A Large, Fast Instruction Window for Tolerating Cache Misses

Alvin R. Lebeck† Jinson Koppanalil‡ Tong Li† Jaidev Patwardhan† Eric Rotenberg‡

† Department of Computer Science
Duke University
Durham, NC 27708-0129 USA
{alvy,tongli,jaidev}@cs.duke.edu

‡ Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC 27695-7914 USA
{jjkoppan,ericro}@ece.ncsu.edu

Abstract

Instruction window size is an important design parameter for many modern processors. Large instruction windows offer the potential advantage of exposing large amounts of instruction level parallelism. Unfortunately, naively scaling conventional window designs can significantly degrade clock cycle time, undermining the benefits of increased parallelism.

This paper presents a new instruction window design targeted at achieving the latency tolerance of large windows with the clock cycle time of small windows. The key observation is that instructions dependent on a long latency operation (e.g., cache miss) cannot execute until that source operation completes. These instructions are moved out of the conventional, small, issue queue to a much larger waiting instruction buffer *(WIB). When the long latency operation completes, the instructions are reinserted into the issue queue. In this paper, we focus specifically on load cache misses and their dependent instructions. Simulations reveal that, for an 8-way processor, a 2K-entry WIB with a 32-entry issue queue can achieve speedups of 20%, 84%, and 50% over a conventional 32-entry issue queue for a subset of the SPEC CINT2000, SPEC CFP2000, and Olden benchmarks, respectively.*

1 Introduction

Many of today's microprocessors achieve high performance by combining high clock rates with the ability to dynamically process multiple instructions per cycle. Unfortunately, these two important components of performance are often at odds with one another. For example, small hardware structures are usually required to achieve short clock cycle times, while larger structures are often necessary to identify and exploit instruction level parallelism (ILP).

A particularly important structure is the issue window, which is examined each cycle to choose ready instructions for execution. A larger window can often expose a larger number of independent instructions that can execute out-of-order. Unfortunately, the size of the issue window is limited due to strict cycle time constraints. This conflict between cycle time and dynamically exploiting parallelism is exacerbated by long latency operations such as data cache misses or even cross-chip communication [1, 22]. The challenge is to develop microarchitectures that permit both short cycle times and large instruction windows.

This paper introduces a new microarchitecture that reconciles the competing goals of short cycle times and large instruction windows. We observe that instructions dependent on long latency operations cannot execute until the long latency operation completes. This allows us to separate instructions into those that will execute in the near future and those that will execute in the distant future. The key to our design is that the entire chain of instructions dependent on a long latency operation is removed from the issue window, placed in a *waiting instruction buffer* (WIB), and reinserted after the long latency operation completes. Furthermore, since all instructions in the dependence chain are candidates for reinsertion into the issue window, we only need to implement select logic rather than the full wakeup-select required by a conventional issue window. Tracking true dependencies (as done by the wakeup logic) is handled by the issue window when the instructions are reinserted.

In this paper we focus on tolerating data cache misses, however we believe our technique could be extended to other operations where latency is difficult to determine at compile time. Specifically, our goal is to explore the design of a microarchitecture with a large enough "effective" window to tolerate DRAM accesses. We leverage existing techniques to provide a large register file [13, 34] and assume that a large active list[1] is possible since it is not on the critical path [4] and techniques exist for keeping the active list

[1] By active list, we refer to the hardware unit that maintains the state of in-flight instructions, often called the reorder buffer.

large while using relatively small hardware structures [31].

We explore several aspects of WIB design, including: detecting instructions dependent on long latency operations, inserting instructions into the WIB, banked vs. non-banked organization, policies for selecting among eligible instructions to reinsert into the issue window, and total capacity. For an 8-way processor, we compare the committed instructions per cycle (IPC) of a WIB-based design that has a 32-entry issue window, a 2048-entry banked WIB, and two-level register files (128 L1/2048 L2) to a conventional 32-entry issue window with single-level register files (128 registers). These simulations show WIB speedups over the conventional design of 20% for SPEC CINT2000, 84% for SPEC CFP2000, and 50% for Olden. These speedups are a significant fraction of those achieved with a 2048-entry conventional issue window (35%, 140%, and 103%), even ignoring clock cycle time effects.

The remainder of this paper is organized as follows. Section 2 provides background and motivation for this work. Our design is presented in Section 3 and we evalute its performance in Section 4. Section 5 discusses related work and Section 6 summarizes this work and presents future directions.

2 Background and Motivation

2.1 Background

Superscalar processors maximize serial program performance by issuing multiple instructions per cycle. One of the most important aspects of these systems is identifying independent instructions that can execute in parallel. To identify and exploit instruction level parallelism (ILP), most of today's processors employ dynamic scheduling, branch prediction, and speculative execution. Dynamic scheduling is an all hardware technique for identifying and issuing multiple independent instructions in a single cycle [32]. The hardware looks ahead by fetching instructions into a buffer—*called a window*—from which it selects instructions to issue to the functional units. Instructions are issued only when all their operands are available, and independent instructions can execute out-of-order. Results of instructions executed out-of-order are committed to the architectural state in program order. In other words, although instructions within the window execute out-of-order, the window entries are managed as a FIFO where instructions enter and depart in program order.

The above simplified design assumes that all instructions in the window can be examined and selected for execution. We note that it is possible to separate the FIFO management (active list or reorder buffer) from the independent instruction identification (issue queue) as described below. Regardless, there is a conflict between increasing the window (issue queue) size to expose more ILP and keeping clock cycle time low by using small structures [1, 22]. Historically, smaller windows have dominated designs resulting in higher clock rates. Unfortunately, a small window can quickly fill up when there is a long latency operation.

In particular, consider a long latency cache miss serviced from main memory. This latency can be so large, that by the time the load reaches the head of the window, the data still has not arrived from memory. Unfortunately, this significantly degrades performance since the window does not contain any executing instructions: instructions in the load's dependence chain are stalled, and instructions independent of the load are finished, waiting to commit in program order. The only way to make progress is to bring new instructions into the window. This can be accomplished by using a larger window.

2.2 Limit Study

The remainder of this section evaluates the effect of window size on program performance, ignoring clock cycle time effects. The goal is to determine the potential performance improvement that could be achieved by large instruction windows. We begin with a description of our processor model. This is followed by a short discussion of its performance for various instruction window sizes.

