SEED: Scalable, Efficient Enforcement of Dependences

Francisco J. Mesa-Martínez, Michael C. Huang[†], Jose Renau
Dept. of Computer Engineering, University of California Santa Cruz
[†]Dept. of Electrical and Computer Engineering, University of Rochester

Abstract

Instruction issue logic is a critical component in modern high-performance out-of-order processors. The ever increasing latencies found in modern processors, mostly associated with memory accesses and longer pipelines, can be attenuated using large issue queues. Conventional designs rely on atomic wakeup-select cycles to ensure compact scheduling. These designs must aggressively utilize broadcasting, compaction, and heavily-ported structures that scale poorly in terms of both power consumption and access time.
To provide high scheduling flexibility and large instruction capacity without incurring prohibitive latency and energy overhead, we propose a novel scheme that uses an out-of-order, broadcast-free instruction wakeup block feeding an in-order scheduler. Multi-banked, index-based structures are used throughout this scheme to provide a high degree of scalability while achieving efficient dependence tracking, resulting in good overall performance and energy efficiency. We call this design "Scalable, Efficient Enforcement of Dependences (SEED)". We present a detailed design and analysis of SEED through an extensive evaluation. Compared to a conventional issue queue design, which is assumed favorably to scale in size without any impact on cycle time, the performance degradation of our design is 3% for both INT and FP suites of SPEC CPU2000. For such a small performance cost, SEED enjoys a 19% reduction in total chip power consumption for a 32-entry configuration. We also synthesize SEED and a conventional issue logic with 90nm standard cell logic. Synthesis results show that SEED can cycle twice the speed of a conventional issue logic of equivalent size. Cycling at the same frequency, SEED consumes ten times less dynamic power and five times less static power while achieving substantial area savings.

Categories and Subject Descriptors

B.7.1 [**INTEGRATED CIRCUITS**]: Microprocessors and microcomputers

General Terms

Design

Keywords

Issue logic, energy-efficient design, scalability

1. INTRODUCTION

As VLSI technologies continue to push toward higher levels of integration, the energy consumption of high-performance microprocessors and the resulting heat dissipation have become perhaps *the* limiting factor to performance in high-end designs. Complex out-of-order designs only increase the problem by requiring more energy [14]. As a result, designers start to opt for simpler processors trading off performance for lower energy consumption and reduced overall design complexity. The high energy consumption of existing out-of-order *implementations* is the result of a long period of dedicated pursuit of performance at the expense of energy consumption: speculations are applied very aggressively and the circuitry for orchestrating out-of-order execution is not designed with energy efficiency as a priority. However, the out-of-order model is not inherently more power-hungry. By executing independent instructions instead of stalling due to dependences, out-of-order execution can in fact cut the energy waste due to unnecessary stalls. Therefore, we need to look for radically different designs for out-of-order microarchitectures and search for more sensible speculation strategies. In this paper, we look at instruction scheduling.

Out-of-order microprocessor cores rely on the instruction issue logic to uncover instruction-level parallelism (ILP) while maintaining data dependences. This circuit component is often very complex, carrying out different functionalities and interacting with almost every microarchitectural component. Centering around the issue queue that buffers instructions waiting for execution, the issue logic has two major scheduling functions: *wakeup* and *selection*. These two functions are closely coupled and form a scheduling loop, thus making the design harder to scale to larger issue queues without impacting cycle time. Moreover, every cycle, several instructions can enter the queue and some may leave, not necessarily in the same order. The management of these entries also makes the design quite complex. On the other hand, modern processors need to buffer a large number of in-flight instructions to partially hide the relatively long and perhaps still increasing memory latency. As a result, the scalability of the issue logic becomes more important.

In this paper, we propose a design that performs scalable, efficient enforcement of dependences (SEED). We decouple the instruction wakeup and selection processes, and employ a broadcast-free wakeup logic using a scalable, multi-banked table for dependence tracking. This table re-orders instructions and then feeds them to a simple in-order execution logic. Compared to a broadcast network, a table structure is more natural for dependence tracking, and can be easily scaled without significant increase in energy consumption. We show that our decoupled instruction scheduler is more energy-efficient compared to conventional implementations and exploits instruction-level parallelism more effectively in a high-frequency core. Compared to an idealized conventional issue logic design with the same buffering capacity and same frequency, the SEED implementation cuts *chip-wide* power consumption by 19% to 39% depending on the capacity when running SPEC CPU 2000 applications. This reduction in power consumption is achieved with a small 3% performance degradation.

The rest of the paper is organized as follows: we discuss the motivations behind our design decisions in Section 2; we describe the microarchitecture in detail in Section 3; we discuss the experimental setup in Section 4; we show the evaluation of our design in Section 5; we discuss related work in Section 6; and we conclude in Section 7.

2. MOTIVATION

The increasing performance gap between modern processor and memory technologies has led to significant penalties associated with memory accesses. High-performance designs need to structurally hide these latencies to successfully exploit ILP. Buffering more in-flight instructions is an effective and straightforward way to hide the latencies associated with many important classes of applications. Unfortunately, this approach requires increasing the size of various resources, such as the issue queues and the register files. This can lead to considerable increases in access time, area, and energy consumption.

The issue queue is particularly challenging to scale in a conventional design, since instruction issue is subdivided into two mutually-dependent steps: *select* and *wakeup*. The select logic examines all ready instructions and determines which ones to execute according to specific priority criteria and constraints in execution resources. The information containing which instructions are selected is propagated back to the issue queue by the wakeup logic, this information is used to update operand readiness status in preparation for the whole process to repeat itself again the next cycle. Increasing the issue queue capacity would slow down both processes, thus making it even more difficult to complete them in one cycle. Without support for atomic execution of the select-wakeup step in one cycle, a conventional scheduler cannot arrange dependent instructions to be executed back-to-back in consecutive cycles. This may lead to significant IPC degradation, especially for applications with heavy use of integer computations.

Moreover, the wakeup process is often implemented by broadcasting the tags of ready instructions to be compared associatively with the operand fields of instructions in the queue [32]. Each simultaneously broadcasted tag requires a dedicated port in the issue queue for matching. With the non-trivial issue widths found in modern processors, this design requires a large, multi-ported structure with a large capacitative load. All these factors lead to the large amount of energy consumption associated with traditional instruction wakeup.

Optimal instruction issuing structures should be, therefore, free of large amounts of associative logic. In the design of SEED, we use indexed tables to keep instructions waiting for wakeup and to also track dependencies. These tables can be implemented using banked structures, which can have much larger capacities than associative queues using comparable amounts of energy and latency. Figure 1 shows the scalability for banked structures which reduce the energy and delay with respect to their flat counterparts.

Latency tolerance is another desirable feature for an optimal instruction issue logic. By this, we mean that the ability to execute dependent instructions back-to-back should be minimally affected even when the scheduling of instructions takes multiple cycles. Finally, an out-of-order issue logic needs to gracefully handle variable-latency instructions such as loads (due to memory hierarchies) and certain floating-point and multimedia operations, lest it loses its appeal over statically scheduled processors.

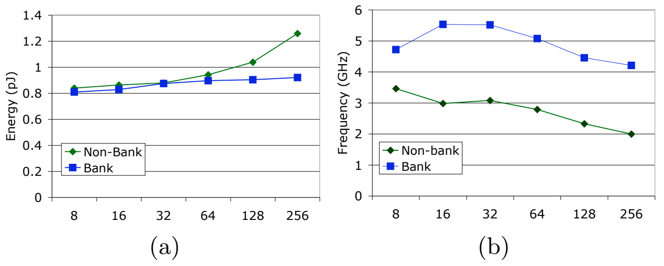


Figure 1. Maximum frequency (a) and energy per access (b) for different SRAM sizes. "Non-bank" corresponds to an SRAM with the same number of ports for all the sizes (8 read, 4 write). "Bank" corresponds to an SRAM that increases the number of banks and decreases the number of ports so that $E * D^2$ is minimized. The data is generated for $70nm$ technology using CACTI [25] for the banked and non-banked configurations.

