# Modulo Schedule Buffers

Matthew C. Merten and Wen-mei W. Hwu
Center for Reliable and High-Performance Computing
University of Illinois, Urbana, IL 61801
{merten, hwu}@crhc.uiuc.edu

## Abstract

*As VLIW/EPIC processors are increasingly used in real-time, signal-processing, and embedded applications, the importance of minimizing code size and reducing power is growing. This paper describes a new architectural mechanism, called the Modulo Schedule Buffers, that provides an elegant interface for the execution of modulo scheduled loops. While the performance is similar to that of kernel-only modulo scheduling, this mechanism has a number of advantages, including minimal code expansion. Rather than generating fully-scheduled kernels, the compiler generates a sequential form of the modulo scheduled loop body. Using the sequential form, the hardware internally synthesizes the prologue, kernel, and epilogue. In addition,* while *loops can be scheduled with fewer constraints and fewer explicit prologues/epilogues than with existing mechanisms. Because the hardware controls loop execution, the burden of modulo schedule loop control is lifted from the predicate register file, allowing for a less rigorous predication implementation. Finally, hardware control limits the interrupt latency when using the EQ explicit latency model to the execution latency of one iteration, rather than the whole loop invocation.*

## 1. Introduction

Horizontal computer architectures, such as *very long instruction word* (VLIW), superscalar, and *explicitly parallel instruction computing* (EPIC) architectures, enable rapid execution of applications by exploiting *instruction-level parallelism* (ILP). Software pipelining is a class of techniques for optimizing loop execution throughput by exploiting the ILP present across loop iterations [1], [2], [3], [4], [5]. Techniques in this class allow instructions from successive iterations to execute in parallel with those of previous iterations, effectively overlapping loop iteration execution. Unlike loop unrolling-based optimization, software pipelining techniques maintain the iteration overlap throughout the execution of the loop. Furthermore, by utilizing hardware rotating registers, code expansion can be held to moderate levels as compared to unrolling, which is an important consideration for embedded applications. *Modulo scheduling* [6] is a form of software pipelining that initiates loop iterations at a constant rate, called the *initiation interval* (II).

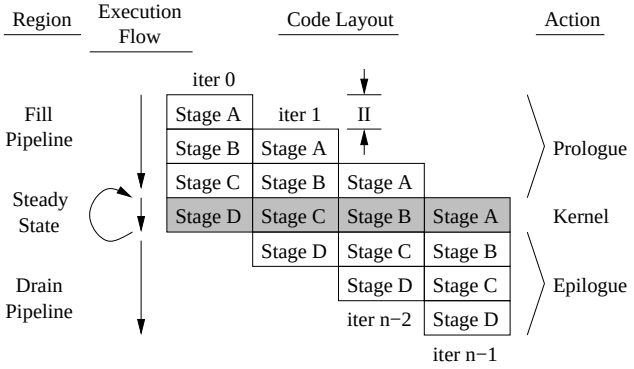



**Figure 1. Generalized execution of modulo scheduled loops with 4 overlapped iterations.**

Figure 1 depicts the general structure of a modulo scheduled loop. Each loop iteration is divided into *stages* of execution of II cycles each, four stages in the example, effectively pipelining the execution of each iteration. Therefore, a new loop iteration can be begin execution every II cycles. At maximum utilization, a steady-state condition, called the *kernel*, is reached where all loop stages are concurrently executing on behalf of successive iterations. Once the steady-state condition is reached, re-executing the kernel advances each active loop iteration by one stage, ending one iteration and beginning another. The code prior to the kernel, called the *prologue*, ramps-up the pipeline by executing selective stages. Likewise, the *epilogue* ramps-down the pipeline from the kernel, finishing the in-flight iterations.

Traditional implementations of modulo scheduling require that the prologue, kernel, and epilogue regions be fully specified in the code. Note that in the worst case, each instruction in the loop body is replicated $n$ times (excluding any necessary versioning), where $n$ is the number of concurrently executing iterations of the loop. Such replication contributes to overall code growth, a serious concern due to tight constraints on instruction memory and cache resources, especially in embedded applications. In order to overlap loop iterations, anti-dependences must often be broken. *Modulo variable expansion* (MVE) [5] is a tech-

nique which can be applied that creates several versions of the kernel using different registers. This, too, can dramatically increase code size. *Kernel-only* modulo scheduling [7] reduces the specification of the loop to just the kernel by utilizing predication to enable execution of select kernel instructions during the prologue and epilogue phases, and rotating registers to eliminate the need for kernel versioning. However, with all of these techniques, a number of difficult issues remain to be addressed. First, kernel-only modulo scheduling requires extensive instruction set architecture (ISA) support for full predication, a feature that many architects are unwilling to implement because of the additional design complexity.

Second, kernel-only modulo scheduling of *while* loops is more complicated than for *counted* loops. *While* loops generally consist of several speculative pipeline stages that are controlled by predicates. However, without support for block alteration of the predicates that control prologue and epilogue stages, a variety of extra scheduling constraints must be observed, complicating the scheduling process. Modulo scheduling on the Itanium Architecture [8], for example, constrains what types of code constructs can be executed in the prologue. This problem will be further discussed in Subsection 2.4.

Third, some architectures utilize the *equals* (EQ) latency model for register result write back. Under this model, the results of operations are not written to their destination registers until their exact architected latency has expired, and never sooner. By exploiting this feature, a single register may hold several in-flight values simultaneously, which may reduce the need for kernel versioning. A side effect of exploiting the EQ model in this way is that execution of the loop cannot be interrupted without storing each of the in-flight register values and their associated write-back times at the point of interruption.

Fourth, repeated fetching of a loop body from the instruction cache or memory unnecessarily wastes energy. By storing a modulo scheduled loop in a dedicated, compiler-controlled buffer close to the functional units, the fetch mechanism can be temporarily disabled, thus saving energy. Currently, *Zero-Overhead Loop Buffers* (ZOLB) [9] are used to reduce energy for *counted* loops, but an extended technique could be employed for both modulo scheduled and *while* loops.

In order to overcome these deficiencies, our proposed solution, called the *Modulo Schedule Buffers* (MSBs) [10], employs a hardware-controlled, modulo schedule execution mechanism and an alternate method for expressing the loop body to the hardware. Our mechanism reads a *sequential* version of the loop body (instead of a kernel version) and *internally* generates and issues prologue, kernel, and epilogue instructions to the functional units. Full predication for loop control becomes unnecessary in this model because the active status of loop body instructions is maintained in the hardware through a series of status registers. Because the hardware manages loop execution, it is free to ramp

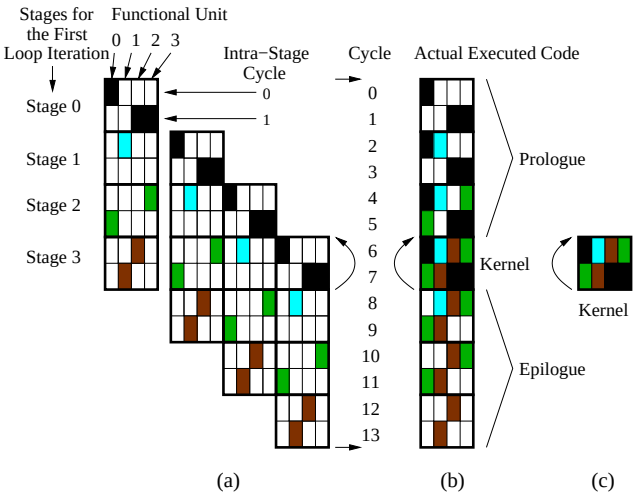


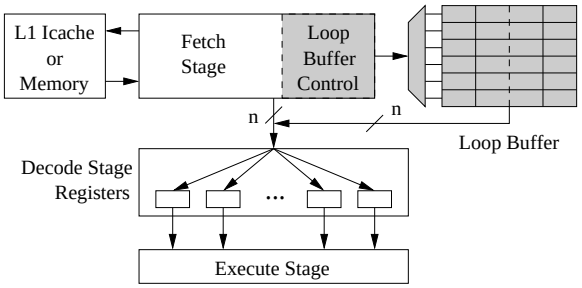

**Figure 2. Modulo scheduled loop with II of 2.**

down the pipeline part way through the loop execution to service an interrupt, and then ramp up execution afterward. This limits the interrupt latency on EQ latency hardware to a mere loop iteration instead of the remainder of the loop, and frees the programmer of interrupt latency concerns. Furthermore, by generating the loop components internally, the instructions can be located in dedicated buffers, thus avoiding repeated access to the entire instruction fetch mechanism. While the compiler must still generate a correct modulo schedule, it is free from generating the prologue and epilogue. Thus, it achieves a significant reduction in code size compared to explicitly generated code, and is comparable to kernel-only code size. This hardware enhanced scheme achieves the same performance as the fully-specified standard method.

