Evaluating the Use Of Register Queues in Software Pipelined Loops

Gary S. Tyson – IEEE member
Contact Author
Advanced Computer Architecture Lab
The University of Michigan
EECS Building, 1301 Beal Ave.
Ann Arbor, MI 48109-2122
Phone: (734) 647 - 7999
Fax: (734) 763 - 4617
email: tyson@eecs.umich.edu

Mikhail Smelyanskiy
Advanced Computer Architecture Lab
The University of Michigan
EECS Building, 1301 Beal Ave.
Ann Arbor, MI 48109-2122
Phone: (734) 926 - 2917
Fax: (734) 763 - 4617
email: msmelyan@eecs.umich.edu

Edward S. Davidson – IEEE Fellow
Advanced Computer Architecture Lab
The University of Michigan
EECS Building, 1301 Beal Ave.
Ann Arbor, MI 48109-2122
Phone: (734) 647 - 1777
Fax: (734) 763 - 4617
email: davidson@eecs.umich.edu

June 11, 2001

Abstract: *In this paper we examine the effectiveness of a new hardware mechanism, called Register Queues (RQs), which effectively decouples the architected register space from the physical registers. Using RQs, the compiler can allocate physical registers to store live values in the software pipelined loop while minimizing the pressure placed on architected registers. We show that decoupling the architected register space from the physical register space can greatly increase the applicability of software pipelining, even as memory latencies increase. RQs combine the major aspects of existing rotating register file and register connection techniques to generate efficient software pipeline schedules. Through the use of RQs, we can minimize the register pressure and code expansion caused by software pipelining. We demonstrate the effect of incorporating register queues and software pipelining with 983 loops taken from the Perfect Club, the SPEC suites, and the Livermore Kernels.*

Index Terms: software pipelining, modulo variable expansion, rotating register file, register queues, VLIW, register connection.

1 Introduction

Many code transformations performed in optimizing compilers trade off an increase in register pressure for some desirable effect (lower instruction count, larger basic block size, etc.). Perhaps this is most clearly shown in the **software pipelining** [3][20][19][4][10][15] of a loop, which interleaves instructions from multiple iterations of the original loop into a restructured loop kernel. This restructuring improves pipeline throughput by enabling more instructions to be scheduled between a value being defined by a high latency operation (e.g., multiplication, memory load) and its subsequent use. Software pipelining thus decreases the time between successive loop iterations by spreading the *def-use* chains in time. This rescheduling increases the number of simultaneously live instances of loop variables from different iterations of the original loop body. To accommodate these variables, each of the simultaneously live instances needs its own register. Furthermore, each instance must be uniquely identified to permit matching a use of a variable to the correct definition; there must be some mechanism to differentiate among live instances of a variable defined in previous iterations and the definition in the current iteration.

Two common schemes that support this form of register naming are **modulo variable expansion** (MVE) [15], and the *rotating register* file (RR) [21][22]. MVE is a software-only approach which gives each simultaneously live variable instance its own name, unrolling the loop body as necessary to insure that any later uses can directly specify the correct instance (more on this later). MVE both increases the architected register requirements and expands the loop body to accommodate the register naming constraints of the software pipelined loops. In contrast, RR is a hardware-managed register renaming scheme that eliminates the code expansion problem by dynamically renaming the register specifier for each instance of a loop variable. This renaming is achieved by adding an additional level of indirection to the register specification to incorporate the loop iteration count; this makes it possible to explicitly access a variable instance that was defined n iterations ago. However, since the rotating register file contains architected registers, RR still requires a large number of architected registers to permit generating efficient schedules.

Each of these techniques satisfy the register requirements for a variable by assigning the instances defined in successive loop iterations to distinct architected registers in some round-robin fashion. The number of architected registers required for a software pipelined (SP) loop therefore grows linearly with increased functional unit latencies [17], i.e., a longer latency operation within the loop will lead to a greater number of interleaved instances of the original loop in the SP loop kernel, and therefore more live instances of the loop variables. Therefore, a shortage of architected registers either limits the number of interleaved loop iterations or introduces spill code, each of which degrades performance. Thus, efficient software pipeline schedules that account for realistic memory latencies are difficult, and often impossible, to achieve with MVE or RR for architectures with small or moderate sized register files. One solution is to dramatically increase the number of architected registers available. This may be achieved when a new instruction set architecture is proposed (e.g., the IA-64 or EPIC instruction set [1] which supports 128 integer and 128 floating point registers). In this paper, we propose an alternative register addressing mechanism which can be integrated into existing instruction set architectures with minimal modification while alleviating the register pressure and register naming issues that are inherent in SP.

In this work we demonstrate that by introducing **Register Queues** (RQs) and the *rq-connect* instructions the architected register space is no longer a limiting factor in achieving efficient software pipelined loop schedules. The design of these register queues is derived from the interprocessor queues that support asynchronous communication in decoupled architectures[23][25]. Software pipelining using queues has also been studied in VLIW architectures [16][7][8] and decoupled processors[24], but not in general purpose superscalar designs. In particular, [8] proposes the use of a queue register file

(QRF) to support the execution of software pipelined loops in VLIW machines. This extends prior work on VLIW processors [12] by making the queues architecturally visible; earlier work scheduled values in pipeline registers, also organized as queues, for a specific VLIW implementation. By making the queues architecturally visible, portability between VLIW implementations is provided. Our work proposes the register queue mechanism for conventional superscalar processors, as well as software/hardware techniques to ease the integration of RQs into existing instruction set architectures and machine implementations with out-of-order pipelines.

In the context of this research, register queues can most clearly be viewed as a combination of the rotating register file ([2], [22]) and register connection [14] concepts. This combination enables a decoupling of the total register space for SP into a small set of architected registers and a large set of physical registers that are organized as circular buffers and accessed indirectly. By using register queues, the architected register requirements of a software pipelined loop are independent of the latencies of the scheduled instructions. Integrating RQs into an existing architecture is also straightforward. We will show that the inclusion of a single new instruction, *rq-connect*, is all that is necessary to add RQs to any instruction set architecture while maintaining full backward compatibility. Experimental results show that the RQ method significantly reduces both the architected register and the code size requirements of software pipelined loops.

The remainder of this paper is organized as follows: Section 2 provides a brief introduction to software pipelining and describes previous work in both software pipelining and register file organization. Section 3 describes the concept of register queues and the architectural modifications required to support this approach. Section 4 presents our experimental evidence of the performance advantage of register queues over existing schemes. We offer conclusions in Section 5.

2 Prior Work

As a simple example of SP, consider Figure 1 which shows the intermediate level code of one iteration of a loop that accumulates the elements of a floating point array into a scalar (loop control instructions have been eliminated for clarity). For this example, we assume a two-wide issue machine with a latency of 3 for the load operation, 2 for floating-point addition, and 1 for integer addition. The scheduling process is governed by two constraints: **resource constraints** determined by the resource usage requirements of the computation, and **precedence constraints** derived from the latency calculations around elementary circuits when they exist in the dependence graph for the loop body due to a loop-carried dependence. With an issue width of 2 and a loop body consisting of 3 instructions, we do not have the resources (issue width in this case) to start a new loop iteration more often than once every 2 cycles. The interval between starting new instances of a loop is termed the initiation interval (or *II*) of the loop (in this case we must make $II \geq 2$). This loop also contains a loop-carried dependence between instances of the floating-point add with a latency of 2, (again, we must make $II \geq 2$).

Figure 1 shows a software pipelined code sequence, for $II = 2$. Instructions at time steps 1-4 form the prologue of the software pipelined loop, time steps 5-6 are the steady-state segment (or kernel of the loop), and 7-10 form the loop epilogue. The prologue and epilogue are executed once and the steady-state kernel is executed repeatedly (n-2 times to execute n iterations of the original loop).

The example in Figure 1 demonstrates a problem with register names in software pipelined schedules. The *fload* instruction from iteration $i + 1$ starts executing before the *fadd* instruction from iteration i uses the value created by the *fload* of iteration i. This creates two simultaneously live instances of the register *f2*. One way to overcome the register overwrite problem (WAR hazard) is to increase the initiation interval to 4 to allow the *fadd* operation from the ith iteration to complete

	time	iteration 1	iteration 2	iteration 3
Prologue	1	iadd r1, r1 #4		
	2	fload f2, 0(r1)		
	3		iadd r1, r1 #4	
	4		fload f2, 0(r1)	
Kernel	5	fadd f6, f6, f2		iadd r1, r1 #4
	6			fload f2, 0(r1)
Epilogue	7		fadd f6, f6, f2	
	8			
	9			fadd f6, f6, f2
	10			

Figure 1: Software pipeline example. This sample program adds elements of a floating-point array and stores the sum in a scalar. Shown are multiple iterations of the loop with an initiation interval of 2 cycles ($II = 2$).

before the *fload* of iteration $i+1$ is issued. However, this would halve the loop throughput to one iteration every four cycles. We now describe several alternative solutions that have been proposed to address this register naming problem.

