

Exploring Wakeup-Free Instruction Scheduling

Jie S. Hu, N. Vijaykrishnan, and Mary Jane Irwin
Microsystems Design Lab
The Pennsylvania State University
University Park, PA 16802, USA
{jhu,vijay,mji}@cse.psu.edu

Abstract

Design of wakeup-free issue queues is becoming desirable due to the increasing complexity associated with broadcast-based instruction wakeup. The effectiveness of most wakeup-free issue queue designs is critically based on their success in predicting the issue latency of an instruction accurately. Consequently, the goal of this paper is to explore the predictability of instruction issue latency under different design constraints and to identify the impediments to performance in such wakeup-free architectures. Our results indicate that structural problems in promoting instructions to the head of the instruction queue from where they are issued in wakeup-free architectures, the limited number of candidate instructions that can be considered for instruction issue, and the resource conflicts due to non-availability of issue ports all have a significant impact in degrading the performance of broadcast free architectures. Based on these observations, we explore an architecture that attempts to overcome the structural limitations by employing traditional selection logic and by using pre-check logic to reduce the impact of resource conflicts while still employing a wakeup-free strategy based on predicted instruction issue latencies. Finally, we improve this technique by limiting the selection logic to a small segment of the issue queue.

1. Introduction

The instruction issue queue is a critical part of a superscalar processor that dynamically schedules instructions for execution. As the quality of the generated instruction schedule is critical to the performance of the processor, the issue queue has been the target of various research efforts [12][4][11][13]. As larger issue queues help to extract more ILP, the trend has been to employ increasingly larger issue queues for improved performance.

However, the complexity of the traditional centralized issue queue design that uses broadcast for waking-up instructions and global selection becomes the limiter to clock frequency as the issue queue size and width increase [12].

Specifically, the relative delay of the wakeup logic increases exponentially when both issue queue size and issue width increase. Consequently, the potential benefits of using a larger issue queue diminish due to its long latency in future technologies [9]. There are two main approaches to address this problem. One is to convert the dynamic scheduling into static compile time scheduling as in VLIW (Very Large Instruction Word) or EPIC (Explicitly Parallel Instruction Computing) architectures. In these architectures, the issue queue is relieved from the scheduling task and acts simply as an instruction buffer. However, the quality of the compile time scheduling can be limited by several factors. Normally, the size of the static instruction window considered for scheduling is limited to basic block, superblock, or hyperblock. Without dynamic (runtime) information, the compiler can make inaccurate assumptions about the branch instructions and load/store instructions that degrade the quality of the scheduling.

Another direction of research focuses on designing complexity-effective issue queues. These techniques attempt to reduce the complexity of the wakeup and selection logic employed in the issue queue. Techniques include adapting the size of the issue queue, the use of segmented issue queues that limit the broadcast and selection to a smaller segment [13], banking of selection logic [8], select-free logic for removing selection from the critical path [2], speculative wakeup based on availability of “grandparent” operands [14], design of dependency chain based ordering within the issue queue [12][11][4][5], the use of secondary schedulers for long latency operations [10][1], and support for broadcast-free issue [6]. Among these techniques, wakeup-free issue is a very recent technique that predicts the issue latency of each instruction at the pre-scheduling stage to avoid wakeup.

The goal of this paper is to explore the predictability of instruction issue latency under different design constraints and to identify the impediments to performance in such wakeup-free architectures. Consequently, the wakeup-free issue queue design (Cyclone) proposed in [6] serves as the starting point of our exploration. Our study of the Cyclone architecture shows that the conflicts arising during instruction flow because of the queue structure and the limitation

that only instructions at the head of the queue can be candidates for issuing are the two main reasons for performance loss. We propose a more flexible model for predictive issue queue design that introduces traditional selection logic to eliminate some of the structural constraints imposed by Cyclone. The evaluation using this model is performed to identify inherent limitations imposed on performance by inaccuracies on the instruction issue latency prediction. Our study indicates that resource (issue ports) conflicts have a significant impact on the prediction accuracy and consequently the performance in wakeup-free architectures. Next, an optimization to minimize this resource conflict is presented. Our experimental evaluation shows that by applying this technique, the wakeup-free architecture can achieve a performance close to that of the broadcast-based architecture. Finally, we present a segmented wakeup-free issue queue to optimize the selection logic used in our generalized model for predictive issue. By avoiding the need for random selection, the segmented issue queue dedicates a specific segment as the instruction pool for selection and issue.

The rest of this paper is organized as follows. We present the experimental setup in Section 2. Section 3 investigates the potential causes of performance loss in a wakeup-free scheduler. In Section 4, we present a general model for wakeup-free architecture and use it to analyze the impact of resource conflicts on prediction accuracy and performance. A technique called Pre-check to eliminate potential resource conflicts in predictive issue queue designs is proposed in Section 5. Section 6 evaluates the impact of load instruction related predictions on the predictability of issue latency. In section 7, a segmented issue queue design to reduce the complexity and delay of the selection logic is presented. The related work is discussed in Section 8. Section 9 concludes this paper.

2. Experimental Setup

We use SimpleScalar/Pisa version 3.0 tool set [3] to implement a contemporary microprocessor similar to Mips R10000 [15] as the conventional wakeup-based architecture Base. The processor and memory hierarchy configuration for this Base architecture is given in Table 1. Our proposed wakeup-free issue queue models are derived from this Base architecture that utilizes separated issue queue and reorder buffer structures. We use a set of eight integer and eight floating point applications from the SPEC2000 benchmark suite and use their PISA binaries and reference inputs for execution. Each benchmark is first fast-forwarded half a billion instructions, and then simulated the next half a billion committed instructions.

