A Scalable Instruction Queue Design Using Dependence Chains

Steven E. Raasch, Nathan L. Binkert, and Steven K. Reinhardt

Electrical Engineering and Computer Science Dept.
University of Michigan
1301 Beal Ave.
Ann Arbor, MI 48109-2122
{sraasch,binkertn,stever}@eecs.umich.edu

Abstract

Increasing the number of instruction queue (IQ) entries in a dynamically scheduled processor exposes more instruction-level parallelism, leading to higher performance. However, increasing a conventional IQ's physical size leads to larger latencies and slower clock speeds. We introduce a new IQ design that divides a large queue into small segments, which can be clocked at high frequencies. We use dynamic dependence-based scheduling to promote instructions from segment to segment until they reach a small issue buffer. Our segmented IQ is designed specifically to accommodate variable-latency instructions such as loads. Despite its roughly similar circuit complexity, simulation results indicate that our segmented instruction queue with 512 entries and 128 chains improves performance by up to 69% over a 32-entry conventional instruction queue for SpecINT 2000 benchmarks, and up to 398% for SpecFP 2000 benchmarks. The segmented IQ achieves from 55% to 98% of the performance of a monolithic 512-entry queue while providing the potential for much higher clock speeds.

1 Introduction

To stay on the microprocessor industry's historical performance growth curve, future generations of processors must schedule and issue larger numbers of instructions per cycle, selected from ever larger windows of program execution [18]. In a conventional dynamically scheduled microarchitecture, the execution window size is determined primarily by the capacity of the processor's instruction queue (IQ), which holds decoded instructions until their operands and an appropriate function unit are available. Unfortunately, processor cycle time constrains the size of this physical structure severely. The latency of wakeup logic—which marks queued instructions as ready to execute when their input dependences are satisfied—increases quadratically with both issue width and instruction queue size [17]. Both wakeup and the following selection phase—which chooses a subset of the ready instructions for execution—generally occur within a single cycle, forming a critical path. Advances in semiconductor technology will not provide a solution: although the number of available transistors will continue to increase exponentially, the number of gates reachable in a single cycle will at best stay constant, and possibly decrease [22, 1, 12].

The poor scalability of conventional wakeup logic results from its broadcast nature. To identify the instructions that become ready as a result of newly available values, the identities of these values (i.e., register tags) are broadcast to all queued instructions. The quadratic dependence of latency on IQ size is a direct result of the wire delay involved in driving these tags across the entire queue [17]. However, only a small fraction of the queued instructions become ready in any given cycle; in fact, a significant number of queued instructions cannot possibly become ready, as they depend on other instructions that have not yet been issued.

Dependence-based instruction queue designs [17, 5, 15, 6] seek to address this inefficiency. These modified queues order buffered instructions based on dependence information, with the goal that an instruction is not considered for issue (and thus need not be searched by wakeup or selection logic) until after the instructions on which it depends have issued. However, designs proposed to date have the potential to introduce dispatch or issue dependences that do not reflect actual data dependences (We use dispatch to refer to the process of sending decoded instructions to the instruction queue, and issue to refer to the process of sending instructions from the instruction queue to function units). These artificial dependences limit the ability of the dynamic scheduling mechanism to tolerate unpredictable, long-latency operations such as cache misses.

This paper presents a novel dependence-based instruction queue design that uses only true dependences to constrain instruction flow, allowing flexible dynamic scheduling in the face of unpredictable latencies. We break the IQ into segments, forming a pipeline. Instructions are issued to function units from only the final segment. The flow of instructions from segment to segment is governed by a combination of data dependences and predicted operation latencies. Ideally, instructions reach the final segment only when their inputs are, or will soon be, available.

Our design dynamically constructs subtrees of the data dependence graph as instructions are inserted into the IQ. These subtrees, referred to as *chains*, typically begin with a variable-latency instruction. The edges of the graph are annotated with the expected latency of the value-producing operation from the time it issues; these edge weights are used to schedule instruction issue within a chain.

Because instruction wakeup and selection logic operate independently on each queue segment, the latency of this critical path is determined by the size of each segment, not the overall queue size. Our design can be scaled across varying window sizes and clock frequencies by varying the number of segments and the number of instruction slots per segment.

Our simulations of this design with thirty-two–instruction segments show that our design can achieve from 55% to 98% of the performance of an idealized, monolithic instruction queue. Average performance is 85% of an ideal queue for a 256-element queue, and 81% of an ideal queue for a 512-element queue.

The remainder of this paper begins with a discussion of related work. Sections 3 and 4 describe our basic design and a set of critical enhancements, respectively. Sections 5 and 6 describe our experimental methodology and results. Section 7 discusses future directions for this research, and Section 8 concludes.

2 Related work

Palacharla et al. [17] performed the initial analysis of complexity-induced circuit delay on superscalar processor clock rates, identified the wakeup/select path as a critical bottleneck, and proposed the first dependence-based instruction queue organization. Their design uses a set of FIFOs for the instruction queue. Only the FIFO heads are considered for issue, meaning that the wakeup/select latency scales with the number of FIFOs rather than the number of instruction slots. Dispatch logic attempts to place each instruction in a FIFO immediately behind a preceding instruction that produces one of its operands. If an instruction's operands are available, or if the single FIFO position that immediately succeeds the producer is occupied, the instruction is placed at the head of an empty FIFO. If there are no empty FIFOs, dispatch stalls until one becomes available.

A second form of dependence-based IQ design was proposed independently by Canal and González [5, 6] and by Michaud and Seznec [15]. The common idea among these schemes is to use predicted operation latencies to build what we term a "quasi-static" schedule at dispatch time. The IQ contains a *scheduling array*, a two-dimensional array of instruction slots. The rows of the array correspond to future issue cycles; the instructions within a given row are predicted to become ready in the same cycle, after instructions in preceding rows and before instructions in later rows. Because the schedule is determined at dispatch time, it is more dynamic and adaptive than a static, compiler-generated schedule. However, operand availability is not perfectly predictable even at dispatch time due to cache misses and resource conflicts. The various proposals deal with these unpredictable latencies by augmenting the static scheduling array with a small fully associative buffer similar to a conventional IQ, though they differ in how this buffer is used.

Canal and González's initial "distance" scheme [5] places the fully associative buffer before the scheduling array. Instructions whose ready time cannot be accurately predicted (e.g., due to dependence on an outstanding load) are held in this buffer until their ready time is known. Instructions are thus guaranteed to be ready when they reach the oldest row of scheduling array.

Michaud and Seznec's "prescheduling" approach [15] and Canal and González's "deterministic latency" scheme [6] place the fully associative buffer after the scheduling array. In Michaud and Seznec's model, the fully associative "issue buffer" is located between the scheduling array and the issue stage; instructions from the oldest row of the scheduling array are written into the issue buffer, and instructions are issued out of the issue buffer only. Canal and González's scheme differs only in that instructions may be issued directly from the oldest row of the scheduling array, and are copied to their equivalent of the "issue buffer" only if a mispredicted latency causes them to reach the oldest row before becoming ready.