Modulo variable expansion (MVE) [15] is a compiler transformation (requiring no hardware support) which schedules a software pipelined loop. The purpose of MVE is to manage the naming problem by making sure that instances of a variable whose lifetimes overlap are allocated to distinct architected registers. So, if the lifetime of a value spans three iterations of the pipelined loop and its lifetime overlaps the instances of that variable in the next two iterations, three registers will be allocated in the loop kernel for that variable. In general, at least $\lceil \frac{l}{II} \rceil$ registers are required for each variable in the loop, where l is the variable's lifetime in cycles. Since successive definitions of a variable must be assigned to different registers (since they are simultaneously live), the kernel has to be unrolled, thus lengthening the steady state loop body. The kernel of the loop must therefore be expanded by a factor of at least $\lceil \frac{l}{II} \rceil$ to account for the different register specifiers required for successive definitions of the variable. The actual degree of unrolling is, however, determined by the requirements for all the variables, given the minimum number of registers required for each variable.

When expanding the loop kernel, two techniques are examined. One technique (which we will call MVE1) minimizes register pressure at the expense of increasing the degree of loop unrolling that is necessary. Each variable v_i is allocated its minimum number of registers, q_i, and the degree of unrolling, u_{mve1}, is given by the **lowest common multiple** (lcm) of the q_i. The other schedule (which we will call MVE2) favors minimizing the number of times that the loop is unrolled, at the expense of more register pressure. This minimum degree of unrolling, u_{mve2}, is the max q_i, which, of course, is never more than $\mathrm{lcm}(q_i)$ required by MVE1. However, rather than requiring exactly q_i registers for each variable as in MVE1, MVE2 requires q_i for a variable if and only if u_{mve2} mod $q_i = 0$, but otherwise requires that the number of registers allocated to store the instances of a variable increase from q_i to the smallest divisor of u_{mve2} that is greater than q_i.

Several additional techniques have been proposed to minimize register requirements in SP loops. In [11], Huff proposes a heuristic based on a bidirectional slack-scheduling method that schedules operations early or late depending on their number of stretchable input and output flow dependences. Integer programming has been used in [5][9] to lower register requirements by optimizing according to several potentially conflicting constraints and objectives, such as resource constraints, scheduling operations along critical dependence cycles, maximizing the throughput, and minimizing the schedule length of the critical path. Stage scheduling [6] breaks the schedule into two steps. In the first step, a modulo scheduler generates a schedule with high throughput and a short schedule length. In the second step, a stage scheduler reduces the register requirements of a modulo schedule by reassigning some operations to different stages. All of these schemes aim at reducing the number of architected registers in the software pipelined loops. The best of these schemes can reduce register pressure by as much as 25% in the configurations studied. However, since all live values must be allocated to architected registers, they are unable to decouple the architected register requirements from the physical requirements. In this paper, we concentrate on modulo scheduling, while recognizing that our results can be applied to other scheduling algorithms as well.

Rau [22] addressed the naming problem in software pipelined loops by employing a new method of addressing a processor register file in the Cydra-5 minisupercomputer [2]. The Rotating Register File (RR) is a register file that supports compiler managed hardware renaming by adding the register address (specified in the instruction) to the contents of an Iteration Control Pointer (ICP) (modulo the number of registers in the RR). This register specifier is then used to index into the architected register space. A special loop control operation decrements the ICP each time a new iteration starts, giving each loop iteration a distinct set of physical registers from those used by the previous iteration (thus a value referenced as r5 in iteration i will be addressed as r6 in i+1). Since register access includes an additional indirection (i.e. adding the ICP to the specifier), unrolling is unnecessary and the loop kernel is not expanded from its original form. RR can therefore eliminate the code expansion problem from SP, but it still requires a large number of architected registers because all of the physically addressable registers are part of the architected rotating register file [21].

The problem of increasing a limited architected register space without dramatically changing an existing instruction set has also been explored. The Register Connection (RC) [14] method tolerates high demand for the architected registers by adding a set of extended registers to the core register set, and incorporating a set of instructions to remap architected register specifiers into the extended set of physical registers. RC architectures use these instructions to dynamically connect architected registers to extended registers. Accesses to an architected register are automatically directed to its most recently connected physical register of the extended register file. A register mapping table with one entry per architected register is used to map each architected register to its own core physical register (by default) or to any register in the extended register file (as setup by a connect instruction). The indirection of RC is similar to that found in register renaming tables [13] used in many superscalar architectures, except that the mapping is performed under compiler control which enables more live values to reside in the extended register file than can be addressed at any point in time by the operand specifiers of the current instruction. This RC work did not target software pipelined loops; however, we show that by decoupling the architected register set from a much larger physical register file, the RC method can greatly reduce the architected register requirements of these loops. Although using RC to perform SP in the context of modulo variable expansion significantly reduces architected register requirements, RC (like MVE) still requires loop unrolling to solve the register naming problem. Furthermore, RC adds some connect instructions to the loop kernel, prologue and epilogue.

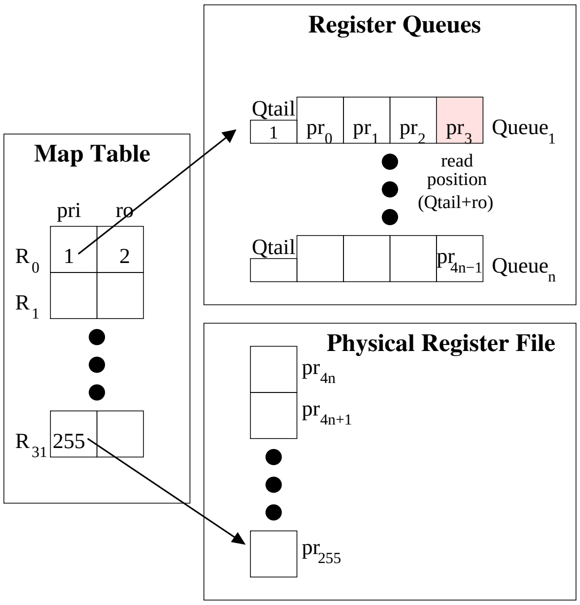


Figure 2: Microarchitectural extensions to support RQs for a machine with 32 architected registers, n queues of length 4, and 256 physical registers.

3 Register Queues

We now propose an alternative scheme called Register Queues (RQs). RQs incorporate both a hardware-managed register renaming feature similar to RR and the register decoupling of RC to ameliorate both the code size and the architected register problems from SP. When scheduling for an SP loop, variables with multiple live instances will be placed in a queue; all other variables in the loop will be assigned to conventional registers. The register file in an RQ design consists of three parts as shown in Figure 2:

- **a set of register queues**: Each queue has a *Qtail* pointer, analogous to the ICP in the RR, and a set of contiguous registers which share a common name-space with the physical register file, but are logically (and probably physically) separate. In Figure 2 the registers that constitute register queue 1 are physical registers pr0 through pr3; physical registers pr_4 through pr_7 make up register queue 2, etc. These registers are analogous to the registers in the RR, and use the same modulo arithmetic to index into the queue. They differ from RR registers in that registers in the queue must be explicitly mapped to an architected register before being accessed. Like the RR registers, the registers in the queues become part of the state of the processor and must be saved during context switch.
- **a physical register file**: The physical register file contains the remaining set of physical registers not allocated to a register queue. This set of registers is equivalent to the physical register file found on most superscalar processors. In Figure 2 the physical register file contains registers pr_{4n} through pr_{255}.
- **an architected register map table**: This table maps each architected register to either a physical register (using standard register renaming logic) or a register queue (using an *rq-*

connect instruction). Each entry in the map table contains a particular physical register index (pri) and a read offset (ro). The index specifies that either some free physical register or a particular register queue is to be mapped to the architected register. The read offset, used only for register queue mappings, contains an offset into the queue specifying which register in the queue is mapped to the architected register.

A single instruction is added to the ISA to manage the RQ: *rq-connect* maps, remaps or unmaps an architected register to one of the register queues. The semantics of the *rq-connect* instructions are:

- **rq-connect \$rq, \$ar, imm**: maps an architected register \$ar to register queue \$rq (\$rq = 1,2,...,n) by writing the queue number into the *pri* field of the map table. Furthermore, the read offset (ro) in the queue is specified by the immediate field *imm*. Subsequent reads of architected register \$ar will now map to the *imm*th entry from the *Qtail* of register queue \$rq. Note that the semantics for a read are different than for real queues; instead of destructively reading from the head of the queue, an architected register is mapped to some location in the queue and reads occur from that location in a nondestructive manner. This greatly increases the flexibility of using register queues (though the term queue is somewhat of a misnomer).
- **rq-connect \$0,\$ar, 0**: remaps architected register \$ar to a free register from the physical register file. By numbering the register queues from 1 to n, we leave the \$rq = 0 operand in the *rq-connect* instruction free to indicate that the architected register \$ar should be disconnected from its register queue.