2.2.1 Methodology

For this study, we use a modified version of SimpleScalar (version 3.0b) [8] with the SPEC CPU2000 [17] and Olden [11] benchmark suites. Our SPEC CPU2000 benchmarks are pre-compiled binaries obtained from the SimpleScalar developers [33] that were generated with compiler flags as suggested at www.spec.org and the Olden binaries were generated with the Alpha compiler (cc) using optimization flag -O2. The SPEC benchmarks operate on their reference data sets and for the subset of the Olden benchmarks we use, the inputs are: `em3d` 20,000 nodes, arity 10; `mst` 1024 nodes; `perimeter` 4Kx4K image; `treeadd` 20 levels. We omit several benchmarks either because the L1 data cache miss ratios are below 1% or their IPCs are unreasonably low (`health` and `ammp` are both less than 0.1) for our base configuration.

Our processor design is loosely based on the Alpha 21264 microarchitecture [12, 14, 19]. We use the same seven stage pipeline, including speculative load execution and load-store wait prediction. We do not model the clustered design of the 21264. Instead, we assume a single integer issue queue that can issue up to 8 instructions per cycle and a single floating point issue queue that can issue up to 4 instructions per cycle. Table 1 lists the various parameters for our base machine. Note that both integer and floating

Active List	128, 128 Int Regs, 128 FP Regs
Load/Store Queue	64 Load, 64 Store
Issue Queue	32 Integer, 32 Floating Point
Issue Width	12 (8 Integer, 4 Floating Point)
Decode Width	8
Commit Width	8
Instruction Fetch Queue	8
Functional Units	8 integer ALUs (1-cycle), 2 integer multipliers (7-cycle), 4 FP adders (4-cycle), 2 FP multipliers (4-cycle), 2 FP dividers (nonpipelined, 12-cycle), 2 FP square root units (nonpipelined, 24-cycle)
Branch Prediction	Bimodal & two-level adaptive combined, with speculative update, 2-cycle penalty for direct jumps missed in BTB, 9-cycle for others
Store-Wait Table	2048 entries, bits cleared every 32768 cycles
L1 Data Cache	32 KB, 4 Way
L1 Inst Cache	32 KB, 4 Way
L1 Latency	2 Cycles
L2 Unified Cache	256 KB, 4 Way
L2 Latency	10 Cycles
Memory Latency	250 Cycles
TLB	128-entry, 4-way associative, 4 KB page size, 30-cycle penalty

Table 1. Base Configuration

point register files are as large as the active list. For the remainder of this paper we state a single value for the active list/register file size, this value applies to both the integer and floating point register files.

The simulator was modified to support speculative update of branch history with history-based fixup and return-address-stack repair with the pointer-and-data fixup mechanism [26, 27]. We also modified the simulator to warm up the instruction and data caches during an initial fast forward phase. For the SPEC benchmarks we skip the first four hundred million instructions, and then execute the next one hundred million instructions with the detailed performance simulator. The Olden benchmarks execute for 400M instructions or until completion. This approach is used throughout this paper. We note that our results are qualitatively similar when using a different instruction execution window [24].

2.2.2 Varying Window Size

We performed simulations varying the issue queue size, from 32 (the base) in powers of 2, up to 4096. For issue queue sizes of 32, 64, and 128 we keep the active list fixed at 128 entries. For the remaining configurations, the active list, register files and issue queue are all equal size. The load and store queues are always set to one half the active list size, and are the only limit on the number of outstanding requests unless otherwise stated. Figure 1 shows the committed instructions per cycle (IPC) of various window sizes normalized to the base 32-entry configuration ($Speedup = IPC_{new}/IPC_{old}$) for the SPEC integer, floating point, and Olden benchmarks. Absolute IPC values for the base machine are provided in Section 4, the goal here is to examine the relative effects of larger instruction windows.

These simulations show there is an initial boost in the IPC as window size increases, up to 2K, for all three sets of benchmarks. With the exception of `mst`, the effect plateaus beyond 2K entries, with IPC increasing only slightly. This matches our intuition since during a 250 cycle memory latency 2000 instructions can be fetched in our 8-way processor. Larger instruction windows beyond 2K provide only minimal benefits. Many floating point benchmarks achieve speedups over 2, with `art` achieving a speedup over 5 for the 2K window. This speedup is because the larger window can unroll loops many times, allowing overlap of many cache misses. A similar phenomenon occurs for `mst`.

The above results motivate the desire to create large instruction windows. The challenge for architects is to accomplish this without significant impact on clock cycle time. The next section presents our proposed solution.

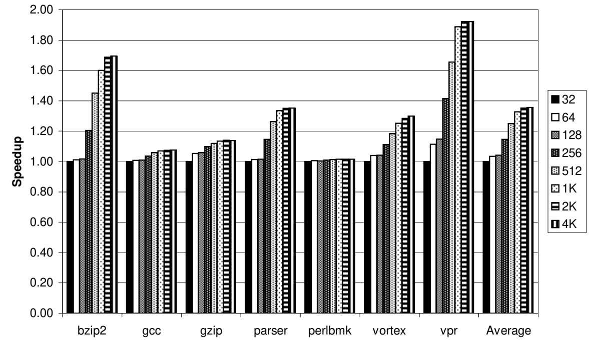
3 A Large Window Design

This section presents our technique for providing a large instruction window while maintaining the advantages of small structures on the critical path. We begin with an overview to convey the intuition behind the design. This is followed by a detailed description of our particular design. We conclude this section with a discussion of various design issues and alternative implementations.

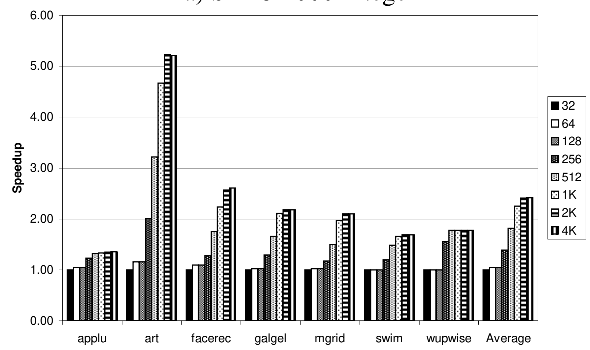
3.1 Overview

In our base microarchitecture, only those instructions in the issue queue are examined for potential execution. The active list has a larger number of entries than the issue queue (128 vs. 32), allowing completed but not yet committed instructions to release their issue queue entries. Since the active list is not on the critical path [4], we assume that we can increase its size without affecting clock cycle time. Nonetheless, in the face of long latency operations, the issue queue could fill with instructions waiting for their operands and stall further execution.