In order to attain the aforementioned goals in SEED, we restructure the scheduling and fully decouple the wakeup-select process. For common instructions, their wakeup in SEED is driven by their parents' wakeup. This shortens the critical loop associated with wakeup-select by completely removing the select. For certain variable-latency instructions, the wakeup is delayed until they are ready (or close to).

3. DESIGN OF SEED

3.1 Overview

To provide sufficient capacity for instructions pending execution and yet avoid the negative energy and latency impact of using large associative broadcast-based queues, SEED holds unready instructions in indexed dependence-tracking tables instead. Instruction wakeup is performed as a simple indexed access which retrieves all dependent instructions from the dependence table and eliminates the need for any broadcasting. Also, unlike in conventional designs, where woken instructions have to go through a *select* process and only selected instructions can wake up their dependents, we remove the select process from the wakeup logic altogether, and allow instructions to wake up their dependents as long as they are woken up. All the woken instructions are sent to a small FIFO buffer where the select is performed. This removes the select logic from the timing-critical process of scheduling dependent instructions to execute back-to-back.

Non-associative queues have already been used in previous proposals dealing with the organization of instruction issue logic, most notably in certain types of "eager" instruction issue [4, 7, 19, 22]. These designs share a common guiding principle: only a subset of instructions "close to ready" need to be present in a final issue queue. Since the amount of almost ready instructions is expected to be significantly lower than the overall number of waiting instructions in the window, this final queue is small in size and can be broadcast-based. The remaining instructions are "pre-scheduled" into an indexed table or queue at decode time based on their *predicted* ready time. Finally, these pre-scheduled instructions are moved into the final issue queue when the predicted ready time arrives.

One drawback of eager issue logic is the need for complex latency calculation logic, especially for variable-latency instructions before the decode stage. Consequently any latency misprediction will negatively impact the performance of its chain of dependent instructions. The indexed table used in these eager issue logic is usually very simple and narrow, disallowing for any dynamic schedule changes. At the same time, wrong-path instructions can only be drained as their ready time arrives one by one. In some designs, instructions in the indexed queue have to be physically moved from one segment to another before injected into the final issue queue. Obviously, this movement of instructions bears an energy cost. Also, these designs tend to use availability time of the registers to propagate the calculation and constantly update the current times. There is the need to checkpoint current availability times for branch misprediction recovery which also increases the overall complexity and energy consumption.

In contrast, SEED is still a wakeup-driven (or "lazy") issue logic design which obviates the need for any latency prediction, update, and checkpoint logic as in an eager issue logic. All instructions in SEED stay in the dependence tracking table until they are woken up, thus there is no physical movement while they stay in the table. Furthermore, as we will discuss in more detail later, the design naturally purges a portion of wrong-path instructions from the table, reducing some capacity loss.

Due to implementation variations and terminology differences we make the following clarifications. In the following discussion, we assume a generic pipeline found in most out-of-order processor implementations. In this pipeline instructions pass through an in-order front-end in which they are fetched, decoded, and renamed. From then on, they are scheduled to execute in an order that strives to maximize parallelism while satisfying data dependences. In typical implementations (*e.g.*, [28]), this scheduling logic is centered around one or more issue queues. Instructions from the front-end flow into these queues in program order. We follow [28] and refer to this process as *dispatch*. Once queued, an instruction waits until its operands are ready, whereupon it is eligible for execution. It is then selected and sent to an execution pipeline. We refer to this process as *issue*. These terms are used to refer to similar processes in our design.

3.2 The Structures and Operation of SEED

In a SEED design, instruction scheduling is broken down into two decoupled components: *wakeup* and *issue*, which are connected by a first-in first-out (FIFO) *issue buffer* (Figure 2). A key difference with respect to conventional designs is that wakeup is performed based on which instructions are woken up, rather than which instructions are selected for execution. Thus the critical loop only contains wakeup in our design.

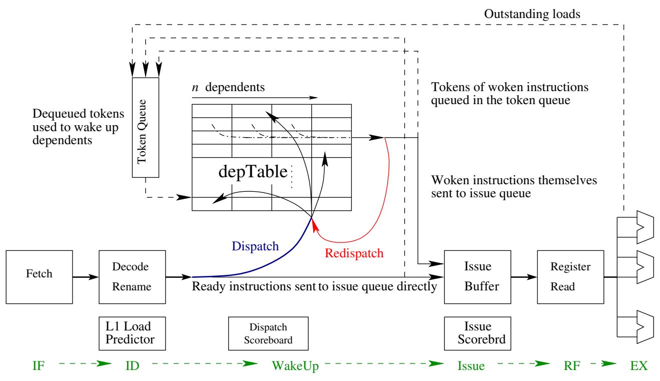
At a high level, issue logic needs to ensure that instructions only execute after all input data are ready. In other words, their execution needs to satisfy two conditions: first, it is after their parents' execution, and second, it is separated from their parents' execution by a minimum of a certain number of cycles to allow the parents' computation to finish. With these conditions in mind, the process of instruction scheduling in a SEED design can be thought of as a two-step process: first, pre-sorting the instructions at wakeup so that they enter the next stage almost always ready for execution and they still follow the partial order dictated by their dependency relationship (order condition). The pre-sorting makes no guarantee of instructions' exact timing, only their relative order. This is why we do not care which instructions are selected when we do wakeup. Then, separately at the issue stage, we fine-tune instruction schedule to satisfy the timing requirement. In the following, we describe the three major stages an instruction flows through in a SEED pipeline: dispatch, wakeup, and issue. We note that the logic described here is implemented in Verilog and is fully synthesizable.

3.2.1 Dispatch

After rename, an instruction is written into a dependence-tracking table called the *depTable* (Figure 2) to clear dependence constraints. In this paper, we focus on register dependence, though the table could also be used to facilitate memory-based dependence tracking. If all source registers are ready (to be exact, all parent instructions have performed their wakeup and have been sent to the issue buffer), the instruction does not need to wait in the depTable and is thus sent to the issue buffer directly.

The depTable is organized as an array of entries. Each entry corresponds to one value-producing instruction and contains a few *sub-entries*, each containing a dependent that should be woken up by this instruction. When an instruction is inserted into the depTable, it is appended to the end of the entry of *one* parent. This is based on the observation that only a small portion of instructions (17% in our study) are dispatched with more than one pending source register. When multiple unready parents are present, one is randomly selected for queuing. For these instructions, a *speculative wakeup* bit is marked to make sure that the system checks whether all sources are ready when the instruction is woken up by one ready source. If the check fails, the instruction is *re-dispatched* the following cycle to be queued under a different source in the depTable. To minimize the occurrence of re-dispatch, we tested using a PC-indexed predictor to predict the source that will be ready last so as to queue a dependent in the depTable under that parent. Somewhat surprisingly, the impact of having such a predictor is negligible, partly because re-dispatch is rare.

During dispatch, a bit vector, which we refer to as the *dispatch scoreboard*, determines which instructions can be sent to the issue buffer directly and which ones need to stay in the depTable. This scoreboard has as many bits as the number of physical registers and is maintained as follows. When a register is allocated, the corresponding bit is reset. When an instruction is sent to the issue buffer from the depTable, the corresponding bit for its destination register is set to one. Essentially, this scoreboard only tracks whether the producer instruction has performed a wakeup of its dependents, rather than whether the value in the register is actually ready. When the bits corresponding to both source registers of an instruction are set, the registers are either ready or the producers have been sent to the issue buffer



Outstanding loads
Dequeued tokens used to wake up dependents
Token Queue
n dependents
depTable
Tokens of woken instructions queued in the token queue
Woken instructions themselves sent to issue queue
Dispatch
Redispatch
Fetch
Decode Rename
Ready instructions sent to issue queue directly
Issue Buffer
Register Read
L1 Load Predictor
Dispatch Scoreboard
Issue Scorebrd
IF
ID
WakeUp
Issue
RF
EX

Figure 2. Structures and operation of a SEED pipeline. Solid and dashed lines show instructions and tokens (depTable entry IDs) respectively. For clarity, the register alias table-like allocator for depTable entries are not shown.

