Exploiting Operand Availability for Efficient Simultaneous Multithreading

Joseph J. Sharkey, *Student Member*, *IEEE*, and Dmitry V. Ponomarev, *Member*, *IEEE*

Abstract—We propose several schemes to improve the scalability, reduce the complexity and delays, and increase the throughput of dynamic scheduling in SMT processors. Our first design is an adaptation of the recently proposed instruction packing to SMT. Instruction packing opportunistically packs two instructions (possibly from different threads), each with at most one nonready source operand at the time of dispatch, into the same issue queue entry. Our second design, termed 2OP_BLOCK, takes these ideas one step further and completely avoids the dispatching of the instructions with two nonready source operands. This technique has several advantages. First, it reduces the scheduling complexity (and the associated delays) as the logic needed to support the instructions with two nonready source operands is eliminated. More surprisingly, 2OP_BLOCK simultaneously improves the performance as the same issue queue entry may be reallocated multiple times to the instructions with at most one nonready source (which usually spends fewer cycles in the queue) as opposed to hogging the entry with an instruction which enters the queue with two nonready sources. For schedulers with the capacity to hold 64 instructions on a 4-way SMT, the 2OP_BLOCK design outperforms the traditional queue by 14 percent, on average, and at the same time results in a 10 percent reduction in the overall scheduling delay. We also present mechanisms to support speculative scheduling with 2OP_BLOCK and introduce the hybrid scheme that dynamically switches between 2OP_BLOCK and instruction packing modes depending on the workload characteristics, to achieve further performance gains.

Index Terms—Issue queue, simultaneous multithreading.

1 Introduction

Simultaneous Multithreading (SMT) is an effective technique to increase the throughput of a traditional superscalar processor via the simultaneous sharing of the key data path components among multiple threads [41], [42]. Such sharing elevates the pressure on these resources and increases their utilization, necessitating the use of a larger number of entries in these components to realize the full performance potential afforded by SMT. Unfortunately, the additional complexities that are incurred by the implementation of these larger structures may exacerbate the critical timing paths and negatively impact the processor's cycle time. As a careful balance must be found between the instruction-level parallelism (ILP) and the cycle time to achieve maximum performance, it is important to reduce the complexity of these critical resources without significantly impacting the ability to extract the ILP.

One such critical data path structure that experiences elevated pressure due to sharing in an SMT processor is the issue queue (IQ). Fig. 1 shows how the processor throughput scales with the size of the IQ for both the superscalar and the 4-threaded SMT machines, based on our simulations (details of our simulation framework are presented in Section 2). Results are shown in the form of the commit IPC improvements over the respective baseline with a 32-entry IQ. The trends clearly indicate that the performance of a superscalar machine (depicted by the bars on the left) increases only slightly as the number of entries in the IQ is increased beyond 32. On average, across the full set of SPEC 2000 benchmarks, the difference between the performance of the processors with 32-entry and 256-entry IQs is only 9.3 percent. The situation is, however, quite different for an SMT machine, where four threads execute simultaneously and much higher pressure is exerted on the IQ. On average, there is a 20 percent IPC difference as the IQ size increases from 32 to 64 entries, an additional 14 percent if the IQ size is 128 entries, and a further 23 percent for the 256-entry IQ.

Thus, SMT machines require very generously sized IQs to realize their full performance potential. However, it is well documented in the recent literature that the wake-up and selection operations associated with the IQ lie on the critical timing path in modern microprocessors [21], [27], [39] and the delays of both increase proportionately with the number of entries in the IQ. Consequently, increasing the number of entries beyond a certain limit may negatively impact the processor's clock frequency. Largely due to these constraints, even modern processor implementations use rather modestly sized IQs. It is therefore important to investigate techniques that can provide the illusion of larger scheduling windows without physically enlarging the IQ.

All of the existing solutions for optimizing the IQ efficiency on SMT processors do so indirectly by controlling the quality of instructions that are fetched into the pipeline from multiple threads, e.g., ensuring that none of the threads clog the issue queue. For example, the I-Count fetching policy [41] gives priority to threads with fewer not-yet-executed instructions already in the pipeline. Some optimizations to the I-Count policy that further increase the efficiency of the IQ usage have also been proposed in the literature. Fundamentally, these solutions attempt to avoid clogging the queue with instructions that reside there for a

• *The authors are with the Department of Computer Science, State University of New York, Binghamton, NY 13902-6000.*
E-mail: {jsharke, dima}@cs.binghamton.edu.

Manuscript received 20 Dec. 2005; revised 18 June 2006; accepted 7 July 2006; published online 20 Dec. 2006.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0454-1205.

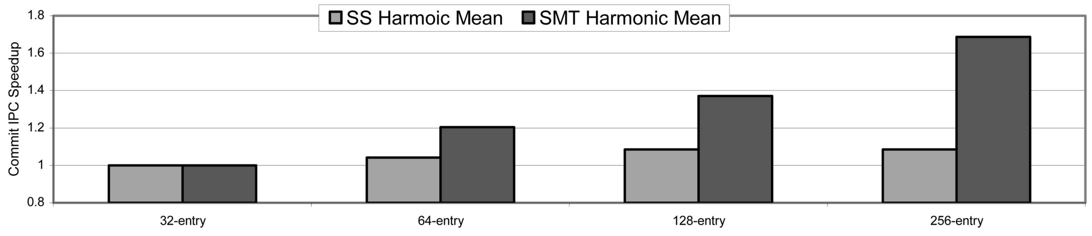


Fig. 1. Average performance improvements for superscalar and 4-way SMT machines for various IQ sizes with respect to the corresponding baseline machine with 32-entry IQs.

large number of cycles before being issued. For example, FLUSH [40], FLUSH++ [11], and the Data Miss Gating technique of [14] combine I-count with a special treatment of threads that experienced misses in various levels of the cache hierarchy. While all these mechanisms are effective to some extent, their inherent limitation lies in the reliance on information that is available at the time of instruction fetch. However, much more effective and precise control can be exercised over the IQ usage if these fetch-centric approaches are augmented with some dynamic register-specific information about the instructions, which is readily available after the register renaming stage. Specifically, we show that the instructions with two nonready source operands at the time of dispatch (we assume an ISA with two source operand instructions for this study) spend a significantly larger number of cycles in the IQ than other instructions and most of these cycles are spent waiting for the arrival of the first source. If such instructions are kept outside of the IQ until one of the source operands becomes available, then the pressure on the IQ will be greatly reduced and the performance will improve. All previous proposals do not make use of such information and treat all instructions equally, regardless of the number of ready source operands at the time of dispatch.

