scheduling as well as on other instruction-level parallel processors which do not. The focus of the paper is on precisely specifying the alternatives and far less on evaluating their relative merit. Hardware support for modulo scheduling includes rotating register files (register files which support compiler-managed register renaming, also known as the MultiConnect in the Cydra 5), predicated execution and the Iteration Control Register (ICR) file (a boolean register file that holds the predicates), certain loop control opcodes [5, 19, 22] and support for speculative code motion [13]. The processor model assumes the ability to initiate multiple operations in a single cycle where each operation may have latency greater than one cycle. For brevity, we shall only discuss code generation for VLIW processors in which each instruction contains multiple operations, where each operation is equivalent to a RISC instruction. Nevertheless, everything discussed in this paper is applicable to RISC and superscalar processors as well; a left-to-right, top-to-bottom scan of the VLIW code would yield the corresponding RISC code.

The examples discussed in this paper assume a processor with seven pipelined functional units as detailed in Table 1. The mnemonics for the various operations that are relevant to the examples are shown along with the unit on which those operations execute as well as their latencies.

Table 1: Description of a sample processor .

Functional Unit	Operations Performed	Mnem-onic	Latency
IALU 1, IALU 2	integer add	iadd	1
	integer subtract	isub	1
Memory Port 1	load	load	5
Memory Port 2	store	stor	1
Floating Point Multiplier	floating multiply	fmul	4
Floating Point Adder	floating add	fadd	2
	floating subtract	fsub	2
	floating ≥	fge	2
Instruction Unit	branch top of loop	brtop	2
	branch out of loop	bquit	2

In the rest of this section, we provide a brief overview of modulo scheduling and discuss the problem that arises from the fact that after modulo scheduling, the successive lifetimes of a loop-variant variable are live concurrently. This section sets up the need for more sophisticated code schemas than have heretofore been discussed in the literature. In Section 2 we define and describe certain hardware capabilities to support modulo scheduling, some of which were present in the Cydra 5. The motivation for them is supplied in Section 3 which discusses the code schemas that must be used for DO-loops and WHILE-loops depending on whether or not these hardware features are provided.

1.2 An Overview of Modulo Scheduling

It is generally understood that there is inadequate instruction-level parallelism (ILP) between the operations within a single basic block and that higher levels of parallelism can only result from exploiting the ILP between successive basic blocks [23, 8, 20, 14, 3, 25]. In the case of innermost loops, the successive basic blocks are the successive iterations of the loop. One method that has been used to exploit such inter-block parallelism has been to unroll the body of the loop some number of times and to overlap the execution of the multiple copies of the loop body [7]. Although this does yield an improvement in performance, the back-edge of the unrolled loop acts as a barrier to parallelism. Software pipelining, in general, and modulo scheduling, specifically, are scheduling techniques which attempt to achieve the performance benefits of extensive unrolling without actually doing so.

The number of instruction issue cycles between the initiation of successive iterations in a modulo schedule is termed the **initiation interval** (II) [16]. This is also the number of (VLIW) instructions in the body of the modulo scheduled code if kernel unrolling (see below) has not been employed. The objective of modulo scheduling is to

0-8186-3175-9/92 $3.00 © 1992 IEEE

engineer a common schedule for all iterations such that when successive iterations are initiated II cycles apart, no resource usage conflict arises between operations of either the same or distinct iterations. This requirement is met by constraining the schedule for a single iteration to be such that the same resource is never used more than once at the same time modulo the II.

Lower bounds on II can be established through a simple analysis of the data dependence graph for the loop body. One bound (**ResMII**) is derived from the resource usage requirements of the graph while the other (**RecMII**) is derived from latency calculations around circuits defining recurrences within the data dependence graph for the loop body. The actual II must be greater than or equal to the maximum of these bounds. A more detailed discussion of **ResMII** and **RecMII** can be found in [9, 11, 5, 18]. Any legal II must be equal to or greater than MAX(ResMII, RecMII).

Figure 1a displays the FORTRAN DO-loop which we shall use as an example, and the corresponding intermediate representation using virtual registers. Figure 1b lists the operations within the body of the loop and the lefthand column of Figure 1b lists names used to refer to individual operations. These names are used in Figure 2a to specify at what time and on which functional unit each operation is scheduled after modulo scheduling is completed with an II of 3.

The schedule for an iteration can be divided into **stages** consisting of II cycles each. The number of stages in one iteration is termed the **stage count** (SC). Each stage of the schedule in Figure 2a is demarcated by heavy lines. Figure 2b shows the record of execution during the steady state of the modulo scheduled loop. The prefix before each operation's name indicates the iteration (relative to the currently issued nth iteration) to which that operation belongs. Note that although a single iteration takes 15 cycles to execute, (as shown in Figure 2a), each additional iteration takes only an incremental 3 cycles. This is the motivation for performing modulo scheduling.

In a modulo schedule, exactly the same pattern of operations is executed in each stage of the steady state portion of the modulo schedule's execution. This behavior can be achieved by looping on a piece of code that corresponds to one stage of the steady state portion of the record of execution. This code is termed the **kernel**. The record of execution leading up to the steady state is implemented with a piece of code called the **prologue**. A third piece of code, the **epilogue**, implements the record of execution following the steady state. Figure 3 shows the code for the loop kernel. Instruction i of the kernel includes all operations that are scheduled at time i modulo the II. Also, shown in Figure 3 is the stage of the schedule from which each operation comes. Operations in the kernel code which are from distinct stages are from distinct iterations of the original loop. The branch operation, B1, determines whether or not another iteration is to be executed, and since its latency is 2 cycles, it must be scheduled in the second to last instruction of the kernel.

Figure 4a uses this example DO-loop to demonstrate the abstracted representation of code that we shall use in this paper. Each square represents one stage's worth of code from a single iteration. The letter label in the square indicates the corresponding stage, with A corresponding to stage 0, B to stage 1, and so on. Thus, the set of rectangles in the leftmost column correspond to all the operations in the first iteration. Each row of squares represents II VLIW instructions. The row of squares that includes the last stage of the first iteration corresponds to the kernel code (Figure 4a). The triangle of squares above the kernel represents the prologue code and the triangle of squares below the kernel represents the epilogue code.

1.3 Overlapped Lifetimes

The code in Figure 3 is incorrect as shown. Consider the operation t03 = iadd(t03,#4) which at time 0 computes a new address value into virtual register t03 (Figure 4a). The lifetime of this value extends to time 12 when it is used for the last time. However, 3 cycles later the same operation is executed again on behalf of the next iteration and will overwrite the previous value in t03, while it is still live, yielding an incorrect result. One approach to fixing this problem is to provide some form of register renaming so that successive definitions of t03 actually use distinct registers. We shall define such a scheme in Section 2. It is important to note that conventional hardware renaming schemes are inadequate. Since, successive definitions of t03 are encountered *before* the uses of the prior definitions, it is impossible even to *write* correct code for the modulo scheduled loop with the conventional model of register storage.

When no hardware support is available, modulo scheduling is made possible by **modulo variable expansion**, (**MVE**), i.e., unrolling the kernel and renaming at compile time the multiple (static) definitions that now exist of each virtual register [11]. The unrolling and renaming prevents successive lifetimes, corresponding to the same loop-variant physical register, from overlapping in time. The minimum degree of unroll, K_{min}, is determined by the longest lifetime among all loop-variants i. Assume that each loop variant i has parameters $start_i$ and end_i marking the beginning and end of the lifetime. Because iterations are initiated every II cycles, K_{min} can be calculated as

$$K_{min} = \underset{i}{MAX}\left(\left\lceil \frac{(end_i - start_i)}{II} \right\rceil \right).$$

In our example, the longest lifetime is 12 cycles corresponding to the definition of t03. For an II of 3, this requires that K_{min} = 4. Every fourth definition of t03 can reuse the same physical register since its previous contents are no longer live. The structure of the code after kernel unrolling is shown in Figure 4b. The labels for the squares now include a numerical suffix which specifies which code version is being used. By looking at the columns one can see that there are K_{min} distinct versions of code for an iteration and that the successive iterations cycle through these four versions. Each version makes use of different sets of physical registers to avoid over-writing live values.