(the order condition is satisfied). Therefore, the instruction is sent to the issue buffer directly, without going through the depTable.

The depTable is heavily banked to allow multiple independent updates to different entries in one cycle. Writing into different entries in the depTable is analogous to writing in multiple reservation stations or writing into issue queues in a clustered microarchitecture. Although the depTable is heavily banked, there is still the possibility that two entries requiring update in the same cycle belong to the same bank. One possible solution is to use multi-ported table, which obviously increases area, latency, and energy consumption. We found out that even with a single-ported banked table, conflict is tolerably infrequent. In our design, therefore, each bank supports a single insertion into one depTable entry (write) or a single wakeup (read) each cycle. This reduces the number of ports required significantly. When there is a bank conflict, we delay the dispatch of certain instructions to a later cycle by storing them in a small FIFO to decouple dispatch from rename. When the FIFO is full, the front end stalls. Also, we may be unable to queue an instruction into a particular entry when it runs out of sub-entries (an exceedingly rare event). At that moment, we simply stall the front-end and wait. Eventually, the owner instruction of the entry will be woken up and sent to the issue buffer. By that time, the stalled instruction can be directly sent to the issue buffer, allowing dispatch to continue. Finally, even during dispatch stall, we may continue to encounter instructions that need re-dispatch but are unable to enter the depTable due to the entry being full already. Such instructions have to be buffered as well. In theory, to guarantee that we can always buffer such instructions requires a queue as big as the capacity of the entire depTable. In practice, such instructions are very rare and a small 4-entry queue suffices. When an overflow happens, we can drop the instructions that we can not buffer and mark them in the re-order buffer (ROB) to generate a soft exception [32] – when the instruction reaches the head of the ROB (if it is not on the wrong path), instruction fetch and decode will restart from that instruction. Notice that forward progress is easily guaranteed: when the dropped instruction is re-fetched, all sources are ready.

Each instruction that has dependents needs to perform their wakeup and thus requires a depTable entry. The management of depTable entries can be tied to that of the physical registers. However, this approach leads to serious underutilization of the space as a depTable entry only serves the purpose to temporarily buffer instructions to be woken up and once they are, the entry is useless and can be recycled. In contrast, the register is recycled much later than when the producer instruction is woken up. Furthermore, tying depTable management to physical register allocation also limits the depTable to register-based dependence tracking only. The depTable can be used for memory-based or other artificial dependences. Therefore, we choose to manage the depTable separately by using a mechanism very similar to register renaming: (1) when needed, an instruction allocates a depTable entry and updates an alias table. (2) Dependent instructions find out the right entry of their parents via this table. (3) Once an instruction performs a wakeup (see below), its entry is de-allocated. The table and the read pointer of the free list are also check-pointed for branch misprediction recovery [32]. The table is indexed with logical register number. In this paper, it is implemented as an extension of the register alias table (RAT). In the following, for ease of discussion, we refer to the ID of the allocated depTable entry as the instruction's *token*. It is essentially a pointer to a depTable entry.

3.2.2 Wakeup

When an instruction is woken up, the instruction itself is sent to the issue buffer. Meantime, its token, if any, is written into a FIFO *token queue* so as to wake up its dependent instructions the next cycle. Every cycle, the wakeup logic selects the first few tokens from the token queue to index the depTable and read out the dependents. How many tokens can be dequeued is subject to various constraints such as bank availability. In this process, those instructions woken up that have the speculative wakeup bit set are double-checked against the dispatch scoreboard and re-dispatched as necessary. The rest (instructions without the speculative bit set and those that do, but find all parents ready) are sent to the issue buffer and have their own tokens sent to the token queue, repeating the cycle. We note that for instructions sent directly to the issue buffer at dispatch time, their token is also queued in the token queue.

Overall, the wakeup process is a simple self-cycle without the "select" component found in conventional issue logic. While this basic operation is straightforward, there are two issues that deserve attention: load-hit speculation and branch misprediction handling.

Load-hit speculation: Due to the use of memory hierarchy, load instructions have variable latencies. In conventional implementations, load-hit speculation is often used [6]. In a SEED implementation, load-hit speculation can also be supported easily. For a load predicted to hit in the cache, its token is entered into the token queue when the load instruction is woken up. For a load predicted to miss, its token is not sent to the token queue until the data returns from the cache. This is illustrated by the examples shown in Figure 3. For instructions with long and variable latencies (*e.g.*, floating-point division), the same strategy can be used: we send the token only when the instruction finishes execution.

When a predicted miss turns out to be a hit, nothing needs to be done. Whereas when a predicted hit turns out to be a miss, we stall the execution pipeline until the data

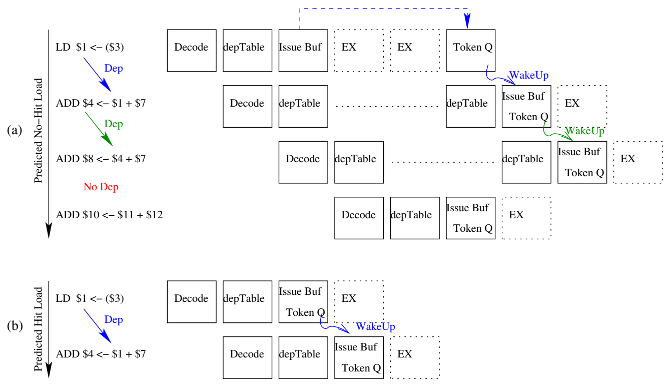


Figure 3. Examples of dependence enforcement. Timing is shown for illustration only, and is not a rigorous representation.

returns. We adopt this simple strategy because we found that in the applications we studied, with a PC-based predictor, mispredicting a cache miss is rare. Our predictor uses an 8192-entry table of 3-bit saturating counters. An L1 data cache hit increments the counter by 1, while a miss resets the counter to zero. We only predict a hit when the counter is saturated.

Branch misprediction handling: DepTable entries allocated to wrong-path instructions are reclaimed via a single action of restoring the read pointer of the free list [32] and will be re-allocated, reset, and reused to hold right-path instructions. This "automatically" purges out many wrong-path instructions from the depTable, but does not remove all of them: Some of them will have parents that are prior to the mispredicted branch and therefore could be queued in an earlier entry unaffected by the recovery. These wrong-path instructions will thus remain in the depTable. For hardware simplicity, we do not proactively purge these remaining wrong-path instructions out of the depTable. They will eventually be woken up, at which time we filter them out. To do so, we use a basic block valid bit vector. Each decoded basic block is assigned a basic block ID (BBID). This ID increases monotonically with wrap-around and is recycled in the same fashion. When a basic block is committed, its ID is recycled. (BBIDs allocated to wrong-path blocks will be recycled when the first subsequent right-path block is committed.) When available IDs run out, decode stalls. When a branch is mispredicted, all the wrong-path BBIDs will be flagged invalid in the bit vector. Instructions belonging to these basic blocks will be dropped when woken up.