3. Performance Bottleneck in A Wakeup-Free Instruction Scheduler

This section analyzes the performance bottleneck in wakeup-free instruction schedulers using the Cyclone ar-

Processor Datapath	
Issue Queue	32 entries
Load/Store Queue	16 entries
Reorder Buffer	128 entries
Fetch/Decode/Commit Width	4 instructions per cycle
Issue Width	4-8 instructions per cycle
Function Units	4-8 IALU, 2 IMULT/IDIV, 4-8 FALU, 2 FMULT/FDIV, 2-4 Memports
Branch Predictor	Bimodal, 2048 entries, 512-set 4-way BTB, 8-entry RAS
Memory Hierarchy	
L1 ICache	32KB, 1 way, 32B blocks, 1 cycle latency
L1 DCache	32KB, 4 ways, 32B blocks, 2 cycle latency
L2 UCache	256KB, 4 ways, 64B blocks, 8 cycle latency
Memory	80 cycles first chunk, 8 cycles rest
TLB	4 way, ITLB 64 entry, DTLB 128 entry, 30 cycle miss penalty

Table 1. Parameters for the simulated processor and its memory hierarchy.

chitecture [6] (See Figure 1) as the basis. The basic idea employed by Cyclone is as follows: the issue latency is predicted for each instruction at the pre-scheduling stage according to a timing table. This table indicates the delay after which the source operands will be ready. Next, the instruction is put into the tail of the countdown queue along with an associated delay value equal to half of the predicted latency. This delay value is decreased every cycle and once an instruction reaches a delay value of zero in the countdown queue, it competes to switch to its corresponding position in the main queue. Once in the main queue, the instruction progresses towards column 0 (the only place from which instructions can be issued) one position each cycle. In this architecture, the instruction is automatically considered for issue when it reaches column 0 and involves no broadcast for wakeup. However, the simplicity of this architecture also imposes several constraints that affect performance. We focus on analyzing these constraints.

The main queue is a critical resource in the Cyclone architecture. In a given cycle where instruction switches occur, entries in column i (top) and column $i - 1$ (right-top) in the countdown queue and the entry in column $i + 1$ (left) in the main queue compete for the corresponding entry in column i in the main queue. Such a competition delays the issue time of certain instructions since Cyclone gives the priority to the entry in the left column in the main queue, then to the top column in the countdown queue, and last to the right-top column in the countdown queue. The effectiveness of this architecture depends on how well this resource conflict (competition in the main queue) can be minimized such that instructions can be issued after its predicted latency decreases to zero. A quantitative analysis of the resource conflicts in the main queue of Cyclone architecture was performed in order to gain insight into the IPC loss as compared to a broadcast style architecture.

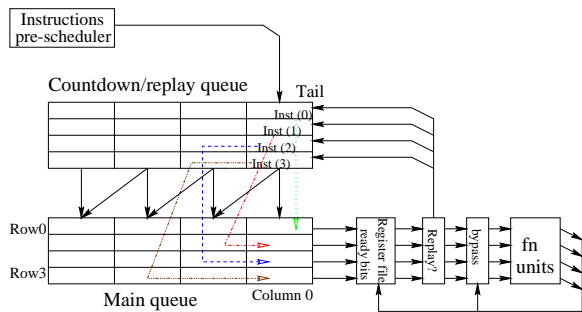
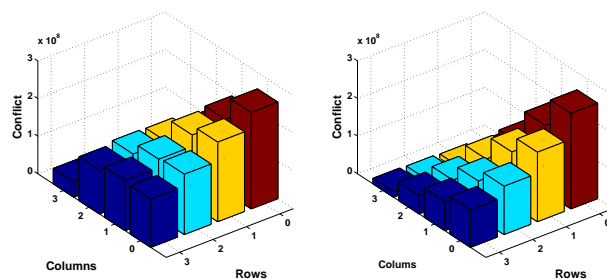


Figure 1. Cyclone instruction scheduler architecture from [6]. The non-solid lines show the promotion paths when applying the optimized instruction placement strategy.

We modeled the Cyclone architecture as shown in Figure 1, in which both the countdown/replay queue and the main queue have four rows and four columns that provides a 32-entry issue queue. The fetch width and issue width are fixed at four instructions per cycle. Figure 2 shows the resource conflict at each entry in the main queue during the scheduling, an average for SPEC2000 floating-point benchmarks and integer benchmarks used in this paper, respectively. In each chart, the floor map is exactly the same as the layout of the main queue in Figure 1. These two charts clearly display that resource conflict increases dramatically when moving from column 3 towards column 0 as well as moving from a lower row (row 3) to a higher row (row 0). The entry at the intersection of column 0 and row 0 incurs the highest conflict.

Intense conflicts in the first two columns of the main queue are detrimental to performance as they delay the predictively ready instructions from issuing. In order to overcome this problem, we proposed an ordered instruction placement strategy according to the predicted latency at the tail of the countdown/replay queue to reduce the conflict in the higher rows and columns in the main queue. This optimized instruction placement enables instructions with different latencies to enter and promote through different rows in the countdown/replay queue and main queue such that the conflict between those instructions can be largely avoided (as the non-solid lines shown in Figure 1). Our evaluation shows that the intensive conflict in higher rows in the main queue is relieved, though the conflict pressure is transferred to lower rows and left columns in the main queue. These conflicts still occur as there is no uniformity in the distribution of the predicted issue latencies. Consequently, the structure of instruction forwarding is a critical bottleneck.

Another main performance bottleneck in the Cyclone architecture is the limited small number of instructions that can be considered for issue. Only instructions in column 0 in the main queue can be selected for issue and the column size is equal to the issue width. If instructions with mis-



(a). An average for FP benchmarks.

(b). An average for INT benchmarks.

Figure 2. Conflict in the main queue of Cyclone.

predicted latency had reached column 0 too early, several problems arise. First, they use and waste issue ports. Second, they prevent some actually ready instructions from issuing. Third, they compete with the newly decoded instructions in the countdown/replay queue since they need to be replayed. However, the issue logic has no choice but to issue these instructions. This limitation is imposed due to the simple selection constraints employed.