It may appear that modulo scheduled code can be generated in conformance with the code schemas of Figures 4a or 4b. We shall see in Section 3 that this is not the case and that, in fact, considerably more complex schemas are needed if performance is not to be compromised. The problem is that with the code schemas of Figures 4a and 4b, it is only possible to execute i+4 and 4*i+4 iterations respectively, where $i \geq 0$. (Four iterations can be executed by branching from the last stage of the prologue to the first stage of the epilogue. Fewer iterations cannot be executed with the codes schemas in their current form.) These code schemas have to be augmented if an arbitrary number of iterations are to be executable.

1.4 Pre-conditioning of Modulo Scheduled DO-Loops

A solution, that is often employed, is to pre-condition the modulo scheduled loop so that only the appropriate number of iterations remain to be executed at the time the prologue is entered. In general, the code schemas of Figures 4a and 4b can execute only certain numbers of iterations, N, where N = K*i + (SC-1) and where K is the degree of unroll, SC is the number of stages in one iteration and $i \geq 0$. When the desired number of iterations, L, is not of this form, a conventional, non-software pipelined version of the loop is first executed until the number of remaining iterations is of the above form. At this point, the modulo scheduled code schema is entered with an appropriate trip count. The number of iterations, M, in the pre-conditioning loop is given by

$$M = \begin{cases} L, & \text{if } L < SC - 1 \\ [L - (SC-1)] \bmod K, & \text{otherwise.} \end{cases}$$

$$N = L - M$$

These M iterations are executed relatively slowly and the remaining N iterations are executed with the full, modulo scheduled level of performance. Assume that the time taken to execute one iteration of the non-software pipelined, pre-conditioning loop is $SL \leq SC*II$ cycles. Then

$$T_{PC} = M*SL + (N + SC - 1)*II$$
$$T_{Ideal} = (L + SC - 1)*II$$

where T_{PC} is the execution time for the pre-conditioned loop and T_{Ideal} is the ideal execution time for the software pipelined loop. The first term in the formula for T_{PC} is the time spent in the pre-conditioning loop and the second term is the time spent in the software pipelined loop. The speedups in the two cases, relative to a non-software pipelined version of the loop are given by

$$S_{PC} = \frac{L*SL}{M*SL + (N + SC - 1)*II}$$

$$S_{Ideal} = \frac{L*SL}{(L + SC - 1)*II}$$

The effectiveness of pre-conditioned code is highly dependent upon the nature of the processor architecture. In order to better illustrate this point, we define four processors: P1, P2, P3, P4 (Table 2). Processor P3 is exactly the sample processor of Table 1. Processors P1 and P2 are versions of the sample processor having identical latency but reduced numbers of functional units, while processor P4 is P3 with increased latencies.

A schedule was generated for the example program of Figure 1 for each of the processors in order to help illustrate the relationship between the amount of processor parallelism and the four parameters which determine pre-conditioned code performance. The four parameters are: the initiation interval (II), the number of stages (SC), the minimum degree of kernel unroll (K_{min}) and the schedule length of a single non-overlapped loop iteration (SL). Note that $K \geq K_{min}$. In this discussion we assume that $K = K_{min}$. Parameters resulting from schedules for the four processors are shown in Table 3.

In all four schedules, II was equal to ResMII because each schedule saturates a resource. For the schedule for processor P1, fifteen total operations were scheduled onto a single functional unit. For P2, six memory operations were scheduled onto a single memory unit. For both P3 and P4, six memory operations were scheduled onto two memory units. Thus, the ResMII and a resulting II can be justified and as we increase the number of functional units within the processor (P1, P2, P3), the II decreases.

SC represents the length of the software pipeline schedule of a single iteration divided by the II and rounded up to the nearest integer. If we were to assume that the schedule length for a single iteration were held constant, than the effect of reducing II is to increase the number of stages. We can see that this increase in SC indeed occurs as one goes from processor P1 to P2 to P3. As II decreases through the values 15, 6 and 3, SC increases through the values 2, 3 and 5, respectively. The parameter K_{min} can be viewed similarly. If we were to assume that the longest lifetime is constant among the different schedules and is then divided by II to yield K_{min} we will see a similar progression of decreasing II and increasing K_{min}. SC and K_{min} are not strictly inversely proportional to II. As we add functional units, the schedule length and the lifetime lengths do not stay absolutely constant. Processor P4 demonstrates the effects of increasing the latency which is to generally increase schedule lengths and, correspondingly, to increase SC, K_{min}, and SL.

The effects of varying these parameters on the speedups, T_{PC} and T_{Ideal}, achieved by the pre-conditioned loop and the ideal case, respectively, are shown in Figure 5. Note that for a machine with little parallelism such as P1 (Figure 5a), pre-conditioning is quite satisfactory because the time lost in non-overlapped pre-conditioning code is small. One reason for this is the small K_{min} which results in a small value for M. Secondly, the small difference between SL and II decreases the benefits of software pipelined execution over non-overlapped execution.

Although pre-conditioning is an acceptable solution for processors with little instruction-level parallelism, in processors with as much parallelism as P3 or P4 (Figures 5c and 5d), the loss in performance due to pre-conditioning is very significant. In particular, only very large trip counts can guarantee that the loop achieves close to asymptotic performance. For P3 and P4, the maximum value of M is large and so is the difference between SL and II. Furthermore, pre-conditioning is not an option with WHILE-loops. Better alternatives are needed with VLIW processors and aggressively superscalar processors or if general-purpose computation involving WHILE-loops is to be supported. These are the subject of Section 3.

Table 2. Definition of Processors P1, P2, P3, P4

P1	A single functional unit which executes all operations. Latencies are as in Table 1.
P2	Three functional units. An IALU unit executes all integer operations and branches. The memory unit performs loads and stores. The floating point unit executes all floating point operations. Latencies are as in Table 1.
P3	The sample processor of Table 1.
P4	The sample processor with all latencies doubled.

Table 3. Results of Scheduling Sample Processors

Machine	II	SC	K_{min}	SL
P1	15	2	2	20
P2	6	3	3	16
P3 (sample processor)	3	5	4	15
P4	3	9	9	25

1.5 Modulo Scheduling of WHILE-Loops

In DO-loops, it is possible to decrement and test the count of the remaining iterations in time to either start the next iteration with an initiation interval of II or exit the kernel. This is not always the case in the broader class of loops which we shall refer to in this paper as "WHILE-loops". This is the class of single entry loops with a single control flow back-edge, one or more exits and for which it is not known, at the time that the loop is entered, what the trip count will be. Whether another iteration is to be executed is known somewhere in the middle of the current iteration and the extent of software pipelining is apparently limited by the fact that the next iteration cannot be initiated until this point in time.

Consider the situation if in Figure 4a it is not known until stage C whether another iteration is to be executed. The earliest time that the next iteration could be initiated would be at the end of stage C. The resulting modulo schedule would have an II that is three times as large (Figure 6a). The limiting dependence is the control dependence between a loop-exiting branch operation in one iteration and all of the operations in the next iteration. Assuming hardware support for speculative code motion [13], this control dependence can be relaxed to yield a smaller II and a better modulo schedule [22].