There are several complementary approaches to decoupling IQ size from latency which could be used in conjunction with a dependence-based IQ design. Most clustered architectures [13, 14, 17, 19, 8] divide the instruction queue among execution clusters, effectively dividing the IQ into "vertical" slices along the width of the machine, rather than the "horizontal" slices provided by our segmented design. Stark et al. [21] propose speculative wakeup based on the availability of "grandparent" values (i.e., the operands of an instruction's producers) to allow pipelining of the wakeup/select operation over two cycles. Brown et al. [3] propose a technique which moves selection logic off the critical path, allowing the wakeup logic to consume a full cycle. Multiscalar architectures [20] expand the instruction window by fetching from multiple points within a logically single-threaded program.

Limiting the number of instructions that can wake up when a single value becomes available allows the use of direct-mapped or low-associativity queue structures [24, 16, 5, 6]. Goshima et al. [10] discuss an alternative wakeup circuit which avoids associative search for small windows.

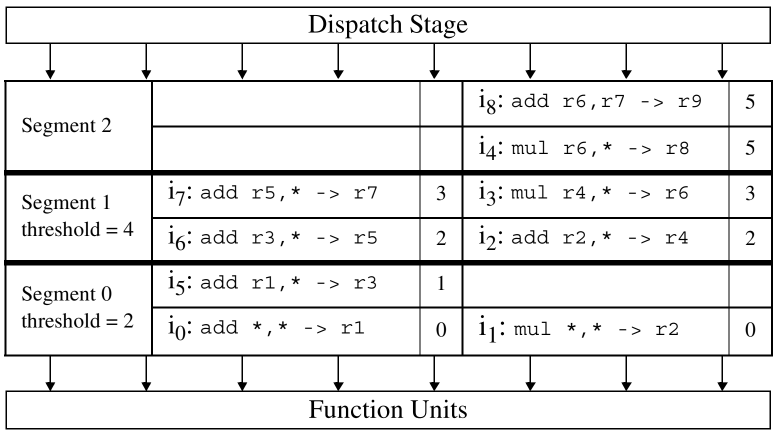
3 The segmented instruction queue

The goal of our design is to exploit dependence information and predictable execution latencies—as do the dependence-based schedulers discussed in Section 2—while maintaining scheduling flexibility to deal with the unpredictable effects of cache misses and resource contention.

As in [15], our scheme issues instructions only from a small "issue buffer", structured like a conventional IQ, and

Instruction	Latency	Delay Value
i_0: `add *,* -> r1`	1	0
i_1: `mul *,* -> r2`	2	0
i_2: `add r2,* -> r4`	1	2
i_3: `mul r4,* -> r6`	2	3
i_4: `mul r6,* -> r8`	2	5
i_5: `add r1,* -> r3`	1	1
i_6: `add r3,* -> r5`	1	2
i_7: `add r5,* -> r7`	1	3
i_8: `add r6,r7 -> r9`	1	5

(a)



(b)

Figure 1. (a) Example code sequence with delay values. Operands denoted by '*' are available. (b) Desired position of instructions within instruction queue after all instructions are dispatched. Numbers to the right of instructions in (b) are the delay values from (a). The column layout shown in part (b) is for illustrative purposes only.

attempts to maximize the efficiency of this small buffer by inserting instructions only when they are expected to be ready to issue. The remainder of the IQ structure is dedicated to staging instructions in such a way that they are available to be inserted in the issue buffer at the time they are predicted to be ready.

The novel aspect of our design is that this staging area is also scheduled dynamically, allowing the IQ to tolerate latencies that are not predictable at dispatch time. Of course, using a large, monolithic, conventional IQ structure as a staging area does not address wakeup/select complexity in a scalable fashion. Instead, we construct the staging area from a pipeline of small, identical queue structures. Each of these structures, or *segments*, is managed using logic similar to the wakeup and select logic of a conventional IQ. However, the individual segments can be sized to meet cycle-time requirements. The overall IQ size—and thus the size of the machine's window for extracting ILP—is determined by the product of the individual segment size and the number of segments. Because each segment is a random-access element, the structure of our segmented IQ does not create inherent scheduling dependences, in contrast to previously proposed FIFO structures.

For the sake of discussion, we present our segmented IQ as a vertical pipeline, with dispatch at the top and issue at the bottom. Instructions are dispatched into the *top segment* and are *promoted* downward from segment to segment until they reach the *bottom segment*, which is the same as the "issue buffer" discussed above. For convenience, we occasionally refer to segments numerically, with segment 0 being the bottom segment and segment $n - 1$ the top in an n-segment IQ. Our design controls the promotion process such that instructions are distributed among the segments according to when they are likely to become ready, with ready instructions in the bottom segment and those furthest from being ready in the upper segments.

Section 3.1 describes our basic scheduling model. Section 3.2 describes how we use dependence chains to efficiently maintain an adaptive schedule. Sections 3.3 and 3.4 provide further details on IQ implementation issues and the chain creation policy.

3.1 Scheduling model

To distribute instructions across the queue segments, we assign each instruction a delay value, which indicates the expected number of cycles until it is ready to issue. We then allow an instruction to promote only when its delay value is less than the *segment threshold* of the destination segment. Initial delay values are assigned by the dispatch stage based on predicted ready times; our process for updating these values as the execution progresses is described in the following sections.

Instructions with a zero delay value are expected to be ready to issue, and are allowed into the bottom segment. To enable back-to-back issue of single-cycle dependent instructions, we also allow instructions with a delay value of one into the bottom segment. We set the threshold of the bottom segment at two, excluding all other instructions, to avoid clogging this segment with instructions which will not soon be ready to issue.

We set the thresholds for subsequent segments using uniform increments of two cycles (resulting in thresholds of 4, 6, 8, etc.) to simplify the promotion logic described in the following section. Instructions may be dispatched into the top segment regardless of their delay value. As a result, a long chain of dependent instructions may fill the top segment; Section 4.1 describes an enhancement which mitigates this behavior.

Figure 1 presents a short example, including a code sequence, a snapshot of the delay values at a particular point in execution, and the desired positions of the instructions in a three-segment queue at that point. For this example, we assume function unit latencies of one cycle for ADD and two cycles for MUL instructions. Instructions i_0 and i_1 are ready to issue, and so are placed in the bottom segment. Instruction i_5 resides there also, so that it can be issued immediately after i_0. The remaining instructions are further from being ready to issue, and so are placed in higher segments.

Once the delay value of an instruction in segment k becomes smaller than the threshold of segment k–1, the instruction becomes eligible for promotion. The instruction queue entry signals its eligibility to the segment promotion logic, which selects some or all of the eligible instructions for promotion in the following cycle, in a manner very similar to the select logic of a conventional IQ. The number of instructions promoted is limited by the inter-segment bandwidth and by the number of available entries in the destination segment. In this paper, we assume the inter-segment bandwidth matches the issue width of the machine. The number of available instruction slots cannot be calculated and propagated through the entire instruction queue in a single cycle, so we assume that each segment's selection logic promotes based on the number of destination slots available in the previous cycle.

