Register Queues: A New Hardware/Software Approach to Efficient Software Pipelining

Mikhail Smelyanskiy, Gary S. Tyson and Edward S. Davidson
Advanced Computer Architecture Lab
The University of Michigan
{msmelyan, tyson, davidson}@eecs.umich.edu

Abstract

In this paper we propose 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.

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 **software pipelining** [14], 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. However, spreading the definition and use increases the variable lifetime as well as the number of simultaneously live (overlapped) instances of that variable from different iterations of the loop body. To accommodate these variables, each of the simultaneously live instances needs its own register. Furthermore, each instance must be uniquely identified to match the use of a variable to the correct definition; there must be some mechanism to differentiate among 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) [14], and the **rotating register file** (RRF) [20][21]. 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, the RRF is a hardware-managed register renaming scheme that eliminates the code size 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 access a register that was defined *n* iterations ago. However, since the register file contains references to architected registers, RRF still requires a large number of architected registers to generate 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 software pipelined (SP) loops therefore grows linearly with increased functional unit latencies [16], i.e., a longer latency operation within the loop leads to a greater number of interleaved instances of the loop and therefore more live instances of each variable. Therefore, a shortage of architected registers limits the number of interleaved loop iterations (to avoid register spill code), which can degrade performance by lowering the loop throughput. Thus, efficient software pipeline schedules that account for realistic memory latencies are difficult, and often impossible, to achieve 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 [10] 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 inherent in SP.

In this work we demonstrate that by using ***Register Queues*** (RQs) 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[22][25]. Software pipelining using queues has also been studied in VLIW architectures [15][7][6] and decoupled processors[24], but not in general purpose superscalar designs. In particular, [6] 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 [11] by making the queues architecturally visible; earlier work scheduled values in *pipeline* regis-

Time	iteration i	iteration i+1	iteration i+2
0	iadd r1, r1, #4		
1	fload f2, 0(r1)		
2		iadd r1, r1, #4	
3		fload f2, 0(r1)	
4	fadd f6, f6, f2		iadd r1, r1, #4
5			fload f2, 0(r1)
6		fadd f6, f6, f2	
7			
8			fadd f6, f6, f2
9			

Figure 1: A software pipeline example. Sample program that adds elements of a floating-point array and stores the sum in a scalar. Shown are multiple iterations with II=2.

ters, 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 easily integrate RQs into an existing instruction set architecture and 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 ([1], [21]) and register connection [13] concepts. This enables a decoupling of the total register space for SP into a small set of architected registers and a large set of physical registers organized as circular buffers which can be 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 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 the architected register and 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 as well as architectural modifications required to support this approach. Section 4 provides the results of our experiments on 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 Fig. 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 $\boldsymbol{II} \geq \mathbf{2}$). This loop also contains a loop-carried dependence between instances of the floating point add with a latency of 2, (again, we must make $\boldsymbol{II} \geq \mathbf{2}$).

Fig. 1 shows a software pipelined code sequence, for **II** = 2. Instructions at time steps 0-3 form the *prologue* of the software pipelined loop, time steps 4-5 are the steady-state segment (or *kernel* of the loop), and 6-9 form the loop *epilogue*. The prologue and epilogue are executed once and the steady-state portion (shaded) is executed repeatedly (*n-2* times for this loop executing *n* iterations).

The example in Fig. 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 (WAR hazard) problem is to increase the initiation interval to 4 to allow the *fadd* operation from the *i*th iteration to complete before load 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 [14] (MVE) 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 $\left\lceil \frac{l}{\mathbf{II}} \right\rceil$ 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 then be expanded by a factor of at least $\left\lceil \frac{l}{\mathbf{II}} \right\rceil$

to account for the different register specifiers required for successive definitions of the variable. The degree of unrolling is determined by the requirements for **all** variables, knowing the 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 necessary. Each variable v_i is allocated its minimum number of registers, q_i, and the degree of unrolling is given by the **lowest common multiple** of the set of registers required for all variables in $\{q\}$. 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 is the $max\ q_i$ which is never more than $lcm\{q_i\}$ required by MVE1. However, rather than always requiring q_i registers as in MVE1, MVE2 requires q_i if $u\ mod\ q_i=0$, but $u> q_i$ otherwise.