In Figure 6b, the operations in stages A and B of a given iteration, instead of being control dependent on the branch operation from stage C of the previous iteration, have been made dependent on the corresponding branch operations from three and two iterations ago, respectively, i.e., they are executed speculatively. This is clear in Figure 6b since stages A and B are executed before or in parallel with stage C of the previous iteration. The remaining stages are scheduled non-speculatively after stage C of the previous iteration. The net result is a schedule that yields the same performance as would be obtained for a DO-loop.

The speculative execution of stages A and B implies that at every instant, after the second stage of the first iteration, we have two iterations that have been initiated speculatively. When the kernel is exited, we can stop executing, and leave unfinished, the two speculative iterations that are in progress at that point. In Figure 6b, this aborted computation is the rightmost two columns of the epilogue which are shown shaded. The code for this is eliminated from the epilogue. In general, if θ stages of each iteration are executed speculatively, the rightmost θ columns of the epilogue are eliminated and the epilogue length reduces by θ stages.

2 Architectural Support for Modulo Scheduling

In this section, we shall describe architectural features that support the use of fast, compact code for modulo scheduled DO-loops and WHILE-loops. The motivation for their existence as well as the manner in which they are intended to be used is deferred to Section 3.

2.1 Rotating Register Files

A rotating register file is addressed by adding the instruction's register specification field to the contents of the **Iteration Control Pointer (ICP)** modulo the number of registers in the rotating register file. Special loop control operations, that are described below, decrement the ICP each time a new stage starts. As a result of decrementing the ICP, a new absolute register now corresponds to the register specifier i, and the register that was previously specified as register i would have to be specified as register i+1. This allows the lifetime of a value generated in one iteration to overlap the lifetimes of corresponding values generated in previous and subsequent iterations without code replication.

The rotating register file is quite similar in concept to vector registers. Instead of moving the pointer every cycle, it is moved once per kernel iteration, and instead of having multiple vector registers, they are pooled into one register file. The use and allocation of rotating registers is described in [17]. One version of rotating registers first appeared in the scratchpad register files of the FPS AP-120B and FPS-164 [4].

2.2 Predicated Execution

The **Iteration Control Register (ICR)** is a rotating register file that stores boolean values called predicates. An operation is conditionally executed based on the value of the predicate associated with it. For example, the operation "a = op(b,c) if p" executes if the predicate in the ICR register p is true (one), and is nullified if the predicate is false (zero). Predicated execution permits the generation of more compact code by conditionally disabling the execution of operations during prologue and epilogue execution. The need to unroll a prologue and epilogue is eliminated, thereby supporting the generation of kernel-only code as described in Section 3.5.

In addition to supporting the combining of prologue, kernel, and epilogue code, predicates are also used to enable modulo scheduling of loops containing conditional branches [5, 19]. Predicates permit the IF-conversion of the loop body [2], thereby eliminating all branches from the loop body. The resulting branch-free loop body is modulo scheduled. This was the primary motivation for providing predicated execution in the Cydra 5. More recently, limited forms of predicated execution have been incorporated or proposed in other machines [6, 1]. In the absence of predicated execution, other techniques must be used requiring either multiple versions of code corresponding to the various combinations of branch conditions [6, 15, 21] or restrictions on the extent of overlap between successive iterations [11]. Predicated execution is conceptually similar to, but more general than, the use of mode bits in the vector mask register of a vector processor.

2.3 Speculative Execution

Speculative execution consists of executing an operation before it is clear that it should, in fact, be executed. One way of achieving speculative execution is by speculative code motion, i.e., by moving an operation up above the branch that could have directed flow of control away from this operation [7]. The main challenge is to report exceptions correctly in the face of speculative execution, i.e., if and only if the exception would have been reported in the non-speculative execution of the program. The hardware support assumed involves having two versions of every operation that can be speculatively executed (one normal opcode and one speculative opcode), and an additional bit in every register to serve as a tag indicating that the register contains an exception tag rather than normal data. A detailed description of this hardware support and its use is described elsewhere [13].

2.4 Loop Control Operations

In this paper we use three loop control operations for modulo scheduling DO-loops: **brtop**, **bquit** and **rotate**. These operations are merely described below; their use is motivated in Section 3.5. The description of the brtop operation provided here follows that by Dehnert, et al. [5]. A flowchart for the brtop is provided in Figure 7a. The brtop operation is scheduled in the second last cycle of a stage within the loop body so as to complete execution in the last cycle. The ICP is decremented every loop iteration so that each iteration can reference a different set of registers. The **loop counter (LC)** which counts the remaining loop iterations is decremented until it reaches zero. Thereafter, the **epilogue stage counter (ESC)** which counts epilogue stages is decremented until it reaches zero. At this point, the brtop branch is not taken and the loop is exited. The ESC supports the execution of the extra iterations of the kernel required to drain the software pipeline. The brtop operation assigns a boolean value to the predicate register ICR(ICP) which controls the conditional execution of the next loop iteration. As the LC is decremented, the assignment to ICR(ICP) sets to true the controlling predicate for the next loop iteration. After LC has reached zero, the assignment to predicate ICR(ICP) sets to false the controlling predicate for subsequent loop iterations, thereby discontinuing the initiation of new iterations. The brtop operation finds LC=0 after the last iteration has been initiated, and finds ESC = LC = 0 when it is time to exit the kernel. The initial value of ESC determines how many additional times the kernel should be executed after LC has become 0.

The bquit operation (Figure 7b) is similar to the brtop operation except that it takes the branch if LC $\leq$ 0. The brtop operation is used when a code generation schema continues execution of a loop by branching (back to the top of the kernel). The bquit operation is used when a code generation schema continues execution of a loop by falling through to the next stage of the prologue. The third loop control operation, rotate, unconditionally decrements the ICP and sets the predictae pointed at by the ICP to 1. The rotate does not affect the flow of control.

Two other loop control operations, **wtop, wquit** are needed in addition to the **rotate** operation for modulo scheduling WHILE-loops. The wtop operation is defined in Figure 8a. In the case of WHILE-loops, the number of iterations (and the loop counter) are not known at the time the loop is entered. Instead of the loop counter, the wtop operation uses two inputs, a boolean and a predicate, and produces an output predicate. The output is true only if both the input boolean and the input predicate are true. This ensures that an iteration completes only if the previous iteration completed and the condition for the WHILE-loop was evaluated to false. The epilogue counter is used just as in brtop to allow the last few iterations to complete before the branch out of the loop is taken. The ICP is decremented as in brtop so that each iteration references a different set of registers. The wquit operation (Figure 8b) is similar to the wtop operation but has been altered much like the bquit operation. Whereas the wtop operation continues loop execution by branching back to the top of the kernel, the wquit operation continues loop execution within the prologue by falling through to the next stage.

In the discussion of code schemas, we shall be considering situations when neither predicated execution nor rotating registers are present. When this is the case, the above five loop control operations degenerate to relatively conventional branch operations. All the necessary loop control operations and their semantics are listed in Table 4.

3 Code Generation Schemas for Modulo Scheduled Loops

When generating code for modulo schedules, two fundamental problems must be overcome. First, a means must be identified to prevent lifetimes, corresponding to successive definitions of the same loop-variant virtual register in successive iterations, from being assigned to the same physical register. One way to accomplish this is to use different versions of the code for successive iterations, with each version making use of different registers as a result of modulo variable expansion. The alternative is to use a single version of the code and to provide a rotating register file that dynamically renames the instruction-specified sources and targets, thereby achieving the same objective. Second, a means must be identified to allow subsets of the steady state software pipeline, the kernel, to be executed. This is required in order to handle the first few and last few iterations of the modulo scheduled loop and to handle the case of a smaller number of loop iterations than that corresponding to a single pass through the prologue, kernel and epilogue. It is possible to generate code for modulo scheduled loops for each assumption regarding the choice of code generation technique and available hardware support. All four code schemas, depending on whether rotating registers, predicated execution, neither or both are present, have been studied [18],. In this paper we shall restrict our discussion to two of the four sets of code schemas. In code schema 1, only speculative code motion is supported. In code schema 4, that plus predicated execution and rotating register files are provided.