4. A General Model and Resource Conflict

The analysis of the Cyclone architecture implies that the conflict in the main queue due to instruction promotion and the small number of candidate instructions for selection are the performance bottlenecks in wakeup-free scheduling. For a high-performance wakeup-free instruction scheduler, it is critical to understand how well the instruction issue latency can be predicted and what are the factors that prevent a high-performance and accurate predictive scheduling. In this section, we propose a general wakeup-free issue queue design that eliminates the performance bottleneck in the Cyclone architecture.

4.1. WF-Replay: A General Model

Our general model, WF-Replay, for a wakeup-free issue queue architecture is presented in Figure 3. The issue queue uses a collapsing scheme and new instructions are added at the end of the queue. Each entry in the issue queue is augmented with a latency counter. The instructions in the issue queue do not move when their latency counters are decreasing. Hence, this eliminates the resource competition possible due to instruction forwarding. This necessitates selection logic that supports random selection among any of the entries of the issue queue that have their latency counters set to zero. Since the whole issue queue serves as the instruction pool for selection, this eliminates the problem of limited instructions for selection. This model helps in

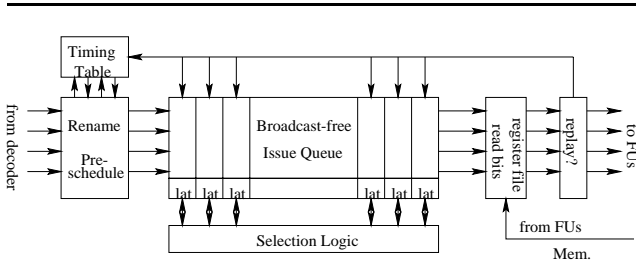


Figure 3. A general model WF-Replay for wakeup-free issue queue supporting replay.

studying the limitations in predicting instruction issue latencies without being constrained by these structural difficulties. We will focus on reducing the complexity of the selection logic in later sections.

Instructions from the decoder predict their issue latencies at the renaming and pre-scheduling stage using a scheme similar to that in [6]. Each physical register is associated with a latency indicating when the register value (result) will be ready, and a set ready bit indicating the availability of the value. These latency values are stored in a separate timing table that is indexed by the physical register number. The ready bits are stored in a special register, ready bit register (RBR), for bit accessing. The renaming table has a copy of the latency and ready bit values for each logical register that is mapped to a physical register. During renaming, the latency values are checked for each source operand. The issue latency of the instruction is computed as the maximum value of the ready latencies of its operands. At the same time, the ready latency of the resulting register (a renamed physical register) is then computed as the sum of the issue latency and the operation latency. This new latency is then updated to the timing table and rename table. When the instruction is dispatched into the issue queue, the latency counter of the issue queue entry is set to the predicted issue latency of that instruction, and decreases by one every cycle. Once the latency counter reaches zero that implies the instruction is predictively ready for issue and the instruction sends a request to the selection logic to compete for the issue port and the function unit.

Instructions are not immediately removed after their issue. Before the instruction starts to execute on an assigned function unit, it first checks whether all its source operands are ready. This task can be performed by checking the corresponding bits in the special ready bit register. If all source operands are ready, the execution proceeds and a feedback is sent to the issue queue to remove the instruction from the issue queue. Otherwise, a replay signal is sent back to the issue queue indicating that this instruction needs to be re-issued. The issue latency of this instruction is re-computed with the new status of the time table. The assigned function unit is idle for one cycle.

Note that the operation latency of instructions except for load instructions can be determined at pre-scheduling (issue latency prediction) stage. ALU instructions have a fixed operational latency, while control instructions do not produce a result so do not need operation latency prediction. However, the issue latency and operation latency prediction for load instructions are much more complicated. Load instruction cannot be issued before the load/store dependence on a previous store is resolved. The latency of a load instruction cannot be determined even after its issue due to cache hit/misses. Here, for our general wakeup-free issue queue model, we use a perfect load hit/miss predictor and a perfect load/store dependence predictor at the pre-scheduling stage; i.e., the predictors have perfect information only at the pre-scheduling stage that may change during later stages.

4.2. Exploring the Performance Constraints in the General Model

Provided with perfect predictors at the pre-scheduling stage, should the general model WF-Replay work perfectly and achieve a performance close to the conventional wakeup-based issue queue? If *not*, what are the hindrances limiting performance in a wakeup-free issue queue architecture? Recall that in our general model, there is no resource conflict within the issue queue and the selection logic is liberated to consider random instructions in the issue queue. Will the relative issue width to the fetch/decode width play some hidden role?

The width of instruction fetch and decode in the general model is fixed at four instructions per cycle. Given a common four-issue width, the performance comparison between our general model (WF-Replay) and a conventional broadcasting issue queue architecture (Base) is shown in Figure 4 (a). Recall that the issue queue size is fixed at 32 entries throughout this paper. A noticeable performance loss of the general model can be observed from Figure 4(a). On average, the WF-Replay has a 9.0% performance loss compared to Base. By comparison, a Cyclone scheduler implemented as given in Figure 1 has an average performance loss of 25.5% compared to WF-Replay. This comparison does not take into account the potential faster clock cycle in Cyclone due to its local communication. Next, we increased the issue width from four to six instructions per cycle. The performance gap between WF-Replay and Base is diminished from 9.0% to 0.2% as shown in Figure 4 (b). When the issue width is further increased to eight instructions per cycle (Figure 4 (c)), the WF-Replay does not lose any performance compared to Base.