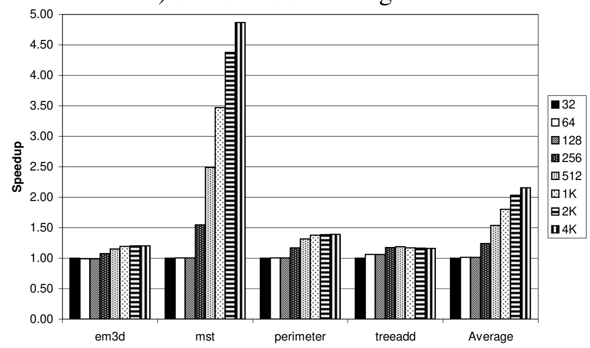
We make the observation that instructions dependent on long latency operations cannot execute until the long latency operation completes and thus do not need to be exam-



a) SPEC 2000 Integer



b) SPEC 2000 Floating Point



c) Olden

Figure 1. Large Window Performance

ined by the wakeup-select logic on the critical path. We note this same observation is exploited by Palacharla, et. al [22] and their technique of examining only the head of the issue queues. However, the goal of our design is to remove these waiting instructions from the issue queue and place them in a *waiting instruction buffer* (WIB). When the long latency operation completes, the instructions are moved back into the issue queue for execution. In this design, instructions remain in the issue queue for a very short time. They either execute properly or they are removed due to dependence on a long latency operation.

For this paper we focus specifically on instructions in the dependence chain of `load` cache misses. However, we believe our technique could be extended to other types of long latency operations. Figure 2 shows the pipeline for a WIB-based microarchitecture, based on the 21264 with two-level register files (described later).

The fetch stage includes the I-cache, branch prediction and the instruction fetch queue. The slot stage directs instructions to the integer or floating point pipeline based on their type. The instructions then go through register rename before entering the issue queue. Instructions are selected from the issue queue either to proceed with the register read, execution and memory/writeback stages or to move into the WIB during the register read stage. Once in the WIB, instructions wait for the specific cache miss they depend on to complete. When this occurs, the instructions are reinserted into the issue queue and repeat the wakeup-select process, possibly moving back into the WIB if they are dependent on another cache miss. The remainder of this section provides details on WIB operation and organization.

3.2 Detecting Dependent Instructions

An important component of our design is the ability to identify all instructions in the dependence chain of a load cache miss. To achieve this we leverage the existing issue queue wakeup-select logic. Under normal execution, the wakeup-select logic determines if an instruction is ready for execution (i.e., has all its operands available) and selects a subset of the ready instructions according to the issue constraints (e.g., structural hazards or age of instructions).

To leverage this logic we add an additional signal—called the *wait bit*—that indicates the particular source operand (i.e., input register value) is "pretend ready". This signal is very similar to the ready bit used to synchronize true dependencies. It differs only in that it is used to indicate the particular source operand will not be available for an extended period of time. An instruction is considered pretend ready if one or more of its operands are pretend ready and all the other operands are truly ready. Pretend ready instructions participate in the normal issue request as if they were truly ready. When it is issued, instead of being sent to the functional unit, the pretend ready instruction is placed in the WIB and its issue queue entry is subsequently freed by the issue logic as though it actually executed. We note that a potential optimization to our scheme would consider an instruction pretend ready as soon as one of its operands is pretend ready. This would allow instructions to be moved to the WIB earlier, thus further reducing pressure on the issue queue resources.

In our implementation, the wait bit of a physical register is initially set by a load cache miss. Dependent instructions observe this wait bit, are removed from the issue queue, and set the wait bit of their destination registers. This causes their dependent instructions to be removed from the issue queue and set the corresponding wait bits of their result

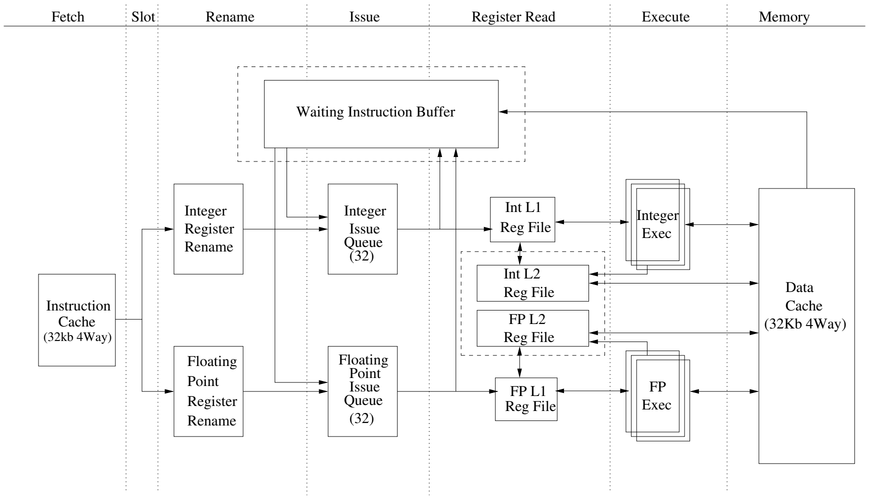


Figure 2. WIB-based Microarchitecture

registers. Therefore, all instructions directly or indirectly dependent on the load are identified and removed from the issue queue. The load miss signal is already generated in the Alpha 21264 since load instructions are speculatively assumed to hit in the cache allowing the load and dependent instructions to execute in consecutive cycles. In the case of a cache miss in the Alpha, the dependent instructions are retained in the issue queue until the load completes. In our case, these instructions move to the WIB.

An instruction might enter the issue queue after the instructions producing its operands have exited the issue queue. The producer instructions could have either executed properly and the source operand is available or they could be in the WIB and this instruction should eventually be moved to the WIB. Therefore, wait bits must be available wherever conventional ready bits are available. In this case, during register rename. Note that it may be possible to steer instructions to the WIB after the rename stage and before the issue stage, we plan to investigate this as future work. Our current design does not implement this, instead each instruction enters the issue queue and then is moved to the WIB if necessary.