All of the code schemas described below have two things in common. First, it is assumed that there is a branch preceding the code schema that checks that the trip count of a DO-loop is at least one; if not, the entire code schema is branched around. Second, whenever a code schema has more than one control flow path out of it and into the code that follows the modulo scheduled loop, it is to be understood that there exists code on each of these paths which copies the scalar live-out values (if any) into the registers in which the subsequent code expects to find them.

Code generation schemas for modulo scheduled WHILE-loops are similar to those for DO-loops. Nevertheless, there are differences that result from the fact that the trip count cannot be predetermined prior to loop entry. Here, we shall consider only the schemas for code generation, not the details of how to modulo schedule WHILE-loops, which is discussed elsewhere [22]. The WHILE-loops referred to in this section correspond to the do-while construct of the C language with an arbitrary number of exits from the loop. One important distinction from DO-loops is that pre-conditioning is not an option with WHILE-loops.

Table 4. Definitions of Loop Control Operations

Name of operation	operation semantics
brtop	defined in Fig. 8a
wtop	defined in Fig. 9a
bquit	defined in Fig. 8b
wquit	defined in Fig. 9b
noop	do nothing
rotate	ICP = ICP-1; ICR(ICP)=1;
bct	If(LC>0) {LC=LC-1; take branch }
bctb	If(LC>0) {LC=LC-1; }else take branch
bc	If(not exit condition) take branch
bcb	If(exit condition) take branch

Table 5. Application of Loop Control Operations

	Placement of Operation	Within DO-Loops	Within WHILE-loops
Schema 1,1s			
	First θ stages of prologue	noop	noop
	Rest of prologue	bctb	bcb
	First K-1 stages of unrolled kernel	bctb	bcb
	Final stage of unrolled kernel	bct	bc
	Epilogue stages	noop	noop
Schema 4			
	First θ stages of prologue	rotate	rotate
	Rest of prologue	bquit	wquit
	Kernel stage	brtop	wtop
	Epilogue stages	rotate	rotate
Schema 4c	Kernel stage	brtop	wtop

We shall avoid detailed discussions of the WHILE-loop schemas since in all cases they closely parallel those for DO-loops, but with the following differences.

- brtop and bquit operations are consistently replaced by wtop and wquit operations, respectively. The loop counter is irrelevant.
- The first θ stages of the prologue do not contain an exit branch because the first exit condition is not evaluated until stage θ +1 of the first iteration. Accordingly, the first θ branch arcs out of the prologue, which are shown as dashed lines, are understood to be absent.
- When rotating registers or predicates are present, these first θ stages contain rotate operations in place of the wquit branches so that the requisite loop control functions are still performed.
- The rightmost θ columns of every epilogue (which are shown shaded in the figures) are deleted since these correspond to the unnecessary completion of speculatively initiated iterations. As a result, the length of each complete epilogue decreases by θ stages.

Table 5 is used to determine which specific loop control operation is to be used in each stage of the various code schema that are discussed in the rest of this section.

3.1 Code Schema 1: Only Speculative Support

We first consider, for the DO-loop example of Figure 1, a code generation schema (Figure 9a) which requires no special hardware. Recall that for this example, SC = 5 and K_{min} = 4. As before, each square is labeled with a letter identifying the stage and a number identifying the code version (register assignment choice) used. All stages of a single iteration (same column) correspond to the same code version. A single stage of the modulo scheduled code (all the squares in a single row) consists of one stage each (and a different one) from successive iterations. A loop-control branch is executed at the end of every stage of the modulo scheduled code as specified by Table 5. Arrows indicate taken branches which, typically, signify transfer of control to an epilogue which completes unfinished portions of the iterations that were in execution when the exit branch was taken.

We can divide the code generated with this schema into a prologue, a kernel, multiple partial epilogues and multiple complete epilogues. Since rotating registers are absent, the code schema must include all the code shown in Figure 4b plus additional code to permit an arbitrary number of iterations. The unique prologue is depicted by the topmost triangle of rows with left hand column A1 ... D1. The kernel is the full width parallelogram consisting of K_{min} = 4 rows with left hand squares labeled E1, E2, E3, E4. Register lifetime overlap requirements necessitate the kernel be unrolled to yield four copies. The last stage of the kernel contains the bct operation which, when taken, closes the loop or, when not taken, enters the complete epilogue depicted by the triangle with righthand column B4 ... D4. The rest of the stages in the prologue and kernel contain bctb operations which, when taken, lead to various versions of complete or partial epilogues. Complete epilogues are reached by exiting the loop at the end of any of the four kernel stages, or by exiting from the the final prologue stage. Partial epilogues are reached by exiting from any of the earlier (first three) prologue stages. The LC must be set initially to one less than the desired trip count.

The code schema of Figure 9a can be seen to be redundant. The epilogues reached by branching out of the final prologue stage and by falling out of the final kernel stage are identical and can be merged into a single epilogue. Each of the partial epilogues reached by branching out of one of the earlier stages of the prologue has a final portion which is identical to the final portion of one of the complete epilogues. This final portion of the partial epilogues can be eliminated and replaced by an unconditional branch to the appropriate stage of the appropriate complete epilogue. The resulting code schema, with this redundancy eliminated, is shown in Figure 9b.

Figure 9 also shows the code generation schema for a WHILE-loop in the absence of hardware support. (The shaded squares and the dashed lines should be viewed as absent for the WHILE-loop schema.) This example loop has SC = 5, K_{min} = 4, θ = 2 and, therefore, the number of epilogue stages, ES = SC-θ-1 = 2. All of the standard differences listed above, between DO-loop schemas and WHILE-loop schemas, apply. Other aspects of this schema are the same as that for DO-loops. As with DO-loops, normal conditional branches are employed and the ESC is unnecessary.

3.2 Code Schema 1s: Aggressive Speculation

Aggressive speculative code motion can be used to minimize the length of the epilogue in both DO-loops and WHILE-loops. In particular, if the loop exit branch can be scheduled in the last stage of an iteration, then θ would be equal to SC-1 and the length of the epilogue would be zero. Since θ rows and θ columns of every partial or complete epilogue are deleted from schema 1, all of the epilogues would disappear. θ = SC-1 corresponds to all but the operations in the last stage being executed speculatively. We shall refer to this as code schema 1s. In certain cases, there may be too many operations (such as stores) to fit in the last stage without compromising the II. In such cases, θ would have to be less than SC-1 and some of the epilogues would be present albeit with reduced length.

3.3 Code Schema 1pc: Pre-conditioned Code (for DO-Loops only)

In the absence of predicates or rotating registers, the multiple epilogues of code schema 1 can be eliminated, yielding the code schemas in Figure 4b, by pre-conditioning the loop. Recall from Section 1.4 that the number of iterations, M, in the pre-conditioning loop is given by

$$M = \begin{cases} L, & \text{if } L < SC - 1 \\ [L - (SC-1)] \bmod K, & \text{otherwise.} \end{cases}$$

$$N = L - M$$

where L is the desired number of iterations, K is the degree of unroll and SC is the number of stages in one iteration. The remaining N iterations are executed in the modulo scheduled code schema. The LC must be initialized prior to entering the modulo scheduled loop with the value [N-(SC-1)] div K. The branch operation at the end of the kernel must decrement the loop counter by 1 each time it is executed (which is every K*II cycles). No other branch operations are needed in either the prologue or the kernel. Alternatively, the LC may be initialized to [N-(SC-1)] and the branch at the bottom of the kernel must decrement the LC by K each time. (Of course, both alternatives are identical when K = 1.) The pre-conditioned version of code schema 1 will be referred to as code schema 1pc .