Several additional techniques have been proposed to minimize register requirements in SP loops. In [9], 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 to lower register requirements in [8][5] by optimizing according to several potentially conflicting constraints, such as fulfilling the resource constraints, scheduling operations along critical dependence cycles, maximizing the throughput of the schedule, and minimizing of the schedule length of the critical path. Stage scheduling [4] breaks the schedule into two steps. In the first step, a modulo scheduler generates a schedule with high throughput and 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 will apply to other scheduling algorithms as well — they simply reduce the physical register requirements of register queues.

Rau, et. al [21] addressed the naming problem in software pipelined loops by employing a new method of addressing a processor register file in the Cydra-5 minisupercomputer [1]. The Rotating Register File (RRF) 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 RRF). 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. RRF 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 register file [20].

The problem of increasing a limited architected register space without dramatically changing an existing instruction set has also been explored. The **Register Connection** (RC) [13] method tolerates high demand for the architected registers by adding a set of extended registers 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 an architected register to an extended register. Once connected, all accesses using the architected register are automatically directed to the appropriate 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[12] 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 instruction. While the RC work did not target software pipelined loops, 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.

Using RC to perform SP in the context of modulo variable expansion significantly reduces architected register requirements. But like MVE, RC still requires loop unrolling to address the register naming problem. Furthermore, RC adds extra connect instructions to the loop kernel, prologue and epilogue. However, by combining the naming solution proposed in RRF with the register decoupling proposed in RC, we now develop a new register naming extension that targets software pipelined loops.

3. Register Queues

We now propose an alternative scheme called Register Queues (RQs). RQs incorporate a hardware-managed register renaming feature similar to RRF and the register decoupling of RC to eliminate 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 Fig. 2:

- **a set of register queues**: Each queue has a *Qtail* pointer, analogous to the ICP in the RRF, and a set of contiguous registers which share a common namespace with the physical register file, but are logically

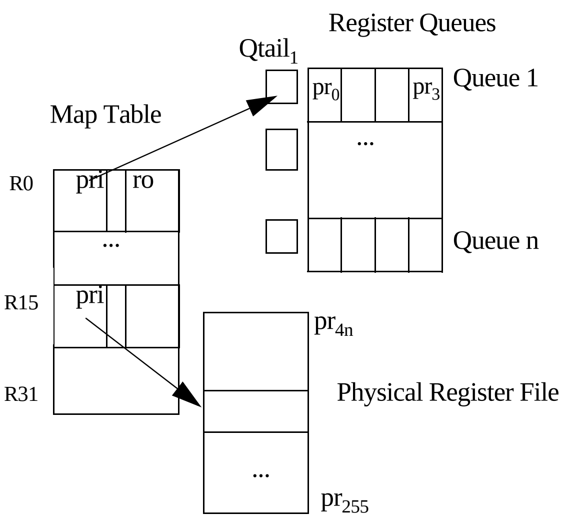


Figure 2: Microarchitectural extensions to support RQ for a machine with 32 architected registers, *n* queues of length 4 and 256 physical registers.

(and probably physically) separate. In Figure 2 the registers that constitute register queue 1 are physical registers pr_0 through pr_3; physical registers pr_4 through pr_7 make up register queue 2, etc. These registers are analogous to the registers in the RRF, using the same modulo arithmetic to index into the queue. They differ in that registers in the queue are not directly indexed by architected registers, but must be explicitly mapped to an architected register. Like the RRF register, 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 either to a physical register (using standard register renaming logic) or a register queue (using an *rq-connect* instruction). Each entry in the map table entry contains a physical register index (*pri*) and a read offset (*ro*). The index identifies either the physical register or the register queue 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 *$rq* by writing the queue number into the *pri* field of the map table. Furthermore, the read offset (*ro*) in the queue is specified by the immediate field *imm*. Any reads of architected register *$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 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:*** remap 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 the register queue.

Once an architected register is mapped to a register queue, when a read access occurs the following events take place:

1. Translate the register specifier in the operand field of the machine instruction to 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* using the *read offset*. To do this the *Qtail* is added to *ro*, modulo the number of registers in the queue. 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 RRF.
3. Read the contents of the physical register specified (or pass the 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. Translate the register specifier in the operand field of the machine instruction to a register queue identifier (the *pri* field) using the register map table. 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 RRF. Note that in RQs the update of *Qtail* occurs on a write,

whereas in the RRF the ICP is updated using a special branch instruction. Both solutions effectively manage the register naming problem.