3.3 The Waiting Instruction Buffer

The WIB contains all instructions directly or indirectly dependent on a load cache miss. The WIB must be designed to satisfy several important criteria. First, it must contain and differentiate between the dependent instructions of individual outstanding loads. Second, it must allow individual instructions to be dependent on multiple outstanding loads. Finally, it must permit fast "squashing" when a branch mispredict or exception occurs.

To satisfy these requirements, we designed the WIB to operate in conjunction with the active list. Every instruction in the active list is allocated an entry in the WIB. Although this may allocate entries in the WIB that are never dependent on a load miss, it simplifies squashing on mispredicts. Whenever active list entries are added or removed, the corresponding operations are performed on the WIB. This means WIB entries are allocated in program order.

To link WIB entries to load misses we use a bit-vector to indicate which WIB locations are dependent on a specific load. When an instruction is moved to the WIB, the appropriate bit is set. The bit-vectors are arranged in a two dimensional array. Each column is the bit-vector for a load cache miss. Bit-vectors are allocated when a load miss is detected, therefore for each outstanding load miss we store a pointer to its corresponding bit-vector. Note that the number of bit-vectors is bounded by the number of outstanding load misses. However, it is possible to have fewer bit-vectors than outstanding misses.

To link instructions with a specific load, we augment the operand wait bits with an index into the bit-vector table corresponding to the load cache miss this instruction is dependent on. In the case where an instruction is dependent on multiple outstanding loads, we use a simple fixed ordering policy to examine the source operand wait bits and store the instruction in the WIB with the first outstanding load encountered. This requires propagating the bit-vector index with the wait bits as described above. It is possible to store the bit-vector index in the physical register, since that space is available. However, this requires instructions that are moved into the WIB to consume register ports. To reduce register pressure we assume the bit-vector index is stored in a separate structure with the wait bits.

Instructions in the WIB are reinserted in the issue queue when the corresponding load miss is resolved. Reinsertion shares the same bandwidth (in our case, 8 instructions per cycle) with those newly arrived instructions that are decoded and dispatched to the issue queue. The dispatch logic is modified to give priority to the instructions reinserted from the WIB to ensure forward progress.

Note that some of the instructions reinserted in the issue queue by the completion of one load may be dependent on another outstanding load. The issue queue logic detects that one of the instruction's remaining operands is unavailable, due to a load miss, in the same way it detected the first load dependence. The instruction then sets the appropriate bit in the new load's bit-vector, and is removed from the issue queue. This is a fundamental difference between the WIB and simply scaling the issue queue to larger entries. The larger queue issues instructions only once, when all their operands are available. In contrast, our technique could move an instruction between the issue queue and WIB many times. In the worst case, all active instructions are dependent on a single outstanding load. This requires each bit-vector to cover the entire active list.

The number of entries in the WIB is determined by the size of the active list. The analysis in Section 2 indicates that 2048 entries is a good window size to achieve significant speedups. Therefore, initially we assume a 2K-entry active list and 1K-entry load and store queues. Assuming each WIB entry is 8 bytes then the total WIB capacity is 16KB. The bit-vectors can also consume a great deal of storage, but it is limited by the number of outstanding requests supported. Section 4 explores the impact of limiting the number of bit-vectors below the load queue size.

3.3.1 WIB Organization

We assume a banked WIB organization and that one instruction can be extracted from each bank every two cycles. These two cycles include determining the appropriate instruction and reading the appropriate WIB entry. There is a fixed instruction width between the WIB and the issue queue. We set the number of banks equal to twice this width. Therefore, we can sustain reinsertion at full bandwidth by reading instructions from the WIB's even banks in one cycle and from odd banks in the next cycle, if enough instructions are eligible in each set of banks.

Recall, WIB entries are allocated in program order in conjunction with active list entries. We perform this allocation using round-robin across the banks, interleaving at the individual instruction granularity. Therefore, entries in each bank are also allocated and released in program order, and we can partition each load's bit-vector according to which bank the bits map to. In our case, a 2K entry WIB with a dispatch width to the issue queue of 8 would have 16 banks with 128 entries each. Each bank also stores its local head and tail pointers to reflect program order of instructions within the bank. Figure 3 shows the internal organization of the WIB.

During a read access each bank in a set (even or odd) operates independently to select an instruction to reinsert to the issue queue by examining the appropriate 128 bits from each completed load. For each bank we create a single bit-vector that is the logical OR of the bit-vectors for all completed loads. The resulting bit-vector is examined to select the oldest active instruction in program order. There are many possible policies for selecting instructions. We examine a few simple policies later in this paper, but leave investigation of more sophisticated policies (e.g., data flow graph order or critical path [15]) as future work. Regardless of selection policy, the result is that one bit out of the 128 is set, which can then directly enable the output of the corresponding WIB entry without the need to encode then decode the WIB index. The process is repeated with an updated bit-vector that clears the WIB entry for the access just completed and may include new eligible instructions if another load miss completed during the access.

The above policies are similar to the select policies implemented by the issue queue logic. This highlights an important difference between the WIB and a conventional issue queue. A conventional issue queue requires wakeup logic that broadcasts a register specifier to each entry. The WIB eliminates this broadcast by using the completed loads' bit-vectors to establish the candidate instructions for selection. The issue queue requires the register specifier broadcast to maintain true dependencies. In contrast, the WIB-based architecture leverages the much smaller issue queue for this task and the WIB can select instructions for reinsertion in any order.