3.4 Code Schema 4: Speculative Support, Rotating Registers and Predicated Execution

The code schema of Figure 10a makes use of both predicates and rotating registers. Starting with code schema 1 (Figure 9b), we see that each partial epilogue is a subset of the complete epilogue below it. So, rather than executing the partial epilogue one could, instead, execute the complete epilogue with the appropriate number of the leftmost columns disabled by predicates. This eliminates the partial epilogues. Next, because of the rotating registers, multiple versions of code are unnecessary. All the epilogues can be merged with the one below the kernel, and the kernel unrolling can be eliminated. The result is the code schema shown in Figure 10a. All the bquit operations have as their target the beginning of the (single) epilogue. With this code schema, too, the LC must initially be set to one less than the desired trip count and the ESC must be initialized to 0.

As before, there are rotate operations instead of branches in the first θ stages of a WHILE-loop schema. Subsequently, there is a wquit operation in every stage of the prologue and a wtop operation in the kernel. ESC must be initialized to 0.

3.5 Code Schema 4c: Kernel-Only Code

With hardware support in the form of rotating registers and predicated execution, it is not necessary to have explicit code even for the prologue and epilogue; a single copy of the kernel is sufficient to execute the entire modulo scheduled loop. This is termed **kernel-only** code. Consider the kernel-only code schema depicted in Figure 10b. Every stage of the code schema in Figure 10a is a subset of this kernel-only schema. The prologue and epilogue can be swept out by executing the kernel with the appropriate operations disabled by predicated execution. Since this is a compact version of code schema 4, we shall refer to the schema in Figure 10b as code schema 4c.

The code corresponding to the kernel-only schema is shown in Figure 11a. All operations from the i-th stage are logically grouped by attaching them to the same predicate, specifically, the contents of the ICR register specified by the predicate specifier i (relative to the ICP). This is represented in Figure 11a by appending "if p_i" to every operation from the i-th stage. This permits all operations from a particular stage (of one iteration) to be disabled or enabled independently of the operations from some other stage (of some other iteration). At every point in time, predicate p_0 is the ICR register that is currently pointed to by the ICP. This predicate is set to 1 by the brtop operation during the ramp up and steady state phases (i.e., while the value of the loop counter is greater than 0) and is set to 0 during the epilogue phase. Because brtop decrements the ICP, a different physical predicate register is written into every II cycles and, a given predicate value must be referred to by different predicate specifiers in different stages.

Figure 11b demonstrates the manner in which this is actually effected with the joint use of rotating registers, predicated execution and the brtop operation. The example assumes that 7 iterations of a loop of 5 stages is desired. The loop counter, LC, is initialized to 6--one less than the number of iterations desired. The epilogue stage counter, ESC, is initialized to 4--one less than the number of stages. Lastly, p_0 (the ICR location that is currently pointed to by the ICP) is set to 1 and p_1 through p_4 are set to 0. At this point, the kernel-only code is entered. Since only p_0 is true, only the operations from the first stage, labelled A, are executed and the rest of the operations are disabled. At the end of the first trip through the kernel, since the LC is greater than 0, the brtop operation loops back to the top of the kernel and decrements the LC by 1. It also decrements the ICP by 1 and, since the ICR is a rotating register file, the true predicate that used to be p_0 is now p_1. Also, because the LC was greater than 0, the new p_0 is set to 1. During the next trip through the kernel code, the operations corresponding to the first two stages, A and B, execute since both p_0 and p_1 are true.

This process is repeated with the operations in the i-th stage being executed when the corresponding predicate, p_i, is 1. Eventually, the brtop operation finds that the LC is 0, but loops back because the ESC is greater than 0. However, it now decrements the ESC, decrements the ICP and inserts a 0 in the new p_0. As a result, the next time around, operations from stage A are not executed. Finally, when both the LC and ESC are 0, the brtop operation falls through to the code following the loop. In the process, seven iterations each consisting of five stages have been swept out by the combined operation of the brtop operation, rotating registers and predicated execution.

As with DO-loops, it is possible to generate kernel-only code for WHILE-loops, but only for the portion after the first θ stages. The first θ stages of the WHILE-loop when the boolean expression for the first iteration is being computed constitutes the minimal length prologue permissible. The kernel can then generate all the remaining stages of the loop iterations. The ESC must be initialized to SC-θ-1.

Predicated execution and the loop-control branches are used in much the same way in code schema 4 as they are in schema 4c, even though an explicit prologue and epilogue are provided. When executing the prologue or when the epilogue is executed due to the brtop operation falling through, the predicates are redundant since only those stages are present for whom the predicate is true. However, the predicates are required, when the epilogue is entered via a bquit operation, so as to disable those stages that are not part of the partial epilogue that needs to be executed.

Two primary benefits result from the fact that schema 4 provides explicit prologues and epilogues unlike its kernel-only counterpart. First, the schedules of the prologue and epilogues can be customized and optimized to take advantage of the reduced requirements for resources within the loop startup and loop shutdown phases. Second, code which originates from outside the innermost loop may be percolated into and scheduled in parallel with prologue and epilogue code. This can result in better performance than with kernel-only code, an effect that is more noticeable when the trip count of the innermost loop is small.

3.6 Bounds on Prologue and Epilogue Lengths

In general, the code schemas described above consist of a prologue, a steady state kernel, and an epilogue (or multiple epilogues). The number of stages of prologue (PS) and epilogue (ES) used in a code schema were assumed to always be SC-1 and SC-θ-1 stages, respectively (with θ set to 0 for DO-loops). For kernel-only code, these numbers are θ and 0, respectively. Whereas, the latter numbers do represent the minimum possible prologue and epilogue lengths, the former number do not quite correspond to the maximum lengths. If the sole objective of the prologue and epilogues is to eliminate the unneeded kernel computation during the ramp up and ramp down of the software pipeline, then SC-1 and SC-θ-1 do, in fact, represent the maximum prologue and epilogue length required. The prologue and epilogue lengths are, however, also dictated by the register allocation strategy [17].

4 Conclusions

Pre-conditioning a modulo scheduled loop, though acceptable on processors with little instruction-level parallelism, leads to significant performance degradation on processors which either are capable of issuing many operations per cycle or are deeply pipelined. In such cases, other code schemas must be employed. The generation of a high performance and correct modulo scheduled code schema is affected by a number of issues: whether or not the loop is a DO-loop, the nature of the hardware support provided, whether or not the loop has live-in or live-out scalar variables, and the nature of the register allocation strategy employed. In this paper we have detailed the code generation schemas, both for DO-loops and WHILE-loops, for certain combinations of assumptions regarding hardware support. Two of these schemas (1 and 4) do not compromise performance. Three other schemas (1pc, 1s and 4c) trade varying amounts of performance for more compact code.

Hardware support for speculative code motion is valuable with all of the modulo scheduled WHILE-loop schemas and for Schema 1s in the case of DO-loops as well. Predicated execution and rotating register files are needed for Schemas 4 and 4c with both types of loops. Predicated execution is also valuable when modulo scheduling either type of loop if control flow is present in the loop body.