3. Pass the physical register identifier at the *Qtail* position in the queue 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. Furthermore, it should now be apparent that the offset (*ro*) field of the map table entry should be 0 to reference the most recently defined variable instance.

3.1. SP Scheduling using register queues

Managing the dynamic mapping as a queue enables efficient software pipeline schedules with little change in code size or architected register requirements. Like the RRF, each register queue provides a set of registers to contain instances of a variable for several successive iterations. While the RRF uses a contiguous set of RRF architected registers enabling unconstrained access to any of the register, RQs use a single register queue to hold all instances of a particular variable. Architected registers 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, an architected register is mapped to the third most recent definition. The two more recent definitions are stored with queue offsets of 0 and 1; these queue locations need not be mapped to architected registers if they will not be referenced until a later iteration. This property eliminates the need for unrolling the loop kernel since architected registers are only mapped to offsets in the queue that reading variable instance values; writes to those variables are always to the decremented Qtail. The architectural register requirements for a variable are calculated by counting the total number of locations in which a variable is referenced, not by counting all simultaneously instances, which can be much greater. RQs therefore reduce architected register pressure for software pipeline loops.

The functionality of RQs can be demonstrated by reexamining the loop fragment from Fig. 1. Fig. 3 (a) illustrates the RQ scheduling, 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*. With a read offset of 1, any reads of *f2* will retrieve the contents of *q1* register (*Qtail*+1) 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 the table 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).

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-disconnect f2

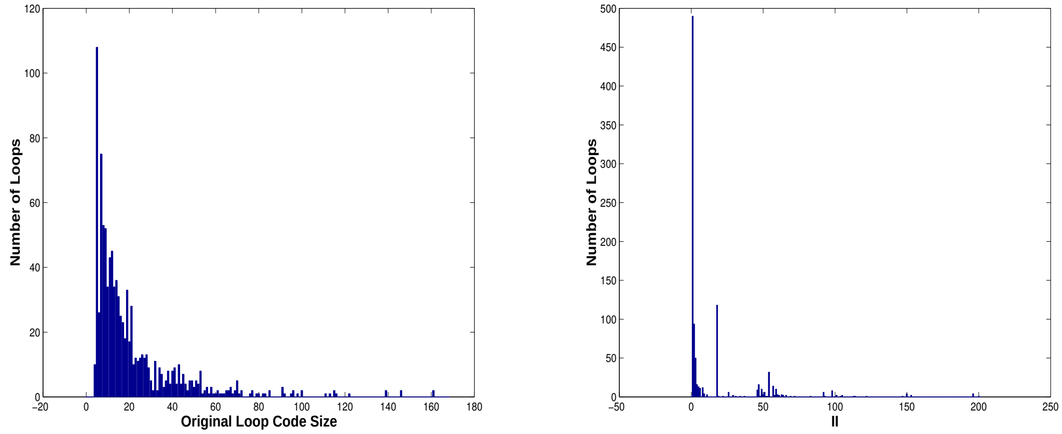
Figure 3: SP schedule for example w/ RQs

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 complete 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 will not be moved in the queue since no further writes to the queue are performed. In this case, 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 accessing 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.

In this example, we have only a single variable with two live instances. In general, there may be many variables with multiple instances. A simple scheduling strategy would employ a single register queue for each variable, holding all live instances of the variable. This works well in reducing the architected register pressure, 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).

Fortunately, since multiple read offsets may be specified, 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 number of writes for all variable instances mapped to the queue. This is simple when writes for each variable are unconditionally performed; it is 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



(a) code size of initial loops (b) initiation interval of loops

Figure 4: Loop statistics.

must carefully determine the possible read offsets or assign the conditionally defined variable to a new 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 (as appropriate for this resource dependency), or we can concatenate two or more queues by copying the head of the first queue to the tail of the second queue. This costs an 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 read offset).

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 allocated to a register queue. This 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.

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 RRF 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.