A read access to an architected register that is mapped to a register queue causes the following events to take place:

1. Use the register specifier in the operand field of the machine instruction to index into the Map Table and extract the register queue identifier (the *pri* field of the register map table entry) and an offset into the queue (the *ro* field).
2. Index into the queue specified by *pri* at the specified read offset. To compute the offset, the *Qtail* is added to *ro*, modulo the number of registers in the queue. The physical register specifier is the index bits in the *pri* field with the least significant 2 bits replaced by the computed offset. Note that in this example, the circuit used to perform the mapping is a 2-bit adder – not a 7-bit adder as used in the Cydra-5 RR.
3. Read the contents of that physical register or pass that physical register identifier to later pipeline stages if the results must be forwarded from an earlier instruction that has yet to retire.

A write into the register queue involves the following sequence of steps:

1. Use the register specifier in the operand field of the machine instruction to index into the map table and extract a register queue identifier (the *pri* field). This selects the register queue; the read offset is not needed since a write value is always appended to the tail of the queue.
2. Decrement the *Qtail* pointer for the queue. This is analogous to decrementing the ICP in the RR. Note that in RQs the update of *Qtail* automatically occurs on each write to the queue, whereas in the RR the ICP is updated using a special branch instruction. Both solutions effectively manage the register naming problem.

3. Pass the physical register identifier of this new *Qtail* position in the queue (along with the instruction) to the appropriate reservation station. Note that at this point all register identifiers found in the reservation station are standard physical register specifiers, leaving the reservation station and operand forwarding logic unchanged.

It should now be apparent that the offset (ro) field of the map table entry is used only for reads from queues; it should be 0 to reference the most recently defined variable instance, 1 to reference the previous instance, etc. Furthermore, since there is only one *ro* field for each architected register it is not possible to read two different queue offsets using a single architected register except by using an intervening connect instruction, and at most one connect instruction can be issued in one cycle for the same architected register. Finally, if a read and a write are issued in the same cycle to the same architected register which is mapped to a queue, which physical register is accessed by that read is unaffected by that write. Furthermore, if the read and the write are to the same physical register, the value in that register prior to that write will be read.

3.1 SP Scheduling using register queues

Managing the dynamic mapping of variable instances as a queue enables implementing efficient software pipeline schedules with little change in code size or architected register requirements. Each register queue, like a rotating register in RR, provides a set of registers to contain instances of a variable for several successive iterations. RR uses a contiguous set of RR architected registers to enable unconstrained access to any physical register. By contrast, RQ assigns each variable that is read one or more iterations after its definition in a software pipelined loop to a distinct register queue that holds all live instances of that variable. [1] Architected registers with unique operand specifiers are then connected to particular locations in the queue which contain live instances that are read. If a value is read three iterations after its definition, i.e., after 2 other intervening writes to the same variable, its architected register is mapped to the third most recent definition by using an *rq-connect* instruction to set the offset, *ro*, for that register to 2. In general, for a particular use of a variable, *ro* is set to the number of intervening writes that occur to that variable (or register queue) between the definition of interest and the use.

The two more recent definitions are (at the time of this read) associated with positions of that queue that now have offsets of 0 and 1; no architected registers need ever be mapped to these queue locations if they will not be referenced until a later iteration. This mapping mechanism eliminates the need for unrolling the software pipelined loop kernel since architected registers are only mapped to offset positions in the queue that are actually read; writes to variables assigned to queues are always to the decremented *Qtail*. This reduces the pressure on the register operand bits of the instruction set architecture since the architected registers that must have unique operand bit patterns is determined by the number of registers that are actually connected to queues at various read offsets. Since at most one register is needed for each read offset, the total register requirements are much less than the total of all simultaneously live instances.

The functionality of RQs can be demonstrated by re-examining the loop fragment from Figure 1. Figure 3 (a) shows the code fragment after SP is applied – including the prologue, kernel and epilogue of the loop. The prologue code includes instructions 1 through 5. Instruction 1 creates a mapping between architected register *f2* and register queue *q1* at read position 1. Writes to *f2* will now decrement *q1*'s *Qtail* and overwrite the register pointed to by the new value of *Qtail*. Once the read offset is set to 1, subsequent reads of *f2* will retrieve the contents of *q1* register (*Qtail*+1)

[1]Several variants of this scheme are discussed in 3.2. In particular using two register queues to accommodate a variable when one queue cannot accommodate the number of simultaneously live instances.

Line	Assembly Code
1	rq−connect q1, f2, 1
2	iadd r1, r1, #4
3	fload f2, 0(r1)
4	iadd r1, r1, #4
5	fload f2, 0(r1)
6	**fadd f6, f6, f2**
7	**iadd r1, r1, #4**
8	**fload f2, 0(r1)**
9	fadd f6, f6, f2
10	rq−connect q1, f2, 0
11	fadd f6, f6, f2
12	rq−connect 0, f2, 0

(a) Scheduled Code

Cyc	Iteration A	Iteration B	Iteration C	Iteration D	Iteration E
1	rq−con q1,f2,1				
2	iadd r1, r1, #4				
3	fload f2, 0(r1)				
4		iadd r1, r1, #4			
5		fload f2, 0(r1)			
6	fadd f6, f6, f2		iadd r1, r1, #4		
7			fload f2, 0(r1)		
8		fadd f6, f6, f2		iadd r1, r1, #4	
9				fload f2, 0(r1)	
10			fadd f6, f6, f2		iadd r1, r1, #4
11					fload f2, 0(r1)
12				fadd f6, f6, f2	
13					rq−con q1,f2,0
14					fadd f6, f6, f2
15					rq−con 0,f2,0

(b) SP execution of 5 original loop iterations

Cyc	Instruction	queue 1 before instruction					queue 1 after instruction					Value read
3	fload f2, 0(r1)	0	–	–	–	–	3	–	–	–	$f2_1$	–
5	fload f2, 0(r1)	3	–	–	–	$f2_1$	2	–	–	$f2_2$	$f2_1$	–
6	fadd f6, f6, f2	2	–	–	$f2_2$	$f2_1$	2	–	–	$f2_2$	$f2_1$	$f2_1$
7	fload f2, 0(r1)	2	–	–	$f2_2$	$f2_1$	1	–	$f2_3$	$f2_2$	$f2_1$	–
8	fadd f6, f6, f2	1	–	$f2_3$	$f2_2$	$f2_1$	1	–	$f2_3$	$f2_2$	$f2_1$	$f2_2$
9	fload f2, 0(r1)	1	–	$f2_3$	$f2_2$	$f2_1$	0	$f2_4$	$f2_3$	$f2_2$	$f2_1$	–
10	fadd f6, f6, f2	0	$f2_4$	$f2_3$	$f2_2$	$f2_1$	0	$f2_4$	$f2_3$	$f2_2$	$f2_1$	$f2_3$
11	fload f2, 0(r1)	0	$f2_4$	$f2_3$	$f2_2$	$f2_1$	3	$f2_4$	$f2_3$	$f2_2$	$f2_5$	–
12	fadd f6, f6, f2	3	$f2_4$	$f2_3$	$f2_2$	$f2_5$	3	$f2_4$	$f2_3$	$f2_2$	$f2_5$	$f2_4$
14	fadd f6, f6, f2	3	$f2_4$	$f2_3$	$f2_2$	$f2_5$	3	$f2_4$	$f2_3$	$f2_2$	$f2_5$	$f2_5$

(c) Reads and writes of register (f2) in sample execution

Figure 3: Software pipeline schedule for the sample code (see Figure 1) using RQs

mod queue size. The remaining instructions in the prologue load the first two memory values and increment the pointer (r1) twice.

Instructions 6-8 in Figure 3(a) represent the loop kernel. The read from register *f2* in instruction 6 returns the second most recent write to *q1* (i.e. ($Qtail + 1$) mod queue size). The variable in f6 has only one live instance at any time and does not require a queue/ Instruction 7 increments the pointer (r1). Instruction 8 writes the next load value into register *f2*, decrements *Qtail* for register queue *q1*, and puts the loaded value into the register pointed to by the new *Qtail*. This loop kernel iterates until the last load operation is performed, leaving uses of the final two memory data for the epilogue.

Instructions 9-12 form the epilogue of the SP loop schedule. Instruction 9 uses the second to last memory value in the same manner as the loop kernel access. However, the last value loaded will remain in the tail of the queue since no further writes to the queue are performed. To use this value, we need to remap *f2* to reference the offset 0 position in *q1*. This is performed with another *rq-connect* instruction (instruction 10). Instruction 11 can then read from *f2* and access the final load value from the *Qtail* position. Finally, architected register *f2* is remapped to a free register in the physical register file, completing the SP schedule.