After an instruction reaches the bottom segment, it is scheduled for issue based on the actual readiness of its operands, as in a conventional IQ. Thus the delay values of instructions in the bottom segment need not be maintained.

3.2 Updating delay values using instruction chains

The key to providing flexible scheduling in our segmented IQ lies in maintaining appropriate delay values for each instruction. Simply decrementing each delay value on every cycle does not allow deviation from the predicted latency calculated at dispatch, and would be equivalent to the quasi-static schemes of Section 2. Instead, we would like the delay values to adapt dynamically to post-dispatch variations in the execution schedule.

Ideally, the delay value for each instruction should be continuously recalculated based on the latest delay values of the instructions which produce its operands. To achieve this effect, an instruction must communicate with its dependents every time its delay value is updated. Because every instruction in the IQ could update its delay value in any given cycle, and because the update must be broadcast to any IQ entry that may hold a dependent instruction, the cost of such communication is prohibitive.

We avoid this cost by managing instructions in groups called *chains*. A chain is made up of a *head* instruction and other instructions which depend directly or indirectly on the head, i.e., a subtree of the data dependence graph rooted at the head. Each instruction maintains its delay value as a fixed latency behind its chain head. This latency is computed at dispatch as the sum of the predicted latencies along the execution path from the head to the target instruction. Each chain-head instruction is itself a non-head member of another chain, and calculates its delay value based on the predicted latency from its respective chain head.

Delay values are thus maintained by broadcasting status-change updates for chain-head instructions only. In our design, a chain head signals the other chain members only when it is promoted between segments. Because each segment corresponds to a two-cycle latency increment, chain members decrement their delay values by two when notified that their head has promoted.

Once a chain head reaches the bottom segment and issues to an execution unit, the remaining instructions in the chain enter *self-timed mode*, in which each instruction decrements its delay value on each clock cycle. In effect, the instructions do not see the head promote beyond the bottom segment, but their notion of their appropriate distance behind the head is reduced cycle by cycle until they reach the bottom segment themselves.

Figure 1(b) divides the example instructions into two columns to illustrate one possible assignment of these instructions to two chains: Instruction i_0 is a chain head and i_5, i_6, and i_7 belong to its chain. Similarly, i_1 is a chain head and i_2, i_3, i_4, and i_8 belong to this chain. If instruction i_0 issues, then i_5, i_6, and i_7 will enter self-timed mode, gradually promote into segment 0, and then issue. Meanwhile, if i_1 does not issue, then i_2, i_3, i_4, and i_8 will remain in place.

Within a chain, then, instructions are scheduled quasi-statically, much like in previous dependence-based IQ designs [15, 6]. However, *between* chains, our segmented IQ provides fully dynamic scheduling. When a chain head reaches the bottom segment, the entire chain will cease advancing until the head issues. If the head is delayed, the remainder of the chain will not promote into the bottom segment, and valuable issue slots will not be consumed prematurely by these instructions.

Note that some instructions, such as i_8 in the example, may depend indirectly on multiple chain heads. Our most general model allows such instructions to belong to two chains, one for each operand. In this case, the instruction maintains two separate delay values, and dynamically chooses the larger value (indicating the later-arriving operand) to control its segment promotion. Section 4.3 describes the use of operand prediction to choose only one chain for such instructions, as illustrated in Figure 1(b).

3.3 Implementation details

In our proposed design, chain-head promotion and issue information is propagated on a set of *chain wires* using a one-hot encoding (i.e., one wire per chain). When a chain head is selected for promotion or issue, it asserts the wire assigned to its chain. The non-head instructions in the chain monitor this wire to decrement their delay values. Because chain members cannot pass the chain head, promotion signals need to propagate only unidirectionally from the chain head location toward the top of the queue. To minimize wire delay, the chain wires are pipelined from segment to segment. That is, the chain wires asserted in segment *k* in a given cycle are the union of the set of wires asserted by chain heads promoting from *k* to *k*–1 in that cycle and the set of wires asserted in segment *k*–1 in the previous cycle.

A *register information table* in the dispatch stage is used to assign chains and delay values to instructions as they are dispatched. This table is indexed by architected register number and contains four fields: the chain ID of the instruction which will produce the register value, the expected latency of this register value relative to when the chain head will issue, the chain head location (segment number), and a flag to indicate if this chain's instructions are currently in self-timed mode. The status of an instruction's source operands in this table determines the chain or chains to which the instruction is assigned and the initial delay value(s). Once the instruction's chain assignment is complete, the table entry for the destination register is updated.

The register information table monitors the chain wires to keep its entries up to date, much as the instruction queue slots do. When the table notes that a chain head has issued and the chain has entered self-timed mode, the latency field decrements once each cycle to indicate more accurately the number of cycles until the register value is ready. Once the delay value reaches zero, we assume that the value is available for scheduling purposes.

Each instruction queue entry maintains four fields for each chain to which the instruction belongs: the chain ID, the delay value, the chain head location (segment number), and the self-timed mode flag. The delay value is initialized in dispatch to $2 \times S_H + D_H$, where S_H is the chain-head segment number and D_H is the relative delay of this instruction from the chain head. Whenever the entry observes a chain-wire assertion for the specified chain, the delay value is decremented by two and the chain-head location is decremented by one. When a chain-wire assertion occurs and the chain-head location is zero, the instruction enters self-timed mode on that chain. The IQ entry also carries a flag to indicate whether the instruction is a chain head, and the ID of the chain it heads (if any).

3.4 Chain creation

Determining which instructions should become the heads of new chains is a key policy issue in our design. Creating too few chains reduces the IQ's dynamic scheduling ability, increasing the impact of dispatch-stage latency mispredictions on performance. However, chain wires are a critical resource, as will be seen in Section 6. If no free chain wires are available for a new chain head, the dispatch stage must stall; these stalls will be aggravated by an overly aggressive chain creation policy.

In this paper, we focus on latency variations induced by cache misses, so our base design creates a new chain on each load instruction. Section 4.4 describes the use of a hit/miss predictor to further conserve chain resources by starting chains only on loads that are likely to be cache misses.

For the most general variant of our design, which allows an instruction to belong to two chains, all such two-chain instructions must themselves be chain heads. Marking each two-chain instruction as a chain head prevents later instructions from needing to follow more than two chains to capture their operand data dependences.

For the design described thus far, it is most appropriate to mark as a chain head every instruction that depends on a variable-latency instruction. The chain heads will reach the bottom segment according to the producer's predicted latency, but their dependents will not advance until the producer completes and the chain heads issue. Unfortunately, this strategy requires the creation of multiple chains to tolerate a single variable-latency instruction, consuming many chains when the fan-out is large. Instead, we make the variable-latency instruction itself the head of a chain. When this instruction issues, the chain members begin to self-time. However, when it becomes apparent that the chain head will not complete within the predicted latency—e.g., when a cache miss is detected for a load—an additional signal is sent up the chain wire, causing the chain members to suspend self-timing. Once the chain head completes, a final chain-wire signal resumes self-timed mode.

4 Design enhancements

The description in the previous section provides a nearly complete picture of a functional segmented IQ with chain-based promotion. This section details a number of design enhancements which improve the performance and/or feasibility of the basic design.