The focus of this paper is to consider the mechanisms that augment the existing fetch-centric approaches with dynamic microarchitectural information to exercise a more effective control of *how* and *when* the instructions are placed in the IQ. To this end, we propose several solutions. First, we extend the previously introduced instruction packing mechanism [34], [35] to SMT. Instruction packing opportunistically places two instructions into the same IQ entry provided that each of these instructions has at most one nonready source operand at the time of dispatch. We show that the percentage of instructions entering the scheduling window with two nonready source operands is smaller on an SMT machine than it is on a superscalar and, therefore, instruction packing is more amenable to SMT processors. Second, we take these ideas one step further and completely disallow the dispatch of instructions with two nonready source operands. This design (called 2OP_BLOCK in the rest of the paper) reduces the scheduler complexity as the capability to support the instructions with two nonready source operands is not needed. Furthermore, 2OP_BLOCK improves the performance as the same IQ entry may be reallocated multiple times to instructions with at most one nonready source (which tend to spend fewer cycles in the queue) rather than clogging the entry for a long time with one instruction that has two nonready sources. Third, we propose adaptive hybrid technique that dynamically switches between instruction packing and 2OP_BLOCK regimes to mitigate the performance losses on 2-threaded workloads. We investigate various triggers that dictate when such switching should occur. Fourth, we present mechanisms to support speculative scheduling and associated instruction replays in the data path that uses 2OP_BLOCK or instruction packing scheduling logic.

The rest of the paper is organized as follows: Our simulation methodology is described in Section 2. Section 3 reviews instruction packing mechanism as proposed for superscalars in [35]. Our dynamic scheduler designs for SMT are described in Section 3. The scheduler designs that improve the SMT efficiency are presented in Section 4. Section 5 presents the adaptive technique to dynamically switch between instruction packing and 2OP_BLOCK modes to sustain the performance in the environment with a limited number of threads. Our results are presented in Section 6. In Section 7, we show how the schemes proposed in this paper can be used in a data path with speculative instruction scheduling. We describe the related work in Section 8 and offer the concluding remarks in Section 9.

2 Simulation Methodology

For estimating the performance impact of the schemes described in this paper, we used M-Sim [37]: a significantly modified version of the Simplescalar 3.0d simulator [5] that supports the SMT processor model. In the SMT model, the threads share the IQ, the pool of physical registers, the execution units, and the caches, but have separate rename tables, program counters, load/store queues, and reorder buffers. While each thread has its own logical reorder buffer, the number of total ROB entries in the machine is kept to 512 and statically partitioned among the executing threads. Specifically, for 2-threaded workloads, each thread has 256 ROB entries, for 3-threaded workloads, each thread has 170 ROB entries, and, for 4-threaded workloads, each thread has 128 ROB entries. Each thread also has its own branch predictor. The details of the studied processor configuration are shown in Table 1. In the baseline SMT model, the I-Count fetch policy [41] was implemented and fetching was limited to two threads per cycle.

We simulated the full set of SPEC 2000 integer and floating-point benchmarks [18], using the precompiled Alpha binaries available from the Simplescalar Web site [5]. We skipped the initialization part of each benchmark, using the procedure prescribed by the Simpoints tool [38], and then simulated the execution of the following 100 million instructions. For multithreaded workloads, we stopped the simulations after 100 million instructions from any thread had committed.

TABLE 1
Configuration of the Simulated Processor

Parameter	Configuration
Machine width	8-wide fetch, 8-wide issue, 8-wide commit
Window size	issue queue – as specified,48 entry load/store queue, 512 ROB entries statically divided among the executing threads
Function Units and Latency (total/issue)	8 Int Add (1/1), 4 Int Mult (3/1) / Div (20/19), 4 Load/Store (2/1), 4 FP Add (2), 4 FP Mult (4/1) / Div (12/12) / Sqrt (24/24)
Phys. Registers	256 integer + 256 floating-point
L1 I–cache	64 KB, 1–way set–associative, 128 byte line, 1 cycles hit time
L1 D–cache	64 KB, 4–way set–associative, 64 byte line, 2 cycles hit time
L2 Cache unified	2 MB, 8–way set–associative, 128 byte line, 6 cycles hit time
BTB	2048 entry, 2–way set–associative
Branch Predictor per thread	2K entry Gshare, 10 bit global history
Pipeline Structure	5 stage front end (fetch-dispatch), scheduling, 2 stages for register file access, execution, writeback, commit.
Memory	150 cycles latency

TABLE 2
Simulated 4-Threaded Workloads

Classification	Mix Name	Benchmarks
4 LOW ILP	Mix 1	*mgrid, equake, art, lucas*
	Mix 2	*twolf, vpr, swim, parser*
4 MED ILP	Mix 3	*applu, ammp, mgrid, galgel*
	Mix 4	*gcc, bzip2, eon, apsi*
4 HIGH ILP	Mix 5	*facerec, crafty, perlbmk, gap*
	Mix 6	*wupwise, gzip, vortex, mesa*
2 LOW ILP + 2 HIGH ILP	Mix 7	*parser, equake, mesa, vortex*
	Mix 8	*parser, swim, crafty, perlbmk*
2 LOW ILP + 2 MED ILP	Mix 9	*art, lucas, galgel, gcc*
	Mix 10	*parser, swim, gcc, bzip2*
2 MED ILP + 2 HIGH ILP	Mix 11	*gzip, wupwise, fma3d, apsi*
	Mix 12	*vortex, mesa, mgrid, eon*

Our multithreaded workloads contain a subset of the possible combinations of the simulated benchmarks. In selecting the multithreaded workloads, we first simulated all benchmarks in the single-threaded superscalar environment and used these results to classify them as low, medium, and high ILP, where the low ILP benchmarks are memory bound and the high ILP benchmarks are execution bound.