Figure 3 (b) shows how the SP schedule interleaves instructions from different loop iterations. The prologue instructions are issued on cycles (time) 1-5; these instructions include the *rq-connect* and *iadd*s and *fload*s for the first two iterations of the original (unscheduled) loop (we will refer to the original loop iterations by uppercase letters – A and B in this case). The kernel of the loop is shaded and in this example executes three times (cycles 6-7, 8-9, 10-11); the first iteration of the SP loop kernel executes the *fadd* instruction from the original loop iteration A along with the *iadd* and *fload* from iteration C. The second time through the kernel will execute instructions from original loop iteration B and D at cycles 8-9; the third time through the kernel executes instructions from iterations C and E at cycle 10-11. Finally, the epilogue of the SP schedule is executed at times 12-15 performing the remaining *fadd* instructions (for iterations D and E).

Figure 3 (c) shows how data is accessed in register queue *q1* for the sample code. Queue reads do not change the state of the register queue; *fload* instructions at times 3,5,7,9 and 11 are shown to decrement the *Qtail* and write the new data in the queue. At cycle time 6, the first *iadd* instruction reads the first value written to *q1*. It retrieves the value $f2_1$ by adding the current *Qtail* (2) to the read offset (which was set to 1 by the *rq-connect* instructions), and accesses *q1* at position 3 (the rightmost, shaded position in *q1* in Figure 3 (c)). The remaining reads are as shown in Figure 3 (c) and operate similarly.

In this example, only a single variable is allocated to a queue and it contains two live instances. In general, there may be many variables with multiple simultaneously live instances. One simple connect strategy employs a single unique register queue for each such variable to hold all live instances of that variable. This mechanism works well in reducing the architected register pressure of SP schedules, but may require a large number of register queues (one for each variable containing multiple instances). Furthermore, with fixed length queues, many of the registers in the queue may not be required (if there are fewer live instances of a variable than registers in a particular queue), whereas some variables with many live instances may not be accommodated by a single queue.

Fortunately, since the read offset values may be changed, the RQ access capabilities for a single queue are flexible enough to hold instances of more than one variable. Thus, we can assign all instances of several variables to the same queue, connecting read offsets accordingly. It is only necessary to determine the read offset for each use, given the sequence of writes for all the variable instances mapped to the queue. This is simple when writes for each variable are unconditionally performed; it is then simply a matter of counting the definitions that occur between the definition and use of a particular instance. It becomes more challenging when writes to a variable are conditionally executed (e.g., instructions in an *if-then-else* statement). In this case, we must carefully determine the possible read offsets or assign the conditionally defined variable to a queue that is not shared. Alternately, a dummy write on the alternate path can be inserted to insure that a value will be written to the queue regardless of the execution path. In the loops studied, predication was performed on all loops prior to SP, thereby eliminating this issue.

A second queue register allocation issue arises when the variables assigned to a particular queue contain more live instances than the number of registers in the queue. In this case, we can either increase the initiation interval of the SP schedule (so as to reduce the number of instances), or we can concatenate two or more physical queues into a larger logical queue by copying the head of the first queue to the tail of the second queue. Each copy costs one extra instruction in the loop body to perform the copy, and one additional architected register to read the oldest value in the first queue (offset 3) as the source field of the copy instruction (any register mapped to the following queue can be used as the destination of the copy since all writes append to the queue tail regardless of the read offset). This will be discussed further in Section 3.2.

Finally, it is possible to run out of architected registers, even using RQs. In this case, we can avoid spilling values to memory by reconnecting architected registers inside the loop body. Indeed, it is possible to use a single architected register throughout the SP schedule by reconnecting prior to each definition or use of a variable that is allocated to a register queue. This strategy would lead to a large number of connect instructions in the loop body (one for each read and write), but it would correctly implement the register requirements of a software pipelined loop. This will be discussed further in Section 3.3.

3.2 Managing queue overflow

If a variable assigned to a particular queue has more live instances than the number of registers in the queues we can concatenate two or more queues by copying the head of the first queue to the tail of the second queue before it is overwritten by a new data instance. Suppose, for example, that the load latency of the machine executing the loop fragment in Figure 1 is increased to 11 cycles. To maintain an initiation interval of 2, the time between definition (*fload*) and use (*fadd*) would span six iterations of the SP kernel, resulting in 6 (rather than just 2) simultaneously live instances of the variable, which exceeds the queue size (4 elements).

In scheduling this loop, the prologue code expands to 14 instructions (1-14) spanning 6 iterations of the original loop (A-F), as shown in Figure 4 (a). The first connect instruction (at position [A,1] in the figure) creates a mapping between architected register *f2* and register queue *q1* with a read offset of 3. Writes to *f2* enqueue data at the tail of the queue, while reads from *f2* access the oldest element in the queue. Since the queue size is insufficient to store six items, a second queue must be allocated to live instances of this variable; the second connect instruction (at [A,2]) maps architected register *f4* to *q2* to provide the remaining queue storage for this variable. The *fmove* pseudo-instruction at [A,11] and [B,13] copy the oldest data from *q0* to the tail of *q1*. Once the oldest element in q0 is copied to *q1*, it is safe to overwrite it by executing another *fload* instruction. The queue management performed in the prologue of this SP scheduled loop is shown in Figure 4 (b). The first four elements are written into *q1* by the *fload* instructions (at [A,4], [B,6], [C,8] and [D,10]). The first element is then copied to *q2* (at [A,11]) freeing that storage for the next write to *q1* (at [E,12]). This process is repeated to move the second element written to *q1* (at [B,13]) and allow the final write to *q1* in the prologue (at [F,14]).

A fourth instruction (at [C,15]) is added to the loop kernel to continue moving head elements from *q1* to the tail of *q2*. The kernel is otherwise unchanged from the schedule in Figure 3 except that the *fadd* instruction now employs architected register *f4* and reads from *q2*. Assuming that this

Cyc	Iteration A	Iteration B	Iteration C	Iteration D	Iteration E	Iteration F	Iteration G
1	rq−con q1,f2,1						
2	rq−con q2,f4,1						
3	iadd r1, r1, #4						
4	fload f2, 0(r1)						
5		iadd r1, r1, #4					
6		fload f2, 0(r1)					
7			iadd r1, r1, #4				
8			fload f2, 0(r1)				
9				iadd r1, r1, #4			
10				fload f2, 0(r1)			
11	fmove f2, f4				iadd r1, r1, #4		
12					fload f2, 0(r1)		
13		fmove f2, f4				iadd r1, r1, #4	
14						fload f2, 0(r1)	
15	fadd f6, f6, f4		fmove f2, f4				iadd r1, r1, #4
16							fload f2, 0(r1)
17		fadd f6, f6, f4					
18			rq−con q2,f4,0				
19			fadd f6, f6, f4				
20							
21				fadd f6, f6, f2			
22					rq−con q2,f2,2		
23					fadd f6, f6, f2		
24						rq−con q1,f2,1	
25						fadd f6, f6, f2	
26							rq−con q1,f2,0
27							fadd f6, f6, f2

(a) SP schedule

Cyc	Instruction	queue 1 after instruction					queue 2 after instruction				
4	fload f2, 0(r1)	3	–	–	–	$f2_1$	0	–	–	–	–
6	fload f2, 0(r1)	2	–	–	$f2_2$	$f2_1$	0	–	–	–	–
8	fload f2, 0(r1)	1	–	$f2_3$	$f2_2$	$f2_1$	0	–	–	–	–
10	fload f2, 0(r1)	0	$f2_4$	$f2_3$	$f2_2$	$f2_1$	0	–	–	–	–
11	fmove f2, f4	0	$f2_4$	$f2_3$	$f2_2$	$f2_1$	3	–	–	–	$f2_1$
12	fload f2, 0(r1)	3	$f2_4$	$f2_3$	$f2_2$	$f2_5$	3	–	–	–	$f2_1$
13	fmove f2, f4	3	$f2_4$	$f2_3$	$f2_2$	$f2_5$	2	–	–	$f2_2$	$f2_1$
14	fload f2, 0(r1)	2	$f2_4$	$f2_3$	$f2_6$	$f2_5$	2	–	–	$f2_2$	$f2_1$

(c) contents of queue1 and queue 2

Figure 4: Software pipeline schedule with queue overflow (*fload* latency = 11).

kernel is executed once (7 iterations of the original loop), the epilogue starts at cycle time 17. To perform more than 7 iterations of the original loop, cycles 15 and 16 are simply repeated once per additional iteration. Note that architected register *f4* and hence queue *q2* is both read and written at cycle 15. Following common design for multiported register files, we assume that the read uses the old value of *Qtail* while the write uses the decremented value as its index, but does not write into the *Qtail* register until after the read access is completed (for the opposite sequence the offset, *ro* for the read, would simply have to be set to 2 rather than 1). Thus at cycle 15, [A,15] reads *f2* from position (2+1) of *q2* while [C,15] write $f2_3$ into position (2-1) of *q2*. The *fload* at cycle 16 then simply overwrites $f2_3$ in position (2-1) of *q1* with $f2_7$.