These results clearly demonstrate that the relative issue width (to fetch/decode width) has a significant impact on the effectiveness of the wakeup-free schedulers. When the relative issue width increases, the WF-Replay approaches perfect prediction accuracy and achieves the same performance as Base. Note that the performance loss in the wakeup-free scheduler is mainly due to the delayed issue of actually

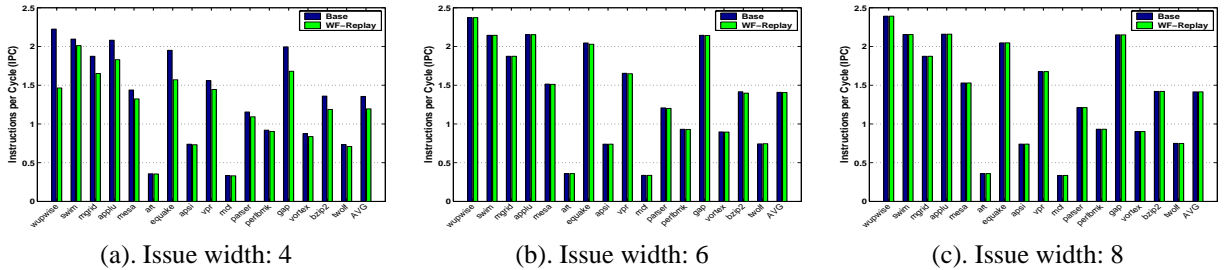


Figure 4. Performance of the general model WF-Replay at different issue widths.

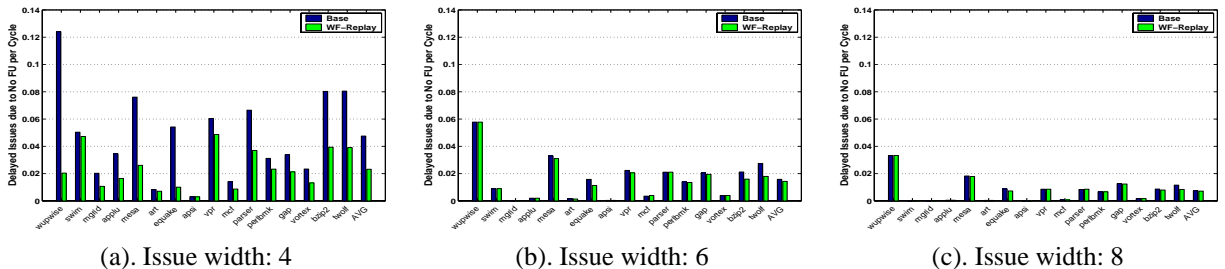


Figure 5. Average number of delayed instructions due to unavailable function unit: Base vs. WF-Replay.

ready instructions. The selection logic in WF-Replay performs in the same way as in Base. Thus, the reason for this delay is due to the delay caused by the non-availability of functional units and issue ports. Figure 5 (a)(b)(c) give the average number of delayed instructions due to non-available functional units per issue cycle for three issue widths for Base and WF-Replay. As the issue width increases, the average delay number decreases as expected. However, the delay number in WF-Replay is much smaller than in Base at 4-issue width. The absolute value of this delay number is very small, < 0.14 instructions per issue cycle.

The comparison of the average number of delayed issues due to non-availability of the issue ports between these two architectures is given in Figure 6 for three issue widths. The results shown in Figure 6 (a) indicate the cause for performance loss of WF-Replay when using a scheduler with 4-issue width configuration. The large number of delayed instructions (up to 15 instructions per cycle) due to the lack of issue ports is the main reason for the performance loss in WF-Replay with an issue width of 4. However, the number is significantly reduced when the issue width is increased to 6 or 8 instructions per cycle as shown in Figure 6 (b)(c), which in turn enables the WF-Replay to achieve a very close performance to Base at these two issue widths.

The explanation to this phenomena is as follows. In the wakeup-free instruction scheduler, the issue latency of each instruction is predicted based on the current state of the timing table. If one instruction gets delayed due to insufficient issue ports, its dependent instructions that already predicted

their issue latencies will be predictively ready and compete with their delayed source-producing (parent) instruction. That creates a ripple effect and the number of delayed instructions cumulate as seen in Figure 6 (a). However, if the issue width is larger than the fetch/decode width, instructions have less chance to be delayed due to insufficient issue ports. In this case, most instructions can be issued at their predicted issue time, thus eliminating this delay and the delay propagation. Consequently, the performance degradation in WF-Replay is minimized.

5. Pre-check Predictive Issue Queue

The general wakeup-free scheduler WF-Replay loses performance when the competition for the issue ports becomes intensive. If the competition can be reduced in an effective way by preventing the blocking of the actually ready instructions from issue, the performance of a wakeup-free scheduler can be fully exploited even when the issue width is same as the fetch/decode width. In this section, we propose a pre-check scheme WF-Precheck to release the competition pressure on the issue ports in a wakeup-free issue queue in order to achieve better performance.

In this new wakeup-free scheduler with pre-check, the instructions with their latency counters of zero need to first check the register file ready bits rather than immediately sending issue request to the selection logic. Each entry in the issue queue is associated with an additional ready bit indicating whether all the source operands of the instruction

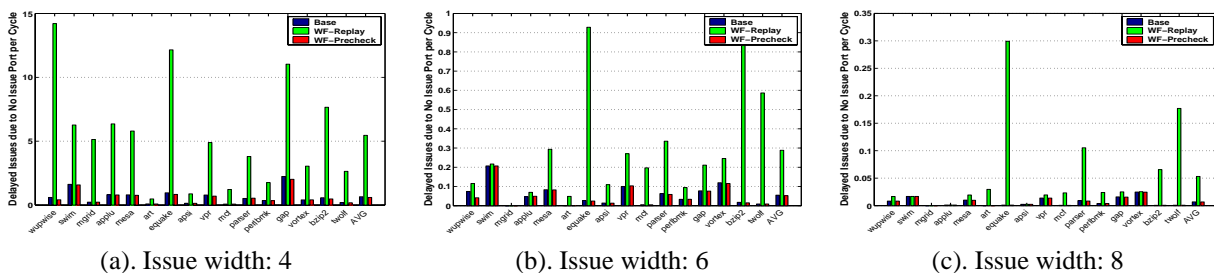


Figure 6. Average number of delayed instructions due to no issue port: Base vs. WF-Replay vs. WF-Precheck. Note that the y axis in (b) is zoomed into the range of 0 - 1, and y axis in (c) is zoomed into the range of 0 - 0.35.