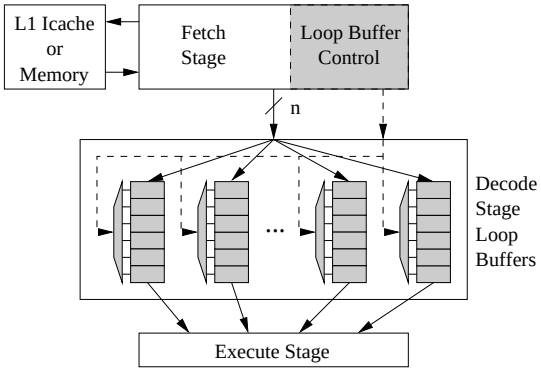
### 1.1. Modulo Scheduling and Kernel-Only Code

Often times an II of 1 cycle is not achievable due to scheduling or resource constraints. Figure 2(a) depicts an example loop that has an II of 2, hence a new iteration of the loop is started once every two cycles. The stages for the first iteration are labeled, and resource usages for each of the stages are shaded (black for the first stage). While executing stage 2 of iteration 1, stage 1 of iteration 2 is also executing. Furthermore, it is not a requirement that only 1 instruction be executed in each cycle for a given loop iteration. For example, two instructions are executed in the second cycle of the first stage, both on behalf of the same iteration. Likewise, no instructions are executed in the second cycle of the second stage. Note that even though no resources are used for this cycle in this stage, the same cycle in other stages may be executing instructions for other iterations. Figure 2(b) depicts the total resource usage during the various regions of loop execution.

*Kernel-only* modulo scheduling [7] utilizes a set of *stage predicates* to control the execution of the various loop stages. This technique relies on the stage predicates to dynamically activate the necessary instructions to comprise




(a) Fetch stage loop buffer




(b) Decode stage loop buffers

**Figure 3. Loop buffer arrangement.**

the prologue and epilogue from the kernel, eliminating the need for an explicit prologue or epilogue. For example, each of the similarly shaded blocks in figure will be predicated on the same stage predicate, so that only the kernel (Figure 2(c)) need be expressed in the code. To perform this task efficiently, extensive use of predication and therefore a fully predicated (ISA) is required. Currently, only a few processors support the level of predication required, while many support a small subset. Sometimes, with some clever data layout and instruction scheduling (specifically of the loop back branch instruction), portions of the prologue and epilogue may be simply eliminated, but this is not always possible [11].

## 2. Architecture

In order to overcome the challenges mentioned in Section 1, our proposed mechanism integrates prologue and epilogue creation and loop iteration management into the architecture itself. This is accomplished by extending and enhancing the concept of a loop buffer present in many DSPs, shown in Figure 3(a). In this traditional configuration, the loop buffer, part of the fetch pipeline stage, acts primarily like a compiler-controlled cache of the instruction stream. This feature reduces power consumption by preventing frequent redundant accesses to the cache and by disabling portions of the fetch mechanism when executing from the buffer. For example, the Lucent DSP16000 utilizes a 31-instruction buffer [9]. However, in an alternate

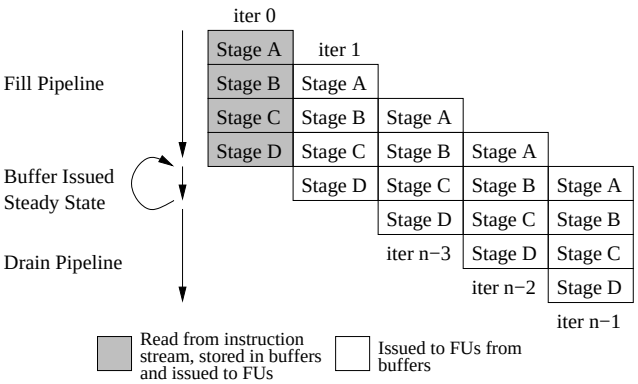



**Figure 4. Process of filling and executing from the Modulo Schedule Buffers.**

configuration, the dispatched instructions could be buffered in the decode stage of the pipeline, Figure 3(b). In this configuration, instructions would be buffered in a location associated with their functional unit and with their cycle within the loop, as opposed to their location in the instruction stream. This configuration enables efficient support for modulo scheduled code, and may further reduce power by allowing decoders to be disabled, if decoded instructions are stored in the buffers.

Without loss of generality, we will use Texas Instruments' TMS320C6000 architecture as a base to illustrate the operation of our mechanism. In the original TI 'C6x architecture, the dispatch stage of the pipeline consists of a network that routes the instructions in the *fetch packet* (8 aligned instructions) to their correct functional units [12]. This network links each instruction dispatch register to each instruction decode register, which is the beginning of the path to the associated functional unit. It is the responsibility of the code generator to insure that there is only a single instruction in each *execute packet* (subset of the instructions in the fetch packet that can be issued in parallel) for any functional unit. In other words, within an execute packet, there is no hardware contention for the routing wires or the functional units. To keep the fetch logic simple, execute packets may not cross fetch packet boundaries.

Rather than specify the prologue, kernel, and epilogue in the instruction stream, we propose to include a *sequential* copy of the loop in the code and use specialized loop buffers to collect the instructions, forming the prologue, kernel, and epilogue at runtime. Figure 4 depicts this process. The shaded region represents the sequential view of the loop iteration, as stored in the instruction stream. As it is read from the stream, it is issued to the functional units on behalf of the first iteration. Meanwhile, it is also stored in the buffers, from where subsequent loop iterations will be issued. For example, while Stage B is read and issued for Iteration 0, Stage A is issued from the buffers on behalf of Iteration 1. The compiler must understand that the execution of the loop iterations will overlap and must properly modulo schedule the loop body.

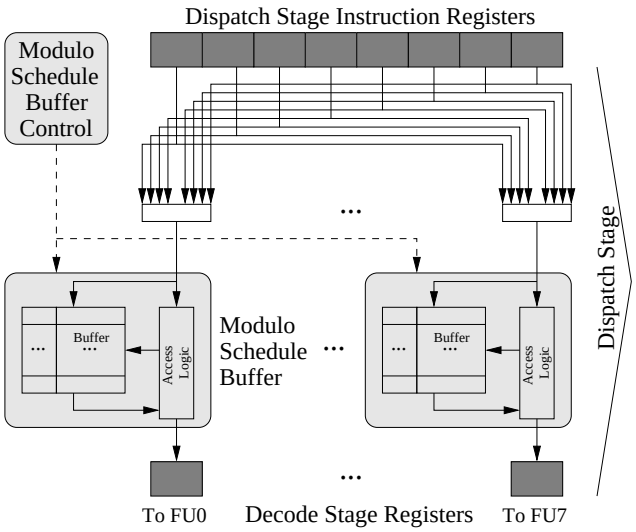



**Figure 5. TI 'C6x dispatch pipeline stage with the addition of the Modulo Schedule Buffers and related control logic.**

## 2.1. Active Stage Issue

Figure 5 depicts a high-level view of the dispatch pipeline stage with the addition of the Modulo Schedule Buffers. The system consists of a MSB for each functional unit pipeline and a single control unit. Each buffer consists of a small table indexed by the current cycle of the executing stages, as shown in center box of Figure 6. Each table entry consists of a stage bit tag and an instruction. In steady-state, the buffers contain the kernel of the modulo scheduled loop. Each instruction is stored in a buffer associated with its functional unit at an entry corresponding to its cycle within the kernel. For EPIC and VLIW machines, the modulo scheduling algorithm ensured there was no contention for functional units and resources, and therefore there is no contention for MSB entries since there is a one-to-one correspondence between schedule location and buffer entry. During steady-state, execution progresses through the buffers cycle-by-cycle, just like it would progress cycle-by-cycle through a software-specified kernel. The cycle decoder selects the current cycle, and is driven by a counter that counts from 0 to (II-1).