It is possible that there are not enough issue queue entries available to consume all instructions extracted from the WIB. In this case, one or more banks will stall for this access and wait for the next access (two cycles later) to attempt reinserting its instruction. To avoid potential livelock, on each access we change the starting bank for allocating the available issue queue slots. Furthermore, a bank remains at the highest priority if it has an instruction to reinsert but was not able to. A bank is assigned the lowest priority if it inserts an instruction or does not have an instruction to reinsert. Livelock could occur in a fixed priority scheme since the instructions in the highest priority bank could be dependent on the instructions in the lower priority bank. This could produce a continuous stream of instructions moving from the WIB to the issue queue then back to the WIB since their producing instructions are not yet complete. The producing instructions will never complete since they are in the lower priority bank. Although this scenario seems unlikely it did occur in some of our benchmarks and thus we use

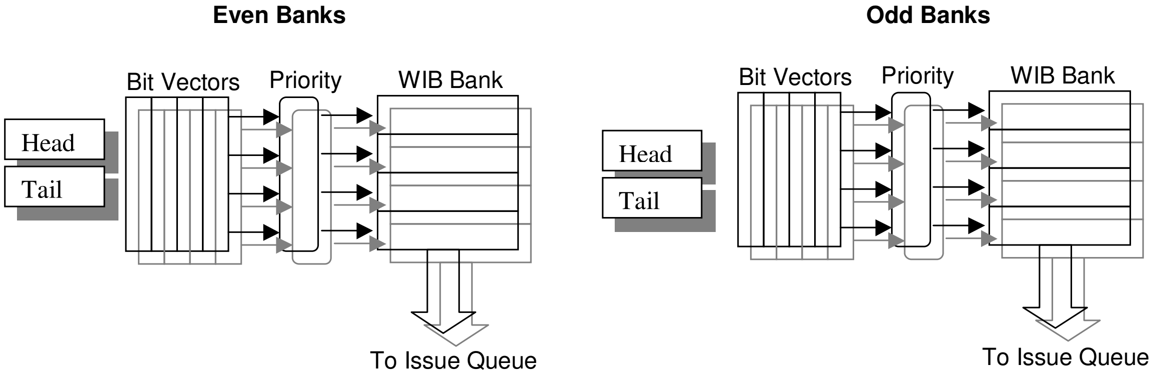


Figure 3. WIB Organization

round-robin priority.

3.3.2 Squashing WIB Entries

Squashing instructions requires clearing the appropriate bits in each bit-vector and reseting each banks' local tail pointer. The two-dimensional bit-vector organization simplifies the bit-vector clear operation since it is applied to the same bits for every bit-vector. Recall, each column corresponds to an outstanding load miss, thus we can clear the bits in the rows associated with the squashed instructions.

3.4 Register File Considerations

To support many in-flight instructions, the number of rename registers must scale proportionally. There are several alternative designs for large register files, including multicycle access, multi-level [13, 34], multiple banks [5, 13], or queue-based designs [6]. In this paper, we use a two-level register file [13, 34] that operates on principles similar to the cache hierarchy. Simulations of a multi-banked register file show similar results. Further details on the register file designs and performance are available elsewhere [20].

3.5 Alternative WIB Designs

The above WIB organization is one of several alternatives. One alternative we considered is a large non-banked multicycle WIB. Although it may be possible to pipeline the WIB access, it would not produce a fully pipelined access and our simulations (see Section 4) indicate pipelining may not be necessary.

Another alternative we considered is a pool-of-blocks structure for implementing the WIB. In this organziation, when a load misses in the cache it obtains a free block to buffer dependent instructions. A pointer to this block is stored with the load in the load queue (LQ) and is used to deposit dependent instructions in the WIB. When the load completes, all the instructions in the block are reinserted into the issue queue. Each block contains a fixed number of instruction slots and each slot holds information equivalent to issue queue entries.

An important difference in this approach compared to the technique we use is that instructions are stored in dependence chain order, and blocks may need to be linked together to handle loads with long dependence chains. This complicates squashing since there is no program order associated with the WIB entries. Although we could maintain information on program order, the list management of each load's dependence chain becomes too complex and time consuming during a squash. Although the bit-vector approach requires more space, it simplifies this management. The pool-of-blocks approach has the potential of deadlock if there are not enough WIB entries. We are continuing to investigate techniques to reduce the list management overhead and handle deadlock.

3.6 Summary

The WIB architecture effectively enlarges the instruction window by removing instructions dependent on load cache misses from the issue queue, and retaining them in the WIB while the misses are serviced. In achieving this, we leverage the existing processor issue logic without affecting the processor cycle time and circuit complexity. In the WIB architecture, instructions stay in the issue queue only for a short period of time, therefore new instructions can be brought into the instruction window much more rapidly than in the conventional architectures. The fundamental difference between a WIB design and a design that simply scales up the issue queue is that scaling up the issue queue significantly complicates the wakeup logic, which in turn affects the processor cycle time [1, 22]. However, a WIB requires a very simple form of wakeup logic as all the instructions in the dependence chain of a load miss are awakened when the miss is resolved. There is no need to broadcast and have all the instructions monitor the result buses.

4 Evaluation

In this section we evaluate the WIB architecture. We begin by presenting the overall performance of our WIB design compared to a conventional architecture. Next, we explore the impact of various design choices on WIB performance. This includes limiting the number of available bit-vectors, limited WIB capacity, policies for selecting instructions for reinsertion into the issue queue, and multicycle non-banked WIB.

These simulations reveal that WIB-based architectures can increase performance, in terms of IPC, for our set of benchmarks by an average of 20%, 84%, and 50% for SPEC INT, SPEC FP, and Olden, respectively. We also find that limiting the number of outstanding loads to 64 produces similar improvements for the SPEC INT and Olden benchmarks, but reduces the average improvement for the SPEC FP to 45%. A WIB capacity as low as 256 entries with a maximum of 64 outstanding loads still produces average speedups of 9%, 26%, and 14% for the respective benchmark sets.

Benchmark	Base IPC	Branch Dir Pred	DL1 Miss Ratio	UL2 Local Miss Ratio	WIB IPC
bzip2	1.19	0.94	0.03	0.47	1.59
gcc	1.34	0.94	0.01	0.09	1.38
gzip	2.25	0.91	0.02	0.04	2.25
parser	0.83	0.95	0.04	0.22	0.95
perlbmk	0.96	0.99	0.01	0.28	0.95
vortex	1.52	0.99	0.01	0.06	1.68
vpr	0.49	0.90	0.04	0.41	0.86
HM	1.00	-	-	-	1.24
applu	4.17	0.98	0.10	0.26	4.28
art	0.42	0.96	0.35	0.73	1.64
facrec	1.47	0.99	0.05	0.48	3.02
galgel	1.92	0.98	0.07	0.26	3.97
mgrid	2.58	0.97	0.06	0.42	2.57
swim	2.41	1.00	0.21	0.27	3.98
wupwise	3.38	1.00	0.03	0.25	3.99
HM	1.42	-	-	-	3.02
em3d	2.28	0.99	0.02	0.16	2.27
mst	0.96	1.00	0.07	0.49	2.51
perimeter	1.00	0.93	0.04	0.38	1.16
treeadd	1.05	0.95	0.03	0.33	1.28
HM	1.17	-	-	-	1.61