In total, we simulated 12 4-threaded workloads, 12 3-threaded workloads, and 12 2-threaded workloads. All workloads were created by mixing the benchmarks with different ILP levels in various ways. Tables 2, 3, and 4 depict the specific benchmarks that constituted each of our workloads. The ILP level of each benchmark is also shown. Note that some differences in the results of this paper compared to the conference version [43] stem from the fact that, in this paper, we eliminated the *mcf* benchmark from 4-threaded *mixes 1 and 7* due to an extremely low IPC of that benchmark.

We used several metrics for evaluating the performance of the multithreaded workloads throughout this paper. The first metric is the total throughput in terms of the commit IPC rate. However, this metric does not accurately reflect changes that favor a thread with high IPC at the expense of significantly hindering a thread with low IPC [25], [41]. Therefore, we also present the "fairness" metric of "harmonic mean of weighted IPCs" [25], [41], which accounts for individual per-thread performance.

For estimating the delays of the various schedulers, we developed handcrafted and highly optimized VLSI layouts of the IQ. Layouts were designed in a 0.18m 6-metal layer TSMC process using Cadence design tools.

3 Instruction Packing

In this section, we briefly describe instruction packing as proposed for superscalar machines in [34], [35] to provide sufficient background for the rest of the paper. Fig. 2a shows the format of an IQ entry used in traditional designs. The following fields comprise a single entry:

1. entry allocated bit (A),
2. payload area (opcode, FU type, destination register address, literals),
3. tag of the first source, associated comparator (tag CAM word 1, hereafter just tag CAM 1, without the "word") and the source valid bit,
4. tag of the second source, associated comparator (tag CAM 2) and source valid bit, and
5. the ready bit.

The ready bit, used to raise the request signal for the selection logic, is derived by AND-ing the valid bits of the two sources.

If at least one of the source operands of an instruction entering the IQ is ready at the time of dispatch, the tag CAM associated with this operand's slot remains unused.

TABLE 3
Simulated 2-Threaded Workloads

Classification	*Mix Name*	*Benchmarks*
2 LOW ILP	Mix 1	*equake, lucas*
	Mix 2	*twolf, vpr*
2 MED ILP	Mix 3	*gcc, bzip2*
	Mix 4	*mgrid, galgel*
2 HIGH ILP	Mix 5	*facerec, wupwise*
	Mix 6	*crafty, gzip*
1 LOW ILP + 1 HIGH ILP	Mix 7	*parser, vortex*
	Mix 8	*swim, gap*
1 LOW ILP + 1 MED ILP	Mix 9	*twolf, bzip2*
	Mix 10	*equake, gcc*
1 MED ILP + 1 HIGH ILP	Mix 11	*applu, mesa*
	Mix 12	*ammp, gzip*

TABLE 4
Simulated 3-Threaded Workloads

Classification	*Mix Name*	*Benchmarks*
3 LOW ILP	Mix 1	*mgrid, equake, art*
	Mix 2	*twolf, vpr, swim*
3 MED ILP	Mix 3	*applu, ammp, mgrid*
	Mix 4	*gcc, bzip2, eon*
3 HIGH ILP	Mix 5	*facerec, crafty, perlbmk*
	Mix 6	*wupwise, gzip, vortex*
2 LOW ILP + 1 HIGH ILP	Mix 7	*parser, equake, mesa*
1 LOW ILP + 2 HIGH ILP	Mix 8	*perlbmk, parser, crafty*
2 LOW ILP + 1 MED ILP	Mix 9	*art, lucas, galgel*
1 LOW ILP + 2 MED ILP	Mix 10	*parser, bzip2, gcc*
2 MED ILP + 1 HIGH ILP	Mix 11	*gzip, wupwise, fma3d*
1 MED ILP + 2 HIGH ILP	Mix 12	*vortex, eon, mgrid*

To exploit this idle tag CAM, instruction packing [34], [35] shares one issue queue entry between two such instructions. With instruction packing, an entry in the IQ can hold one or two instructions, depending on the number of ready operands in the stored instructions at the time of dispatch. The key difference between instruction packing and the previously proposed Tag Elimination mechanism of [15] is that the queue partitioning into the entries with various numbers of comparators is done dynamically in the former scheme and statically in the latter.

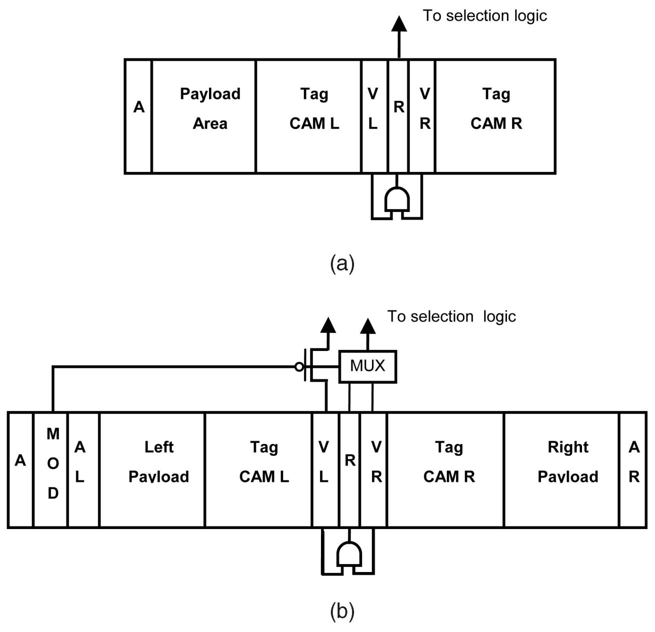
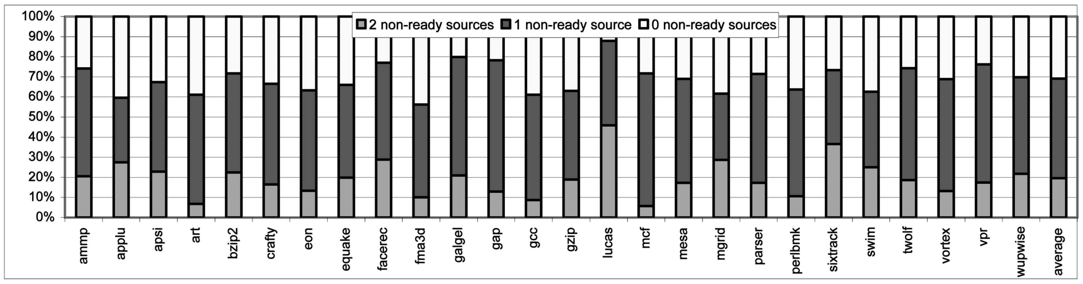