4.1 Improving utilization via instruction pushdown

A potential problem with static thresholds is that they are unlikely to result in uniform segment utilization. In particular, instructions at the end of long dependence chains may reside in the top segment for many cycles before they are eligible for promotion. The top segment then fills up and stalls the dispatch stage, even when many lower segments are empty. Adaptive thresholds could improve utilization, but would be complex to implement.

Instead, we address this problem by allowing a full segment to "push down" otherwise ineligible instructions into the next lower segment if entries are available. Specifically, if a segment has less than *IW* free entries (where *IW* is the issue width) and the segment below it has more than $1.5 \times IW$ free entries, the upper segment will consider up to *IW* of its oldest non-eligible instructions as eligible for promotion. In situations where many instructions have large delay values, this policy forces some instructions down into the lower segments to make room for more newly dispatched instructions. The pushdown mechanism is designed to augment the promotion mechanism: an instruction made eligible by pushdown will never take the place of an instruction promoting as a result of the normal chain-promotion process.

4.2 Reducing pipeline depth penalties via segment bypassing

A key shortcoming of an *n*-segment IQ as described thus far is that it adds at least *n*–1 stages to the pipeline before execution, increasing the branch misprediction penalty by *n*–1 or more cycles. With this penalty, a large segmented IQ has a severe negative impact on a number of integer benchmarks (e.g., gcc). To alleviate these effects of the extended pipeline, we allow instructions to *bypass* empty queue segments at dispatch time. We observed the best performance when the dispatch stage bypasses all empty queue segments, regardless of segment thresholds and the delay values of the dispatched instructions.

The bypass wires that allow the dispatch stage to direct newly dispatched instructions into any segment are the only wires in our IQ design that span more than one segment. For this reason, we designed the bypass scheme carefully to minimize its impact on cycle time. The bypass wires are driven unidirectionally from the dispatch stage, so large drivers and repeaters can be used to optimize signal propagation. The number of loads on these wires is equal to the number of segments, not the number of IQ entries, so the load grows slowly with IQ size. Finally, because only the first sequence of empty segments is bypassed, a segment will receive instructions on a given cycle either from the dispatch stage (if it is the highest non-empty segment) or from the segment above it (if any higher segments are non-empty)—never some from each—resulting in a simple two-input mux structure at each segment whose select signal should be available well in advance of the instruction data.

4.3 Reducing IQ complexity and chain count via operand prediction

The design we have describes thus far assumes that each instruction may belong to one or two chains. Our baseline design reveals that about 35% of all instructions have two unmet dependencies produced in different chains. Dynamically following two chains provides the best scheduling, guaranteeing that an instruction does not occupy a precious slot in segment 0 before both its operands are expected to be ready. Unfortunately, providing logic in every IQ entry to track two chains, and to decide dynamically which chain should be used to determine the appropriate segment, is a potentially significant overhead. Additionally, each instruction following two chains requires the allocation of a new chain, as described in Section 3.4.

If we can accurately predict which of the two operands will be available later, we can assign the instruction to that chain alone, and simplify the IQ design by having at most one chain per processor. In Section 6, we study the impact of using a table of two-bit counters, indexed by program counter, to predict which operand (“left” or “right”) will be the critical path. A similar predictor was previously proposed by Stark et al. [21]. In addition to simplifying the IQ design, our left/right predictor reduces demand for chain wires. We will see in Section 6 that reducing the number of chains created is critical for maximizing performance with a fixed number of chain wires.

4.4 Reducing chain count using hit/miss prediction

Load instructions exhibit highly variable latencies depending on the level in the memory hierarchy that they access. Due to this variability, loads are prime candidates for chain heads; in fact, they are the primary source of chain heads considered in this paper, and account for an average of 65% of the chains in our base design. In most programs, however, most loads are cache hits, and can be scheduled with a known latency. We explore the use of a dynamic cache hit/miss predictor (HMP) [14, 25] to reduce the number of chains. We use the HMP to identify loads which have a high probability of hitting in the primary cache, and use this information to not start chains for these instructions.

In our scheme, predicting a hit reference as a miss incurs the small cost of an unnecessary chain head. On the other hand, predicting a miss reference as a hit, and not creating a new chain, will cause a potentially large number of instructions dependent on the load value to flood segment 0 well in advance of becoming ready. If segment 0 fills with non-ready instructions, performance degrades severely. As a result, we would like to predict a hit only when we have very high confidence in our prediction. We use a table of four-bit saturating counters, indexed by program counter. We increment a counter on a hit, clear it to zero on a miss, and predict a hit only if the counter is greater than 13. We show in Section 6 that this predictor achieves over 98% accuracy for hit predictions while achieving very good coverage of hits on most benchmarks.

Dependence-based prescheduling schemes that rely on accurate prediction of latencies could also benefit from a hit/miss predictor. Predicted hits that miss will have dire consequences similar to our scheme, but predicted misses that hit may still effectively suffer most or all of the miss latency if the scheduling of their dependents is delayed accordingly. Thus these schemes require high accuracy for *all* predictions, both hit and miss. In addition, these schemes must predict a specific latency—i.e., they must predict at which level in the memory hierarchy an access will hit, and cannot directly tolerate variable timing due to memory-system contention.

4.5 Deadlock recovery

The explicit scheduling dependences introduced by our segmented IQ reflect true data dependences, and thus cannot lead to scheduler deadlock. However, the resource dependence between segments—i.e., the fact that an instruction can be promoted only if the next segment has available entries—can, in rare circumstances, lead to a deadlock situation. This possibility arises because chains reflect only a subset of the dependence-graph edges. If the dispatch stage assigns a two-input instruction to the "wrong" chain—that is, the chain of the operand that becomes available earlier—then that instruction may be promoted beyond the instruction that produces its other operand. Additional instructions that depend on the incorrectly assigned instruction and are assigned to the same chain may also pass this producer; if a sufficient number do so, they may occupy all the entries of a segment below the producer. At this point the producer cannot be promoted, and deadlock occurs.

This situation is extremely rare, occurring during only 0.05% of the cycles that we simulate in Section 6. Fortunately, it is also straightforward to detect and resolve. We detect IQ deadlock when the IQ is not empty and no progress is being made; i.e., no instructions are issued or promoted from any segment, and no instructions are in execution. In this situation, we are guaranteed two things. First, there is at least one ready instruction (the oldest) that is eligible to promote to the bottom segment. Second, that instruction is not being promoted because of a lower segment that is full of instructions, none of which are eligible for promotion.

Our recovery scheme is very simple: for one cycle, we force every full segment to choose one of its ineligible instructions and promote it. This step guarantees a free entry in every segment to receive a promoted instruction. Segments with eligible instructions will promote one of those candidates. If the bottom segment is full of non-ready instructions, we recycle an instruction back to the top segment. After this cycle, we are guaranteed to have at least one eligible instruction closer to the bottom segment. Usually a single such cycle is sufficient to clear the deadlock condition; if not, the detection/recovery cycle will remain active until the deadlock is cleared. As long as eligible instructions are always promoted in preference to ineligible instructions, the oldest ready instruction is guaranteed to reach the bottom segment and issue eventually, generating forward progress.