Queue accesses in the epilogue differ from earlier references. Since the epilogue code will not enqueue new data into *q1* (i.e. there are no *fload* instructions in the epilogue) *fmove* instructions are not required; instead, *rq-connect* instructions are added to change the read offset to access the correct entry in the queues. Notice that the first two *fadd*s in the epilogue reference *q2* (through *f4*) and the final four references access *q1* (through *f2*). Two final *rq-connect* instructions (not shown in Figure 4(a) could be added if necessary to reconnect *f2* and *f4* to free registers (as done by instruction 12 of Figure 3 (a)).

3.3 Reducing architected register pressure

In the event that too few architected registers are available to support the SP schedule, it is possible to reconnect architected registers inside the loop body. Figure 5 illustrates this approach by re-examining a modified version of the loop fragment from Figure 1. For demonstration purposes we added a new instruction (*fadd f8, f6, f2*) requiring an additional reads of the queue mapped to *f2* with a different offset. [2] Normally, each different read location in the queue would be mapped to a different architected register so as to eliminate reconnecting; however, in this example, we assume that only a single architected register is free for use by this variable. The modified source code is shown in Figure 5 (a) and the resulting SP scheduled loop is shown in Figure 5 (b).

In the modified loop body there are two reads from register *f2* at different read offsets. If we had extra architected register, it would be connected to the second read position in the queue. Instead, we reconnect register *f2* inside the loop body to the corresponding read positions. Prologue instruction [A,1] (Figure 5 (b)) maps architected register *f2* to register queue *q1*. *fload* instructions enqueue data in *q1*. The first *fadd* instruction (at [A,6]) accesses *q1* with read offset1 (to access the value loaded at [A,3] despite the intervening *fload* at [B,5]). The second *fadd* instruction (at [A,8]) accesses *q1* with read offset 2 (since there has been another intervening *fload* at [C,7]). Further iterations connect similarly.

The loop kernel requires six instructions: the original four instructions and two additional *rq-connect* instructions. In the first cycle of the loop kernel (cycle 8), *f2* is mapped to *q1* with a read offset of 2 to enable access to the value required for the *fadd f8, f6, f2* instruction in [A,8]. The second cycle of the loop kernel (cycle 9) remaps *f2* to access read offset 1 of *q1* to access the value required for the *fadd f6, f6, f2* instruction in [B,9]. The *fload* instruction writes a new element, as before, into the queue and updates the *Qtail*.

Keeping the initiation interval at 2 cycles requires careful management of the queue resources; for example, in cycle 9 we are changing the mapping of register *f2*, reading the queue specified by *f2*, and writing a new element into the queue specified by *f2*. The ordering of these operations is as follows:

[2] We also change the latency for *fadd* to 1 for this example in order to simplify the discussion of the resulting schedule.

Line	Assembly Code

```
iadd   r1, r1, #4
fload  f2, 0(r1)
fadd   f6, f6, f2
fadd   f8, f6, f2
```

Original loop body before SP scheduling

(a)

Cyc	Iteration A	Iteration B	Iteration C	Iteration D
1	rq−con q1,f2,1			
2	iadd r1, r1, #4			
3	fload f2, 0(r1)			
4		iadd r1, r1, #4		
5		fload f2, 0(r1)		
6	fadd f6, f6, f2		iadd r1, r1, #4	
7			fload f2, 0(r1)	
8	rq−con q1,f2,2 fadd f8, f6, f2			iadd r1, r1, #4
9		rq−con q1,f2,1 fadd f6, f6, f2		fload f2, 0(r1)
10		rq−con q1,f2,2 fadd f8, f6, f2		
11			rq−con q1,f2,1 fadd f6, f6, f2	
12			fadd f8, f6, f2	
13				rq−con q1,f2,0 fadd f6, f6, f2
14				fadd f8, f6, f2

(b)

Cyc	Instruction	queue 1 before instruction					queue 1 after instruction					Value read
3	fload f2, 0(r1)	0	–	–	–	–	3	–	–	–	$f2_1$	–
5	fload f2, 0(r1)	3	–	–	–	$f2_1$	2	–	–	$f2_2$	$f2_1$	–
6	fadd f6, f6, f2	2	–	–	$f2_2$	$f2_1$	2	–	–	$f2_2$	$f2_1$	$f2_1$
7	fload f2, 0(r1)	2	–	–	$f2_2$	$f2_1$	1	–	$f2_3$	$f2_2$	$f2_1$	–
8	rq_con q1,f2,2 fadd f8, f6, f2	1	–	$f2_3$	$f2_2$	$f2_1$	1	–	$f2_3$	$f2_2$	$f2_1$	$f2_1$
9a	rq_con q1,f2,1 fadd f6, f6, f2	1	–	$f2_3$	$f2_2$	$f2_1$			$f2_3$	$f2_2$	$f2_1$	$f2_2$
9b	fload f2, 0(r1)	1	–	$f2_3$	$f2_2$	$f2_1$	0	$f2_4$	$f2_3$	$f2_2$	$f2_1$	–

(c)

Figure 5: Software pipeline schedule with architected register pressure (*fadd* latency = 1).

- the read from *f2* uses the queue mapping (*q1*) and read offset forwarded from the *rq-connect* instruction issued the same cycle (instead of reading the current *ro* field from the map table) and adds that to the *Qtail* value at the start of the cycle (1), accessing element $f2_2$.

- the write to *f2* uses the queue mapping forwarded from *rq-connect* instruction issued the same cycle, decrements the *Qtail* value (to 0) and writes $f2_4$ into *q1* at position 0.

Figure 5 (c) shows the queue requests that occur in the prologue and the first iteration of the loop kernel. Writes to the queue occur on cycles 3,5,7 and 9. Reads occur on cycles 6 and 9 for the *fadd f6,f6,f2* instruction (using read offset 1) and on cycle 8 for the *fadd f8,f6,f2* instruction (using read offset 2). The ninth cycle is separated into two parts to illustrate the details of the queue access: cycle 9a shows the queue read performed by *fadd f6, f6, f2* using an offset of 1 from the current *Qtail* of 1, returning element $f2_2$. The queue location for $f2_4$ is allocated and the *Qtail* of *q1* is updated in the second phase of that cycle. Remapping *f2* continues as accesses by the *fadd* instructions are executed; otherwise instruction flow is similar to previous examples.

4 Experimental Results

To demonstrate the capability of the RQ approach, we compare the register space and kernel code requirements for various load latencies in the RR and both MVE methods (labeled MVE1 and MVE2) and compare the results to the RQ scheme. We then vary the load latency from 1 cycle to 45 cycles to assess how the resource requirements might vary are across a wide variety of machine models.

We use an iterative modulo scheduler (IMS) [18] which produces a near optimal steady-state throughput for machines with realistic machine models. IMS constructs a schedule that minimizes the number of architected registers required for given a loop L, a machine architecture M, and initiation interval *II*.

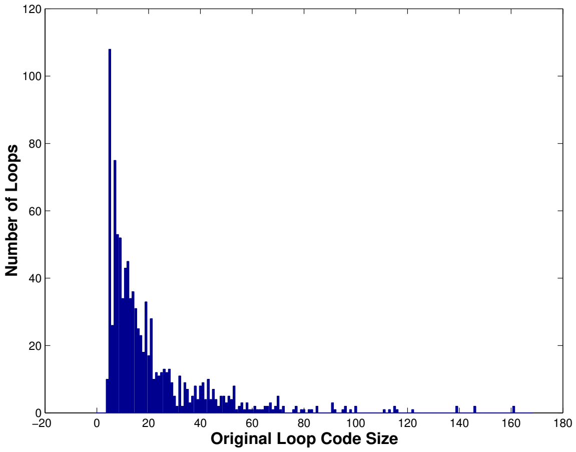
The benchmark loops studied were obtained from the Perfect Club Suite, SPEC and the Livermore Kernels. These loop kernels were provided by B.R. Rau from HP Labs. Loops were compiled by the Cydra 5 Fortran77 compiler performing load-store elimination, recurrence back-substitution and IF-conversion. The input to our scheduler consists of the intermediate representation; SP is then performed generating a new intermediate representation with support for RQs. Of the 1327 loops extracted from the applications, 983 were selected for this study; the remaining 344 loops did not perform memory references.

In our experiments we used two target machine models. One machine model has limited resources, while the other has no resource constraints. The code sizes of the 983 loops studied (before SP was performed) are shown in Figure 6(a). A majority of the loops ranged from 5 to 20 instructions, with the largest loops exceeding 100 instructions.

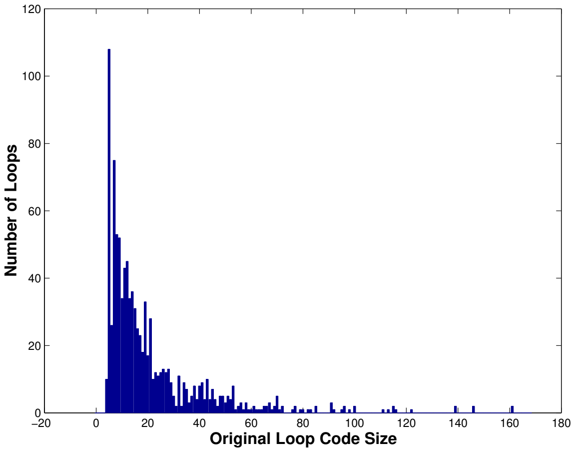
Figure 6(b) shows the initiation intervals for the loops assuming no resource dependencies and with a load latency of 13 cycles. A majority of the loops have *II* between 2 and 15 cycles, with a few loops requiring 200 cycles.