Table 2. Benchmark Performance Statistics

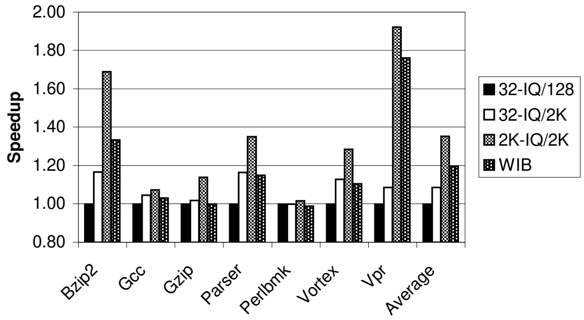
4.1 Overall Performance

We begin by presenting the overall performance improvement in IPC relative to a processor with a 32-entry issue queue and single cycle access to 128 registers, hence a 128-entry active list (32-IQ/128). Figure 4 shows the speedups (IPC_{new}/IPC_{old}) for various microarchitectures. Although we present results for an 8-issue processor, the overall results are qualitatively similar for a 4-issue processor. The WIB bar corresponds to a 32-entry issue queue with our banked WIB organization, a 2K-entry active list, and 2K registers, using a two-level register file with 128 registers in the first level, 4 read ports and 4 write ports to the pipelined second level that has a 4-cycle latency. Assuming the 32-entry issue queue and 128 level one registers set the clock cycle time, the WIB-based design is approximately clock cycle equivalent to the base architecture. For these experiments the number of outstanding loads (thus bit-vectors) is not limited, we explore this parameter below. Table 2 shows the absolute IPC values for the base configuration and our banked WIB design, along with the branch direction prediction rates, L1 data cache miss rates, and L2 unified cache local miss rates for the base configuration.

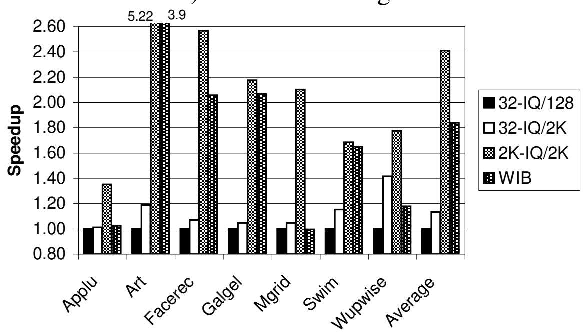
For comparison we also include two scaled versions of a conventional microarchitecture. Both configurations use a 2K-entry active list and single cycle access to 2K registers. One retains the 32-entry issue queue (32-IQ/2K) while the other scales the issue queue to 2K entries (2K-IQ/2K). These configurations help isolate the issue queue from the active list and to provide an approximate upper bound on our expected performance.

From the results shown in Figure 4, we make the following observations. First, the WIB design produces speedups over 10% for 12 of the 18 benchmarks. The average speedup is 20%, 84%, and 50% for SPEC INT, SPEC FP, and Olden, respectively. The harmonic mean of IPCs (shown in Table 2) increases from 1.0 to 1.24 for SPEC INT, from 1.42 to 3.02 for SPEC FP, and from 1.17 to 1.61 for Olden.

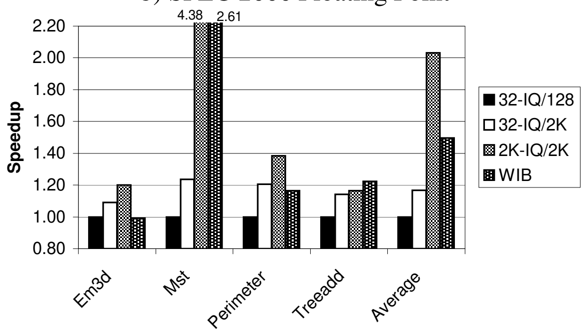
For most programs with large speedups from the large 2K issue queue, the WIB design is able to capture a significant fraction of the available speedup. However, for a few programs the 2K issue queue produces large speedups when the WIB does not. `mgrid` is the most striking example where the WIB does not produce any speedup while the 2K issue queue yields a speedup of over two. This phenomenon is a result of the WIB recycling instructions through the issue queue. This consumes issue bandwidth that the 2K issue queue uses only for instructions ready to execute. As evidence of this we track the number of times an instruction is inserted into the WIB. In the banked implementation the average number of times an instruction is inserted into the



a) SPEC 2000 Integer



b) SPEC 2000 Floating Point



c) Olden

Figure 4. WIB Performance

WIB is four with a maximum of 280. Investigations of other insertion policies (see below) reduces these values to an average insertion count of one and a maximum of 9, producing a speedup of 17%.

We also note that for several benchmarks just increasing the active list produces noticable speedups, in some cases even outperforming the WIB. This indicates the issue queue is not the bottleneck for these benchmarks. However, overall the WIB significantly outperforms an increased active list.

Due to the size of the WIB and larger register file, we also evaluated an alternative use of that space by doubling the data cache size in the base configuration to 64KB. Simulation results reveal less than 2% improvements in performance for all benchmarks, except `vortex` that shows a 9% improvement, over the 32KB data cache, indicating

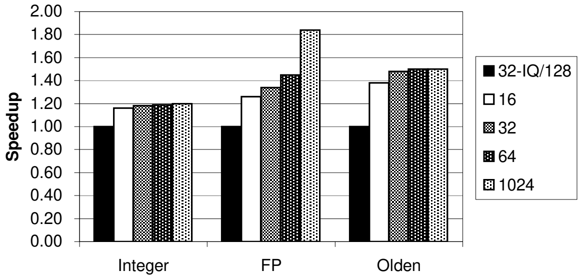


Figure 5. Performance of Limited Bit-Vectors

the WIB may be a better use of this space. We explore this tradeoff more later in this section.