5 Evaluation methodology

We evaluated our scheme by developing an execution-driven simulator based on the SimpleScalar toolkit [4]. Although our simulator was derived originally from SimpleScalar's *sim-outorder*, it has been largely rewritten to model a simultaneous multithreaded processor with separate instruction queue, reorder buffer, and physical register resources; a realistic pipeline depth; and a detailed event-driven memory hierarchy. The simulator executes Compaq Alpha binaries.

As in *sim-outorder*, memory reference instructions are split into an effective-address calculation, which is routed to the IQ, and a memory access, which is stored in a separate load/store queue (LSQ). The IQ schedules the effective-address calculation as an ordinary integer operation. On completion, its result is forwarded to the LSQ. The LSQ marks a memory access eligible for issue when its effective address is available and is known not to conflict with any pending memory access that precedes it in program order. Although the IQ designs modeled in this paper rely on the LSQ to enforce memory dependences, Michaud and Seznec [15] illustrate how a similar scheme can be augmented to enforce predicted memory dependences using store sets [7].

Processor parameters are listed in Table 1. Because we focus in this paper on the execution variability introduced by caches, we use a generous supply of function units to reduce variability due to resource constraints. For the same reason, we configure the ROB to be three times the size of the IQ. To account for added complexity, we add an extra cycle to the dispatch stage for both the segmented and prescheduling IQs.

Table 1: Processor parameters

Parameter	Value
Front-end pipeline depth	10 cycles fetch-to-decode, 5 cycles decode-to-dispatch
Fetch bandwidth	Up to 8 instructions per cycle; max 3 branches per cycle
Branch predictor	Hybrid local/global (a la 21264); global: 13-bit history reg, 8K-entry PHT local: 2K 11-bit history regs, 2K-entry PHT choice: 13-bit global history reg, 8K-entry PHT
Branch target buffer	4K entries, 4-way set associative
Dispatch/issue/ commit bandwidth	Up to 8 instructions per cycle
Function units	8 each: integer ALU, integer mul, FP add/sub, FP mul/div/sqrt, data-cache rd/wr port
Latencies	integer: mul 3, div 20, all others 1 FP: add/sub 2, mul 4, div 12, sqrt 24 all operations fully pipelined except divide & sqrt
L1 split I/D caches	Both: 64 KB, 2-way set associative, 64-byte lines Inst: 1-cycle latency (to simplify fetch unit) Data: 3-cycle latency, up to 32 outstanding misses
L2 unified cache	1 MB, 4-way set associative, 64-byte lines, 10-cycle latency, up to 32 outstanding misses, 64 bytes/cycle bandwidth to/from L1 caches
Main memory	100-cycle latency, 8 bytes/CPU cycle bandwidth

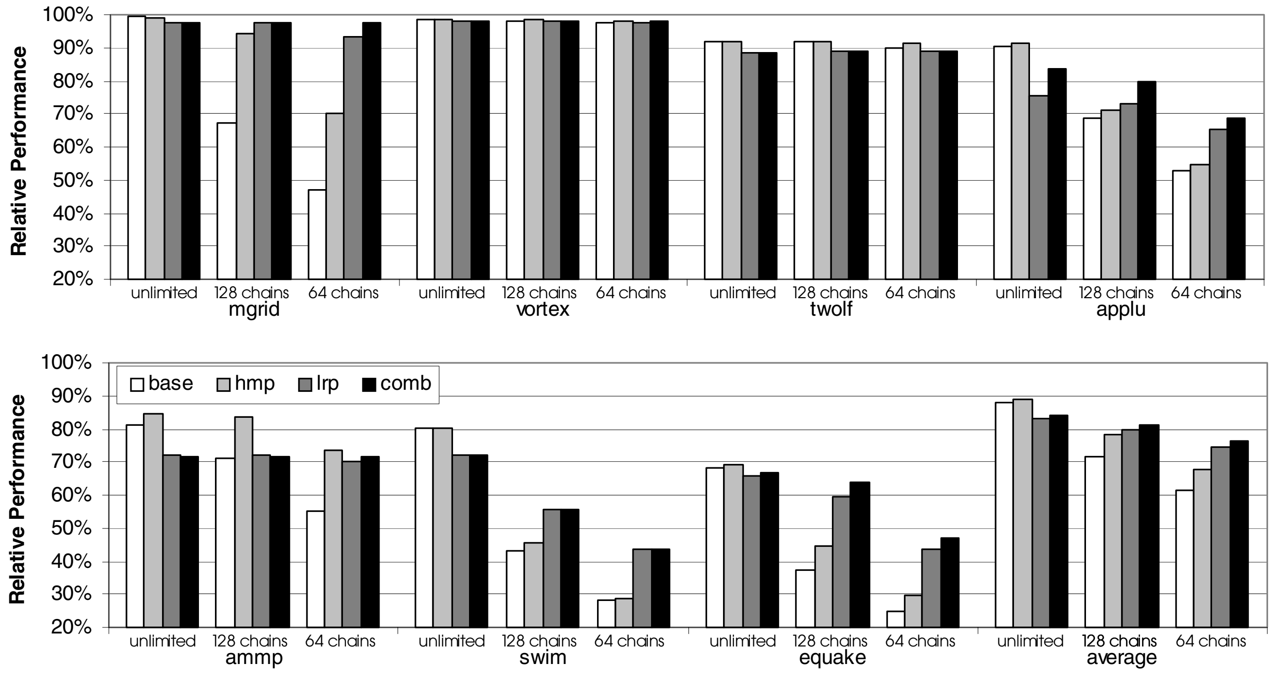


Figure 2. Performance of 512-entry segmented IQ configurations relative to ideal 512-entry IQ. Labels below the bars indicate the maximum number of chains available. "Comb" indicates a configuration using both the hit/miss predictor (HMP) and left/right predictor (LRP).

We use a subset of the SPEC CPU2000 benchmarks for our study. In all our studies, we start from a checkpoint 20 billion instructions into the benchmark's execution and simulate a sample of 100 million instructions. We compiled all of the CPU2000 benchmarks using Compaq's GEM compiler with full optimizations and simulated them using a range of IQ sizes. We then selected the two integer benchmarks (twolf and vortex) and five floating point benchmarks (ammp, applu, equake, mgrid, and swim) that show the greatest performance improvement as IQ size is increased. The FP benchmarks show the largest speedups: L2 cache misses limit their performance, and a large IQ (coupled with high branch-prediction accuracies) allows them to overlap large numbers of main-memory accesses. We also simulate gcc, which does not benefit from a larger IQ, to calibrate the impact of our design on applications with a high misspeculation rate and low ILP. The behavior of other benchmarks that do not benefit from a large instruction queue is similar to that of gcc.