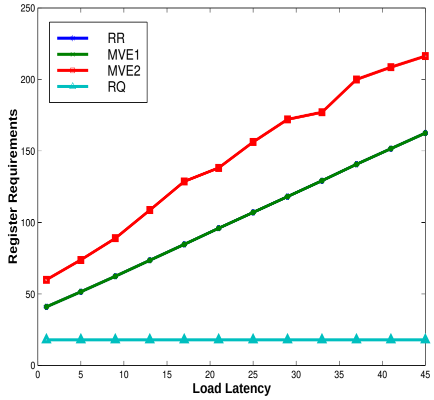
We use an iterative modulo scheduler (IMS) [18] which produces a near optimal steady-state throughput for machines with realistic machine models. IMS constructs a schedule that minimizes the number of architected registers required for given a loop *L*, a machine architecture *M*, and initiation interval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

The benchmark loops studies 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.

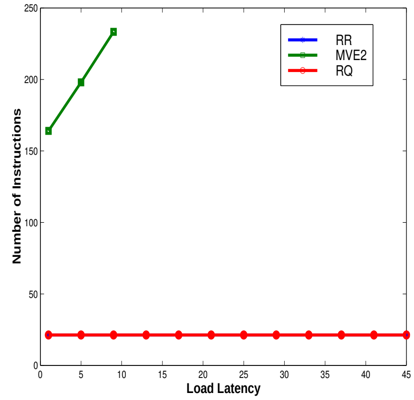
In our experiments we used two target machine models. One machine model has limited resources, while the other has no resource constraints. The code sizes of the 983 loops studied (before SP was performed) are shown in Fig. 4(a). A majority of the loops ranged from 5 to 20 instructions, with the largest loops exceeding 100 instructions. Figure 4(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

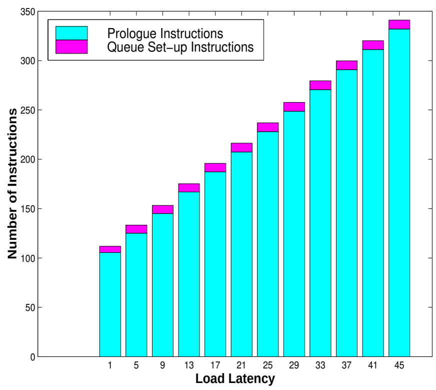
The results of our experiments show the effects on architected and physical register requirements as well as the code expansion of the loop due to software pipelining. Software pipelining was performed using both methods of MVE (minimizing register requirements (MVE1) and min-



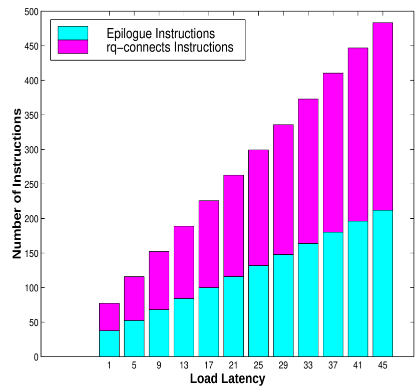
(a) Architected Register Requirements



(b) Code Size Requirements



(c) Prologue Code Size



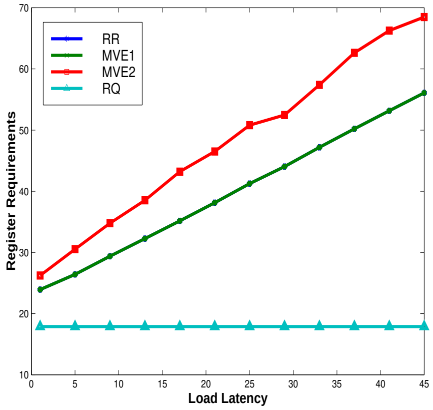
(d) Epilogue Code Size

Figure 5: Study of RRF, MVE and RQ schemes for Machine Model 2 (with unlimited resources).