3.2.3 Issue

After being reordered by the wakeup process, instructions in the issue buffer are in an order that is amenable to simple, in-order issue. We simply employ another bit vector, which we call *issue scoreboard*, to enforce proper pipeline interlocking. This scoreboard indicates whether a physical register is ready or the value can be read off the by-pass network by the time the sourcing instruction travels through intermediate pipeline stages and reaches the functional unit. Each cycle, the issue logic selects as many ready instructions as the functional units can accommodate from the head of the issue buffer and sends them down the pipeline for register access and execution. The issue scoreboard is adjusted to reflect the issue of these instructions: for single-cycle instructions, the corresponding scoreboard bit of their destination register is set to ready the same cycle; for instructions with an n-cycles latency, the bit is reset when the instruction issues and will be set in cycle $n-1$ after issue.

To correctly handle branch mispredictions, the corresponding bit in the issue scoreboard is reset when a register is allocated, just like with the dispatch scoreboard. Wrong-path instructions in the issue buffer are discarded upon issue to save energy down-stream.

4. EVALUATION SETUP

In order to characterize the behavior of our proposed *SEED* design we compare it against a conventional system denoted as *Base* which uses a compacting issue queue (IQ). Both configurations use a reorder buffer (ROB) and separate register files (RF).

Shared Parameters	
Processor core: 70nm technology @ 4GHz ROB/LQ/SQ/INT RF/FP RF: 640/96/64/384/256 Fetch/issue/commit width: 8/6/6 LdSt/branch units/other ALU/FP units: 3/2/5/4 Latencies (cycles): RF 3, LSQ 3, min. mispred. penalty 14 Branch: 1 taken br per cycle, 32-entry RAS, 4K BTB 4-way, YAGS: spec. update, 32K-entry T/NT cache, 6-bit tags	
Memory subsystems: 32 MSHRs, 16-stream stride prefetcher IL1: 32KB, 2-way, 64B line, 2 cycles, 1 port DL1: 32KB, 4-way, 64B line, 3 cycles, 2 ports L2: 2MB, 4-way, 64B line, 12 cycles, 1 port TLB (fully associative): Data 32-entry, Inst. 64-entry TLB Latency: 2 sequential accesses + 16 cycles Main memory: 78ns round-trip w/ 16GB/s b/w	
SEED(S)	*Base(W)*
depTable: size S, 4 instr./entry, $\frac{S}{16}$ banks, 2 ports 1 cycle. in-order issue buffer: 8-entry DL1 hit predictor: 8K-entry (3-bit)	Issue Logic: size W 4 ports, 1 cycle

Table 1. Simulation parameters.

We perform our evaluation using execution-driven simulations with a detailed model of a state-of-the-art processor and its memory subsystem [23]. All the pipeline details are modeled faithfully. For example, SEED increases the pipeline by one wakeup stage. Additionally, latency, occupancy, and contentions of all structures in the processor, caches, bus, and memory are modeled in great details. Table 1 summarizes the parameters used.

4.1 Energy Model

We model and aggregate the dynamic energy consumed in all chip structures, including processor core and the cache hierarchies. We incorporated Wattch [2] into our simulation infrastructure. Clock-gated structures are set to consume 5% of their active dynamic energy. We created models for the depTable using CACTI [25]. Although we have synthesized *Base* and *SEED* with 90nm standard cell logic, we use the CACTI energy values to maintain the simulation infrastructure consistent. This also simplifies the comparison with other research works because most of them use CACTI or some derivatives for their energy models.

4.2 Applications Evaluated

We use SPEC CPU 2000 applications for evaluation. A few applications are excluded: *eon* causes some special problems in the simulator and does not execute correctly; benchmarks written in F90 are excluded as our infrastructure does not yet support them. All other applications are compiled with gcc 3.4 with the -O3 optimization flag to generate MIPS binaries. We insert “simulation markers” in the application binaries. After fast-forwarding over the initialization (several billion instructions), we simulate for a given number of markers such that more than 750 million instructions are committed.

4.3 Verilog Code Evaluated

The evaluation also compares the synthesis results from a traditional compacting window like the one used in Alpha 21264 and the proposed SEED design. To do so, we implement the SEED design on synthesizable Verilog-2001. The IVM [30] project provides Verilog code for a complete Alpha 21264. We optimize the IVM compacting window design to make it fully synthesizable with our synthesis tools. At the same time, we also optimize the design to decrease area and increase the frequency achieved.

4.4 Synthesis Tools

Synthesis is the process of converting an HDL functional description into a gate-level net-list in the case of a standard-cell ASIC, or into a logic-block mapping in the case of an FPGA. For ASIC synthesis, we use Synopsys Design Compiler 2005.12 [26] with a 90nm standard cell library (ASIC) target; for FPGA synthesis, we use Synplify Pro 8.4 from Synplicity [27] and targeted the best Altera Stratix II device (EP2S180). We synthesize the SEED and IVM designs with the same optimizations and frequency targets. In the ASIC target, we assume a 15% clock skew with a load of 8 medium strength AND-gates for each output.

5. EVALUATION

The evaluation consists of two major components: cycle-accurate results assuming same frequency (Sections 5.1, 5.2, and 5.3) and synthesis results (Section 5.4). We first compare a SEED design with a conventional issue logic (Section 5.1), we then proceed with a more detailed analysis and sensitivity study (Section 5.2), and we finish the cycle-level evaluation by comparing SEED against a compacting issue queue with ideal banking and gating (Section 5.3).

5.1 Main Results

We start by comparing the performance of *SEED* against that of the conventional design *Base*. For all the configurations, we keep the same pipeline parameters as shown in Table 1 while varying the capacities of both issue logic designs. This clearly favors *Base* because the structure is assumed to scale to larger sizes while maintaining atomic wakeup-select without impacting cycle time.

Figure 4 shows the speedups of different hardware configurations over a baseline configuration *Base(32)* when running SPEC applications. Therefore, the first bar (*Base(32)*) in every application is always 1. We can see that both *Base* and *SEED* benefit from larger capacities in the issue queues (or depTable). For SPEC INT, *Base(64)* and *Base(128)* are 1.16 and 1.21 times faster than *Base(32)*, respectively. Figure 4 also shows that increasing the size of the issue buffer from 128 to 256 entries yields a meager 2% additional performance improvement. Therefore, we focus our discussion on configurations with 32, 64, and 128 entries in the following. We can observe that on average, for the same issue logic buffer size, *SEED* is slightly slower than the idealized *Base*: around 3% slower on average for SPEC INT as well as SPEC FP.

Figure 5 shows the comparison of power consumption. The power results include all the dynamic power consumed on-chip. For clarity, we only break down the results into three components: issue logic, register renaming logic (RAT), and the rest (Other). We show the energy in RAT because the management of the depTable involves an extension of the register alias table logic. From the figure, it is clear that *Base* does not scale well. The issue logic represents around 30% of the power consumption in *Base(32)*. At 20-28W, the absolute power of the issue logic in *Base(128)* is more than 3 times that in *Base(32)* and accounts for close to half of the power in *Base(128)*.

After replacing the conventional issue logic (*Base*) with a SEED issue logic, power consumption reduces drastically, especially for larger configurations. The issue logic and the RAT combined accounts for only 3-4W in the low-end *SEED(32)* and about 4-7W in *SEED(128)*. The relative contribution to the total chip power is about 17% and changes minimally from one configuration to another. This reduction in issue logic power translates to about 19% total chip power reduction at the small configurations (*SEED(32)* vs. *Base(32)*) and more than 39% in the large configurations (*SEED(128)* vs. *Base(128)*). The cost for this power reduction is 3% performance degradation.

In terms of scalability, from *Base(32)* to *Base(128)*, performance improves about 17% (FP) to 21% (INT) at a power increase of 75% (INT) to 85% (FP). For *SEED*, the same improvements comes at much lower increase in chip power: 34% (FP) to 37% (INT).