6 Experimental results

We begin by comparing the performance of a 512-entry segmented IQ composed of sixteen 32-entry segments with that of an ideal, monolithic, single-cycle 512-entry conventional IQ. Section 6.1 discusses the segmented IQ's performance using an unlimited number of chains, and examines the impact of adding a hit/miss predictor (HMP) and a left/right operand predictor (LRP) on performance and chain count. Section 6.2 repeats this analysis using realistic segmented IQs with finite chain resources. Finally, Section 6.3 examines the performance of realistic segmented IQs across a variety of IQ sizes, and compares their performance with our implementation of Michaud and Seznec's prescheduling scheme [15].

6.1 Segmented IQ with unlimited chains

Figure 2 plots the performance of several benchmarks using a 512-entry segmented IQ relative to their performance with an ideal single-cycle IQ of the same size. For space reasons, we omit gcc, whose behavior in this portion of the study is uninteresting (much like vortex). In this section, we focus on the first cluster of four bars for each benchmark, which indicate performance assuming an unlimited number of chain wires. The bars within the group correspond to four configurations. The first, labeled *base*, creates a new chain on every load and on every instruction with two outstanding input operands. The latter instructions are dynamically associated with two chains.

Examining the average results for the base configuration, we see that the segmented IQ's performance is within 16% of the ideal IQ. This performance gap is due to the segmented IQ's additional pipeline stages and its inability to issue instructions from all slots in the queue. Mgrid achieves the best relative performance, at 99.4% of the ideal. Our chain-based scheduling is very effective for mgrid: on average, the 32 entries in segment zero hold 16 ready instructions, representing more than 25% of all the ready instructions in the entire IQ.

Vortex and twolf also perform well because they actively use only a small fraction of the queue (no more than 136 out of 512 entries). The bypass mechanism moves the majority of their instructions past the top 8 queue segments, drastically reducing the impact of the pipeline delay. The lower IQ occupancy also means that a smaller fraction of all instructions wait in the upper queue segments; for both benchmarks, more than 33% of ready instructions reside in segment zero.

Unfortunately, most benchmarks require an excessive number of chains to achieve this performance. The first two columns of Table 2 show the measured average and peak chain counts for this unlimited-chains model. The peak chain usage numbers can be larger than the IQ size because we do not deallocate chains until the chain head instruction has written its result back to the register file.

For the segmented queue to be viable, we must reduce the required number of chains significantly. A hit/miss-predictor (HMP) avoids creating new chains for loads which are predicted to be L1 cache hits, as discussed in Section 4.4. A left/right operand predictor (LRP), discussed in Section 4.3, avoids creating chains on instructions with two outsting operands, and also simplifies the IQ by restricting each instruction to a maximum of one chain.

The second bar for each benchmark in Figure 2 shows the relative performance of the segmented IQ with the hit/miss predictor (HMP). Our predictor has a prediction accuracy of over 98%, predicting over 83% of all cache hits. Referring to Table 2, we see that the HMP reduces the average number of chains by 33%. The maximum savings is limited by the cache hit rate; swim sees only a negligible decrease in chains because over 90% of its loads miss in the L1 cache. (Only 20% of these misses cause L2 accesses; the remainder are “delayed hits”, where a load references a block which is in the process of being fetched.) Figure 2 shows that the HMP actually improves performance slightly. We believe this effect occurs because the delay counter values assigned by the dispatch stage do not compensate for the latencies of pipelining the chain promotion wires; thus giving a chain a small head start by using the hit latency for a delayed-hit access may allow some dependent instructions to issue sooner.

The LRP eliminates all multiple-chain instructions, reducing the number of chains and simplifying the IQ implementation. As with the HMP, however, an LRP misprediction will cause the mispredicted instruction, and its dependents on the same chain, to enter segment zero before all its operands are ready. In fact, an instruction may enter segment zero before the producer of its second operand, leading to potential deadlock as discussed in Section 4.5. Even if deadlock does not occur, the additional unissuable instructions in segment zero can block ready instructions from entering. Again referring to Table 2, we see that use of the LRP reduces the average number of chains required by 58%. Figure 2 shows that, unlike the HMP, LRP mispredictions do cause noticeable performance losses in several benchmarks, particularly ammp and applu.

Since the HMP and LRP address different sources of chain creation, their combination produces an even greater reduction in chain count: an average of 67% fewer than the base configuration. The performance effects are also mostly additive; performance with both predictors is at or slightly above the performance of LRP alone.

6.2 Evaluation of realistic queues

Of course, a real-world segmented IQ must be constructed using a finite number of chains. An eight-wide processor with a conventional 512-entry IQ using CAM-based wakeup logic would require $8 \times \log_2 512 = 72$ tag lines. To keep the wiring area comparable, we constrain the number of chain wires in our segmented IQ to a similar range. Specifically, we examine configurations of 64 and 128 chain wires. In these configurations, the dispatch stage will stall when it tries to dispatch a chain-head instruction but no

Table 2: Chain usage for 512-entry segmented IQ with unlimited chains

Benchmark	Baseline		HMP		LRP		Combined	
	Average	Peak	Average	Peak	Average	Peak	Average	Peak
AMMP	143	453	82	352	64	221	49	214
APPLU	294	661	202	646	119	360	99	358
EQUAKE	414	620	313	568	177	342	129	329
GCC	22	379	20	367	18	253	17	248
MGRID	389	577	139	577	102	246	52	246
SWIM	305	522	292	526	150	268	148	272
TWOLF	47	357	37	327	33	279	27	276
VORTEX	64	293	36	262	47	212	33	150
Average	210	483	140	453	89	273	69	261