Fig. 2. Formats of (a) a traditional IQ entry and (b) an entry of the IQ that supports instruction packing.

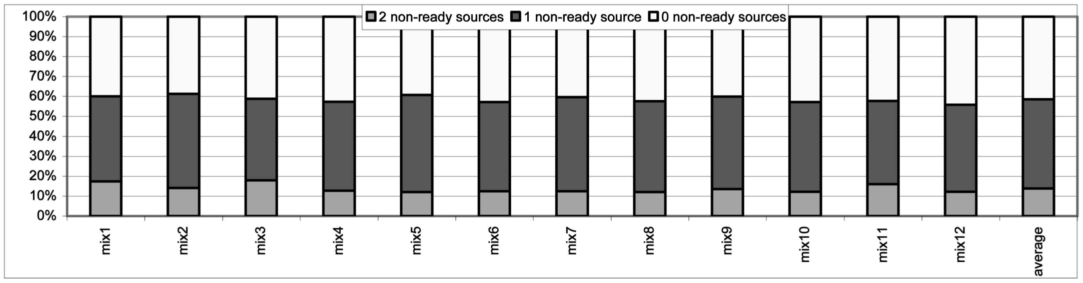
Fig. 2b shows the format of an issue queue entry that supports instruction packing. Each IQ entry is comprised of the "entry allocated" bit (A), the ready bit (R), the mode bit (MODE), and the two symmetrical halves: the left half and the right half. The structure of each half is identical, so we will use the left half for the subsequent explanations.

The left half of each IQ entry contains the following fields:

- Left half allocated (AL) bit. This bit is set when the left half-entry is allocated.
- Source tag and associated comparator (Tag CAM). This is where the tag of the nonready source operand for an instruction with at most one nonready source is stored.
- Source valid left bit (VL). This bit signifies the validity of the corresponding source operand connected to the comparator in this half of the entry, similarly to traditional designs. This bit is also used to indicate if the instruction residing in a half-entry is ready for selection (as explained later).
- Payload area. The payload area contains the same information as in the traditional design, namely, opcode, bits identifying the FU type, destination register address, and literal bits. In addition, the payload area contains the tag of the second source. Notice that the tag of the second source does not



(a)



(b)

Fig. 3. Distribution of nonready operands at the time of dispatch in (a) superscalar and (b) 4-threaded SMT machines for 32-entry IQs.

participate in the wake-up because, if an instruction is allocated to a half-entry, the second source must be valid at the time of dispatch. Compared to the traditional design, the payload area is increased by the number of bits used to represent a source tag.

The process of instruction wake-up remains exactly the same as in the traditional design for an instruction that occupies a full IQ entry (i.e., enters the queue with two nonready register sources). Here, the ready bit (R) is set by AND-ing the valid bits of both sources. For instructions that occupy half of an IQ entry, the wake-up simply amounts to setting of the valid bit corresponding to the source that was nonready when the instruction entered the IQ. The contents of the source valid bits are then directly used to indicate that the instruction is ready for selection (the validity of the second source is implicit in this case).

An increase in the complexity of the selection logic is avoided by sharing one request line between the R and the VR bits. The shared request line is raised if at least one of the bits (the R or the VR) is set. The R and the VR bits are both connected to the shared request line through a multiplexer, which is controlled by the "mode" bit of the IQ entry (Fig. 2b). In order to avoid additional delays during instruction issue (due to the ambiguity in the location of register tags needed to start the register file read access), the following solution is used. When an instruction with two nonready sources is allocated to the IQ, the tag that is connected to the left half comparator is also replicated in the payload area storage for the second tag in the right half. As a result, both tags will be present in the right half of the queue, so these tags can simply be used for register file access, without regard for the IQ entry mode.

More details describing this technique can be found elsewhere [34], [35]. In the next section, we discuss the application of instruction packing to SMT processors and show that it is an even more effective technique for those machines.

4 Scheduler Designs for SMT

This section describes our proposed designs to maximize the scheduling efficiency of SMT processors.

4.1 Instruction Packing on SMT

Our first step is to simply apply instruction packing to the SMT machine. As follows from the discussions in Section 3, the effectiveness of instruction packing directly depends on the percentage of instructions that enter the scheduling window with two nonready sources: The lower this percentage is, the better the trade-offs are that can be realized by packing.

Fig. 3 presents the percentage of instructions that are dispatched with zero, one, and two nonready source operands for both superscalar (Fig. 3a) and SMT (Fig. 3b) processors. While about 20 percent of the instructions are dispatched with two nonready sources on a superscalar machine, this percentage drops to 13 percent on a 4-threaded SMT configuration. It is not surprising that the percentage of such instructions is lower on an SMT machine than it is on a superscalar. The fundamental reason behind this lower percentage is that each thread runs relatively slower on SMT, which in turn increases the likelihood of a source operand being produced before the consuming instruction is dispatched. As a result, fewer instructions require a full-width entry in the instruction packing scheduler on SMT, allowing for a more efficient use of the available comparators.