References

1. (Special issue on IBM RISC System/6000 processor). *IBM Journal of Research and Development 34*, 1 (1990).
2. Allen, J.R., Kennedy, K., Porterfield, C., and Warren, J. Conversion of control dependence to data dependence. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages*, (1983).
3. Butler, M., *et al.* Single instruction stream parallelism is greater than two. In *Proceedings of the Eighteenth Annual International Symposium on Computer Architecture*, (Toronto, 1991).
4. Charlesworth, A.E. An approach to scientific array processing: the architectural design of the AP-120B/FPS-164 family. *IEEE Computer 14*, 9 (1981), 18-27.
5. Dehnert, J.C., Hsu, P.Y.-T., and Bratt, J.P. Overlapped loop support in the Cydra 5. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, (Boston, Mass., 1989), 26-38.
6. Ebcioglu, K., and Nakatani, T. A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture. In *Languages and Compilers for Parallel Computing*, Gelernter, D., Nicolau, A., and Padua, D., Editor. 1989, Pitman/The MIT Press: London. p. 213-229.
7. Fisher, J.A. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers C-30*, 7 (1981).
8. Foster, C.C., and Riseman, E.M. Percolation of code to enhance parallel dispatching and execution. *IEEE Transactions on Computers C-21*, 12 (1972), 1411-1415.
9. Hsu, P.Y.-T. Highly Concurrent Scalar Processing. Coordinated Science Lab. Technical Report CSG-49. University of Illinois, 1986.
10. Jain, S. Circular scheduling: a new technique to perform software pipelining. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, (1991), 219-228.
11. Lam, M. Software pipelining: an effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, (1988), 318-327.
12. Lee, R.L., Kwok, A.Y., and Briggs, F.A. The floating point performance of a superscalar SPARC processor. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, (Santa Clara, California, 1991), 28-37.
13. Mahlke, S.A., *et al.* Sentinel scheduling for VLIW and superscalar processors. In *Proceedings of the The Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, (Boston, Massachussetts, 1992).
14. Nicolau, A., and Fisher, J.A. Measuring the parallelism available for very long instruction word architectures. *IEEE Transactions on Computers C-33*, 11 (1984), 968-976.
15. Nicolau, A., and Potasman, R. Realistic scheduling: compaction for pipelined architectures. In *Proceedings of the 23th Annual Workshop on Microprogramming and Microarchitecture*, (Orlando, Florida, 1990), 69-79.
16. Rau, B.R., and Glaeser, C.D. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the Fourteenth Annual Workshop on Microprogramming*, (1981), 183-198.
17. Rau, B.R., Lee, M., Tirumalai, P., and Schlansker, M.S. Register allocation for software pipelined loops. In *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation*, (San Francisco, 1992).
18. Rau, B.R., Schlansker, M.S., and Tirumalai, P.P. Code generation schemas for modulo scheduled DO-loops and WHILE-loops. Technical Report HPL-92-47. Hewlett Packard Laboratories, 1992.
19. Rau, B.R., Yen, D.W.L., Yen, W., and Towle, R.A. The Cydra 5 departmental supercomputer: design philosophies, decisions and trade-offs. *IEEE Computer 22*, 1 (1989).
20. Riseman, E.M., and Foster, C.C. The inhibition of potential parallelism by conditional jumps. *IEEE Transactions on Computers C-21*, 12 (1972), 1405-1411.
21. Su, B., and Wang, J. GURPR*: a new global software pipelining algorithm. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, (Albuquerque, New Mexico, 1991), 212-216.
22. Tirumalai, P., Lee, M., and Schlansker, M.S. Parallelization of loops with exits on pipelined architectures. In *Proceedings of the Supercomputing '90*, (1990), 200-212.
23. Tjaden, G.S., and Flynn, M.J. Detection and parallel execution of parallel instructions. *IEEE Transactions on Computers C-19*, 10 (1970), 889-895.
24. Touzeau, R.F. A FORTRAN compiler for the FPS-164 scientific computer. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, (1984), 48-57.
25. Wall, D.W. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, (1991), 176-188.

```
         DO 10 I = 1,N
            Q    = U(I) * Y(I)


            Y(I) = X(I) + Q


            X(I) = Q - V(I) * X(I)



10       CONTINUE
```
(a)

```
P1       t01 = iadd(t01,#4)
R1       t02 = load(t01)
P2       t03 = iadd(t03,#4)
R2       t04 = load(t03)
M1       t05 = fmul(t02,t04)
P3       t06 = iadd(t06,#4)
R3       t07 = load(t06)
A1       t08 = fadd(t07,t05)
W1             stor(t03,t08)
P4       t09 = iadd(t09,#4)
R4       t10 = load(t09)
M2       t11 = fmul(t10,t07)
S1       t12 = fsub(t05,t11)
W2             stor(t06,t12)
B1             brtop
```
(b)

Figure 1: (a) A sample FORTRAN DO-loop. (b) The intermediate representation of the body of the loop.

Time	Time Modulo 3	IALU 1	IALU 2	Memory Port 1	Memory Port 2	Multiplier	Adder	Instruction Unit
0	0	P1	P2					
1	1	P3		R1	R2			B1
2	2		P4	R3				
3	0	---	---		R4			
4	1	---		---	---			
5	2		---	---				
6	0	---	---		---	M1		
7	1	---		---	---			
8	2		---	---		M2		
9	0	---	---		---	---		
10	1	---		---	---		A1	
11	2		---	---		---		
12	0	---	---	W1	---	---	S1	
13	1	---		---	---		---	
14	2		---	---	W2	---		

(a)

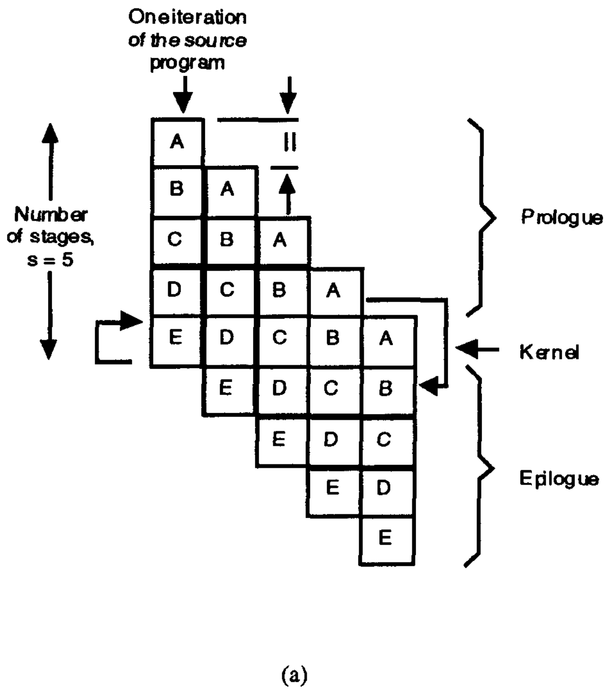
Time Modulo 3	IALU 1	IALU 2	Memory Port 1	Memory Port 2	Multiplier	Adder	Instruction Unit
0	n:P1	n:P2	n-4:W1	n-1:R4	n-2:M1	n-4:S1	
1	n:P3		n:R1	n:R2		n-3:A1	n:B1
2		n:P4	n:R3	n-4:W2	n-2:M2		

(b)

Figure 2: (a) Modulo schedule for the example of Figure 1. (b) Record of execution for a single stage during the steady state (assuming register renaming). The label before the colon indicates the iteration to which the operation belongs. Iteration n has just begun.

Instruction	Stage	Modulo Scheduled Kernel Code
0	0	`t01 = iadd(t01,#4),     t03 = iadd(t03,#4),`
	1	`t10 = load(t09),`
	2	`t05 = fmul(t02,t04),`
	4	`      stor(t03,t08),     t12 = fsub(t05,t11);`
1	0	`t06 = iadd(t06,#4),     t02 = load(t01),     t04 = load(t03),     brtop,`
	3	`t08 = fadd(t07,t05);`
2	0	`t09 = iadd(t09,#4),     t07 = load(t06),`
	2	`t11 = fmul(t10,t07),`
	4	`      stor(t06,t12);`

Figure 3: Kernel code after modulo scheduling. (Operations in a single instruction are separated by stage for illustrative purposes only. There is no such distinction in the code.)