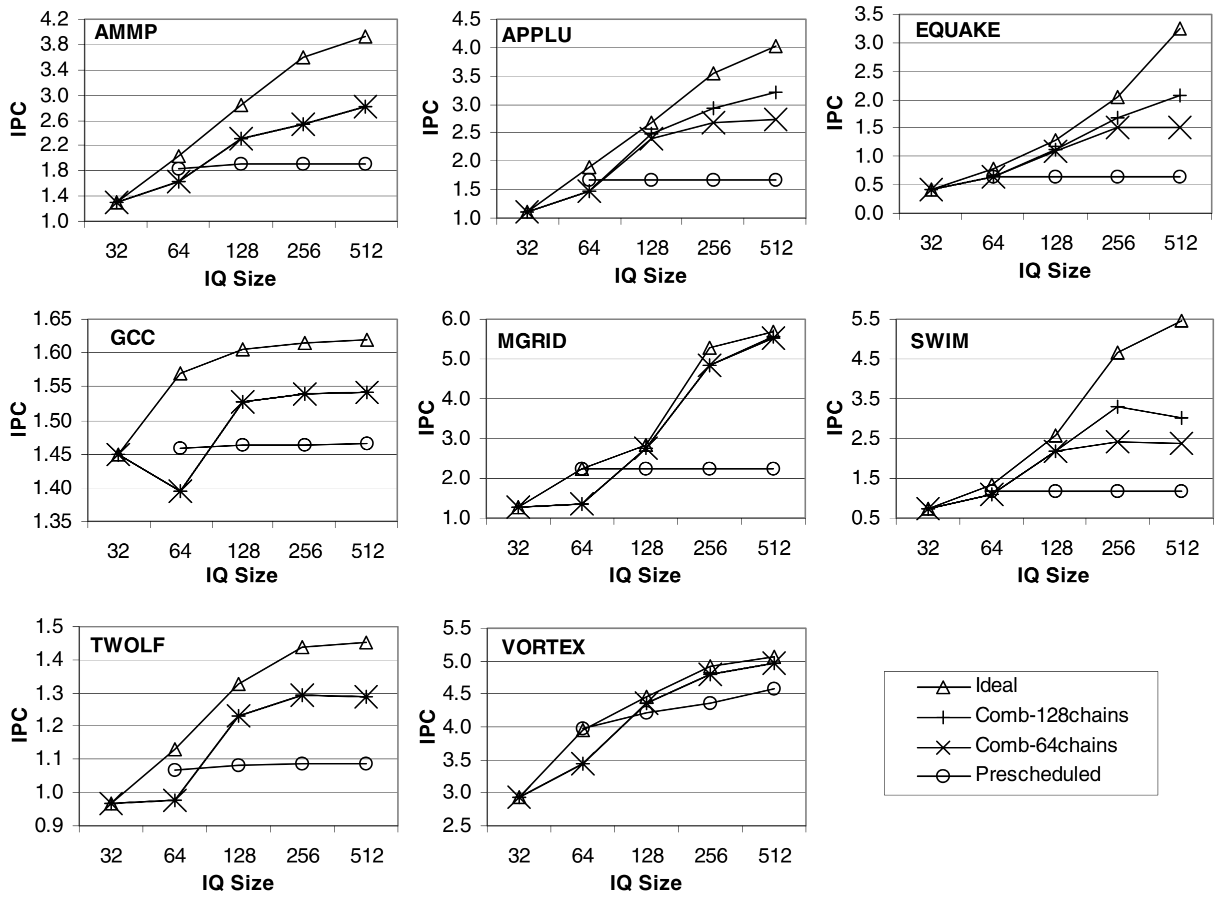


Figure 3. Performance of all benchmarks for varying queue sizes and configurations. The datapoints for the Prescheduled curves represent queue structures totaling 128, 320, 704, and 1472 instructions.

free chains are available. Table 2 indicates that these values should cover the average, though not the peak, chain demand when both an HMP and an LRP are used.

The second and third groups of bars in Figure 2 indicate the performance of a segmented IQ as the number of chains is fixed at 128 and 64, respectively. The first bar in each group, representing the performance without HMP or LRP, shows the importance of having a sufficient number of chains. On average, the 128-chain queues posted an additional 17% performance reduction over the unlimited-chains queue model (29% lower performance than the ideal queue), and the 64-chain queues posted a 27% reduction compared to the unlimited-chains model (39% lower than the ideal queue). Among the benchmarks, those requiring the fewest chains (vortex and twolf) suffered less than those requiring more chains (mgrid, equake and swim).

Adding the HMP reduces the chains required for loads, providing a significant performance improvement: an average 9% for 128 chains and 10% for 64 chains. As in the previous section, benchmarks with large cache miss rates (e.g., swim) do not benefit much from the HMP. Those with low chain usage (vortex and twolf) do not often run out of chains thus do not benefit much either. Mgrid and ammp, which have fairly high chain usage and queue occupancy rates, but low cache-miss rates, benefit the most.

Using the LRP to reduce chain usage generally works well also. With the exception of ammp and twolf, the performance decreases seen in the unlimited-chains configuration are more than compensated for by the reduction in dispatch stalls due to lower chain usage.

As in the previous section, using the HMP and the LRP together generally provides additive benefits. The key exceptions are ammp and twolf, where HMP cannot address the performance loss due to LRP mispredictions.

6.3 Performance across multiple IQ sizes

In this section, we examine the benefits of our segmented IQ structure across a range of IQ sizes. Figure 3 presents performance results for IQs with 32 to 512 entries. The top line in each graph shows the performance of an ideal, single-cycle instruction queue at that size. We also plot the performance of our segmented IQ using both the HMP and LRP with 64 and 128 chains, assuming 32-entry segments. At an IQ size of 32 entries, our scheme degenerates to a single segment, and is thus equivalent to the conventional IQ. As IQ size increases, all three IQs show improved performance. The benefit tapers off quickly for gcc, and to some extent for twolf, due to branch mispredictions. The remaining benchmarks exhibit significant performance gains out to 512 entries.

Although the segmented queues generally show continued performance improvements for larger queues, the rate of improvement is less than that of the ideal queue. Gcc shows a 0.05 IPC drop in performance between 32 and 64 entries, due largely to the benchmark's sensitivity to pipeline depth and the fact that very little useful scheduling can be done in just two queue segments. Both swim and twolf show some reduction in performance when going from 256 to 512 entries. Neither of these benchmarks can make much use out of the additional queue slots, yet they suffer from the increased pipeline depth and reduced predictor accuracies that result. Equake, swim, and applu also suffer at larger queue sizes from the limited number of chains in the 64-chain IQ.

However, since the cycle time of our segmented IQ design is determined by the complexity of the individual 32-entry segments, we expect cycle times to be fairly constant across the range of sizes. In contrast, the cycle time of the ideal queue would be expected to grow quadratically with its size [17]. In fact, the complexity of a 32-entry segment zero is similar to that of a 32-entry conventional IQ; thus the performance gains of the segmented IQ over the ideal 32-entry queue can be viewed as the improvement made possible by adding queue segments.

We also compare our segmented queue design to Michaud and Seznec's prescheduling scheme [15]. Michaud and Seznec indicate that their prescheduling scheme outperforms Palacharla et al.'s FIFOs [15], while Canal and González indicate that their deterministic-latency scheme outperforms their distance scheme [6]. We believe that the performance of the prescheduling and distance schemes would be similar due to their structural similarity.

We implemented a similar prescheduling IQ in our simulator framework to provide a direct comparison with our segmented IQ. Unlike Michaud and Seznec's design, we continue to use a separate LSQ to manage memory dependences. Our prescheduling IQ is configured as suggested by the authors for best performance. It uses a 32-entry issue buffer (similar to our segment zero) and twelve instructions per line in the prescheduling array. Because the entries in the prescheduling array are much simpler than our IQ segment entries, we allot roughly three prescheduling-array entries for each additional segmented IQ entry; thus the four data points for the prescheduling scheme correspond to a 32-entry issue buffer plus 8, 24, 56, or 120 lines of 12 instructions (totaling 128, 320, 704, or 1472 total instruction slots).

For all benchmarks, the 128-entry prescheduling scheme performs better than the 64-entry segmented IQ. However, vortex is the only benchmark which shows any appreciable improvement as the size of the prescheduling array is increased. Our 128-entry segmented IQ outperforms any prescheduling-array size for every other benchmark. For vortex, our 256-entry IQ outperforms all prescheduling configurations. In general, the performance gap widens as IQ size is increased.

7 Future work

Power consumption is a significant concern in modern architectures; instruction-queue power consumption is particularly significant, as it already constitutes a sizable fraction of the power budget in high-performance processors [11]. Copying an instruction from segment to segment consumes more dynamic power than keeping the instruction in a single storage location between dispatch and issue; whether the performance benefit of the segmented IQ justifies this power consumption will depend on the detailed design and the target market. In any case, the segmented structure lends itself naturally to dynamic resizing by gating clocks and/or power on a segment granularity, based on power constraints or power/performance trade-offs [2, 9]. Individual segments are also amenable to power optimizations proposed for conventional IQ structures [9].