Fig. 4 compares the performance achieved by using various configurations of the traditional schedulers as well as the schedulers supporting instruction packing for both superscalar (SS) and SMT processors. When comparing the different scheduler designs throughout the paper, we adopt the following notation: Each particular scheduler is referred to as N-Scheme, where N refers to the maximum number of instructions that the scheduler can hold simultaneously (or its capacity) and Scheme refers to the scheduling mechanism

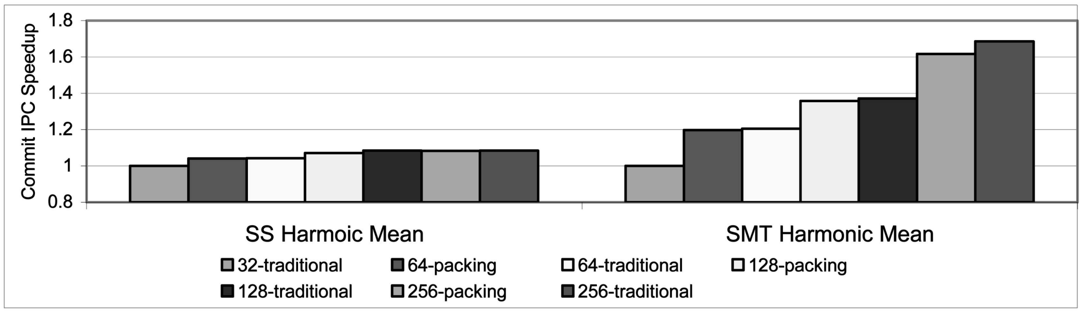


Fig. 4. Performance of instruction packing on superscalar (SS Harmonic Mean) and 4-way SMT (SMT Harmonic Mean) machines for various IQ configurations.

that is in use (traditional, packing, or 2OP_BLOCK). For example, 64-traditional refers to the traditional issue queue with 64 entries and 64-packing refers to the scheduler that supports instruction packing and has a capacity to hold up to 64 instructions. Note that, in terms of the logic presented in Fig. 2b, such a scheduler will only have 32-entries in the traditional sense and each of these entries will support up to two instructions. In practice, the maximum number of instructions held by this scheduler is often less than 64 because some instructions occupy full-sized entries. Note further that this terminology is different from that used in our previous work [34], [35].

While instruction packing provides some benefits for a superscalar machine (the set of bars on the left), the advantages are generally limited to the schedulers of smaller sizes because the superscalar's performance is virtually insensitive to the IQ size if that exceeds 32 entries on our simulation framework. In contrast, one can observe that, on an SMT machine, instruction packing continues to be a very effective technique even for much larger schedulers, as there are significant IPC improvements when the IQ size is increased in the baseline SMT case up to at least 256-entry queues. For SMT processors, a 32-entry IQ represents a large performance bottleneck—the difference between the performance of the machines with a 32-entry IQ and a 64-entry IQ is 22 percent on average.

In all of the cases considered and depicted in Fig. 4 (the set of bars on the right), the performance of the N-packing scheduler comes within a few percentage points of the performance of the N-traditional scheduler. This is not surprising as almost 90 percent of the instructions (see Fig. 3b) can be actually packed within the IQ entries and do not require a separate entry of their own. Specifically, the 64-packing scheduler achieves IPC within 2 percent of the 64-traditional scheduler. Results of a similar nature are achieved for other IQ sizes.

While achieving essentially the same performance levels as the traditional queue with the same maximum capacity, instruction packing significantly reduces the access delays and, therefore, presents a more attractive solution to scaling the dynamic scheduling logic than simply increasing the number of entries in the queue. The reduced amount of associative logic in the packing scheduler results in a drastic reduction of the wake-up access delays compared to the traditional designs of similar capacity. Even though the delays of the selection logic are slightly increased (due to the extra MUX), the overall reduction in the scheduling delays is still substantial. Specific delays of instruction packing schedulers are presented elsewhere [35], the important result to note at this point is that the difference in the scheduling delay between N-traditional and 2*N-traditional designs is roughly twice as much as the difference in the delay between N-traditional and 2*N-packing schedulers. Therefore, simply applying instruction packing to a traditional queue achieves almost the same maximum capacity and the same performance as doubling the queue, but with significantly less additional delay.

4.2 2OP_BLOCK: Blocking Instructions with Two Nonready Source Operands

To motivate our next technique, we first present the microarchitectural-level statistics related to the number of cycles that the instructions spend in a 32-entry IQ of the SMT processor as a function of the number of nonready register source operands at the time of instruction dispatch. Fig. 5 shows this data.

The leftmost set of bars in Fig. 5 depicts the number of cycles spent in the IQ by the instructions which enter the queue with two nonready register sources. On average, such instructions wait 20 cycles before being issued. The next set of bars shows the average number of cycles elapsed between the dispatch of such instructions into the queue and the arrival of the first source. It is interesting to observe that most of the cycles are spent waiting for the first-arriving source operand. After that, the instruction typically issues very fast (in two cycles on average). Finally, the third set of bars in Fig. 5 shows the number of cycles spent in the queue by all other instructions (the ones that enter with at least one register source ready or have no more than one such source in the first place). As can be seen, the instructions encapsulated by the third bar spend significantly fewer cycles in the queue—they reside in the queue for only 10 cycles as opposed to 20 cycles spent in the queue by the instructions that enter with two nonready sources.

We now examine how to exploit the statistics presented in Fig. 5 to increase the efficiency of the SMT scheduling logic. First, we observe that the SMT environment opens up an additional dimension for maximizing the performance and efficiency of instruction scheduling. Specifically, when one of the threads is expected to supply instructions which will spend a large number of cycles in the IQ, the dispatching of instructions from that thread can be temporarily suspended. In contrast to the single-threaded execution in a superscalar processor, such thread suspension will not result in performance degradation if the supply of instructions from other threads can be sustained.