In order to produce a prologue and an epilogue, a mechanism is provided to selectively issue instructions from within the buffers. Referring back to Figure 2(b), the epilogue begins immediately following the kernel. Notice that in the beginning of the epilogue, only Stages 1 to 3 execute, but not Stage 0. In this way, previously initiated iterations are allowed to complete, but no new iterations are begun. In order to generate the epilogue from within the hardware, select instructions must be deactivated. This is accomplished by tagging the individual instructions with their appropriate stage and by maintaining a set of the active stages in the Active Stages Bit Vector. At the end of a stage, all of the bits in the vector are shifted to the next sequential bit to indicate that all of the active iterations have moved to their next stages. If the instruction stage tag stored in the buffer is also stored as a bit vector, then a simple logic function can be used to determine if the instruction should be issued. If none of the bits in the Active Stages Bit Vector match the bits in the entry's tag, then a NOP is issued to the functional unit because that entry is not active in that stage. The value filled into the first bit of the Active Stages Bit Vector after the shift depends on type and state of the loop, which be discussed in subsequent sections.

## 2.2. Buffer Initialization

Prior to loop execution, all of the Stage Bit Tags are cleared to zero, indicating that no valid instructions are held in the entries. As the first iteration of the loop progresses, its instructions are inserted from the instruction stream into the their appropriate entries and are tagged with their appropriate stage. These instructions are also issued directly to functional unit pipelines. As other iterations are initiated, instructions for those iterations are all issued from the Modulo Schedule Buffers, even as instructions for later stages of the first iteration are issued and being inserted into other entries. The control mechanism that directs the loading and issuing process is shown in Figure 7. Four values, shaded gray at left, are required to properly initialize the loop hardware. These values are communicated to the hardware by writing into control registers, similar to the setup for traditional *counted* loop support. First, the initiation interval (II) is required to limit the Current Cycle Counter's range, which in turn cycles through the entries in the buffers.

Second, the total number of stages in the loop body is required to control the loading of the buffers. Using this counter and the II, the controller fetches execute packets from the instruction stream and stores them into the modulo schedule buffers until the entire loop is read (II*NumStages cycles). `NOP` instructions that serve as filler in the instruction stream are not associated with any particular functional unit and are not written to any of the buffers. The Buffer Access Logic, detailed in Figure 6, enables the write of `non-NOP` first iteration instructions into the buffer. Current Cycle Counter is used to select the correct entry in the buffer, while the Current 1st Iteration Stage Counter is used to tag each instruction with its stage. When all loop execute packets are read, the first iteration of the loop is complete. In fact, the next execute packet waiting in the instruction decode registers contains the first instructions along the fall-through path of the loop back branch. At this point, the fetch unit may be disabled, saving power, until all iterations (which are completely dispatched from the Module Schedule Buffers) are complete.

The number of epilogue stages (NumEpilogueStages) serves two purposes. First, when a loop exit condition is detected (loop back branch condition is false or side exit condition is true) previous loop iterations may not have com-

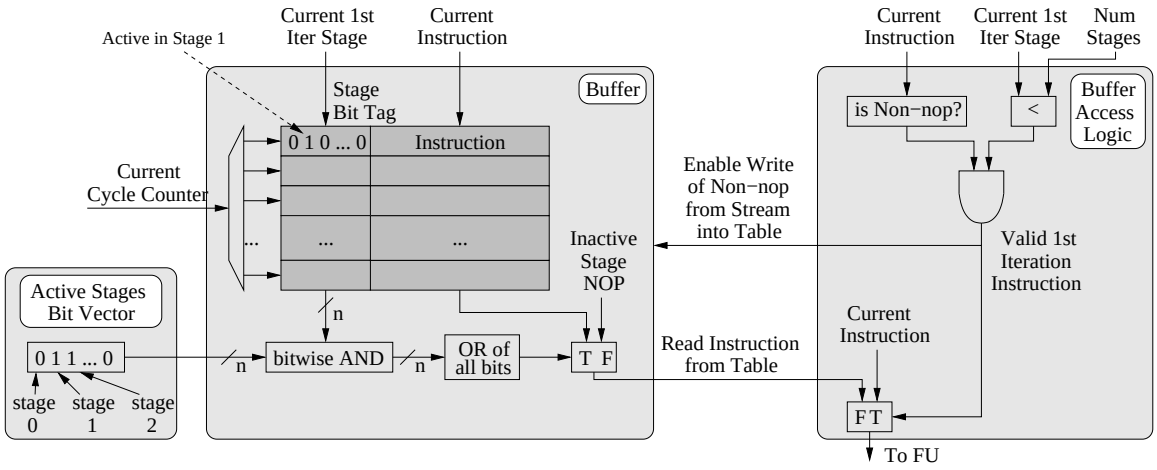



**Figure 6. The Modulo Schedule Buffer and the Buffer Access Logic.**

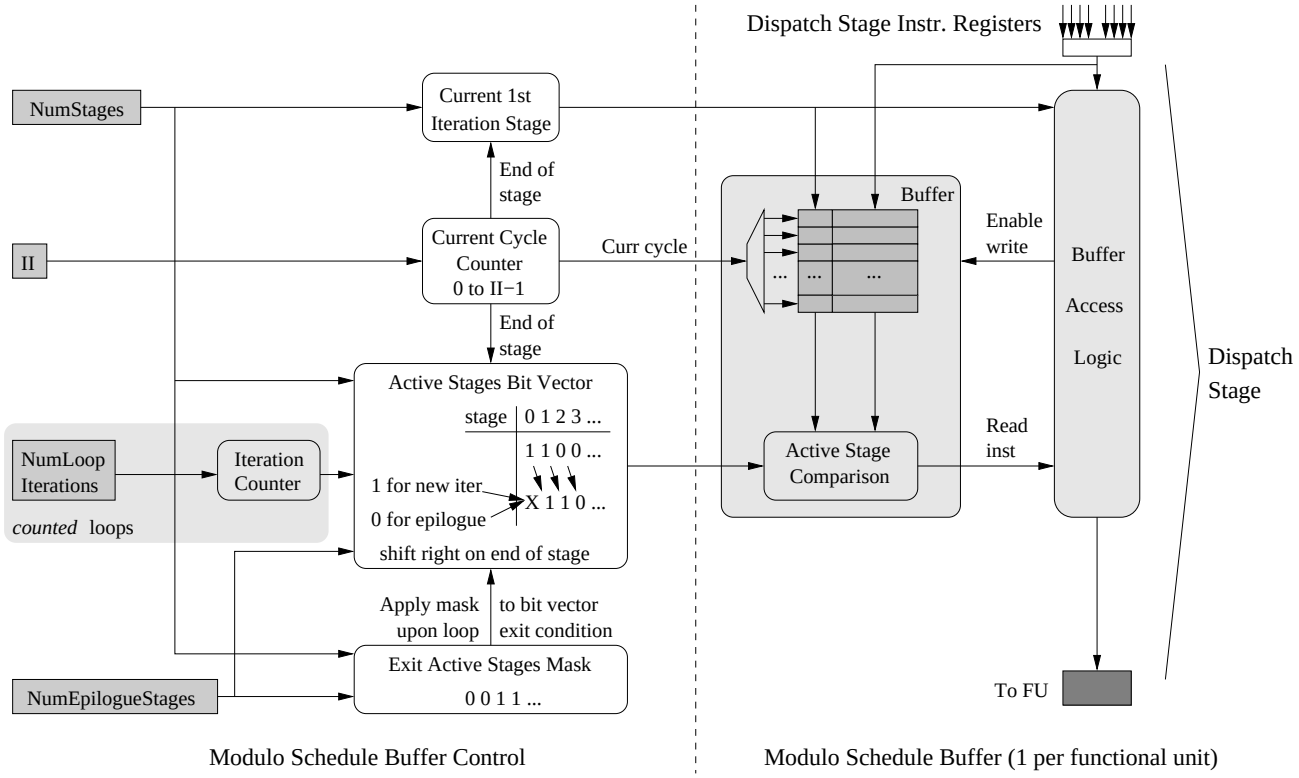



**Figure 7. Detailed view of Modulo Schedule Buffer Control unit.**

pleted execution. This register contains a count of the number of further kernel iterations that must be completed to finish the prior iterations. Second, this value enables the production of an Exit Active Stages Mask required to support *while* loops. *While* loop operation will be discussed in more detail in Subsection 2.4. This parameter could be passed in by the compiler, or it could be detected at run time. Since modulo scheduling schema require that all branches occur in the same pipeline stage, the hardware could monitor the buffer fill process for any branch instruction. The number of epilogue stages can be then be determined by subtracting the branch's stage number from NumStages.