5.2 Analysis and Sensitivity Study

5.2.1 Port requirement

Every cycle, instructions are being dispatched into and read out of the depTable. In the base configuration, the depTable has only one port per bank. When the depTable bank is busy, it can potentially stall the fetch engine or delay the wakeup of dependent instructions. Therefore, every bank conflict potentially incurs some performance penalty. Adding multiple ports to the structure will reduce the frequency of conflicts and the resulting stalls. We performed simulation with single-, dual-, and quad-ported configurations. Increasing the number of ports from one to two results in less than 1% performance improvement on average and no more than 3% for any application. When further increasing the number of ports to four, we observe no performance impact. Clearly, a single-ported structure is sufficient. Having a single port is more energy-efficient and requires less area. It is a very important factor in the high energy efficiency seen above. According to CACTI [25], going from single- to dual-port, both area and energy per access double.

5.2.2 Number of sub-entries

Each entry of the depTable contains several sub-entries to hold dependent instructions. Figure 6 shows the impact of the number of sub-entries per depTable entry on performance. We evaluated configurations with 1, 2, 4, and 8 sub-entries. For brevity, we do not show the results of individual applications. Instead, we only show the average of the SPEC INT and the SPEC FP suites.

Not surprisingly, a configuration that allows the tracking

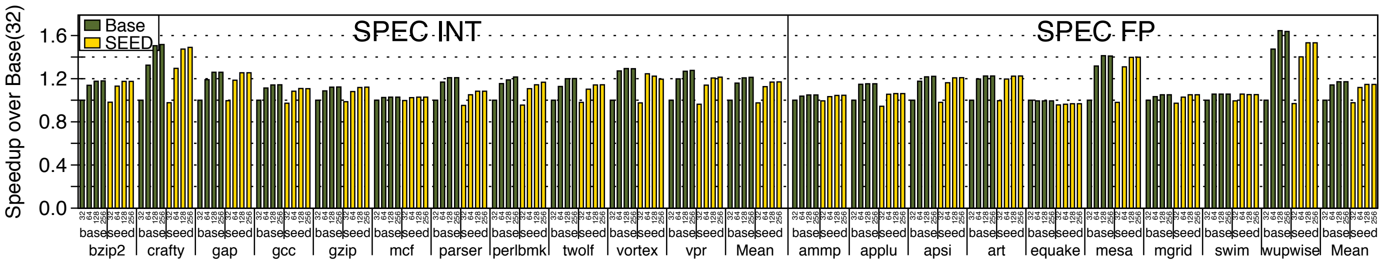


Figure 4. Normalized speedups for *SEED* and *Base* architectures.

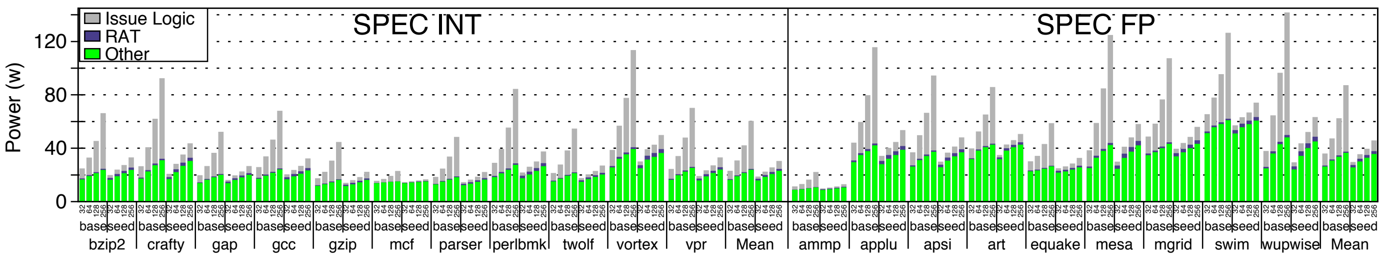


Figure 5. Power consumption for *SEED* and *Base* architectures.

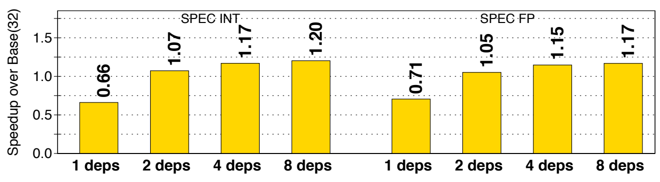


Figure 6. Performance impact of the number of sub-entries. The tested configuration is *SEED(128)*.

Bench	miss rate	% Predicted Corr.	Miss	Hit	Bench	miss rate	% Predicted Corr.	Miss	Hit
ammp	20.2	99.0	0.9	0.1	bzip2	2.7	93.3	6.1	0.6
applu	0.2	99.8	0.2	0.0	crafty	1.1	92.7	6.7	0.6
apsi	7.1	98.6	1.2	0.1	gcc	1.4	94.4	5.2	0.4
art	42.9	86.5	13.4	0.1	gap	1.6	94.4	5.2	0.4
equake	4.4	91.5	7.8	0.7	gzip	2.5	82.6	16.0	1.3
mesa	0.5	98.5	1.4	0.1	mcf	24.2	86.0	13.6	0.4
mgrid	2.0	85.4	13.1	1.6	parser	3.4	88.7	10.5	0.8
swim	9.7	38.9	57.8	3.3	perl.	0.4	99.1	0.8	0.1
wup.	0.8	89.9	9.2	0.8	twolf	7.7	91.2	8.5	0.3
					vortex	1.5	90.4	8.8	0.8
					vpr	6.0	89.2	10.4	0.4
FP	9.7	87.6	11.7	0.8	INT	4.8	91.1	8.3	0.6

Table 2. Load-hit predictor accuracy. For each application we show the L1 data cache miss rate, the percentage of loads whose hit-miss status is correctly predicted (Corr.), incorrectly predicted to be a miss (Miss), and incorrectly predicted to be a hit (Hit).

of only a single dependence per instruction performs very poorly (about a third slower than *Base(32)*). However, the performance level quickly rises as the number of sub-entries increases. With 8 sub-entries, the performance is indistinguishable from that of a configuration with infinite capacity in every entry. In our design, we choose 4 sub-entries.

5.2.3 Load-hit predictor

As explained in Section 3, we implement an L1 hit predictor to support load-hit speculation since we use a simple in-order issue buffer. Without an L1 hit predictor or an out-of-order issue buffer, we either have to conservatively wake up a load's dependents after the data is returned or optimistically wake them up early and run the risk of blocking the issue buffer when a load misses.

Table 2 shows the accuracy of the L1 hit predictor. We note that when a load is incorrectly predicted to be a miss, we waste the opportunity to execute some dependent instructions earlier, whereas when a load is incorrectly predicted to be a hit, we will stall the entire issue stage for a long time. This is what we want to minimize. We can see that this latter case is indeed infrequent using our predictor. No more than 1% of loads are incorrectly predicted to be a hit.

To further understand the actual performance impact of using a load-hit predictor, we compared our design with an idealized design using an oracle predictor. The performance difference between the two configurations is merely 0.6% for the integer applications and 0.3% for the floating-point applications. However, if we remove the load-hit predictor altogether and only wake up dependents when the load data is returned, we get a 7% and 3% slowdown for integer and floating-point applications, respectively. Therefore, we conclude that using a load-hit predictor and simply stalling the pipeline is a good design alternative that obtains most of the advantages of load-hit speculation without the necessity of supporting load-hit mis-speculation recovery.