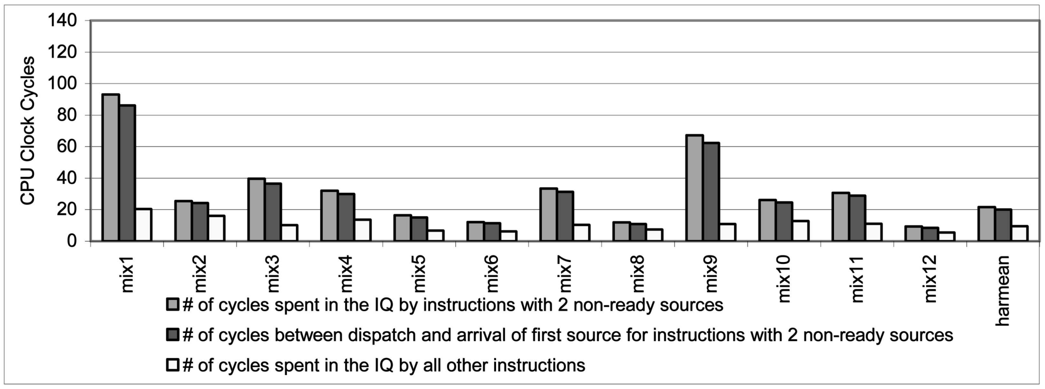


Fig. 5. Number of cycles spent by the instructions in the IQ for the 4-way SMT machine.

Existing techniques partially take advantage of this opportunity by controlling the order in which instructions are fetched from multiple threads [11], [40], [41]. However, in a typical deeply pipelined machine, the number of stages between the fetch and dispatch (i.e., insertion into the IQ) can be significant (we assume five such stages in our simulations). As a result, the specific situation that led to the decision not to fetch from a certain thread can be completely reversed by the time the fetched instructions actually reach the IQ. Furthermore, additional dynamic microarchitectural information about the instruction (i.e., the status of its physical registers) is readily available after register renaming. Consequently, if the final decision regarding which instructions to place into the IQ is postponed until after the register renaming stage, then the information presented in Fig. 5 can be directly exploited by not dispatching instructions with two nonready source operands into the queue, but instead blocking the dispatching from such a thread until one of the operands becomes ready.

Following these observations, we propose a design where the instructions can only enter the scheduling window if they have no more than one nonready register source operand, i.e., will only require one comparator. All instructions with two nonready operands (and their corresponding threads) will stall until one of the sources becomes ready. In the rest of the paper, we refer to this design as 2OP_BLOCK.

While, generally, the two design targets of improving the performance and reducing the complexity are at odds with each other, the 2OP_BLOCK scheduler, surprisingly, achieves both of these goals at the same time. The scheduler complexity is reduced because each entry in the IQ can be simplified to have only one set of comparators. A more nonintuitive result is the increased IPC performance compared to a similarly sized traditional IQ. Despite the fact that the dispatch stage can be completely blocked in some cycles (when all threads have their oldest nondispatched instructions with two nonready source operands at the same time), higher overall performance is still realized, even for fairly sizable IQs, because the queue is utilized much more efficiently. Specifically, the same IQ entry can be reused multiple times by the instructions with at most one nonready source operand instead of allowing a single instruction with two nonready sources to occupy the entry for a long time. Since the instructions with two nonready register sources are likely to wait for a long time in the IQ anyway, it is more beneficial for performance to have them wait at the dispatch stage, freeing up the valuable space in the scheduler for other instructions which are likely to be issued faster (such as the instructions from other threads which enter with some of their source operands ready).

Notice that, while one thread is blocked, the other threads can continue moving through the front end as long as they do not encounter instructions with two nonready sources. Every cycle when the instructions from a blocked thread are considered for dispatching, the ready bits associated with the source operand registers of the blocked instruction are reexamined. If one of these registers becomes ready, the thread is unblocked and further fetches, renames, and dispatches from that thread resume. Notice that such checking of the ready bits is nothing unique to our scheme; such checks are routinely performed in the baseline machine to determine the status of the source register operands before the instruction is moved into the IQ.

This mechanism naturally works in a data path that has completely separate front-end pipelines (including the fetch buffers and register renaming logic) for each thread. In our evaluations, we assume this model. However, even if the rename logic is shared among the threads, the selective blocking of certain threads can be accomplished as follows: Consider Intel's Hyperthreaded P4 [44], which implements a fully shared pipeline from fetch to commit. In this design, the rename bandwith is shared between the threads, but separate fetch buffers per thread are used before the rename stage in order to allow the stalling of one thread in the buffer without disrupting the processing of instructions from the other threads. To support the 2OP_BLOCK design on such a pipeline, we propose keeping the instructions in the per-thread fetch buffers during the rename process and only invalidate the entries in the fetch buffers after the instructions are successfully placed into the IQ. If an instruction with two nonready operands is encountered, it is not placed into the IQ and, instead, its fetch-buffer entry is marked as "not-sent" again so that it will be resubmitted when one of the operands arrives. In this way, instructions from the other threads can still harness the full rename bandwidth without being disturbed by the blocked thread. Finally, in order to unblock a thread from the fetch buffer, we maintain one bit per logical register in the rename table called the *offending_source_bit*. This bit is set to 1 for each source register of an offending instruction. When a register value is produced and the register is marked as "ready," the corresponding offending_source_bit is checked. If this bit is set, the "unstall" signal is sent to the fetch buffer for that

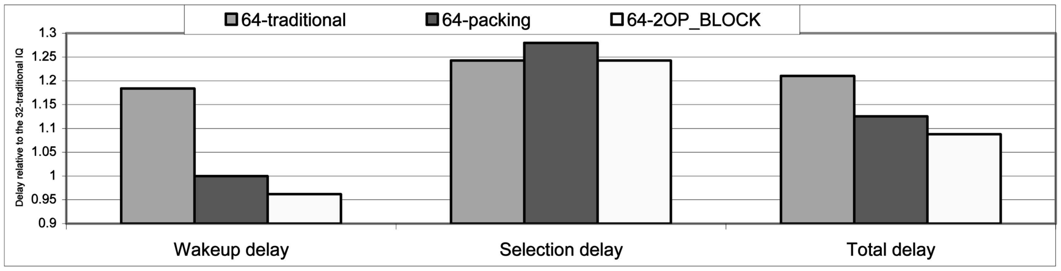


Fig. 6. Circuit delays of various scheduler configurations (relative to the 32-traditional design).

thread and processing of instructions from the thread continues.