We also performed two sensitivity studies by reducing the memory latency from 250 cycles to 100 cycles and by increasing the unified L2 cache to 1MB. The results match our expectations. The shorter memory latency reduces WIB speedups to averages of 5%, 30%, and 17% for the SPEC INT, SPEC FP, and Olden benchmarks, respectively. The larger L2 cache has a smaller impact on the speedups achieved with a WIB. The average speedups were 5%, 61%, and 38% for the SPEC INT, SPEC FP, and Olden benchmarks, respectively. The larger cache has the most impact on the integer benchmarks, which show a dramatically reduced local L2 miss ratio (from an average of 22% to 6%). Caches exploit locality in the program's reference stream and can sometimes be sufficiently large to capture the program's entire working set. In contrast, the WIB can expose parallelism for tolerating latency in programs with very large working sets or that lack locality.

For the remainder of this paper we present only the average results for each benchmark suite. Detailed results for each benchmark are available elsewhere [20].

4.2 Limited Bit-Vectors

The number of bit-vectors is important since each bit-vector must map the entire WIB and the area required can become excessive. To explore the effect of limited bit-vectors (outstanding loads), we simulated a 2K-entry WIB with 16, 32, and 64 bit-vectors. Figure 5 shows the average speedups over the base machine, including the 1024 bit-vector configuration from above. These results show that even with only 16 bit-vectors the WIB can achieve average speedups of 16% for SPEC INT, 26% for SPEC FP, and 38% for the Olden benchmarks. The SPEC FP programs (particularly `art`) are affected the most by the limited bit-vectors since they benefit from memory level parallelism. With 64 bit-vectors (16KB) the WIB can achieve speedups of 19%, 45%, and 50% for the three sets of benchmarks, respectively.

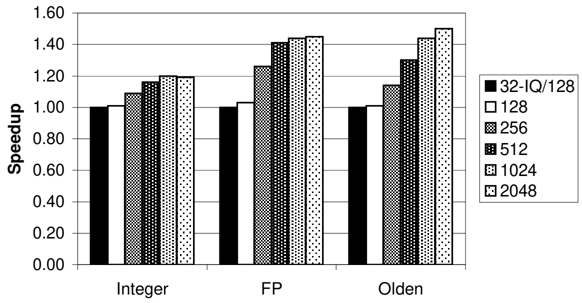


Figure 6. WIB Capacity Effects

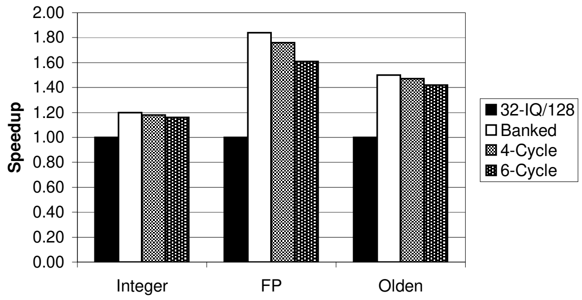


Figure 7. Non-Banked WIB Performance

4.3 Limited WIB Capacity

Reducing WIB area by limiting the number of bit-vectors is certainly a useful optimization. However, further decreases in required area can be achieved by using a smaller capacity WIB. This section explores the performance impact of reducing the capacity of the WIB, active list and register file.

Figure 6 shows the average speedups for WIB sizes ranging from 128 to 2048 with bit-vectors limited to 64. These results show that the 1024-entry WIB can achieve average speedups of 20% for the SPEC INT, 44% for SPEC FP, and 44% for Olden. This configuration requires only 32KB extra space (8KB for WIB entries, 8KB for bit-vectors, and 8KB for each 1024-entry register file). This is roughly area equivalent to doubling the cache size to 64KB. As stated above, the 64KB L1 data cache did not produce noticable speedups for our benchmarks, and the WIB is a better use of the area.

4.4 WIB to Issue Queue Instruction Selection

Our WIB design implements a specific policy for selecting from eligible instructions to reinsert into the issue queue. The current policy chooses instructions from each bank in program order. Since the banks operate independently and on alternate cycles, they do not extract instructions in true program order. To evaluate the impact of instruction selection policy we use an idealized WIB that has single cycle access time to the entire structure. Within this design we evaluate the following instruction selection policies: (1) the current banked scheme, (2) full program order from among eligible instructions, (3) round robin across completed loads with each load's instructions in program order, and (4) all instructions from the oldest completed load.

Most programs show very little change in performance across selection policies. `mgrid` is the only one to show significant improvements. As mentioned above, `mgrid` shows speedups over the banked WIB of 17%, 17%, and 13% for each of the three new policies, respectively. These speedups are due to better scheduling of the actual dependence graph. However, in some cases the schedule can be worse. Three programs show slowdowns compared to the banked WIB for the oldest load policy (4): `bzip` 11%, `parser` 15%, and `facerec` 5%.

4.5 Non-Banked Multicycle WIB Access

We now explore the benefits of the banked organization versus a multicycle non-banked WIB organization. Figure 7 shows the average speedups for the banked and non-banked organizations over the base architecture. Except the different WIB access latencies, the 4-cycle and 6-cycle bars both assume a non-banked WIB with instruction extraction in full program order. These results show that the longer WIB access delay produces only slight reductions in performance compared to the banked scheme. This indicates that we may be able to implement more sophisticated selection policies and that pipelining WIB access is not necessary.

5 Related Work

Our limit study is similar to that performed by Skadron et al. [28]. Their results show that branch mispredictions limit the benefits of larger instruction windows, that better branch prediction and better instruction cache behavior have synergistic effects, and that the benefits of larger instruction windows and larger data caches trade off and have overlapping effects. Their simulation assumes a very large 8MB L2 cache and models a register update unit (RUU) [29], which is a unified active list, issue queue, and rename register file. In their study, only instruction window sizes up to 256 are examined.

There has been extensive research on architecture designs for supporting large instruction windows. In the multiscalar [30] and trace processors [23], one large centralized instruction window is distributed into smaller windows among multiple parallel processing elements. Dynamic multithreading processors [2] deal with the complexity of a large window by employing a hierarchy of instruction windows. Clustering provides another approach, where a collection of small windows with associated functional units is used to approximate a wider and deeper instruction window [22].

Recent research [7, 18] investigates issue logic designs that attemp to support large instruction windows without impeding improvements on clock rates. Michaud and Seznec [21] exploit the observation that instructions dependent on long latency operations unnecessarily occupy issue queue space for a long time, and address this problem by prescheduling instructions based on data dependencies. Other dependence-based issue queue designs are studied in [9, 10, 22]. Zilles et al. [35] and Balasubramonian et al. [4] attack the problem caused by long latency operations by utilizing a future thread that can use a portion of the issue queue slots and physical registers to conduct precomputation. As power consumption has become an important consideration in processor design, researchers have also studied low power instruction window design [3, 16].