Last, the total number of loop iterations (NumLoopIterations) can be used to support efficient *counted* loops. This value is stored in a counter that is decremented each time a loop iteration is initiated. The iteration is initiated by shifting a 1 bit into the Active Stages Bit Vector. When the loop iteration counter reaches zero, no more iterations need be initiated. This is accomplished by shifting a 0 bit into the Active Stages Bit Vector for stage 0. However, previous iterations may have yet to complete (represented by 1 bits remaining in the vector). NumEpilogueStages kernel iterations will be executed to complete prior iterations (shifting in 0 bits at the end of each stage) before execution will fall

through to post-loop code. For *counted* loops, no loop back branch is required since the iteration length and iteration count are predetermined values.

In some cases, control will leave a modulo scheduled loop, execute other code, and then return to the loop. This is common when the modulo scheduled loop is nested in an outer loop. In these situations, it is possible to reinitialize the loop control registers while using the previously buffered loop, thus avoiding the repeated fill process. Several alternatives exist, including compiler determined reuse and run-time detected reuse.

### 2.3. Operational Example

Figure 8 depicts the *counted* loop operation of the Modulo Schedule Buffer system. In this example, three iterations of the loop are executed on a 4-issue machine. The loop body, as illustrated at the upper right corner of the figure, consists of two loads, one multiply, one add, and one shift instruction. The loop is modulo scheduled into three stages. Thus, each iteration requires 6 cycles to execute, with a throughput of one iteration per two cycles. The figure depicts the contents of each functional unit's MSBs for the ten cycles of the loop execution. Shaded buffer entries indicate the active stages that are issued to the FUs from the buffers, while the outlined and shaded entries indicate those that are filled into the buffers and issued to the functional units from the instruction stream. Actions during select cycles are described below:

**Initialization:** Initialize NumStages to 3, NumLoopIterations to 3, II to 2, NumEpilogueStages to 2, and Current Cycle Counter to 0. As in TI's 'C6x architecture, the initialization instructions may need to be scheduled several cycles prior to the loop body in order to overcome initialization latency. Assignment to the II or NumStages registers could be used to trigger filling of the buffer since these fields are required for both *counted* and *while* loops.

**Cycle 0:** Loop execution begins by shifting a 1 bit into the stage 0 location of the Active Stages Bit Vector. This initiates a new iteration and thus the Iteration Counter is decremented to 2. The first execute packet contains a load instruction. Since the Current 1st Iteration Stage counter is less than NumStages, a valid first iteration instruction is present in the execute packet. It is filled into the memory unit's buffer for cycle 0 and marked with stage 0, and issued to the memory functional unit.

**Cycle 1:** The Current Cycle Counter is incremented to 1. The second instruction is also a valid first iteration instruction. It is a load and is placed into cycle 1 of the memory unit's buffer and marked with stage 0.

**Cycle 2:** The Current Cycle Counter equals the II, so it is reset to 0, and the End of Stage condition is raised. Since the Iteration Counter is non-zero, a new iteration is initiated by shifting a 1 into the Active Stages Bit Vector, activating stages 0 and 1. The Iteration Counter is decremented to 1. From the instruction stream, a nop is read, which is not a useful instruction and is not written into any buffer. The nop is simply a place holder to contain the cycle stop bit for this empty cycle due to the two-cycle latency between the load in cycle 1 and the multiply in cycle 3. However, the instruction in cycle 0 of the memory unit is issued because its stage matches an active stage.

**Cycle 5:** Two instructions in the execute packet are filled into the buffers for the ALU and shifter units at cycle 1.

**Cycle 6:** The Current Cycle Counter is reset to 0. However, the Iteration Counter is already at zero, indicating that no new iterations should be initiated. Therefore a 0 bit is shifted in, leaving only stages 1 and 2 active. The NumEpilogueStages counter is decremented since execution is only completing previously initiated iterations. Coincidentally, the Current 1st Iteration Stage has also exceeded the loop body length. The next execute packet contains post-loop code. Therefore, no new instructions will be accumulated into the buffers.

**Cycle 10:** The Current Cycle Counter is reset to 0. NumEpilogueStages is also zero, indicating all epilogues are complete. Loop execution has ended.

### 2.4. Active Stage Management

In order to support *while* loops, iterations must be initiated speculatively, which requires that 1 bits be shifted into the Active Stages Bit Vector to enable those stages prior to any loop back decision. However, when a while loop exits, all speculative iterations must be squashed from the modulo schedule pipeline so their speculative results do not commit. Consider the loop in Figure 9(b) which consists of four stages, with a side exit and a loop back branch in stage 1. Note that iteration 1 must be speculatively initiated before the loop back decision in iteration 0 is reached. Now suppose that the loop back condition in iteration 1 fails. This indicates that the current iteration (1) and all previous iterations (0) should complete, but all future iterations (2 and 3) should be squashed. Once the loop back branch fails, a 0 will be shifted into the Active Stages Bit Vector so that iteration 3 is not initiated, and the 1 bit (circled) for iteration 2 must be cleared. In generalized loop execution, 1 bits must be shifted in during the prologue and kernel regions, speculative stage bits must be cleared on an exit condition, and 0 bits must be shifted in during the epilogue region.

Other architecture mechanisms have been proposed to support kernel-only scheduling of *while* loops through the use of a rotating predicate register file and special branching instructions [11]. In these architectures, the rotating predicates serve the same function as the Active Stages Bit Vector, however there is no support for selectively squashing the predicates of speculative iterations on an exit. Two primary options exist to alleviate this problem. First, explicit epilogues can be generated that issue the correct non-speculative stages on an exit, while the rotating predicates are used to issue the prologue from the kernel. However, this may require extra epilogue code at each loop exit point.