Another area of future work involves investigating the performance of segmented IQs under simultaneous multithreading (SMT) [23]. By scheduling across multiple threads, an SMT processor may obtain even larger benefits out of increased IQ sizes. Unlike other prescheduling schemes, the dynamic inter-chain scheduling of our segmented IQ should allow chains from independent threads to exploit thread-level parallelism effectively.

Finally, as mentioned in Section 2, we believe that future large IQs will employ both vertical segmentation, as we have proposed, and horizontal clustering, as in the Alpha 21264 [14]. There may be exploitable synergies between our chain-based scheme and a clustered approach. For example, chains seem to form a natural unit for assignment to function-unit clusters, and such an assignment may allow a more distributed our hierarchical broadcast of chain head promotion signals.

8 Conclusions

Two key paths to higher performance—larger instruction windows and lower cycle times—conflict directly in conventional instruction-queue designs. An associative search of a large structure to identify issuable instructions results in an inherently large cycle time. Previous work has shown that this trade-off may be overcome by constraining the search for issuable instructions to a likely subset of the instruction window, identified using data dependence information. Unfortunately, opportunities for instruction-level parallelism can easily be lost in the process of constraining this search. In particular, the use of predicted latencies for data-dependence scheduling can hamper the processor's ability to tolerate unpredictable latencies—one of the key benefits of dynamic scheduling.

This paper presents a novel instruction queue design that provides both quasi-static data-dependence scheduling to limit the scope of wakeup logic and flexible dynamic scheduling in the face of unpredictable latencies. We accomplish this combination by grouping instructions into *chains* representing subtrees of the dynamic data dependence graph. Instructions within a chain are scheduled quasi-statically based on predicted latencies; however, scheduling across chains is fully dynamic and can tolerate unpredictable latencies. We use these chains to manage the flow of instructions in a *segmented* instruction queue, effectively a pipeline of small structures similar to a conventional IQ. The cycle time of the segmented IQ is determined by the size of each segment, not the overall queue size. Our design can be scaled across varying window sizes and clock frequencies by varying the number of segments and the number of instruction slots per segment.

We also identify and evaluate a number of enhancements to the segmented IQ design, including a bypassing mechanism to reduce the pipeline depth penalty, and hit/miss and left/right operand predictors to reduce the number of chain wires needed and the complexity of IQ entries.

Despite its roughly similar circuit complexity, simulation results indicate that our segmented instruction queue with 512 entries and 128 chains improves performance by up to 69% over a 32-entry conventional instruction queue for SpecINT 2000 benchmarks, and up to 398% for SpecFP 2000 benchmarks. The segmented IQ achieves from 55% to 98% of the performance of a monolithic 512-entry queue while providing the potential for much higher clock speeds.

References

[1] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate vs. IPC: The end of the road for conventional microarchitectures. In *Proc. 27th Int'l Symp. on Computer Architecture*, pp. 248–259, June 2000.

[2] David H. Albonesi. Dynamic IPC/clock rate optimization. In *Proc. 25th Int'l Symp. on Computer Architecture*, pp. 282–292, June 1998.

[3] Mary D. Brown, Jared Stark, and Yale N. Patt. Select-free instruction logic. In *34th Int'l Symp. on Microarchitecture*, pp. 204–213, December 2001.

[4] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating future microprocessors: the SimpleScalar tool set. Technical Report 1308, Computer Sciences Department, University of Wisconsin–Madison, July 1996.

[5] Ramon Canal and Antonio González. A low-complexity issue logic. In *Proc. 2000 Int'l Conf. on Supercomputing*, pp. 327–335, May 2000.

[6] Ramon Canal and Antonio González. Reducing the complexity of the issue logic. In *Proc. 2001 Int'l Conf. on Supercomputing*, June 2001.

[7] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *Proc. 25th Int'l Symp. on Computer Architecture*, pp. 142–153, June 1998.

[8] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *30th Int'l Symp. on Microarchitecture*, pp. 149–159, December 1997.

[9] Daniele Folegnani and Antonio González. Energy-effective issue logic. In *Proc. 28th Int'l Symp. on Computer Architecture*, July 2001.

[10] M. Goshima, K. Nishino, Y. Nakashima, S. Mori, T. Kitamura, and S. Tomita. A high-speed dynamic instruction scheduling scheme for superscalar processors. In *34th Int'l Symp. on Microarchitecture*, pp. 225–236, December 2001.

[11] Michael K. Gowan, Larry L. Biro, and Daniel B. Jackson. Power considerations in the design of the alpha 21264 microprocessor. In *Proc. 35th Design Automation Conf.*, pp. 726–731, June 1998.

[12] Ron Ho, Kenneth W. Mai, and Mark A. Horowitz. The future of wires. *Proc. IEEE*, 89(4):490–504, April 2001.

[13] Gregory A. Kemp and Manoj Franklin. PEWs: A decentralized dynamic scheduler for ILP processing. In *Proc. 1996 Int'l Conf. on Parallel Processing (Vol. 1)*, pp. 239–246, 1996.

[14] R. E. Kessler. The Alpha 21264 microprocesor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[15] Pierre Michaud and André Seznec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *Proc. 7th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pp. 27–36, January 2001.

[16] Soner Önder and Rajiv Gupta. Superscalar execution with dynamic data forwarding. In *Proc. 1998 Conf. on Parallel Architectures and Compilation Techniques*, pp. 130–135, October 1998.

[17] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proc. 24th Int'l Symp. on Computer Architecture*, pp. 206–218, June 1997.

[18] Yale N. Patt, Sanjay J. Patel, Marius Evers, Daniel H. Friendly, and Jared Stark. One billion transistors, one uniprocessor, one chip. *IEEE Computer*, 30(9):51–57, September 1997.

[19] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. Trace processors. In *30th Int'l Symp. on Microarchitecture*, pp. 138–148, December 1997.

[20] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. 22nd Int'l Symp. on Computer Architecture*, pp. 414–425, June 1995.

[21] Jared Stark, Mary D. Brown, and Yale N. Patt. On pipelining dynamic instruction scheduling logic. In *33rd Int'l Symp. on Microarchitecture*, pp. 57–66, December 2000.

[22] Dennis Sylvester and Kurt Keutzer. Rethinking deep-submicron circuit design. *IEEE Computer*, 32(11):25–33, November 1999.

[23] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. 22nd Int'l Symp. on Computer Architecture*, pp. 392–403, June 1995.

[24] Shlomo Weiss and James E. Smith. Instruction issue logic in pipelined supercomputers. *IEEE Trans. Computers*, C-33(11):1013–1022, November 1984.

[25] Adi Yoaz, Mattan Erez, Ronny Ronen, and Stephan Jourdan. Speculation techniques for improving load related scheduling. In *Proc. 26th Int'l Symp. on Computer Architecture*, pp. 42–53, May 1999.