5.2.4 Queuing policy

Table 3 shows some additional statistics about the configuration *SEED(128)*. Recall that when we dispatch an instruction, we queue it under a random source if there is more than one pending source. Later, this instruction may be re-dispatched. We show the frequency of re-dispatch. When a re-dispatched instruction can not be queued because the destination entry runs out of sub-entries, the instruction has to be buffered. A buffer overflow may incur a pipeline flush (Section 3). We show the frequency of such flushes. We see that in general, the percentage of instructions that need re-dispatch is low and that flushes are very rare except for a

Bench	IPC	% Inst. with				$\frac{cycles}{flush}$
		0deps	1dep	2deps	re-disp	
ammp	0.257	27.8	58.5	13.7	9.0	>50k
applu	1.583	36.0	38.6	25.5	14.6	211
apsi	1.263	35.6	47.3	17.1	7.4	2681
art	0.962	41.2	44.5	14.3	6.4	>50k
equake	0.705	31.5	39.8	28.7	15.1	>50k
mesa	1.846	38.6	45.2	16.2	8.2	716
mgrid	1.429	56.0	18.3	25.7	15.7	>50k
swim	1.457	45.1	26.8	28.1	13.4	>50k
wupwise	1.952	32.3	45.9	21.8	10.2	>50k
SPEC FP	1.272	38.2	40.5	21.2	11.1	
bzip2	0.945	21.8	58.8	19.4	9.0	34866
crafty	1.333	28.8	52.1	19.1	12.2	>50k
gap	0.716	28.1	55.0	16.9	5.5	5920
gcc	0.954	28.9	55.4	15.7	10.8	2876
gzip	0.632	25.0	60.2	14.8	8.7	>50k
mcf	0.176	23.2	66.4	10.4	6.4	>50k
parser	0.602	24.8	58.7	16.5	7.0	>50k
perlbmk	1.194	30.1	57.4	12.5	7.1	7171
twolf	0.707	25.3	57.9	16.7	8.0	>50k
vortex	1.623	32.4	55.3	12.3	6.1	276
vpr	0.949	27.6	56.0	16.4	9.4	>50k
SPEC INT	0.893	26.9	57.6	15.5	8.2	

Table 3. Execution statistics of *SEED(128)*. From left to right, the columns show the application, IPC, the percentage of dynamic instructions dispatched with 0, 1, 2 pending source registers, percentage of instructions re-dispatched, and the number of cycles between flushes due to special situation in re-dispatch handling.

few applications. For instance, for application *applu*, flushing is unusually frequent. This explains its poor performance shown in Figure 4.

We evaluated different queuing policies for instructions with more than one pending source during dispatch: our default policy that queues at a randomly selected source, queuing at both sources, and using a PC-based (16K-entry) predictor to predict which source to queue. We note that queuing at both sources obviates the re-dispatch logic: If a woken instruction has the speculative wakeup bit set and is not ready, we can simply drop it since it must be queued in the remaining parent instruction's depTable entry. In our simulations, we saw no significant performance difference (less than 1%) between different policies. Nevertheless, we found that queuing at both sources does incur a slightly higher energy overhead (about 4%).

5.2.5 In-order vs. out-of-order issue

After an instruction is woken up, it is sent to the in-order issue buffer. We experimented with an out-of-order issue buffer and found that the performance difference is an insignificant 1%. This suggests that the depTable is doing an adequate job reordering instructions in a way that lends to simple, in-order issue logic.

5.3 Ideal Banking and Gating

As previously shown (Section 5.1), the base configuration does not scale well in terms of energy. However, it is conceivable that some sort of adaptation can adjust the active size of the queue to achieve better energy-efficiency. There have been previous attempts to simplify the overall hardware requirements related with instruction scheduling, either by speculatively removing the select phase from the wakeup-select cycle [3], or by further reducing the number of

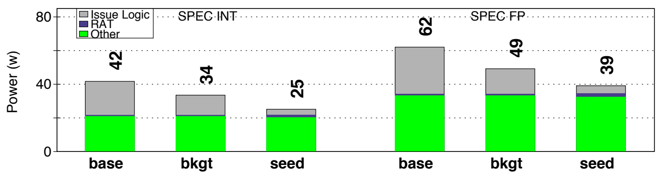


Figure 7. Power consumption with ideal baseline banking and gating (*bkgt*).

operands being considered in the dependency resolution for the wakeup [17]. To better understand the potential of our approach with respect to possible optimized conventional designs, we performed an study with an idealized configuration based on conventional issue queue where the banking of the queue can be dynamically adjusted and unused entries can be optimally gated (which we label *bkgt* for banked-gated queue). Essentially, every cycle, based on the simulated occupancy of the queue (say, n instructions), we assume the energy consumption of the best banked design of a n-entry issue queue. This, obviously, represents a unrealistically optimistic design. We see from the figure, that with such an idealized configuration, the power consumption indeed is significantly lower compared with *Base(128)*: about 20% chip power reduction is achieved. Yet, the SEED implementation is still about 20% more efficient. The primary reason for the difference is the that SEED uses single-ported indexed table rid of broadcast circuitry which is far more energy-efficient.

5.4 Synthesis Results

This section reports the synthesis results for *SEED* and *Base*. Previous sections on the evaluation do not use synthesis results because the energy models are based on CACTI and Wattch and we have assumed up to now that all the circuits cycle at the same frequency.

We synthesize *SEED* and *Base* designs for 90nm ASIC and for Altera Stratix-II EP2S180 90nm. (Section 4 has more details on the tools used.) The *SEED* Verilog implementation includes the depTable, the token queue, the issue buffer, and the extension to the rename table as shown in Figure 2. The *Base* implementation is based on IVM as explained in Section 4. Figure 8 shows the synthesis results for frequency (a,b), dynamic (d) and static (e) power consumption, and area (c). The results show four different *SEED* designs (16, 32, 64, 128, and 256) and three *Base* designs (16, 32, and 64). We do not include bigger *Base* designs because Synopsys requirements are too high. While *Base(32)* requires 4GB of memory and 3 days of CPU time, *Base(64)* requires 9GBs of memory and over 7 days of CPU time. Bigger designs are not practical without partitioning which would affect synthesis results, so we only synthesize up to 64 entries for *Base*.

5.4.1 Frequency

Figure 8-(a) shows the frequency obtained for ASIC and Figure 8-(b) shows the frequency for FPGA. The first thing to observe is that all *SEED* designs are over two times faster than *Base*. A *SEED(64)* achieves 485MHz while *Base(64)* only achieves 173MHz. *SEED* also achieves higher frequency for FPGAs. Even for the worst FPGA case, *SEED(64)* is 2.2 times faster than *Base(64)*. The *SEED(128)* FPGA frequency improves with respect to *SEED(64)* due to better placement performed by the tool. Although it may seem

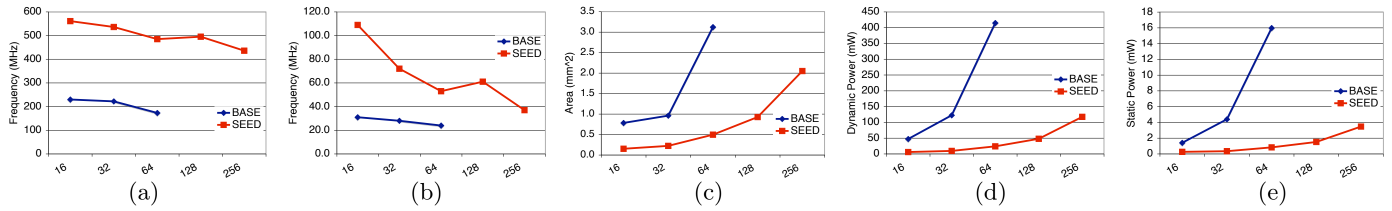


Figure 8. Maximum frequency for ASIC (a) and FPGA (b), ASIC area (c), and dynamic (d) and static (e) ASIC power consumption synthesis results for *SEED* and *Base* architectures. ASIC uses a 90nm standard cell technology, and FPGA targets the Altera Stratix-II 90nm. Synopsys reports static and dynamic power consumption at the achieved frequency, we normalize it to 200MHz so that it can be compared across synthesis results.

that *SEED* has worse scalability on FPGAs, the reason is that Synplicity uses memory banks for depTables. As more depTable entries are needed, they need to be placed farther and farther apart and the wiring overhead increases substantially. *SEED(128)* is an interesting point on Figure 8-(b). It achieves higher frequency than *SEED(64)*. The reason is that Synplicity Pro finds a better placement, and therefore shorter wires, with 128 entries. Manual placement removes this opportunity, but we leave the tool to choose the best placement automatically.