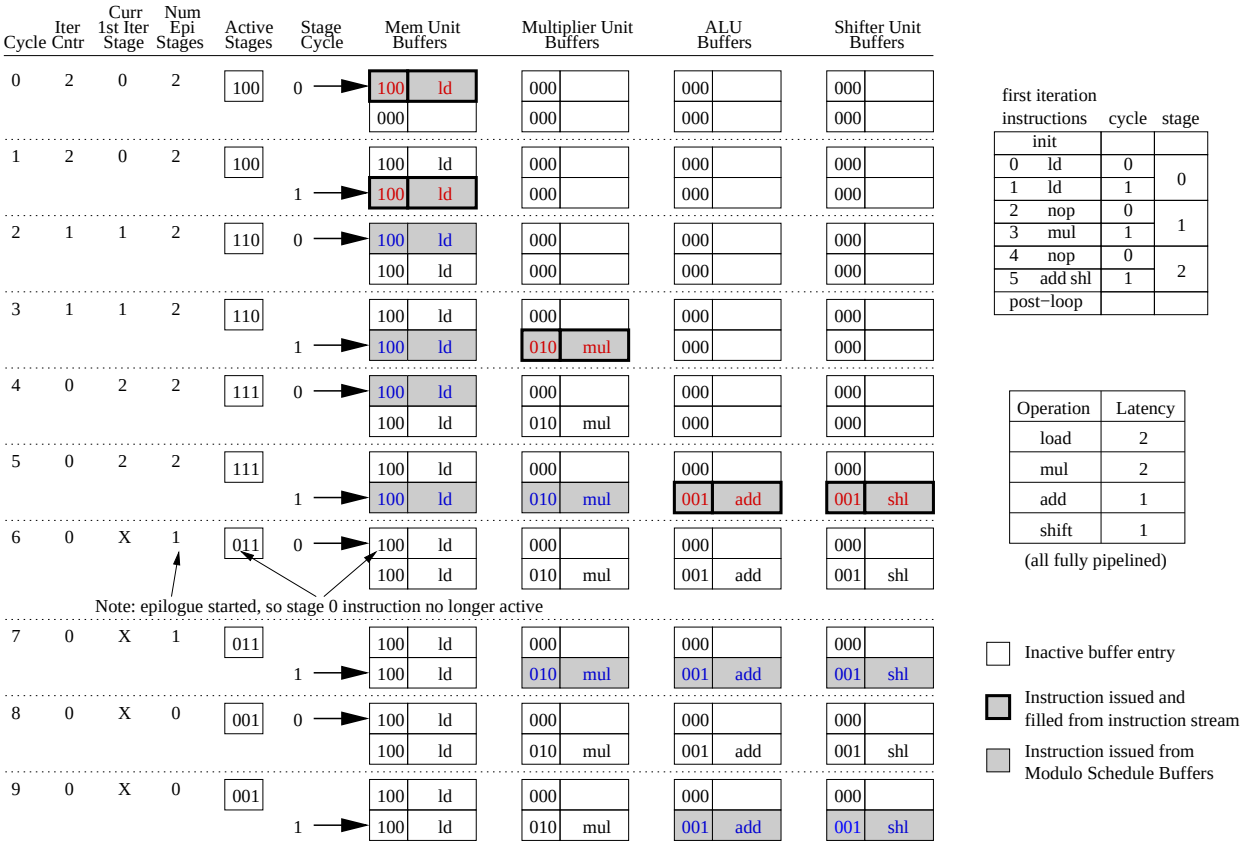



**Figure 8. Example loop execution using the Module Schedule Buffers.**

Second, the rotating predicates may be used to issue the epilogue from the kernel while an explicit prologue is provided. In this scheme, 0 bits are instead shifted into the predicate registers to disable unneeded stages in the epilogue region. Similar to this scheme, the Itanium Architecture [8] utilizes the predicates for the epilogue, but suggests that speculative stages be generated so that they can always safely execute, rather than specify an explicit prologue. However, this places a number of restrictions on scheduling. Consider one example from the shaded loop iteration in Figure 10(a). Normally, the predicate `p5` could only be set when stage 1 is actually active, as controlled by stage predicate `s1`, and cleared otherwise (unconditional-type predicate define of `p5`). Therefore, predicate `p5` actually contains the logical AND of the defining condition (`cond`) and the stage predicate. This predicate can then be used in place of the stage predicate on the branch in stage 3. However, as shown in Figure 10(b), the speculative stages no longer have stage predicates and execute unguarded in the Itanium Architecture. In the first cycle, stage 1 erroneously executes, potentially setting predicate `p5'` (`p5'` is different than `p5` due to versioning, as described in the next section). Since `p5'` may now be set to true in cycle 0, the branch in cycle 2 may erroneously execute. Therefore, predicate definitions that control non-speculative instructions cannot be speculatively generated, thus limiting the stage they may be scheduled in. To prevent this and other similar situations, a significant number of extra dependences must be obeyed during scheduling.

In our approach, we utilize an Exit Active Stage Mask to disable speculative iterations on an exit, allowing us to use the Active Stage Bit Vector for both prologue and epilogue code. In the example in Figure 9(b), the mask is applied to disable stage 1 of speculative iteration 2. Each stage of the loop is classified as speculative, exit, or non-speculative. After execution of the loop back branch, only the non-speculative iterations must complete their execution. Therefore, the mask consists of bits representing the non-speculative stages. This mask is constructed as shown in Figure 9(a). Figure 9(c) depicts the same loop, only with a taken early side exit. In this case, the remainder of the current iteration should not complete its execution. Therefore, the same mask that enables only the non-speculative stages can be applied. When a side exit is detected, execution continues in the kernel to complete previous iterations while the fetch unit begins fetching from the exit's target. The target instructions are buffered in dispatch registers until the loop has completed.

Lastly, as with *counted* loops, a loop back branch is not necessary for *while* loops since the hardware controls the iterations of the kernel. However, an instruction that communicates to the hardware the result of the loop back condition is required. Similarly, side exit branches do not actually alter control flow, but rather notify the hardware of

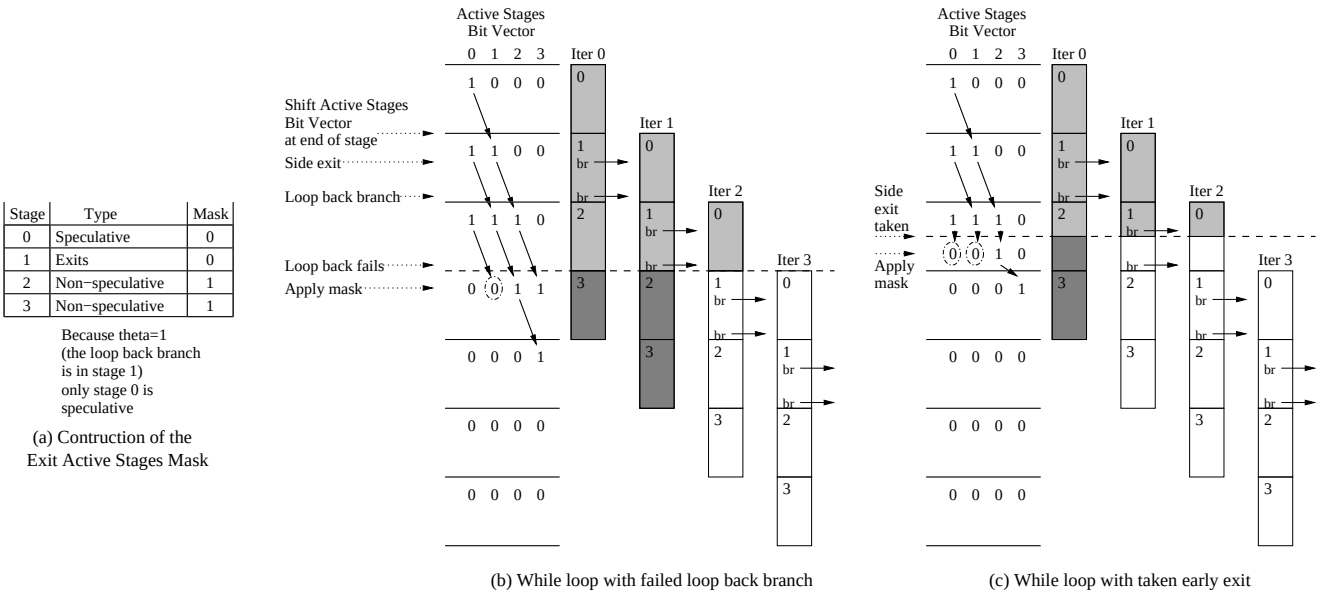



**Figure 9. Four stage *while* loop with loop back branch in stage 1.**

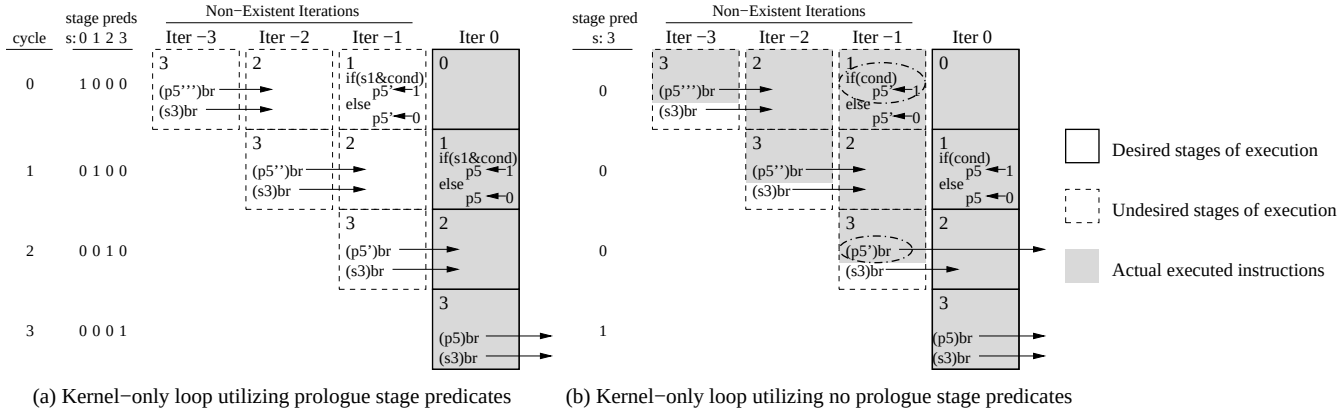



**Figure 10. Four stage *while* loop with unpredicated speculative stages.**

the side exit condition and report the target address. A condition failure or any taken side exit branch will cause the mask to be applied. The hardware will automatically iterate through the kernel for NumEpilogueStages, decrementing the counter at the taken side exit condition (because the current stage is now considered an epilogue stage for previous iterations) and at the end of each II.