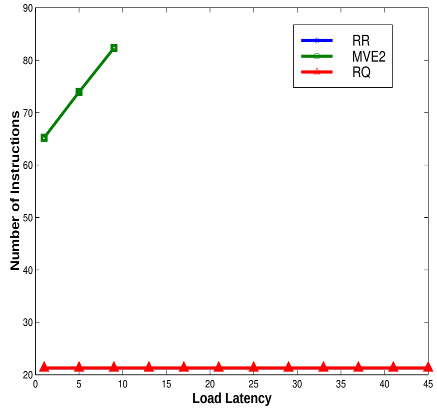
imizing unrolling (MVE2)) with no hardware support. SP was also performed targeting each of the two machine configurations with hardware support: RRF, and RQ. These results are presented in Fig. 5 and Fig. 6 for the two machine models (with unlimited and limited resources, respectively). Fig. 5 (a) and Fig. 6 (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 RRF 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.

Fig. 5(b) and Fig. 6(b) show the code expansion caused by SP as memory latency increases. Code size remains unaffected by memory latency for both RRF 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!

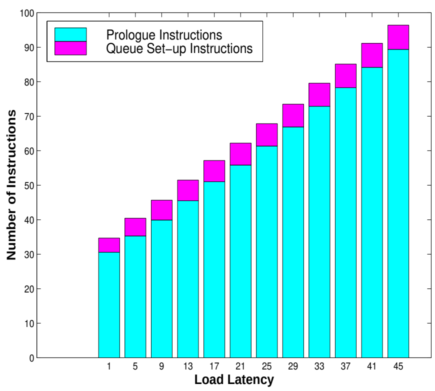
Fig. 5(c) and Fig. 6(c) show the code expansion of the prologue code as latency increases. Each bar shows the number of instructions moved from early iterations of the original loop to initialize the software pipeline, as well as



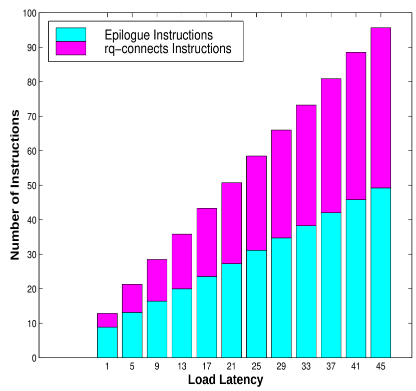
(a) Architected Register Requirements



(b) Code Size Requirements



(c) Prologue Code Size



(d) Epilogue Code Size

Figure 6: Study of RRF, MVE and RQ schemes for Machine Model 1 (with limited resources).

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 additional overhead in the prologue to initialize the register mappings required in the RQ scheme is seen to be minimal. Fig. 5(d) and Fig. 6(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.

Fig. 7 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.

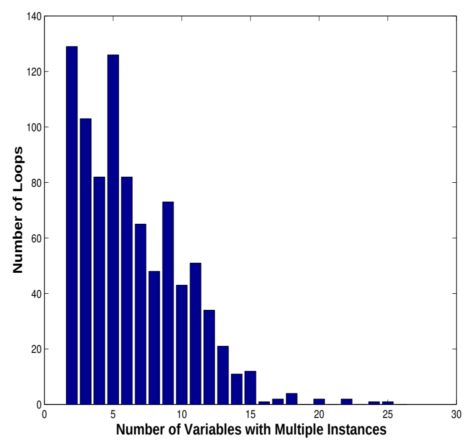


Figure 7: Histogram of the number of multi-instance variables in a loop.

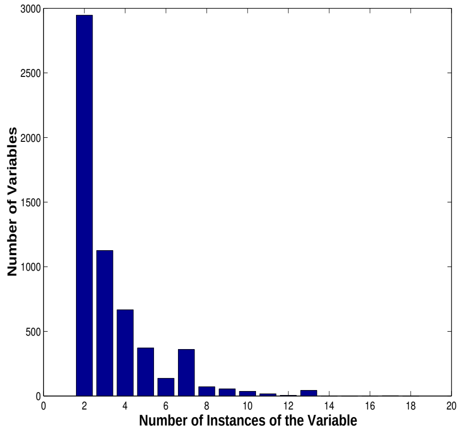


Figure 8: Histogram of number of instances of variables (containing multiple instances) over all loops (Machine Model 1, memory latency 13).