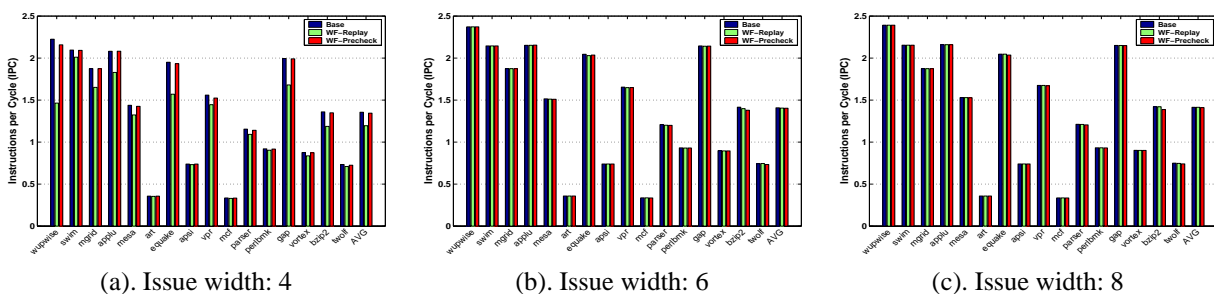


Figure 7. Performance of WF-Precheck scheme compared to Base and WF-Replay with issue widths of 4, 6, and 8 instructions per cycle.

are ready. The issue request to the selection logic is filtered by the values returning from the register ready bit checking. At the same time, the instruction ready bit is set according to this returning value. If the returning value is one, the issue requesting will go through to the selection logic, and the instruction ready bit is set to one such that there is no need to perform this register ready bit checking in a later cycle, if the instruction is not selected for issue in the current cycle. Otherwise, the issue request is blocked, and the latency value is recomputed by either setting it to a fixed value or checking the current state of the timing table. Working in such a way, only actually ready instructions are permitted to compete for the issue port, significantly reducing the competition. The register file ready bits are updated one cycle before the results are ready (produced). The proposed wakeup-free scheduler WF-Precheck with pre-check scheme is shown in Figure 8.

As noted, the register ready bit check and the instruction selection need to be done within a single cycle. The structure that stores these register ready bits might become a critical resource or performance bottleneck (We will address the delay problem of the selection logic in Section 7). Our experimental results show that a large portion (an average of 40.2%) of the instructions have both of two source operands ready (thus the instruction ready bit is set) at the

pre-scheduling stage, and another 45.4% have one operand ready at the pre-scheduling stage. Thus, the actual number of register ready check requests is small, less than two per cycle for 4-issue WF-Precheck, on the average. Note that the structure is very small and only stores one bit for each physical register. This structure can be made very fast for a small number of read ports.

The effectiveness of the pre-check scheme on reducing the competition at the issue port can be seen from Figure 6. Figure 6 (a) shows the significantly reduced number of delayed instructions due to issue port competition in the WF-Precheck (compared to the general model WF-Replay) for an issue width of 4. With larger issue widths, a reduction in the delay number is also achieved as shown in Figure 6 (b) and (c). These results meet the expectation of the WF-Precheck design.

At this point, the main performance constraints in the wakeup-free scheduler have been addressed using this new model WF-Precheck. Figure 7 presents the performance comparison for Base, WF-Replay, and WF-Precheck for different issue widths. For issue widths of 6 and 8, the WF-Precheck works consistently well as the WF-Replay and Base as seen in Figure 7 (b) and (c), respectively. Our focus is on the issue width of 4 which is equal to the fetch/decode

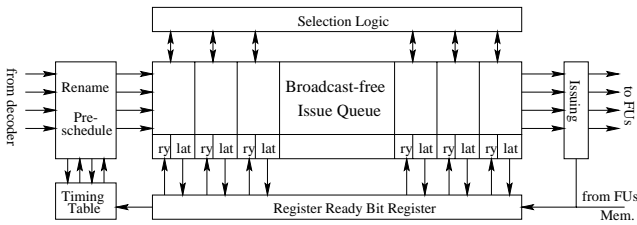


Figure 8. A wakeup-free scheduler WF-Precheck applying pre-check scheme.

width. Figure 7 (a) shows that WF-Precheck successfully attacks the performance bottleneck that presents in wakeup-free instruction schedulers including our general model WF-Replay.

6. Load Instruction Related Predictions

Our wakeup-free instruction scheduler model WF-Precheck presented in the previous section is a significant step towards designing a high performance wakeup-free architecture. For all three models discussed in the previous sections, a perfect load hit/miss predictor and a perfect load/store dependence predictor are always assumed for the latency prediction at the pre-scheduling stage. In this section, several load hit/miss predictors and a simple load/store dependence predictor are used to evaluate the impact on WF-Precheck. The issue width is fixed at 4 instructions per cycle (equal to the fetch/decode width) for the following evaluation.