## 2.5. Register Versioning

Many loop bodies contain values that have long lifetimes. When a variable's lifetime is greater than II cycles, its lifetime will overlap with the lifetime of the same variable in the next iteration. In order to keep the values for each iteration separate, register versioning (renaming) is employed. In software, this technique is called *Modulo Variable Expansion* (MVE) [5], [11], [13]. MVE unrolls the loop body and assigns a different register for each iteration. The loop kernel then consists of copy of each version of the loop body, and iterates through the different versions. Specialized epilogues are required, one for each exit of each version. However, these techniques can dramatically increase code size and can be tricky to generate.

A *rotating register file* [14] can also be used to support versioning. This features provides a mechanism such that uses of a particular architectural register in successive iterations actually utilize different physical registers. The physical register number is the sum of a *rotating register base* with specified architecture register number, modulo the size of the register file. At the end of each II, the rotating register base is decremented such that the new iteration utilizes a new physical register. Note that the register mappings in the second stage of the first iteration have also been affected by the update to the rotating register base. Therefore, register names in subsequent loop stages must be compensated to counteract this effect. This technique is proposed to handle renaming with the Modulo Schedule Buffers.

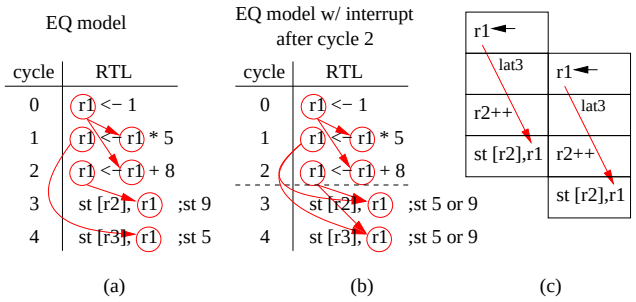



**Figure 11. Data delivery graph for code utilizing *multiple assignment* on the result of a 3-cycle multiply instruction.**

## 2.6. Interrupt Handling

Two primary scheduling models exist for *non-unit assumed latency* (NUAL) architectures. The TI 'C6x is an example of the *equals* model (EQ), where each operation executes for exactly its specified latency, and writes back the result exactly after that latency has expired. Consider the example in Figure 11(a). In cycle 0, register `r1` is initialized to a value of 1. In cycle 1, the multiply operation begins using the value of 1 for `r1`. Because the latency of the multiply is 3 cycles, the result of the multiply will not be written back to `r1` until the end of cycle 4. Meanwhile, in cycle 2, the value of `r1` is guaranteed to still be 1, and the add completes writing a value of 9 to `r1`. The store of `r1` in cycle 3 is also guaranteed to be unaffected by the multiply and will correctly write to memory a value of 9. As can be seen in this example, registers can effectively be reused during long latency operations, often resulting in fewer registers needed for a computation. TI calls this particular type of reuse *multiple assignment*. Similarly in Figure 11(c), a single register can be used to support versioning if the result is used by the consumer shortly it is available. Because the latency of `r1` is three cycles, the `r1` for the second iteration will not prematurely overwrite the value for the first iteration. Use of multiple assignment can often reduce the reliance on explicit versioning, but rarely eliminates the need for it.

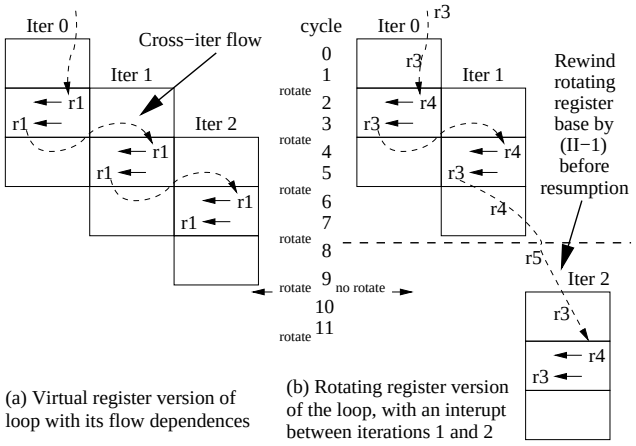
The other model is called the *less than or equal* model (LE). Under this model, the result latency is specified as the maximum time that a particular operation may take to complete and write back its result. In other words, the result may be written back at any time up until and including the cycle at which its latency expires. Coding for this model disallows the register reuse allowed in the EQ model. TI calls this type of register allocation *single assignment*.

Clearly, when scheduling and register allocating for an LE machine, a single assignment methodology must be used. If an instruction finishes early and overwrites a register, another usage of that register might read the new value rather than the old value. However, for an EQ machine, either single assignment or multiple assignment scheduling and allocation may be used. If the instruction is guaranteed to take a certain number of cycles, assuming that it can finished early is a safe, conservative assumption.

Though registers cannot be reused during long latency operations, interrupt handling in code scheduled for the LE model is much simpler. Precise interrupts are maintained by completing execution of code already in the functional units and squashing code in the fetch or decode units. After processing the interrupt, execution can continue from a single program counter value, the instruction immediately following the last one executed. Likewise, correctly handling interrupts in the EQ model under single assignment is simple, as all instructions prior to the interrupt can be allowed to finish. Since the schedule assumed that they may finish early, the computation will be correct. However, interrupt handling in the EQ model under multiple assignment is more difficult. Consider the situation when an interrupt is taken immediately after instruction 2 in the example Figure 11(b). A precise interrupt cannot be taken because there is no single program counter value where all instructions prior to the PC have been executed and all after have not. The multiply was issued prior to the interrupt and has not yet completed. Furthermore, if that instruction is allowed to complete before the interrupt is actually taken, then the value of r1 would be prematurely overwritten with the result of the multiply. Hardware techniques, such as *snapshot buffers* [15] and *replay buffers* [16], have been proposed to save the result and its relative write-back time upon a context switch. These features are often costly to implement and are not present in the TI architecture. Therefore, in the TI processors, interrupts must be postponed during any portion of the code that uses multiple assignment.

However, by using the Modulo Schedule Buffers, interrupts will only have to wait until the end of the current loop iteration before they can be processed. This is a benefit of having the control logic actually issuing iterations of the loop. When an interrupt occurs, new iterations will stop being issued from the MSBs, like when the last iteration of the loop has been issued. After the epilogues of previous iterations have completed, the interrupt can be taken. Note that the Iteration Counter register contains the number of iterations that remain in the loop and will need to be saved across context switches. Similarly, the instruction stream address of the beginning of the initialization code also needs to be saved in case the buffers need to be reloaded upon returning.