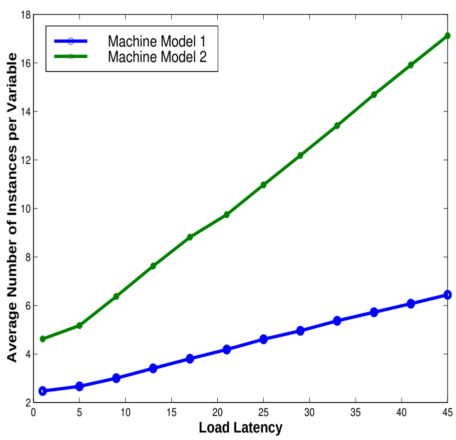


Figure 9: Average queue size as latency increases for machine models with limited (Machine Model 1) and unlimited (Machine Model 2) resources.

Fig. 8 shows the number of simultaneously live instances for each of the variables identified in Fig. 7. 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 (Fig. 7), the number of instances for each of those variables will increase in proportion to the memory latency.

Fig. 9 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.

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 RRF (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 RRF, 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 RRF 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.

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 RRF, 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 architected 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. We would also like to thank the anonymous referees for their helpful comments and suggestions for improvement of this paper.

7. References

[1] G.R. Beck, D.W.L. Yen, and T.L. Anderson, “The Cydra-5 minisupercomputer: Architecture and implementation”, *The Journal of Supercomputing*, Vol. 7, pages 143-180, May 1993.

[2] D. Callahan, S. Carr, K. Kennedy, “Imrpoving register allocation for subscripted variables”, *SIGPLAN ‘88 Conference on Programming Language Design and Implementation*, pages 20-22, June 1990.

[3] A.E. Charlesworth, “An approach to scientific array processing: the architectural design of the AP-120B/FPS-164 family”, *IEEE Computer*, Vol. 14, No. 9, pages 18-27, 1981.

[4] A. E. Eichenberger, E. S. Davidson, “Stage scheduling: A technique to reduce the register requirements of a modulo schedule”, In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 85-94, 1994.

[5] A. E. Eichenberger, E.S. 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] M. A. Fernandes, “Clustered VLIW Architecture Based on Queue Register File”, Ph. D. Thesis, Department of Computer Science, University of Edinburgh.

[7] M. Fernandes, J. Llosa. and N. Topham, “Partitioned Schedules for Clustered VLIW Architectures”, In *Proceedings of 12th International Parallel Processing Symposium*, April 1998.

[8] R. Govindarajan, E. R. Altman, 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.

[9] R. A. Huff, “Lifetime-sensitive modulo scheduling”, In P*roceedings of the ACM SIGPLAN’93 Conference on Programming Language Design and Implementation*, pages 258-267, June 1993.

[10] IA-64 Application Developer’s Architecture Guide, Rev 1.0, Intel Document #245188. Available at: http://developer.intel.com/design/ia64/.

[11] V. Kathail, M. Schlansker, B. R. Rau, “HPL PlayDoh Architecture Specifications: Version 1.0”, HP Laboratories Technical Report # HPL-93-80, February 1994.

[12] R. Keller, “Lookahead processors”, *ACM Computing Surveys*, pages. 177-195, Dec. 1975.

[13] K. Kiyohara, S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, S. Anik, 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.

[14] 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.

[15] 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*, Sep. 1994.

[16] 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.

[17] J. H. Patel and E. S. Davidson, “Improving the Throughput of a Pipeline by Insertion of Delays”, *In Proceedings of the 3d International Conference on Computer Architecture*, pages 159-164, Jan. 1976.

[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 and J. A. Fisher, “Instruction-level parallel processing: History, overview, and perspective”, *The Journal of Supercomputing*, Kluwer Academic Publishers, Vol. 7, pages 9-50, July 1993.

[20] B. R. Rau, M. Lee, P. P. Tirumalai, 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.

[21] B. R. Rau, D. W. L. Yen, R. A. Towle, “The Cydra 5 departmental supercomputer”, *IEEE Comp*. 22, Vol. 1, pages 12-34, Jan. 1989.

[22] J. E. Smith, “Decoupled Access/Execute Computer Architectures”, In *Proceedings of the 9th Annual International Symposium on Computer Architecture*, pages 112-119, June 1982.

[23] G. Sohi‘, “Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers”, *IEEE Transactions on Computer*, Vol. 39, pages 349-359, March 1990.

[24] G. S. Tyson, “Evaluation of a Scalable Decoupled Microprocessor Design”, Ph. D. 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 Architecture*, pages 382-390, May 1992.