6.1. Impact of Load Hit/Miss Predictors

Different from the ALU or control instructions, the latency of a load instruction is variable due to the memory behavior. However, the latency of the load instruction must be specified somehow since the latency prediction of all following instructions dependent on this load is based on this load's latency. Consequently, the wakeup-free issue queue design must have some prediction scheme for the load latency in order to proceed the pre-scheduling task. We investigated several off-the-shelf load hit/miss predictors that were originally designed for load-related speculative issue in the conventional issue queue designs [16]. These predictors include a bimod predictor, a 2-level gshare predictor, and a combinational predictor (consisting of a bimod predictor, a 2-level gshare predictor, and a selector). In these load hit/miss predictors, the load latency is simply determined by the outcome of the prediction. If the prediction is "hit", the load latency is set to the access latency of the L1 data cache. Otherwise, the load latency is set to the access latency of the L2 data cache. The resource cost of the

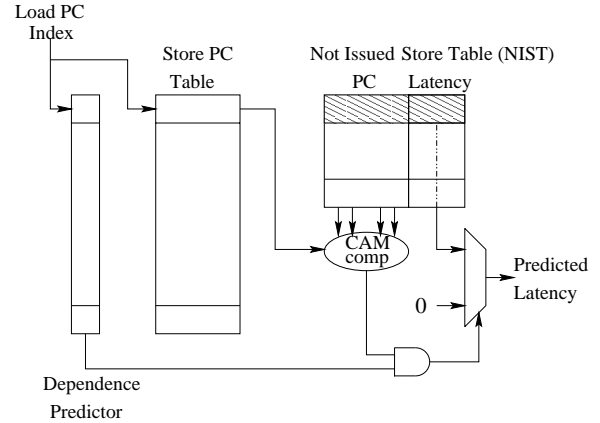


Figure 9. The load/store dependence predictor.

predictors configured are around 0.5 KB. The evaluation results (not shown due to the space limitation) show that these predictors achieve very close prediction accuracy and performance. We use the simple bimod predictor for the following analysis.

6.2. Impact of Load/Store Dependence Predictors

The load hit/miss predictor predicts the latency of a load instruction. However, a more complicated problem is to determine when a load instruction is safe for issue. A load instruction cannot be issued if it has a true memory dependence (load after store) on a previously not-issued store instruction. Further, the dependence will not be known before the memory addresses accessed by the load and store are computed. Intuitively, the load instructions have to wait until their memory addresses are computed and all previous stores have their memory addresses computed. However, this is infeasible from a performance perspective. Consequently, a load/store dependence predictor must be used.

In this section, we present a load/store dependence predictor to investigate the impact on the performance of the wakeup-free issue queue with pre-check scheme. The schematic of the predictor is given in Figure 9. This predictor has three components: a bimod dependence predictor, a store PC table, and a not-issued store table (NIST). The bimod predictor and store PC table are indexed by the PC address of a load instruction, and are updated according to the result of dependence check at the issue stage. The store PC table is initialized as empty. NIST stores the PC address (as the tag of the entry in NIST) of current not-issued store instructions (in the issue queue) and their issue latency, and is updated when store instructions are pre-scheduled or issued. The store PC part of NIST is implemented as a CAM. The work-flow of this load/store dependence predictor is quite sim-

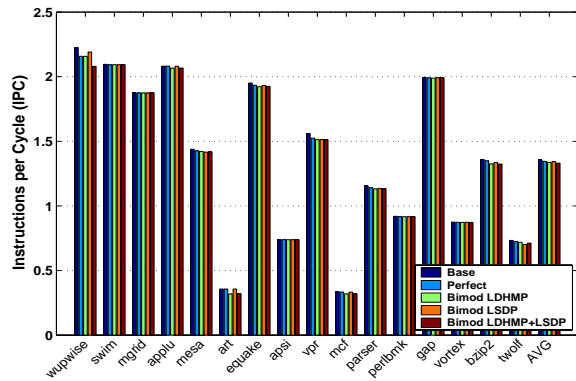


Figure 10. Performance impact of using load hit/miss predictors and load/store dependence predictors.

ple. Given a load instruction, its PC address is used to index the bimod dependence predictor and store PC table. If the predictor predicts no-dependence, a zero latency is returned immediately. Otherwise, the fetched store PC is sent to the tag part of NIST. If a hit is signaled, the corresponding latency value is returned. If a miss in NIST, the dependence prediction is automatically updated to no-dependence and a zero latency is returned. The load instruction uses this returned latency along with the latencies of source operands to predict the issue latency of this load instruction. We evaluated this predictor with 2K entries for the dependence predictor and store PC table and 16 entries for NIST.

Figure 10 gives the performance of the WF-Precheck adopting both load hit/miss predictor and load/store dependence predictor (shown as the rightmost bar), compared to the Base (leftmost bar), the WF-Precheck using perfect predictors (the second left bar), using a bimod hit/miss predictor with a perfect load/store dependence predictor (the third bar), and using a bimod load/store dependence predictor with a perfect hit/miss predictor (the fourth bar). The results in Figure 10 show that the WF-Precheck applying these simple predictors achieves a very close performance to the one using perfect predictors.

7. Segmented Issue Queue

In conventional issue queues, the delay of wake-up logic dominates the overall scheduling delay and increases linearly as either issue queue size increases or the issue width increases [12][7]. On the other hand, the selection logic delay increases at the logarithmic speed of increase of the issue queue size. However, this selection delay will become a significant part of the scheduling latency in a wakeup-free issue queue, especially when the issue queue size is a large one. In this section, we present a segmented wakeup-free is-

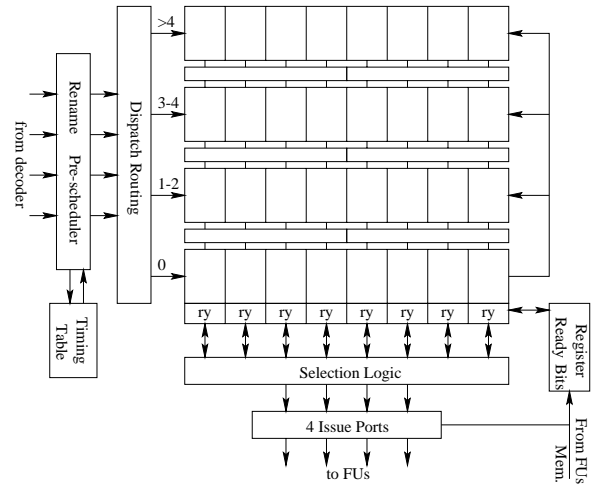


Figure 11. Wakeup-free instruction scheduler WF-Segment with segmented issue queue design.

sue queue design based on WF-Precheck to optimize the selection logic.