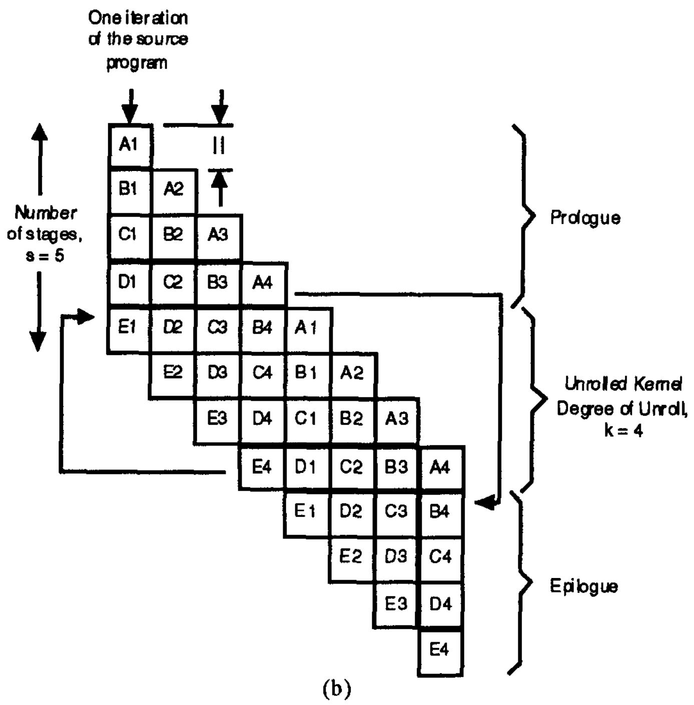
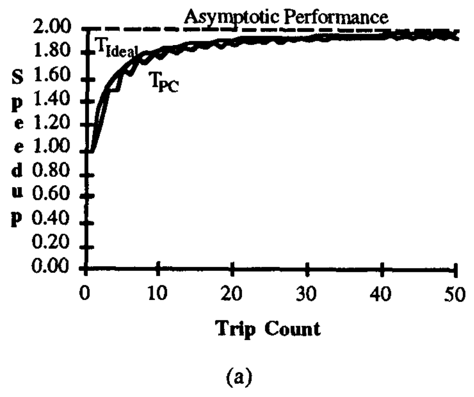
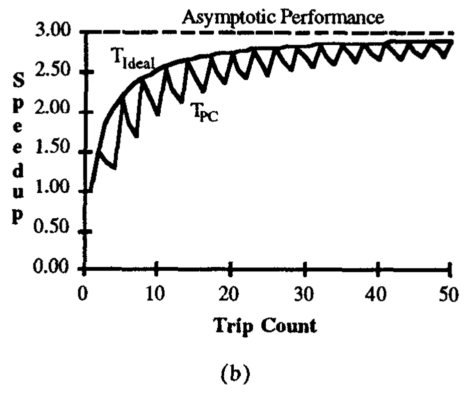
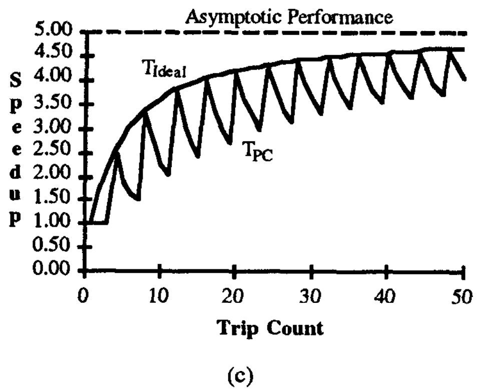


Figure 4: (a) The code schema for the modulo scheduled loop. (b) Kernel-unrolled loop structure.







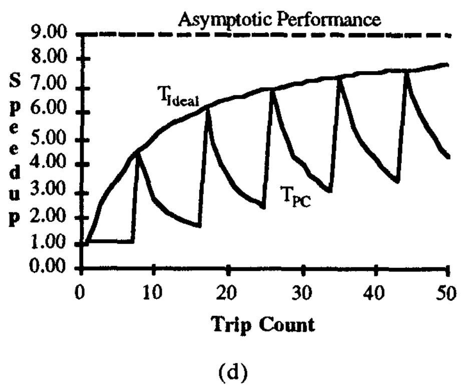


Figure 5: Speedup as a function of trip count for a pre-conditioned loop vs. the best achievable code schema. (a) For II = 15, SC = 2, K_{min} = 2, and SL = 20. (b) For II = 6, SC = 3, K_{min} = 3, and SL = 16. (c) For II = 3, SC = 5, K_{min} = 4, and SL = 15. (d) For II = 3, SC = 9, K_{min} = 9, and SL = 25.

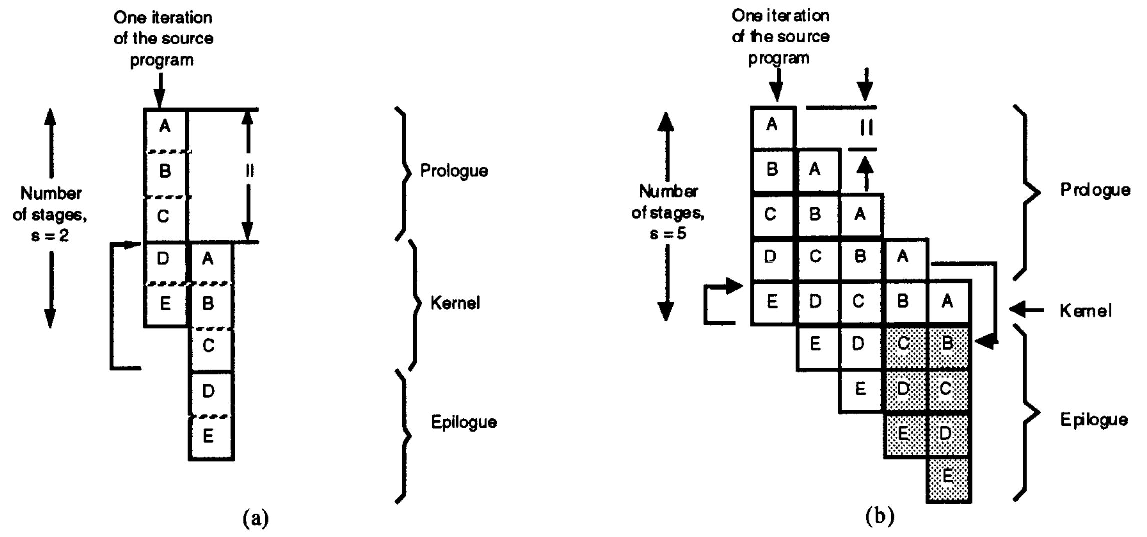


Figure 6: A modulo scheduled WHILE-loop (a) without and (b) with speculative execution