A possible concern on the validity of the results is the low frequency achieved for the *Base* design. After all, the original Alpha 21264 achieves 550MHz with a 350nm technology [8]. If we apply linear scaling, such a design should achieve 2100MHz instead of 225MHz achieved with our synthesis results. There are four factors for such a difference: (1) the original alpha uses full-custom dynamic logic while we use standard cell ASIC; (2) wires do not scale linearly; (3) we use a 15% clock skew while the original alpha design achieved a 5% clock skew; (4) our designs use flip-flops while the original Alpha 21264 uses dual clock domino logic. When clock skew difference is factored in, the original Alpha 21264 issue logic is about 8 times faster than our synthesized *Base* design. This is in line with the expectation: It is widely known that full-custom design results in a circuit that is much faster than its standard-cell ASIC counterpart. Indeed, Keutzer [5] reports that ASICs are between 6 to 8 times slower than full-custom design for the same technology target. As a result, we expect that a full-custom *SEED* implementation would maintain the frequency advantage over *Base*.

After analyzing the synthesis results, we found two major reasons why *SEED* is faster than *Base*: (1) *SEED* does not use heavily ported structures like *Base*; (2) *SEED* breaks the wakeup-select in two cycles, wakeups happen back to back so the IPC impact is nearly negligible as shown in Section 5.1.

5.4.2 Power consumption

The dynamic and static power consumption are shown in Figure 8-(d) and Figure 8-(e) respectively. In both plots, the power is shown in mW for a 90nm ASIC synthesis. Synopsys reports the dynamic power consumption at the target frequency. Since each synthesis achieves a different frequency target, we normalize all the dynamic power consumption to 200MHz.

As expected, reducing the number of ports has a substantial reduction on power consumption and significantly improves design scalability. *SEED* outperforms *Base* even for the smallest configuration where the RAT overhead is more significant. *SEED(32)* is over ten times more power efficient than *Base(32)*. The difference gets even more substantial as we increase the size. Going from *SEED(16)* to *SEED(64)*, the power consumption increases 4 times as the size quadruples. The increase is linear because the overhead for the RAT stays constant. In contrast, the same size increase in *Base* results in an 8.7 times increase in dynamic power consumption. These results are achieved without considering clock gating. In other words, all the banks are active all the time. Clock gating is a big opportunity for SEED, but not so much for the *Base* design (IVM) because it is not banked.

Figure 8-(e) shows that *SEED* also has lower static power consumption as well. Synopsys reports 5.4 times less static power consumption for the smallest configuration (16-entries). *SEED* has a sub-linear leakage increase, going from *SEED(16)* to *SEED(64)* the static power consumption increases 3.3 times. The reason is that the RAT overhead stays nearly constant as we add more entries. In comparison, quadrupling the number of entries in *Base* increases static power consumption by 11 times.

5.4.3 Area requirement

The depTable used in our SEED implementation has 1 read and 1 write port (1-rd/1-wr) SRAM structure. Using a 1-rd/1-wr port structure not only provides significant power savings but also reduces silicon area. Figure 8-(c) shows the area reported by Synopsys for 90nm technology. As reported for power and frequency, *SEED* outperforms *Base* in all the configurations. While *Base(32)* requires $0.96mm^2$, *SEED(32)* requires $0.23mm^2$. Looking at the Synopsys results, we can observe that the area increase is not linear as we increase the number of entries. For example, going from *Base(16)* to *Base(32)* only has a 1.23X area increase, while going from *Base(32)* to *Base(64)* has a 3.25X area increase.

As a reference, we apply linear scaling to the original Alpha 21264 area report [8]. While *Base(16)* has $0.78mm^2$ and *Base(32)* has $0.96mm^2$, the original Alpha 21264 with 20 entries should require around $0.66mm^2$. Therefore, we believe that the reported area estimations are consistent with previous results.

6. RELATED WORK

As discussed before, our SEED design has three main features to address the scalability problem of conventional instruction scheduling: indexing-based wakeup, decoupled wakeup and select, and a design that supports graceful misprediction recovery or replay. While many prior proposals also address the same problem, they tend to offer only a subset of the features.

Indexing-based wakeup is proposed in a very early design by Weiss and Smith to reduce complexity by removing the tag broadcast in the Tomasulo's algorithm [31]. A similar idea is exploited in [4] where a register ID-indexed "first-use table" records the identity of only the first consumer instruction, which can be woken up through indexing when the producer is selected. In this design, wakeup is triggered by select. In contrast, our depTable stores all instructions and our wakeup is triggered by wakeup, reducing the difficulty to support back-to-back execution of dependent instructions.

Indexed tables are used in a class of eager schedulers where the execution schedule is determined after decode and rename. In [4], Canal and Gonzalez explored the idea of a distance-based scheduling, where instructions are scheduled into a table based on the latencies and the schedule of their parents. This concept is also explored by Michaud and Seznec in [19] where instructions are pre-woken up into a buffer and only moved to a much smaller issue queue when they reach the head of the pre-wakeup buffer. Rassch *et al.* proposed an implementation in which both pre-wakeup and dynamic scheduling are used [22]. Instructions are promoted from a series of issue window segments before entered into the final segment, which allows issue. The Cyclone scheduler also relies on pre-wakeup [7]. However, even the main issue window is broadcast-free and issues instructions based on the predicted ready time. Thus, a sanity check is required, and the instruction is re-deposited into the pre-wakeup table if the sources are not ready.

As discussed before, although an eager scheduler avoids the challenge of implementing atomic wakeup-select loop, it is not without cost. In addition to requiring non-trivial latency prediction and calculation circuit and schedule tracking logic, an eager scheduler handles branch misprediction and other selective squash inefficiently. As instructions that need to be discarded are widely commingled with other instructions, they have to be individually screened and discarded, often at the end of propagation stages inside the scheduler. This unnecessarily wastes capacity of the queues and, more importantly, energy. Furthermore, similar to statically-scheduled VLIW machines, although to a lesser degree, it is challenging to handle variable-latency instructions (including loads) efficiently in an eager scheduler. In SEED, due to lazy wakeup, and the fact that depTable entries for wrong-path instructions are reclaimed *en masse*, these disadvantages are largely avoided.

Lebeck *et al.* proposed a two-level instruction (issue) window design [18]. When a long-latency instruction (*e.g.*, a load that misses in the L2 cache) is detected, all dependents are removed from the level-1 window to a waiting buffer to make room for independent instructions. They are reinserted when the long-latency event finishes. Our depTable naturally holds a large amount of instructions, including dependents of long-latency instructions. There is no need to actively remove and reinsert dependents.

Brown *et al.* proposed an optimization that speculatively removes select from the wakeup-select loop [3]. This design is motivated by the observation that usually no more than one instruction becomes ready every cycle. Special logic is designed to detect and resolve conflicts. In contrast, we decouple wakeup and select (issue) in a straightforward and *non-speculative* fashion. Without the need to handle misspeculation recovery and to minimize its occurrence, our implementation is thus simpler.