Recall that a larger instruction selection pool is necessary for high performance wakeup-free issue queue design. The architecture of our segmented wakeup-free issue queue WF-Segment is presented in Figure 11. In WF-Segment architecture, the issue queue is divided into several segments. Each segment is associated with a latency range, which increases from the bottom segment to the top segment. The bottom segment is used to host ready-to-issue (latency equal to zero) instructions, while the top segment is to buffer long latency instructions (latency is larger than a preset threshold). The latency counter of each instruction in segments other than the bottom segment is decreasing cycle by cycle. When the counter value reaches the latency range of the immediately lower segment, the instruction tries to move down (sink) to the lower segment. Instructions in the bottom segment do not need a latency counter because the instructions enter the bottom segment only when their latency value reaches zero. Instead, each instruction in the bottom segment is associated with a ready bit, which is not present in higher segments. These ready bits implement the pre-check scheme described in Section 5. The size of the segment is designed to be larger than the issue width such that the selection logic is provided with a larger selection pool. Also, the selection logic only interacts with the bottom segment reducing the complexity and delay of the selection logic.

Instructions are dispatched into different segments through the dispatch routing according to the issue latency predicted at the pre-scheduling stage. In this example, instructions predicted ready (latency = 0) are

dispatched into the bottom segment, instructions with predicted latency of 1 or 2 cycles go to the second segment, instructions having latency from 3 to 4 cycles are routed the third segment, and all other instructions with latency larger than 4 cycle are treated as long latency instructions and put into the top segment. If the current segment is full, an instruction is routed to a higher segment. If it is the top segment and there is no free space, the dispatching is stalled and re-tried in a later cycle.

The sinking path is designed in such a way that the instructions at the both sides of a segment are given the priority to sink to the next lower segment. This design helps simplify the new instruction dispatching and instruction switching back for replay. At a given cycle, each free entry in a lower segment might receive at most three sink requests from its next upper segment and gives different priorities to these requests. The priority granting policy is different from entries in the left half of a segment and entries in the right half. A free entry in the left half uses the left-first priority policy. It first gives the priority to the request coming from the top-left entry, then to top entry, and last to the top-right entry. On the other hand, entries in the right half adopt a right-first priority policy that is reverse to the one used in the left half. Employing these two priority policies at the left half and the right half of a segment respectively, instructions are sinking towards the center. The space at each side of a sinking segment will be freed in the best case. These freed entries can be immediately used to host the switch-back instructions and the newly dispatched instructions at two other sides thus reducing the routing cost.

The pre-check scheme in the WF-Segment scheduler is different from the one employed in WF-Precheck scheduler. When an instruction in WF-Segment sinks to the bottom segment, it immediately checks the register ready bits of its source operands. If all the operands are ready, the instruction sends issue request to the selection logic and sets its ready bit. These operations are performed within an issue cycle. Otherwise, the instruction is removed from the bottom segment and sent back to a higher segment according to its updated latency. The WF-Segment enforces a high-level priority policy for instruction movement within the issue queue. An instruction switching back is always granted the highest priority, followed by instruction sinking. Instruction dispatching has the lowest priority to compete for the free entries in the issue queue. The enforcement of this priority policy guarantees the issue queue design to be deadlock free.

In our experiments, the segmented issue queue WF-Segment is configured as follows. The issue width of WF-Segment is same to the fetch/decode width, 4 instructions per cycle. The segment size is configured as two times larger than the issue width. The number of segments for the issue queue is 4. The latency range associated with each segment is as follows: 0 cycle for the bottom segment, 1-2 cycles for the second segment, 3-4 cycles for the third segment, and >4 cycles for the top segment.

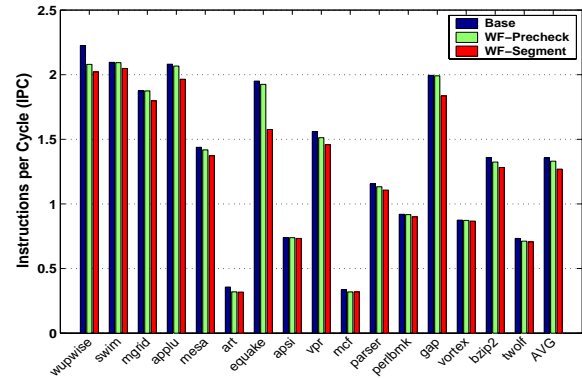


Figure 12. The performance of the segmented wakeup-free issue queue WF-Segment.

The new latency of a switching-back instruction is recomputed based on the new state of the timing table. This latency re-computation can be simplified to setting a fixed latency value in order to reduce the complexity of the timing table, which has a very slight performance impact [6]. Figure 12 shows the performance of the WF-Segment compared to the WF-Precheck and Base. With minimized selection logic, the WF-Segment presents a very effective wakeup-free issue queue design. On the average, the WF-Segment only trades 3.5% performance(IPC) loss to the WF-Precheck and 5.8% loss to the Base for its segmented design to optimize the selection logic. Note that this performance loss does not count the fact that WF-Segment can use a much faster clock due to its shorter scheduling delay.

8. Related Work

Reducing the complexity of issue queue has been the focus of several efforts. The work by Palacharla et. al. [12] clearly highlights the complexity bottleneck imposed by the use of larger and wider issue queues. They also propose an effective technique for reducing the complexity of instruction issue using dependency-based ordering of instructions into a set of FIFO queues and by considering only the head of the FIFO queues for issue.