4.3 Synergy of 2OP_BLOCK with Instruction Packing and Timing Issues

While the 2OP_BLOCK scheduler can be described and implemented completely independently of instruction packing (as presented in the previous section), additional advantages, primarily from the timing and power perspectives, can be realized if the physical layout of the IQ which supports instruction packing (and is shown in Fig. 2b) is leveraged. In this section, we describe the synergistic advantages of instruction packing and 2OP_BLOCK.

The 2OP_BLOCK mechanism simplifies the instruction packing scheduler of Fig. 2b in a number of ways. First, as only instructions with at most one nonready operand are present in the scheduler, both the multiplexer and the p-device that drive the request signals to the select logic are no longer needed. Second, the MODE bit, the A bit, and the R bit can be removed. Furthermore, the AND gate (shown at the bottom of Fig. 2b) is not required as the readiness of the instruction is manifested by the ready bit of a single operand. Finally, the capability of searching several allocation bits in parallel in the course of setting up the IQ entries is no longer necessary. The resulting combination of 2OP_BLOCK and the instruction packing scheduler is also less complex than the traditional scheduler because of the use of fewer comparators, shorter tag buses, and the absence of the AND gates to drive the final match signal. It is important to understand that the IPC advantages of the 2OP_BLOCK design come only because of the more efficient usage of the queue and have nothing to do with the underlying circuitry (such as instruction packing logic). The instruction packing logic only provides additional advantages in terms of lower access delay and power dissipation.

We now examine the delays of the 2OP_BLOCK scheduler (which uses the simplified IQ layout derived from instruction packing, as described above) and compare it to that of the traditional IQ as well as the IQ supporting a full-fledged implementation of instruction packing (as presented in Fig. 2b). Fig. 6 presents the wake-up, selection, and overall delays for various schedulers with the maximum capacity of 64 instructions relative to the traditional scheduler design with 32 entries. As seen from the graph, for the same capacity, the 2OP_BLOCK design has the lowest overall delay, which is 9.5 percent smaller than the delay of the 64-traditional queue and 3.5 percent smaller than the delay of the full-fledged packing queue. Compared to packing, this delay reduction comes from two main sources. The wake-up delay is reduced because the AND gate used to drive the final match signal in both traditional and instruction packing designs has been eliminated (this is also the reason why the wake-up delay of 64-2OP_BLOCK is somewhat smaller than the wake-up delay of the 32-traditional design). Second, the selection delay is reduced because of the elimination of the multiplexer.

5 Adaptive Techniques for Supporting a Limited Number of Threads

The 2OP_BLOCK scheduler increases the efficiency of the IQ usage, but relies on the abundant instruction supply from multiple threads to overcome the performance barriers imposed by blocking the instruction dispatching from some threads for possibly prolonged periods of time. With a small number of threads from which to choose the instructions, such a limitation can have a huge impact on performance. The extreme situation occurs, of course, when only a single thread is executing, in which case, the performance losses are very significant (up to 40 percent, on average, for 64-entry schedulers).

To address the issues of 2OP_BLOCK performance in the environments with a limited number of threads (the case of a single-threaded execution being an extreme example), one can deploy the full-fledged instruction packing logic to implement the queue (as opposed to the simplified circuitry that only supports instructions with at most one nonready source) and dynamically switch between the packing mode and the 2OP_BLOCK mode depending on the number of active threads. Using the instruction packing circuitry from Fig. 2b in conjunction with the 2OP_BLOCK design is especially attractive because, as seen in the results presented by Fig. 6, the increase of the circuit delay is very small if the full-fledged packing is implemented as opposed to the simplified logic, which does not support the instructions with two nonready source operands.

The switch between the two modes is a simple matter of changing instruction dispatch to allow or disallow the instructions with two nonready sources to enter the scheduler. In this simple implementation, when the number of threads is no greater than two, the scheduler always operates in the packing mode; otherwise, it switches to the 2OP_BLOCK regime.

This switch between the packing mode and the 2OP_BLOCK mode can even be done dynamically, depending on the program phases and corresponding microarchitectural statistics, to better address the scheduling needs of the specific workloads or the phases thereof. In this paper, we evaluate such dynamic switching between packing and 2OP_BLOCK modes.

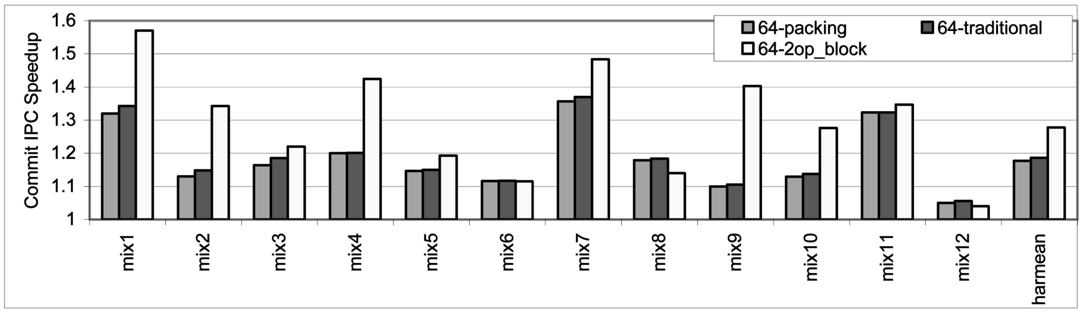


Fig. 7. Speedups (in terms of throughput IPC) over the 32-traditional IQ for various scheduler designs for the 4-way SMT machine.