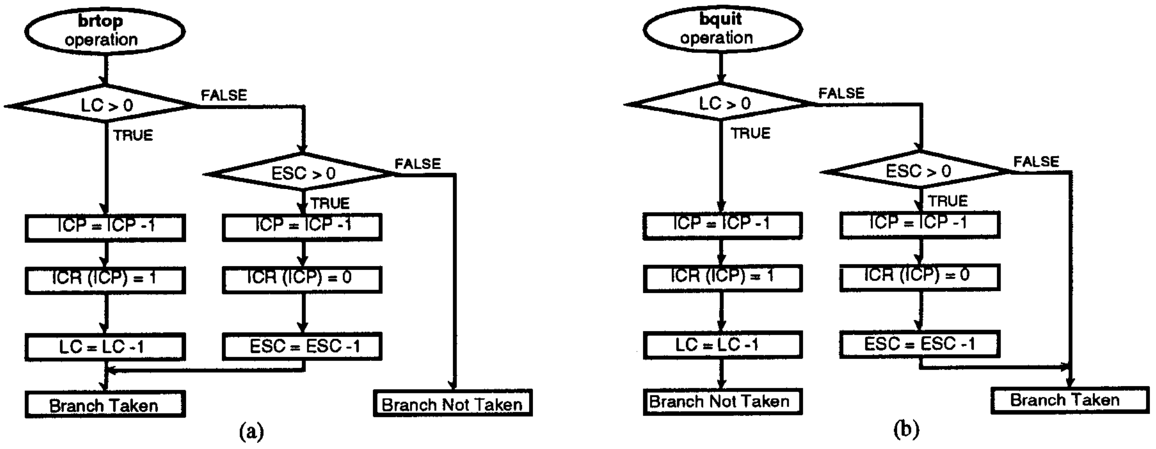


Figure 7: (a) The brtop operation. (b) The bquit operation

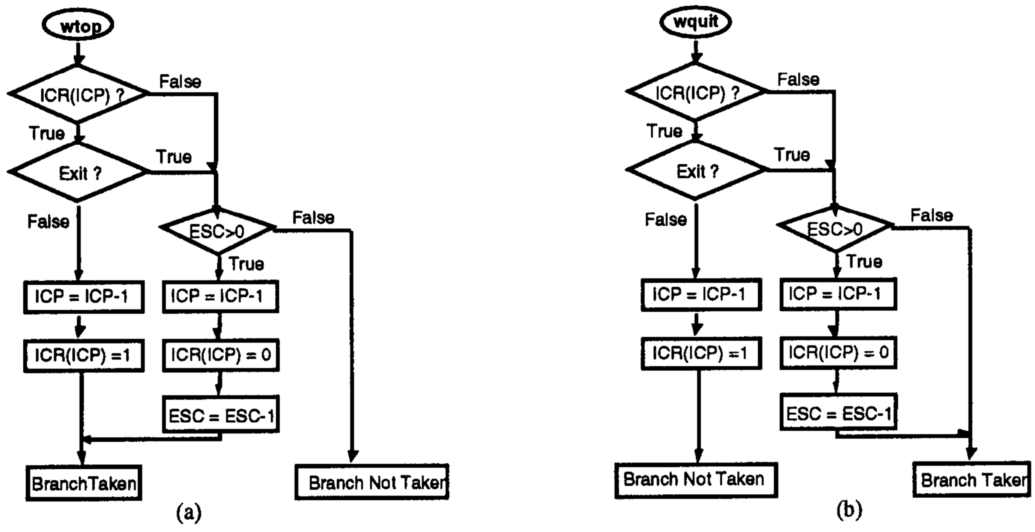
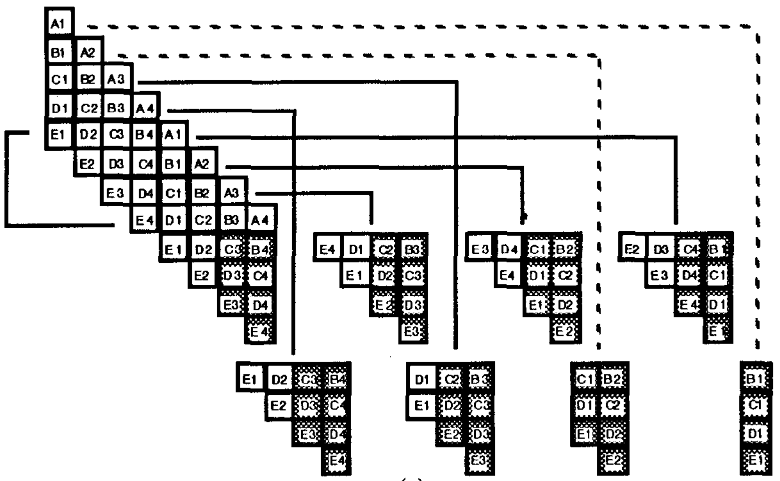
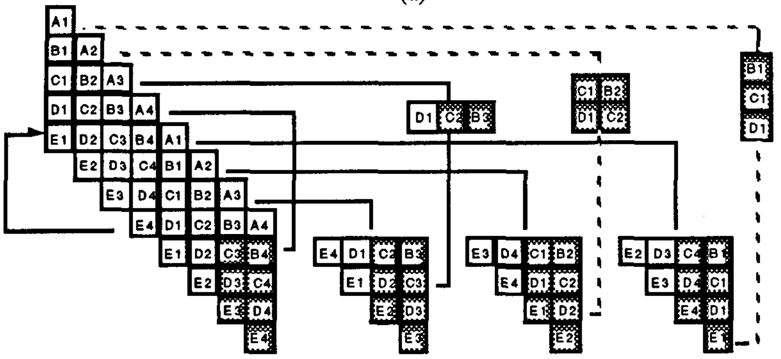


Figure 8: (a) The wtop operation. (b) The wquit operation.

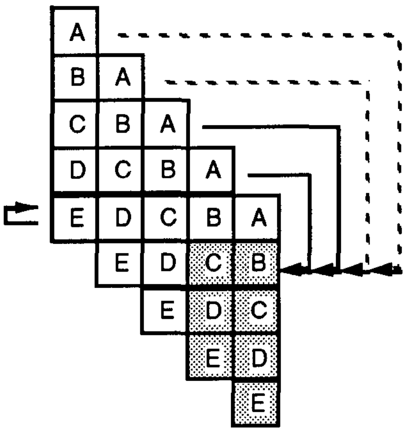


(a)

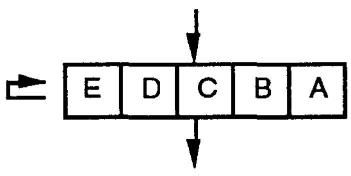


(b)

Figure 9: (a) Code schema 1 (without predicated execution or rotating register files). (b) Code schema 1 after removal of redundant code.



(a)



(b)

Figure 10: (a) Code schema 4 (with predicated execution and rotating register files). (b) Kernel-only code schema.

Instruction	Stage	Modulo Scheduled Loop Code
0	0	`r21 = iadd(r22,# 4) if p0,  r00 = iadd(r01,#4)  if p0,`
	1	`r04 = load(r15)     if p1,`
	2	`r22 = fmul(r21,r19) if p2,`
	4	`stor(r04,r17) if p4,  r14 = fsub(r24,r14) if p4`
1	0	`r08 = iadd(r09,#4)  if p0,  r19 = load(r21)     if p0,  r17 = load(r00) if p0,     brtop,`
	3	`r16 = fadd(r08,r23) if p3`
2	0	`r14 = iadd(r15,#4)  if p0,  r05 = load(r08)     if p0,`
	2	`r12 = fmul(r05,r07) if p2`
	4	`stor(r12,r14) if p4`

(a)

Start of Itern. #	Loop Counter	ESC																ICR 43210
	6	4																00001
1	6	4	(E)	(D)	(C)	(B)	A											00001
2	5	4		(E)	(D)	(C)	B	A										00011
3	4	4			(E)	(D)	C	B	A									00111
4	3	4				(E)	D	C	B	A								01111
5	2	4					E	D	C	B	A							11111
6	1	4						E	D	C	B	A						11111
7	0	4							E	D	C	B	A					11111
	0	3								E	D	C	B	(A)				11110
	0	2									E	D	C	(B)	(A)			11100
	0	1										E	D	(C)	(B)	(A)		11000
	0	0											E	(D)	(C)	(B)	(A)	10000
	0	0																00000

(b)

Figure 11: (a) Kernel code for the kernel-only code schema. (b) Operation of the brtop instruction while executing kernel-only code for 7 iterations of a loop with 5 stages