The dependency based scheduling techniques were improved by predicting operation latencies at dispatch time and creating a schedule at dispatch time. Canal and Gonzalez [4] propose a scheme where the instructions whose latencies are predicted are placed in a FIFO queue and can be issued deterministically. However, they use an associative buffer for instructions whose ready time cannot be determined due to memory dependences. Michaud and Sezec [11] propose a prescheduling technique that progresses the oldest elements from the scheduling array to a small fully-associative issue buffer. Canal and Gonzalez [5]

propose a “deterministic” latency approach that can issue directly from the scheduling array but places those with mispredicted latencies into a delayed issue queue. In our approach, the precheck logic helps to eliminate issue of instructions that have mispredicted latencies. Further as compared to those techniques that employ an associative buffer, wakeup in our approach occurs only based on predicted latencies.

Our approach is most similar to that proposed by [13] that uses a segmented issue queue based on the predicted delays. In their approach, instructions are dispatched into the top segment and are advanced from one segment to another based on whether their predicted latencies before issue is below a specific threshold. Eventually, the instructions in the final segment are woken up and issued like in a traditional issue queue. In contrast, our approach does not involve any broadcast for instruction wakeup. Hence, in our approach, it is essential to avoid the possibility of issuing instructions before their operands become available. Hence, the precheck logic is critical in eliminating this bottleneck.

As mentioned earlier, the broadcast free approach mentioned in Cyclone is also closely related to our work.

9. Conclusions and Future Work

The relative delay of the dynamic instruction scheduler in high-performance superscalar microprocessors is increasing due to the fast expanding wakeup logic that dominates the scheduling latency. In order to take the full advantage of the advancing clock speed, instruction scheduler (i.e., the issue queue) must be designed in such a way that the wakeup logic will not be the performance bottleneck. In this paper, we explored the designs of wakeup-free dynamic instruction schedulers. We proposed a general wakeup-free scheduler WF-REPLAY based on the study of a previously proposed broadcast-free scheduler Cyclone. Our design avoids the performance constraints present in Cyclone. The analysis of the WF-REPLAY shows that the resource conflict at the issue port has a significant impact on the latency prediction accuracy and consequently the performance in a wakeup-free scheduler. By preventing the unnecessary issue port competition, the pre-check scheme applied in our new WF-PRECHECK scheduler helps achieve a performance close to the wakeup-based conventional instruction scheduler. The impact of a load hit/miss predictor and a load/store dependence predictor on the WF-PRECHECK wakeup-free scheduler is investigated. Finally, we propose a segmented wakeup-free scheduler WF-SEGMENT to minimize the complexity and latency of the selection logic, while maintaining a high performance. A detailed analysis of the design complexity of the WF-SEGMENT scheduler is one of our ongoing work.

The issue latency prediction at the prescheduling stage in a wakeup-free architecture is performed based on the state of the timing table and load related predictions. As for future work, a much smarter or sophisticated latency predic-

tion scheme can be designed to utilize the information about the usage and reservation of functional units to make more accurate prediction. If the prediction can specify the target functional unit for each instruction, the selection logic can be eliminated.

Acknowledgments

This work was supported in part by an NSF CAREER Award 0093085, an NSF grant 0103583, and a grant from MARCO/GSRC-PAS.

References

- [1] E. Brekelbaum, J. R. II, C. Wilkerson, and B. Black. Hierarchical scheduling windows. In *Proc. of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2002.
- [2] M. D. Brown, J. Stark, and Y. N. Patt. Select-free instruction scheduling logic. In *Proc. of the International Symposium on Microarchitecture*, Dec. 2001.
- [3] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, 1997.
- [4] R. Canal and A. Gonzalez. A low-complexity issue logic. In *Proc. of 2000 International Conferences on Supercomputing*, May 2000.
- [5] R. Canal and A. Gonzalez. Reducing the complexity of the issue logic. In *Proc. of 2001 International Conferences on Supercomputing*, June 2001.
- [6] D. Ernst, A. Hamel, and T. Austin. Cyclone: a broadcast-free dynamic instruction scheduler selective replay. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, June 2003.
- [7] D. Ernst and T. Austin. Efficient dynamic scheduling through tag elimination. In *Proc. of the 29th Annual International Symposium on Computer Architecture*, May 2002.
- [8] J. Hennessey and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, 3rd edition, 2002.
- [9] M. S. Hrishikesh, D. Burger, S. W. Keckler, P. Shivakumar, N. P. Jouppi, and K. I. Farkas. The optimal logic depth per pipeline stage is 6 to 8 for 4 inverter delays. In *Proc. of the 29th Annual International Symposium on Computer Architecture*, May 2002.
- [10] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proc. of the 29th Annual International Symposium on Computer Architecture*, May 2002.
- [11] P. Michaud and A. Sez nec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *Proc. of the 7th International Symposium on High Performance Computer Architecture*, Jan. 2001.
- [12] S. Palacharla, N. P. Jouppi, and J. Smith. Complexity-effective superscalar processors. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.
- [13] S. Raasch, N. Binkert, and S. Reinhardt. A scalable instruction queue design using dependence chains. In *Proc. of the 29th Annual International Symposium on Computer Architecture*, May 2002.
- [14] J. Stark, M. D. Brown, and Y. N. Patt. On pipelining dynamic instruction scheduling logic. In *Proc. of the International Symposium on Microarchitecture*, Dec. 2000.
- [15] K. C. Yeager. The mips r10000 superscalar microprocessor. *IEEE Micro*, pages 28–40, 1996.
- [16] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation techniques for improving load related instruction scheduling. In *Proc. of the 26th Annual International Symposium on Computer Architecture*, May 1999.