At the end of each iteration of the loop body, a set of registers and memory locations exist that communicate state to the next iteration. Figure 12(a) depicts an example loop body with such a cross-iteration register lifetime `r1`. This lifetime is live into the loop, but is not live out along the fall-through path of the loop back branch. Since the example lifetime is not live out, it may be allocated into two rotating registers `r3` and `r4`. However, by delaying iteration 2 by two stages due to the interrupt (to allow iterations 0 and 1 to complete), two additional rotations have been performed as shown in Figure 12(b). This value now appears live out the




(a) Virtual register version of loop with its flow dependences

(b) Rotating register version of the loop, with an interupt between iterations 1 and 2

**Figure 12. Register allocation for interrupt tolerance of a sample loop body.**

bottom of the loop in iteration 1 as `r5` and live into the top of iteration 2 as `r3`. In order to maintain correctness, cross iteration lifetimes must be allocated as if they were live in and live out of the loop, and upon restarting the loop, the rotating register base must be rewound by (II-1) stages to move the live out values to their live in locations.

### 2.7 Design Complexity

The proposed Modulo Schedule Buffer mechanism has been designed as a set of components in order to manage complexity. Each component consists primarily of a single counter or register (or table of registers) with some attached control logic. While the interconnect may seem complex, there is clear flow of control information that originates at the Current Cycle Counter and flows to the Buffer Access Logic, with minimal feedback. Because the Current Cycle Counter simply counts from 0 to II-1, its behavior, and therefore the behavior of its dependent components, is highly cyclic. This feature would allow the control logic to be run slightly ahead of the buffers themselves if cycle time became a limiting factor. The only significant feedback loop in the system concerns the detection of exit conditions and thus application of the Exit Active Stages Mask. By the time an exit condition is detected by a functional unit at the end of the execute hardware pipeline stage, the MSB system has already prepared the next set of instructions for execution. These instructions were issued based on what is effectively a *speculative* Active Stages Bit Vector, since the feedback from the previously issued instructions has not yet been applied. It may be possible to nullify select instructions based on the updated bit vector, but more likely, the execute packet must be flushed and reissued.

In terms of cycle time, minimal logic has been added to the dispatch pipeline stage in order to write to or fetch from the MSBs when appropriate. A multiplexor has been added to the stage to select and instruction from either the MSBs from the fetch unit. The selector logic for this multiplexor must check for an incoming NOP from the fetch stage, and must check for a valid first iteration instruction (which can run ahead, as previously mentioned). Essentially, the critical path through the dispatch stage is only increased by a few gates.

The size of the buffers is directly related to the maximum supported II, maximum number of stages, and number of functional units. For example, an aggressive implementation could support a maximum II of 32 and a maximum stage count of 16, with 8 functional units. Therefore, each entry would contain a 32-bit instruction and a 16-bit stage tag. Each buffer would contain 32 entries for a total of 192 bytes. Therefore, the total system could require 1536 bytes of storage plus a handful of registers for the control unit.

## 3. Experimental Results

In order to explore the effectiveness of our system, the IMPACT Compiler [17] was enhanced to generate code for kernel-only modulo scheduling and for the Modulo Schedule Buffers using Iterative Modulo Scheduling [6]. Twelve MediaBench [18] programs, a MP3 player, and a GSM codec [19] were emulated to verify the correctness of our system while examining loop characteristics. These benchmarks were compiled using the IMPACT compiler to generate aggressively-optimized superblock and hyperblock code for the IMPACT EPIC architecture [20]. Function inlining (to an estimated 50% static code size increase) was also employed along with use of compiler intrinsics to support typical DSP operations to broaden the scope of available loops for pipelining.

The base processor model is an 8-issue, single cluster, EPIC architecture with a functional unit mix and instruction latencies similar to that of the Texas Instruments TMS320C6700 line of processors [12]. Table 1 lists the machine specifications. Large register files were chosen to emulate the effects of multiple assignment and to provide ample resource for aggressive modulo scheduling. These experiments are designed to provide insight into the opportunity available for utilizing modulo scheduling in DSP applications.

For these experiments, we limited the maximum II to 48 cycles. Very few modulo schedulable loops were observed that would exceed this II, even with more aggressive predication. Observations of many of these large loops also indicate minimal overlap of the iterations, mitigating the potential benefits of modulo scheduling. The costs of increasing the size of the Modulo Schedule Buffers to include these larger loops also often out-weighs the potential benefits.

Table 2 shows the number of issued cycles in each of the applications studied (first data column), along with the percentage of those cycles spent in modulo scheduled loops (second column), and the percent spent executing solely from the buffers (third column). The applications average 72% of their execution in modulo scheduled loops, indicating that significant benefits can be achieved by maintaining

| Benchmarks | Description: Input | Number Cycles | % Modulo Sched Cycles | % MSB-only Issued Cycles | Kernel-Only Static Ops (ind. nops) | MSB Static Ops (ind. nops) | Kernel-Only Static Ops (Multi-nop) | MSB Static Ops (Multi-nop) |
|---|---|---|---|---|---|---|---|---|
| adpcmdec | Adaptive diff. pulse code mod. [18]: | 1.33M | 99.9 | 99.6 | 456 | 464 | 432 | 432 |
| adpcmenc | clinton.pcm | 2.96M | 99.9 | 99.7 | 480 | 504 | 464 | 464 |
| epicdec | Experimental image codec [18]: | 3.83M | 64.4 | 51.7 | 7768 | 7880 | 7080 | 7080 |
| epicenc | test_image | 32.37M | 40.6 | 29.3 | 10096 | 10272 | 9392 | 9400 |
| g721dec | CCITT G.721 voice codec [18]: | 86.19M | 37.8 | 26.7 | 2872 | 2896 | 2776 | 2768 |
| g721enc | clinton.pcm | 98.87M | 35.4 | 24.2 | 3008 | 3032 | 2936 | 2928 |
| jpegdec | IJPEG Group image codec [18]: | 1.23M | 84.0 | 71.4 | 49368 | 49408 | 45488 | 45464 |
| jpegenc | testimg.jpg | 5.08M | 46.1 | 34.9 | 51952 | 52056 | 48208 | 48192 |
| mpeg2dec | MPEG2 video codec [18]: | 62.65M | 84.9 | 65.5 | 17048 | 17328 | 16144 | 16176 |
| mpeg2enc | mei16v2.m2v | 390.35M | 90.2 | 68.8 | 29256 | 29512 | 27592 | 27560 |
| pgpdec | Pretty Good Privacy codec [18]: | 22.14M | 82.1 | 72.5 | 99968 | 100192 | 96736 | 96720 |
| pgpenc | pgptest.plain | 28.42M | 80.3 | 69.4 | 105296 | 105552 | 101920 | 101848 |
| g724dec | ETSI GSM 06.60 speech transcoding [19]: | 16.85M | 82.7 | 65.4 | 6440 | 6640 | 6192 | 6184 |
| g724enc | 363 frames of speech and noise | 107.57M | 85.8 | 71.3 | 11600 | 12080 | 11320 | 11288 |
| mpg123 | MPEG-2 Layer 3 audio dec: short.mp3 | 34.72M | 73.2 | 54.8 | 29696 | 29936 | 27272 | 27288 |
| average | | N/A | 72.4% | 60.4% | N/A | (norm) +1.5% | N/A | (norm) -0.1% |

**Table 2. Benchmark results for Modulo Schedule Buffer utilization.**

| Name | Functionality | Num Units |
|---|---|---|
| L | Logic/Int ALU/Fp ALU/Fp Convert | 2 |
| S | Branch/Shifter/Logic/Int ALU/Fp ALU | 2 |
| M | Multiplier | 2 |
| D | Memory/Basic Int ALU | 2 |

| Inst Type | Latency | Inst Type | Latency |
|---|---|---|---|
| Int ALU | 1 | Fp ALU | 3 |
| Int Multiply | 2 | Fp Multiply | 4 |
| Int Divide | 10 | Fp Divide | 15 |
| Int Logic | 1 | Fp Reciprocal/Sqrt | 2 |
| Int Shift | 1 | Fp Conversion | 4 |
| Int Load | 4 | Branch | 1 |

| Reg Type | Num Available | Max Subset Rotating |
|---|---|---|
| Integer | 128 | 64 |
| Double/Float | 64 | 32 |
| Predicate | 128 | 128 |

**Table 1. Emulated machine characteristics.**

continuous overlap of loop iterations. Of greater interest is that the applications issue an average of 60% of their cycles solely out of the buffers. This percentage directly correlates to a reduction in the number of execute packets that must be served by the instruction cache and fetch unit. By issuing from the buffers instead, these portions of the processor can be disabled through clock gating or other means, thus saving energy. Furthermore, by reusing a loop when it is already stored in the MSBs, the percentage of MSB cycles will likely grow toward the percentage of modulo scheduled cycles (its maximum).