6 Conclusion

Two important components of overall execution time are the clock cycle time and the number of instructions committed per cycle (IPC). High clock rates can be achieved by using a small instruction window, but this can limit IPC by reducing the ability to identify independent instructions. This tension between large instruction windows and short clock cycle times is an important aspect in modern processor design.

This paper presents a new technique for achieving the latency tolerance of large windows while maintaining the high clock rates of small window designs. We accomplish this by removing instructions from the conventional issue queue if they are directly or indirectly dependent on a long latency operation. These instructions are placed into a waiting instruction buffer (WIB) and reinserted into the issue queue for execution when the long latency operation completes. By moving these instructions out of the critical path, their previously occupied issue queue entries can be further utilized by the processor to look deep into the program for more ILP. An important difference between the WIB and scaled-up conventional issue queues is that the WIB implements a simplified form of wakeup-select. This is achieved by allowing all instructions in the dependence chain to be considered for reinsertion into the issue window. Compared to the full wakeup-select in conventional issue queues, the WIB only requires select logic for instruction reinsertion.

Simulations of an 8-way processor with a 32-entry issue queue reveal that adding a 2K-entry WIB can produce speedups of 20%, 84%, and 50% for a subset of the SPEC CINT2000, SPEC CFP2000, and Olden benchmarks, respectively. We also explore several WIB design parameters and show that allocating chip area for the WIB produces signifcantly higher speedups than using the same area to increase the level one data cache capacity from 32KB to 64KB.

Our future work includes investigating the potential for executing the instructions from the WIB on a separate execution core, either a conventional core or perhaps a grid processor [25]. The policy space for selecting instructions is an area of current research. Finally, register file design and management (e.g., virtual-physical, multi-banked, multi-cycle, prefetching in a two-level organization) require further investigation.

Acknowledgements

This work was supported in part by NSF CAREER Awards MIP-97-02547 and CCR-0092832, NSF Grants CDA-97-2637 and EIA-99-72879, Duke University, and donations from Intel, IBM, Compaq, Microsoft, and Ericsson. We thank the anonymous reviewers for comments and suggestions on this work.

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock Rate Versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000.

[2] H. Akkary and M. A. Driscoll. A Dynamic Multithreading Processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 226–236, December 1998.

[3] R. I. Bahar and S. Manne. Power and Energy Reduction via Pipeline Balancing. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 218–229, July 2001.

[4] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically Allocating Processor Resources Between Nearby and Distant ILP. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 26–37, July 2001.

[5] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Reducing the Complexity of the Register File in Dynamic Superscalar Processors. In *Proceedings of the 34th International Symposium on Microarchitecture*, December 2001. To appear.

[6] B. Black and J. Shen. Scalable Register Renaming via the Quack Register File. Technical Report CMuArt 00-1, Carnegie Mellon University, April 2000.

[7] M. D. Brown, J. Stark, and Y. N. Patt. Select-Free Instruction Scheduling Logic. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, December 2001. To appear.

[8] D. C. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors-the SimpleScalar Tool Set. Technical Report 1308, University of Wisconsin–Madison Computer Sciences Department, July 1996.

[9] R. Canal and A. González. A Low-Complexity Issue Logic. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 327–335, May 2001.

[10] R. Canal and A. González. Reducing the Complexity of the Issue Logic. In *Proceedings of the 2001 International Conference on Supercomputing*, pages 312–320, June 2001.

[11] M. C. Carlisle, A. Rogers, J. H. Reppy, and L. J. Hendren. Early Experiences with Olden. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 1–20, August 1993.

[12] Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*, July 1999.

[13] J.-L. Cruz, A. González, M. Valero, and N. P. Topham. Multiple-Banked Register File Architectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 316–325, June 2000.

[14] 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.

[15] B. A. Fields, S. Rubin, and R. Bodik. Focusing Processor Policies via Critical-Path Prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 74–85, July 2001.

[16] D. Folegnani and A. González. Energy-Effective Issue Logic. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 230–239, July 2001.

[17] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[18] D. S. Henry, B. C. Kuszmaul, G. H. Loh, and R. Sami. Circuits for Wide-Window Superscalar Processors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 236–247, June 2000.

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

[20] T. Li, J. Koppanalil, A. R. Lebeck, J. Patwardhan, and E. Rotenberg. A Large, Fast Instruction Window for Tolerating Cache Misses. Technical Report CS-2002-03, Department of Computer Science, Duke University, March 2002.

[21] P. Michaud and A. Seznec. Data-flow Prescheduling for Large Instruction Windows in Out-of-Order Processors. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 27–36, January 2001.

[22] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[23] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace Processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 138–148, December 1997.

[24] S. Sair and M. Charney. Memory Behavior of the SPEC2000 Benchmark Suite. Technical Report RC-21852, IBM T.J. Watson, October 2000.

[25] K. Sankaralingam, R. Nagarajan, D. Burger, and S. W. Keckler. A Design Space Evaluation of Grid Processor Architectures. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, December 2001. To appear.

[26] K. Skadron. *Characterizing and Removing Branch Mispredictions*. PhD thesis, Department of Computer Science, Princeton University, June 1999.

[27] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Improving Prediction for Procedure Returns with Return-Address-Stack Repair Mechanisms. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 259–271, December 1998.

[28] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Branch Prediction, Instruction-Window Size, and Cache Size: Performance Tradeoffs and Simulation Techniques. *IEEE Transactions on Computers*, 48(11):1260–1281, November 1999.

[29] G. S. Sohi. Instruction Issue Logic for High-Performance, Interruptable, Multiple Functional Unit, Pipelined Processors. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[30] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[31] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. White Paper, IBM, October 2001.

[32] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal*, pages 25–33, January 1967.

[33] C. T. Weaver. Pre-compiled SPEC2000 Alpha Binaries. Available: http://www.simplescalar.org.

[34] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Two-Level Hierarchical Register File Organization For VLIW Processors. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, pages 137–146, December 2000.

[35] C. B. Zilles and G. S. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 172–181, June 2000.