4.1 Software pipelining using MVE, RR and RQs

The results of our experiments show the effects on architected and physical register requirements as well as the code expansion of the loop due to software pipelining. Software pipelining was performed using both methods of MVE (minimizing register requirements (MVE1) and minimizing unrolling (MVE2)) with no hardware support. SP was also performed targeting each of the two machine



(a) code size of initial loops



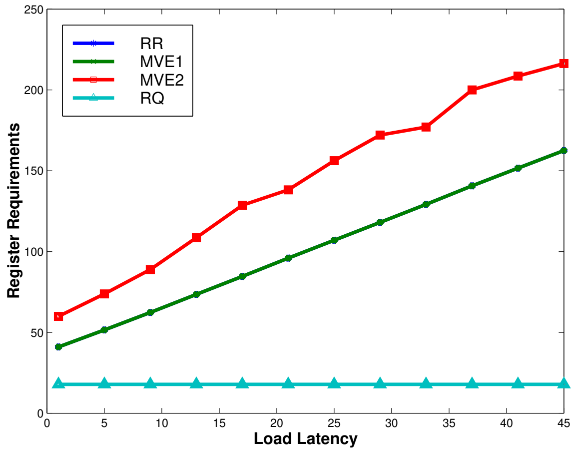
(b) initiation interval of loops

Figure 6: Loop statistics.

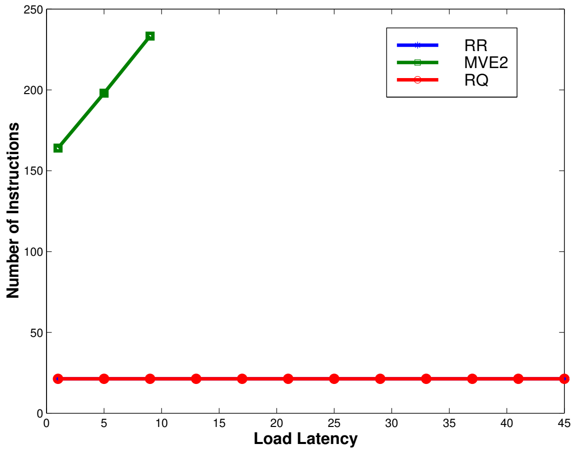
configurations with hardware support: RR, and RQ. These results are presented in Figure 7 and Figure 8 for the two machine models (with unlimited and limited resources, respectively). Figure 7 (a) and Figure 8 (a) show the architected register requirements after performing software pipelining on each loop (averaged over all loops). The graphs show the increase in register requirements and code expansion of the loop kernel as memory latency is increased from 1 to 45 cycles. The two models differ significantly in the registers required to achieve the best software pipelining. The unlimited resource model (which has no resource constraints and therefore a small *II*) requires 2-3 times as many registers as the more realistic machine model. However, the trends seen in both models are similar. In the RR and MVE1 schemes, the number of architected registers are identical, growing at a linear rate. The architected register requirements for MVE2 increase more rapidly, since extra registers are added to reduce the code expansion; this growth rate is also fairly linear. Architected register requirements for the RQ scheme remain constant as long as all live instances of each variable can fit in one register queue. The increased latency only affects the RQ schedule by increasing the offset specified in the *rq-connect* instructions in the loop prologue; as more instances of a variable are needed to support higher latencies the offset is increased to account for the change in the location of the instance that is read. The number of architected registers in the RQ scheme is bounded by the number of consumers (instructions in the loop body which read from the queue) and is not affected by the latency of the instructions.

Figure 7(b) and Figure 8(b) show the code expansion caused by SP as memory latency increases. Code size remains unaffected by memory latency for both RR and RQ due to the hardware support for renaming the instances of a variable. The code size drastically increases in the MVE schemes because of the additional unrolling required to handle the explicit, distinct naming of the additional live instances of the variables defined by load instructions as latency increases. MVE1 is not shown in this graph because of its tremendous code expansion; for a load latency of 13, the kernel code size in MVE1 averages 149,256 instructions!

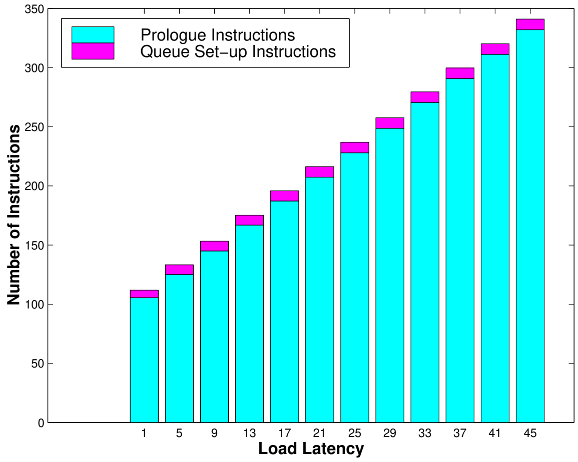
Figure 7(c) and Figure 8(c) show the code expansion of the prologue code as latency increases. Each bar shows the number of instructions moved from early iterations of the original loop to initialize the software pipeline, as well as extra instructions required in the RQ model to connect architected registers to register queues (the darker shaded portion at the top of each bar). The



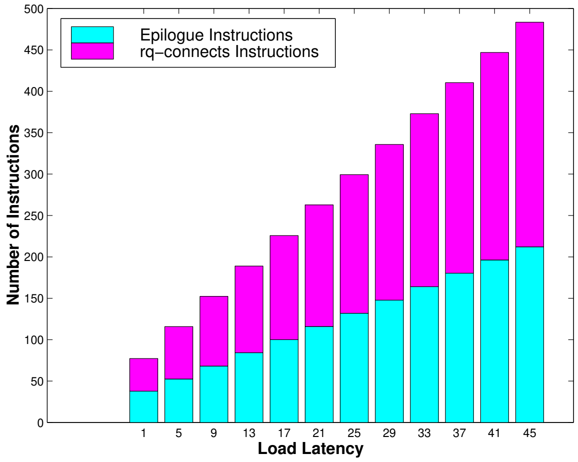
(a) Architected register requirements



(b) Code size requirements

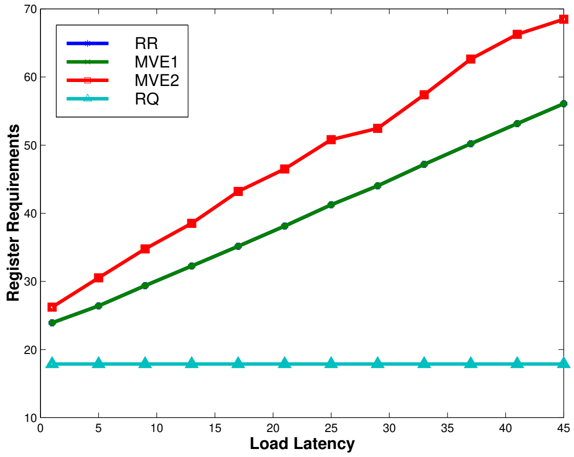


(c) Prologue code size

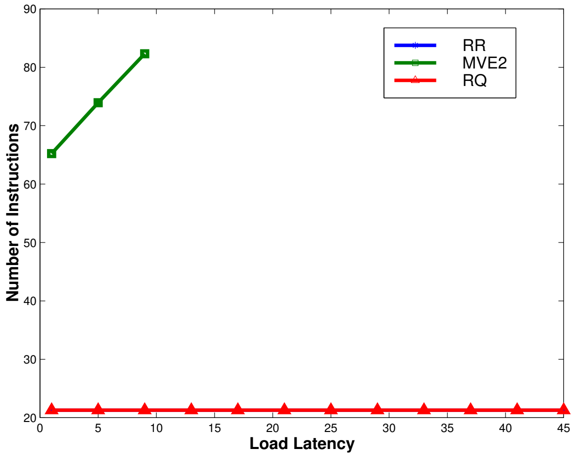


(b) Epilogue code size

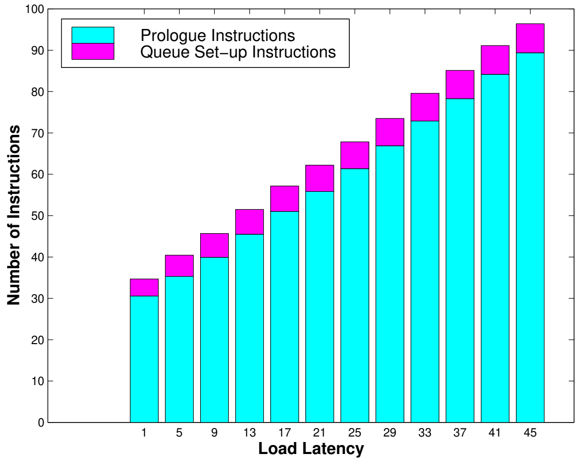
Figure 7: A study of RR, MVE and RQ schemes for Machine Model 2 (with limited resources).