Columns four and five of Table 2 show the effects of the sequential representation of the modulo scheduled loops on code size. The MSB scheduled code sizes are compared to ideal kernel-only code without prologues or epilogues. To take the MSB code size measurement, we applied a greedy-style bundler to the entire program, packing execute packets into the 8-slot fetch packets. The results indicate an average code size percentage increase of 1.5% due to the extra `nops` for cycle stop bits that must be inserted to represent empty cycles in the sequential representation of the loops. Because the loop body instructions are spread over the entire height of an iteration, many more nop cycles are present than when spread over II cycles in a kernel-only representation. The problem was most noticeable in floating-point code due to the long latency instructions. However, by using a special `multi nop` instruction capable of representing *several* empty cycles with one instruction, the code size increase becomes negligible (columns six and seven). In fact, for *pgpenc* (and several other benchmarks) the sequential representation was actually smaller. By representing the code sequentially, the loop body consists of *more* but *smaller* execute packets, thus making them easier to pack into 8-slot fetch packets with less waste.

In order to examine the desired number of entries for the MSBs, the fraction of each benchmark executed in modulo scheduled loops of each II was calculated. Figure 13 depicts the fraction of issued modulo schedule cycles for a given II for several representative benchmarks. The normalized average of all the benchmarks reveals that an implementation that supported an II of 16 cycles could achieve about 70% of the potentially modulo schedulable cycles. Clearly, as the II is increased beyond 32, the benefits are less likely to outweigh the extra costs of larger Modulo Schedule Buffers.

## 4. Conclusion and Future Work

The mechanism presented in this paper has been designed to provide an elegant interface for executing modulo scheduled code. While it achieves the same performance as fully-specified and kernel-only methods, it has a number of advantages. The mechanism provides the ability to limit interrupt latency to a single loop iteration rather than the remainder of the loop, and moves the burden of loop control to the dedicated hardware, thus allowing for a less costly predication schemes. It handles all possible possible loop trip counts cleanly, unlike fully-specified methods which require a handful of different epilogues, and most im-

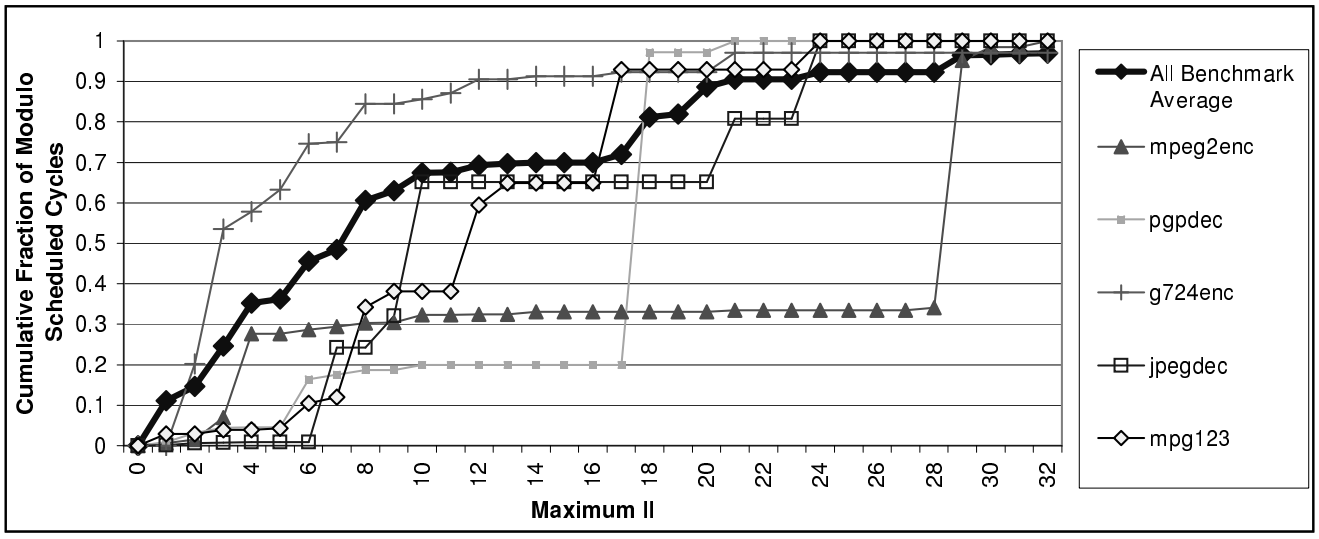



**Figure 13. Cumulative percentage of the modulo scheduled cycles for a given II.**

portantly, handles all forms of *while* loops without any prologue or epilogue. Future work includes development of an integrated register versioning mechanism that does not require the explicit use of rotating registers.

## 5. Acknowledgments

Special thanks to Chris Shannon, John Sias, Hillery Hunter, and the rest of the IMPACT Research Group for their comments and assistance. This research was pursued while the authors were with IMPACT Technologies, Incorporated.

## References

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

[2] K. Ebcioglu and T. Nakatani, "A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture," in *Proc. of the Second Workshop on Languages and Compilers for Parallel Computing*, pp. 213–229, 1989.

[3] A. Aiken and A. Nicolau, "A realistic resource-constrained software pipelining algorithm," in *Advances in Languages and Compilers for Parallel Processing* (A. Nicolau, D. Galernter, T. Gross, and D. Padua, eds.), pp. 274–290, London: Pitman/The MIT Press, 1991.

[4] M. Rajagopalan and V. H. Allan, "Efficient scheduling of fine grain parallelism in loops," in *Proc. of the 26th International Symposium on Microarchitecture*, pp. 2–11, December 1993.

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

[6] B. R. Rau, "Iterative modulo scheduling," *International Journal of Parallel Processing*, vol. 24, pp. 3–64, February 1996.

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

[8] Intel Corporation, *Intel IA-64 Architecture Software Developer's Manual Volume 1: Application Architecture*, Jan 2000.

[9] G.-R. Uh, Y. Wang, D. Whalley, S. Jinturkar, C. Burns, and V. Cao, "Effective exploitation of a zero overhead loop buffer," in *Proc. of the ACM SIGPLAN Workshop on Language, Compilers, and Tools for Embedded Systems*, May 1999.

[10] W. W. Hwu and M. C. Merten, *Method and Apparatus for Modulo Scheduled Loop Execution in a Processor Architecture*. United States Patent Application, IMPACT Technologies, Inc., December 1999.

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

[12] Texas Instruments, "TMS320C6000 CPU and instruction set reference guide," Tech. Rep. SPRU169D, Texas, March 1999.

[13] D. M. Lavery, *Modulo Scheduling for Control-Intensive General-Purpose Programs*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1997.

[14] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, vol. 22, pp. 12–35, January 1989.

[15] G. R. Beck, D. W. Yen, and T. L. Anderson, "The Cydra 5 minisupercomputer: Architecture and implementation," *The Journal of Supercomputing*, vol. 7, pp. 143–180, January 1993.

[16] K. W. Rudd, "Efficient exception handling techniques for high-performance processor architectures," Tech. Rep. CSL-TR-97-732, Coordinated Science Lab, Stanford University, October 1997.

[17] W. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, and D. I. August, "Compiler technology for future microprocessors," *Proc. of the IEEE*, vol. 83, pp. 1625–1995, December 1995.

[18] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. of the 30th Annual International Symposium on Microarchitecture*, pp. 330–335, December 1997.

[19] ETSI TC-SMG, "Digital cellular communications system; enhanced full rate (EFR) speech transcoding (GSM 06.60)," Tech. Rep. ETS 300 726, European Telecomm. Standards Institute, Mar. 1997.

[20] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu, "Integrated predicated and speculative execution in the IMPACT EPIC architecture," in *Proc. of the 25th International Symposium on Computer Architecture*, pp. 227–237, June 1998.