While the above-mentioned work tries to make the issue queues more scalable, there are a number of other studies that focus primarily on reducing the energy consumption of a broadcast-based issue logic [9, 12, 15, 17, 21, 33]. Another related work focuses on steering the instructions early in the pipeline to different smaller queues or windows to reduce complexity [20]. Goshima *et al.* proposed a circuit-level technique that allows fast wakeup in a small portion of the buffering structure and slower wakeup in the remainder [13]. Finally, a large body of work explored the scalability of other related microarchitectural resources such as register files (*e.g.*, [1, 29]) and the load-store queue (*e.g.*, [10, 11, 16, 24]).

7. CONCLUSIONS

With relatively long memory latencies seen in high-frequency designs, processors are buffering more in-flight instructions to maintain high performance. Power constraints are quickly becoming the dominant barrier to higher levels of performance in microarchitectures. Therefore, conventional designs in which energy efficiency is not a first-class design consideration will no longer be viable. In this paper, we have presented a novel dependence tracking and instruction issue mechanism called SEED that is effective, energy-efficient, scalable, and has low complexity.

SEED uses broadcast-free, indexing-only tables to keep track of dependences and rearranges instructions in an optimal order to feed into a simple, in-order issue buffer. This design has a number of advantages. Compared to a conventional issue logic, our wakeup logic is driven by what has been woken up. This takes the complex select logic out of the critical loop, making it much more scalable. Furthermore, the wakeup process is non-speculative. Without the need to recover from mis-speculation, the design complexity is very low. Compared to designs that eagerly schedules incoming instructions, our buffering structure and organization of instructions obviates complex pre-scheduling logic and lend themselves well to handling variable-latency instructions and recovering efficiently from common branch mispredictions.

Simulation-based evaluations show that the *SEED* design is effective in exploiting ILP. Compared to a processor using conventional issue logic with the same instruction capacity, a SEED-based processor suffers only a small 3% performance degradation but occupies a much smaller area and enjoys a 19-39% power savings depending on the exact capacity.

In addition, the proposed SEED issue logic is synthesized and compared against a traditional compacting issue queue. The synthesis results show that *SEED* can cycle at twice the frequency of the conventional design. More importantly, the dynamic power consumption of a conventional issue logic is over ten times higher than that of a SEED logic with equivalent capacity and five times higher in static power consumption.

8. ACKNOWLEDGMENTS

We would like to thank Carlos Andrés Cabrera for his help on the Verilog implementation of SEED and the anonymous reviewers for their feedback. This work was supported in part by the National Science Foundation under grants 0546819 and 0509270, Special Research Grant from the University of California, Santa Cruz, and gifts from SUN.

References

[1] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Reducing the Complexity of the Register File in Dynamic Superscalar Processors. In *International Symposium on Microarchitecture*, pages 237–248, Dec. 2001.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[3] M. Brown, J. Stark, and Y. Patt. Select-Free Instruction Scheduling Logic. In *International Symposium on Microarchitecture*, pages 204–213, Dec. 2001.

[4] R. Canal and A. Gonzalez. A Low-Complexity Issue Logic. In *International Conference on Supercomputing*, pages 327–335, May 2000.

[5] D. G. Chinnery and K. Keutzer. Closing the gap between asic and custom: An asic perspective. In *Design Automation Conference*, pages 637–642, June 2000.

[6] Compaq Computer Corporation. *Alpha 21264/EV6 Microprocessor Hardware Reference Manual*, Sept. 2000. Order number: DS-0027B-TE.

[7] D. Ernst, A. Hamel, and T. Austin. Cyclone: a Broadcast-Free Dynamic Instruction Scheduler with Selective Replay. In *International Symposium on Computer Architecture*, pages 253–263, June 2003.

[8] J. A. Farrell and T. C. Fischer. Issue logic for a 600-MHz Out-of-Order Execution Microprocessor. *IEEE Journal of Solid-State Circuits*, 33(5):707–712, May 1998.

[9] D. Folegnani and A. González. Energy-Effective Issue Logic. In *International Symposium on Computer Architecture*, pages 230–239, June–July 2001.

[10] A. Garg, F. Castro, M. Huang, L. Pinuel, D. Chaver, and M. Prieto. Substituting Associative Load Queue with Simple Hash Table in Out-of-Order Microprocessors. In *International Symposium on Low-Power Electronics and Design*, Oct. 2006.

[11] A. Garg, M. Rashid, and M. Huang. Slackened Memory Dependence Enforcement: Combining Opportunistic Forwarding with Decoupled Verification. In *International Symposium on Computer Architecture*, pages 142–153, June 2006.

[12] K. Ghose. Reducing Energy Requirements for Instruction Issue and Dispatch in Superscalar Microprocessors. In *International Symposium on Low-Power Electronics and Design*, pages 231–233, July 2000.

[13] M. Goshima, K. Nishino, T. Kitamura, Y. Nakashima, S. Tomita, and S. Mori. A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors. In *International Symposium on Microarchitecture*, pages 225–236, Dec. 2001.

[14] M. Gowan, L. Biro, and D. Jackson. Power Considerations in the Design of the Alpha 21264 Microprocessor. In *Design Automation Conference*, pages 726–731, June 1998.

[15] M. Huang, J. Renau, and J. Torrellas. Energy-Efficient Hybrid Wakeup Logic. In *International Symposium on Low-Power Electronics and Design*, pages 196–201, Aug. 2002.

[16] R. Huang, A. Garg, and M. Huang. Software-Hardware Cooperative Memory Disambiguation. In *International Symposium on High-Performance Computer Architecture*, pages 248–257, Feb. 2006.

[17] I. Kim and M. Lipasti. Half-Price Architecture. In *International Symposium on Computer Architecture*, pages 28–38, June 2003.

[18] A. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A Large, Fast Instruction Window for Tolerating Cache Misses. In *International Symposium on Computer Architecture*, pages 59–70, May 2002.

[19] P. Michaud and A. Seznec. Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors. In *International Symposium on High-Performance Computer Architecture*, pages 27–38, Jan. 2001.

[20] S. Palacharla, N. Jouppi, and J. Smith. Complexity-Effective Superscalar Processors. In *International Symposium on Computer Architecture*, pages 206–218, June 1997.

[21] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources. In *International Symposium on Microarchitecture*, pages 90–101, Dec. 2001.

[22] S. Raasch, N. Binkert, and S. Reinhardt. A Scalable Instruction Queue Design Using Dependence Chains. In *International Symposium on Computer Architecture*, pages 318–329, May 2002.

[23] J. Renau et al. SESC simulator, January 2005. http://sesc.sourceforge.net.

[24] S. Sethumadhavan, R. Desikan, D. Burger, C. Moore, and S. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *International Symposium on Microarchitecture*, pages 399–410, Dec. 2003.

[25] P. Shivakumar and N. Jouppi. CACTI 3.0: An integrated cache timing, power and area model. Technical Report 2001/2, Compaq Computer Corporation, August 2001.

[26] Synopsys Inc. Design Compiler Product Information, 2005. http://www.synopsys.com.

[27] Synplicity Inc. SynplifyPro Product Information, 2005. http://www.synplicity.com.

[28] J. Tendler, J. Dodson, J. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, Jan. 2002.

[29] J. Tseng and K. Asanovic. Banked Multiported Register Files for High-Frequency Superscalar Microprocessors. In *International Symposium on Computer Architecture*, pages 62–71, June 2003.

[30] N. Wang, J. Quek, T. Rafacz, and S. Patel. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. In *International Conference on Dependable Systems and Networks*. IEEE Computer Society, Jun 2004.

[31] S. Weiss and J. Smith. Instruction Issue Logic in Pipelined Supercomputers. *IEEE Transactions on Computers*, 33(11):1013–1022, Nov. 1984.

[32] K. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, Apr. 1996.

[33] V. Zyuban and P. Kogge. Optimization of High-Performance Superscalar Architectures for Energy Efficiency. In *International Symposium on Low-Power Electronics and Design*, pages 84–89, July 2000.