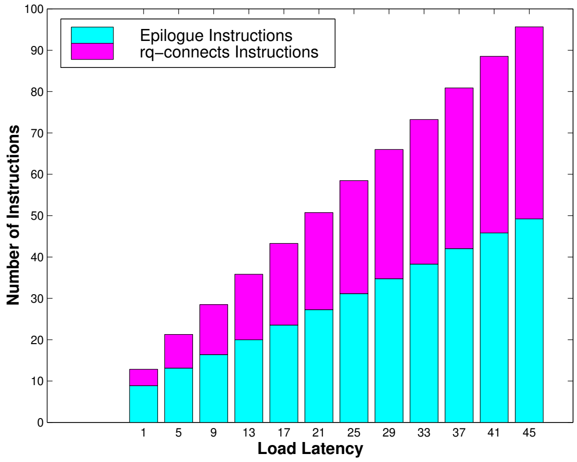
(a) Architected register requirements



(b) Code size requirements



(c) Prologue code size



(b) Epilogue code size

Figure 8: A study of RR, MVE and RQ schemes for Machine Model 1 (with unlimited resources).

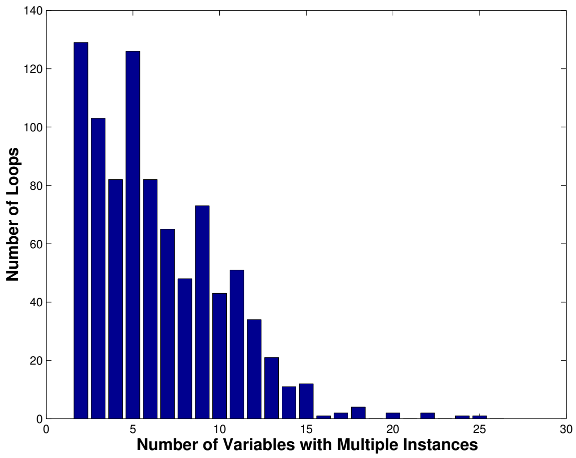


Figure 9: Histogram of the number of multi-instance variables in a loop.

additional overhead in the prologue to initialize the register mappings required in the RQ scheme is seen to be minimal. Figure 7(d) and Figure 8(d) show similar code requirements for the loop epilogue. Here the overhead for the RQ scheme is higher; caused by the necessity to remap architected registers to read the final instances in the queue, since no more writes to the queue are performed to align the queue read offset automatically.

Figure 9 shows the number of variables with multiple instances over all loops. The vertical axis shows how many loops have a specified number of variables with multiple live instances. For instance, the leftmost column (at 2 on the horizontal axis) shows that 130 loops have exactly 2 variables with multiple live instances. Almost all of the loops have fewer than 16 variables with multiple live instances. Since the register queues are allocated only to those variables with multiple live instances, the register queue allocation problem need only address those (few) variables.

Figure 10 shows the number of simultaneously live instances for each of the variables identified in Figure 9. Over half of the variables require only 2 instances, resulting in little physical register pressure in the queue. This result also makes finding a very close to optimal bin-packing solution to register queue mapping quite easy. The largest number of live instances found was 13. Unlike the number of variables with multiple instances (Figure 9), the number of instances for each of those variables will increase in proportion to the memory latency.

Figure 11 shows the rate of increase in the number of instances (averaged over all variables in all loops) as load latency increases. The growth is linear, ranging from 2.5 with low latency (since we only count variables with multiple instances, 2 is an absolute minimum) to 6.5 (for the machine model with limited resources) or 17 (for the machine with unlimited resources) when memory latency is 45. This number is very large when allocating the small number of physical registers found on most machines, making SP intractable. However, since these are only physical register requirements in the RQ model it becomes much more feasible to perform SP.

4.2 Scheduling multiple-use lifetimes for FIFO queues with and without destructive reads

The implementation of register queues presented in this paper might more descriptively be called circular register buffers. Unlike FIFO queues, reads can access any element in the buffer and reads

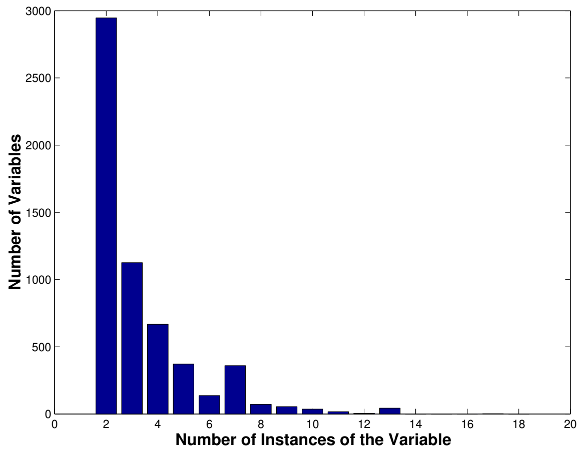


Figure 10: Histogram of the number of instances of variables containing multiple live instances (averaged over all loops).

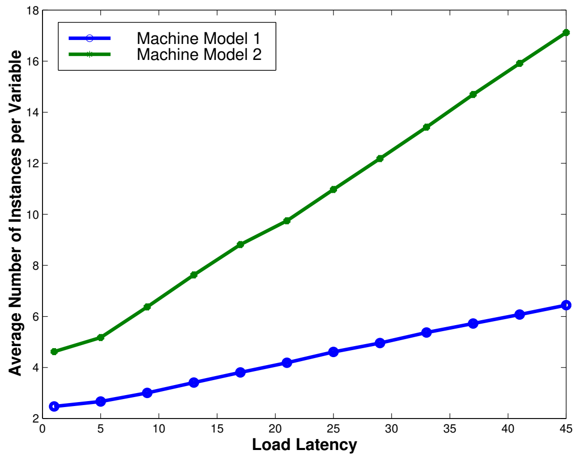


Figure 11: Average queue size as latency increases for machine models 1 (limited resources) and 2 (unlimited).

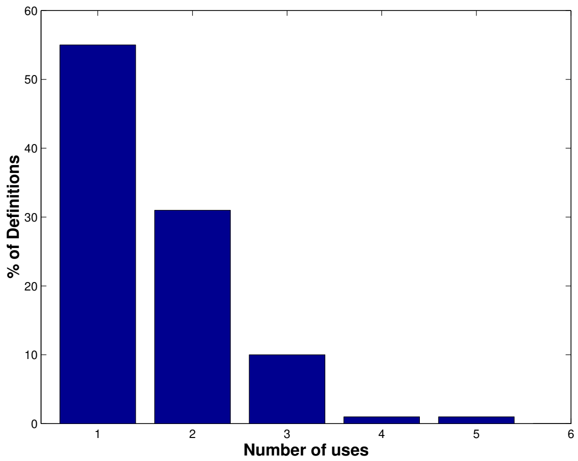


Figure 12: Number of uses for each element of variables with multiple instances.

are non-destructive. We chose this design to enable more flexible access to live variable instances. In this section we examine the effects of this flexible access on the SP schedules by comparing out access mechanism with conventional FIFO organizations which read from the head of the queue. We examine two FIFO designs, one which utilizes a destructive read and a second which employs both destructive and non-destructive reads of the queue.

The first experiment performed to evaluate the effectiveness of our more flexible queue access mechanism was to characterize the usage of all variables that are allocated to a queue in the benchmark loops. For each write to a queue, the number of reads of that element are counted. If there is a single read, then a FIFO organization with destructive reads is sufficient to access the element; destroying the data is allowed since it will not be re-accessed and the data will be located at the head of the queue when accessed provided that the queue does not hold instances of some other variable. However, writes of a variable which is read multiple times make destructive reads unacceptable since a destructive read of the first read access eliminates the data, preventing further reads. In this event, additional code must be inserted to retain the data in some other storage (e.g. a general purpose register) to support multiple reads. Figure 12 shows how many variables have multiple readers. The horizontal axis shows the number of readers for a variable and the vertical axis shows what percent of all the variables allocated to register queues have a given number of readers. Note that 55% of a variables written to a register queue are read only once; these references require no access method more sophisticated than a FIFO queue structure. The remaining 45% of writes require at least 2 reads making destructive reads problematic. At least one FIFO queue design [24] proposed allowing both destructive and non-destructive reads from the head of a queue to provide more flexible access to queue elements. This approach increases the number of variables that can be allocated to a queue without the overhead of moving them to a conventional register prior to using the data; however, it is still too restrictive for some SP pipelined variables. About one third of the variables with multiple reads require reads of the same instance in different iterations of the SP kernel. This means that the first read does not occur when the element is at the head of the queue, since it must remain in the queue until the last read. In this event, a FIFO queue structure requires that multiple queues; one queue to store the data from the write to the first read, a second to store the data between successive reads in different SP kernel iterations, and possibly additional queues if reads occur on three or more different iterations.

To determine the instruction overhead required within the SP kernel if FIFO queues were used, we rescheduled each of the loops with at least one variable that was written to a queue and read two or more times. The loop kernel instruction count was increased by 60% for those loops when only destructive FIFO accesses was provided. This overhead included register queue to general purpose register moves and re-queueing instructions when reads occurred in different loop iterations. The overhead was reduced to 19% when non-destructive reads from the head of the queue were allowed. This overhead consisted of register queue to register queue moves for variables with reads in different SP kernel iterations.

5 Conclusions

Existing SP implementations have limited effectiveness due to their high architected register requirements, particularly as operational latencies grow. In this paper, we have introduced the RQ technique which limits architected register pressure and code size increases from software pipeline schedules by combining a modification to the architecture and microarchitecture of a processor with a modified register allocation algorithm in the compiler.

RQ achieves this goal by combining the features of RR (to enable instances of a variable defined in earlier iterations to be accessed efficiently) with the features of RC (to decouple architected registers from the physical registers holding live variable instances). By including the dynamic register name mechanisms found in RR, we can achieve a software pipelined loop without unrolling the kernel, and by adding the register decoupling capabilities of RC we can allocate multiple instances of a loop variable without increasing architected register pressure. This enables RQ to schedule loops for expected memory latencies when a cache miss occurs; the alternative is to assume that all memory accesses will hit in the L1 cache and stall the processor when a miss occurs which leads to non-optimal schedules, particularly when cache miss rates are high.

Our experiments on the loops from a large benchmark suite showed that RQ provides a significant reduction in the number of architected registers and code size requirements (compared to RR and MVE). Furthermore, memory latency increases have little effect on either code size or architectural register requirements. RQ thus enables more aggressive implementation of software pipelining. Finally, by allowing reads to occur non-destructively and from any location in the queue, RQ can significant reduce the instruction overhead required to access the values stored in conventional FIFO queues.

RQ can also be incorporated into existing instruction set architectures with the addition of a single new instruction and a modification of the register renaming microarchitecture. Furthermore, the complexity of the implementation approximates that of RR, requiring a single level of indirection and modulo arithmetic of small (4 or 5 bit) offsets to address the physical registers in the queue (for queues of length 16 or 32). The physical register requirements of RQ can also be scaled by reducing the number of registers in a queue and/or by restricting the number of queues. The results show that a small number of modest size queues is sufficient to support software pipelining, even as instruction latencies increase.

6 Acknowledgments

This work has been funded by NSF Career award MIP9734023 and gifts from IBM and Intel.

References

[1] *IA-64 Application Developer's Architecture Guide, Rev 1.0.* Intel Document 245188, http://developer.intel.com/design/ia64/, 1986.

[2] G. Beck, D. Yen, and T. Anderson. The cydra-5 minisupercomputer: Architecture and implementation. *Journal of Supercomputing*, 7(1):143–180, May 1993.

[3] A. Charlesworth. An approach to scientific array processing: the architectural design of the ap-120b/fps-164 family. *IEEE Computer*, 14(9):18–27, Sept. 1981.

[4] R. G. Cytron. *Compiler-time Scheduling and Optimization for Asynchronous Machines.* Dept. of Computer Science Report UIUCDCS-R-84-1177, University of Illinois at Urbana-Champaign, Urbana, IL, 1984, 1984.

[5] A. E. Eichenberger, E. Davidson, and S. G. Abraham. Minimum register requirements for a modulo schedule. In *Proceedings of 27th International Symposium on Microarchitecture*, pages 75–84, Nov. 1994.

[6] A. E. Eichenberger and E. S. Davidson. Stage scheduling: A technique to reduce the register requirements of a modulo schedule. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 338–349, Nov. 1995.

[7] M. Fernandes, J. Llosa., and N. Topham. Partitioned schedules for clustered vliw architectures. In *Proceedings of 12th International Parallel Processing Symposium*, pages 386–391, Mar. 1998.

[8] M. A. Fernandes. *Clustered VLIW Architecture Based on Queue Register File.* Department of Computer Science, University of Edinburgh.

[9] R. Govindarajan, E. R. Altman, and G. R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proceedings of 27th International Symposium on Microarchitecture*, pages 85–94, Nov. 1994.

[10] P. Y.-T. Hsu. *Highly Concurrent Scalar Processing.* PhD Dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 1986.

[11] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 258–267, June 1993.

[12] V. Kathail, M. Schlansker, and B. R. Rau. *HPL PlayDoh Architecture Specifications: Version 1.0.* HP Laboratories Technical Report HPL-93-80, 1994.

[13] R. Keller. Lookahead processors. *ACM Computing Surveys*, pages 177–195, Dec. 1975.

[14] K. Kiyohara, S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, S. Anik, and W. W. Hwu. Register connection: A new approach to adding registers into instruction set architectures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 247–256, May 1993.

[15] M. Lam. Software pipelining: an effective scheduling technique for vliw machines. In *Proceedings of the ACM SIGPLAN '88 Conference on programming language Design and Implementation*, pages 318–327, 1988.

[16] J. Llosa, M. Valero, J. Fortes, and E. Ayguade. Using sacks to organize registers in vliw machines. In *Proceedings of International Conference on Parallel and Vector Processing*, Sept. 1994.

[17] W. Mangione-Smith, S. G. Abraham, and E. S. Davidson. Register requirements of pipelined processors. In *Proceedings of the International Conference on Supercomputing*, pages 260–271, July 1992.

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

[19] B. R. Rau, C. D. Glaeser, and R. L. Picard. Efficient code generation for horizontal architectures: Computer techniques and architectural support. In *Proceedings of the ninth Annual International Symposium on Computer Architecture*, pages 131–139, 1982.

[20] B. R. Rau, P. J. Kuekes, and C. D. Glaeser. *A Statically Scheduled VLSI Interconnect for Parallel Processors.* Computer Science Press, 1981.

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

[22] B. R. Rau, D. W. L. Yen, and R. A. Towle. The cydra 5 departmental supercomputer. *IEEE Computer*, 1(22):12–34, Jan. 1989.

[23] J. E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th Annual International Symposium on Computer Architecture*, pages 112–119, June 1982.

[24] G. S. Tyson. *Evaluation of a Scalable Decoupled Microprocessor Design.* PhD Dissertation, The University of California - Davis, 1997.

[25] W. A. Wulf. Evaluation of the wm architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architectur*, pages 382–390, May 1992.

Gary S. Tyson

Gary S. Tyson received a BS in Computer Science (California State University, Sacramento, 1986), an M.S. in Computer Science (California State University, Sacramento, 1988), and a PhD. in CS (University of California, Davis, 1997). While completing his Degree he worked as an independent consultant and an assistant Professor at the University of California, Riverside. Upon completion of his degree, he joined the faculty of the University of Michigan as an assistant Professor. His research interests include compiler design, computer architecture and the interaction between code transformation and instruction set design. His current research focuses on techniques to improve memory system performance for workstations and instruction set design for embeded processors. He is a member of ACM, IEEE and Tau Beta Pi.

Mikhail Smelyanskiy

Mikhail Smelyanskiy received a BS in Computer Science (University of Michigan at Dearborn, 1996) and an MS in Computer Science and Engineering (University of Michigan, 1999). He is currently a research assistant in the Advanced Computer Architecture Laboratory, where he is pursuing a PhD in computer science and engineering. His current research interests are in performance modeling and evaluation of the computer systems.

Edward Davidson

Edward S. Davidson received a BA in mathematics (Harvard University, l96l), an M.S. in communication science (University of Michigan, l962), and a PhD. in EE (University of Illinois, l968). After working at Honeywell ('62-65), and on the faculties of Stanford University ('68-73), and the University of Illinois ('73-87), he joined The University of Michigan as professor of EECS in '98, served as its chair through 1990, and as chair of computer science and engineering (1997-2000). He retired from Michigan in June 2000, was appointed Professor Emeritus and works part time at IBM Research. He managed the hardware design of the Cedar parallel supercomputer at the Center for Supercomputing Research and Development (1984-87) and directed Michigan's Center for Parallel Computing (1994-97). His research interests include computer architecture, supercomputing, parallel and pipeline processing, performance modeling, application code assessment and tuning, and intelligent caches. With his graduate students, he developed the reservation table approach to optimum design and cyclic scheduling of pipelines, designed and implemented an 8 microprocessor SMP system in 1976, proposed multiple-stream shared pipelines in 1978, structured memory access architectures to decouple the access and execute processes in 1983, and developed a variety of systematic methods for computer performance evaluation and optimal design. He has supervised 46 PhD and 38 MS students, was elected Fellow of the IEEE (1984) and Chair of ACM-SIGARCH (l979-l983), and is a recipient of the Harry M. Goode Memorial Award (1992), the Taylor L. Booth Education Award (1996), and the Eckert-Mauchly Award (2000).