The central aspect of such a hybrid packing/2OP_BLOCK scheme is in identifying the right triggers to force the switching between two modes. We experimented with several alternatives and our best results (presented in Section 6) were achieved for the following scheme:

Two separate counters are maintained for aiding in the decision to switch between packing and 2OP_BLOCK modes. These counters are sampled at the end of a periodic timing interval, which is called the *SAMPLING_PEROID*. In our experiments, we tried various sampling periods and present results in Section 6 for the period of 1,024 cycles, which provided the best performance.

The first counter is used to determine when to switch out of 2OP_BLOCK mode (and into packing mode). For this trigger, the number of cycles at which the dispatch of all threads is stalled due to the presence of instructions with two nonready operands is used. If this count exceeds 64 cycles (5 percent of the sampling period; the reason for such selection will become clear when one examines the trends presented in Fig. 9, Section 6) at the end of SAMPLING_PERIOD, then the 2OP_BLOCK mode is exited. The intuition here is that the 2OP_BLOCK mode alone fails to perform well when the amount of TLP is not sufficient. In these cases, the number of cycles at which the dispatch of instructions from all threads is stalled due to the presence of instructions with two nonready operands becomes very large and is not sufficiently compensated for by the TLP. For such workloads (or such phases), it is beneficial to operate in the packing mode.

The second counter is used for the decision to switch out of packing mode and into 2OP_BLOCK mode. For this, a counter of the average IQ occupancy is used. If the average occupancy at the end of the sampling period exceeds 90 percent of the IQ size, then packing mode is exited. In these cases, the IQ is highly utilized and, thus, there may be sufficient parallelism in the workload to benefit from the 2OP_BLOCK mode.

The implementation of these triggers requires two counters. The first counter (which counts the number of dispatch stalls due to the 2OP_BLOCK condition) can be implemented as an N-bit saturating counter for the sampling period T, where $N = \log_2(T * 5\%)$. For the sampling period of 1,024 cycles, this requires a 6-bit saturating counter. The second counter, which maintains the average IQ occupancy, requires a counter of $\log_2(IQ\_Size * Sampling\_Period)$ bits. For an IQ size of 64 entries and the sampling period of 1,024, this requires 16 bits.

Finally, note that, when the queue operates in packing mode, then only the queue occupancy counter is monitored. When the queue operates in the 2OP_BLOCK mode, only the counter of the number of dispatch stalls is examined. For this reason, the physical implementation can only require one counter to be maintained—this counter is reset in the beginning of every *SAMPLING_PERIOD* and it is used in the next period in the fashion determined by the current mode of operation.

6 Results and Discussions

As mentioned previously, when we compare the performance of different schedulers, the sizes of the IQ are measured in terms of the maximum number of instructions that may be present in the queue simultaneously. Specifically, an N-traditional IQ holds up to N instructions; an N-2OP_BLOCK scheduler can store up to N instructions, each with at most one nonready source; and an N-packing scheduler can hold up to N-instructions (if all the instructions have at most one nonready source), but typically holds fewer than N instructions due to the presence of some instructions with two nonready source operands.

Fig. 7 presents the IPC improvements of various scheduler designs considered in this paper with the total capacity of 64 instructions. The results are shown as relative improvements over the baseline IQ with 32 entries. On average, across all simulated workloads, the performance gains with respect to the 32-traditional design are 17 percent, 18 percent, and 28 percent for 64-packing, 64-traditional, and 64-2OP_BLOCK schedulers, respectively. It is interesting to observe that the 2OP_BLOCK design outperforms the traditional scheduler of the same capacity in all simulated mixes, except for mixes 6, 8, and 12. On the average, 64-2OP_BLOCK outperforms 64-traditional scheduler by 14 percent. Some mixes show especially large gains, up to 39 percent on mix 1.

Fig. 8 presents similar results in terms of the fairness metric (which is a harmonic mean of weighted IPCs). Here, the relative difference between the various schedulers is somewhat smaller, but, still, the 2OP_BLOCK design outperforms the traditional queue on all but three mixes (mixes 3, 8, and 12) as well as on average. On average, the performance of the 64-2OP_BLOCK is better than the performance of 64-traditional design by 10 percent according to the fairness metric.

While the 2OP_BLOCK scheduler disallows all instructions with two nonready operands from entering the scheduling window, the thread-level parallelism present in SMT designs naturally allows such limitations to be overcome. Even as the dispatch from one thread may be

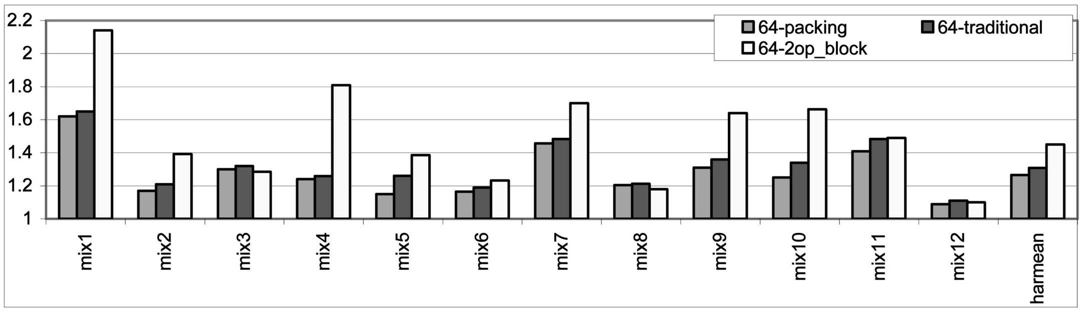


Fig. 8. Fairness improvements (in terms of the harmonic mean of weighted IPC) over the 32-traditional IQ for various scheduler designs for the 4-way SMT machine.

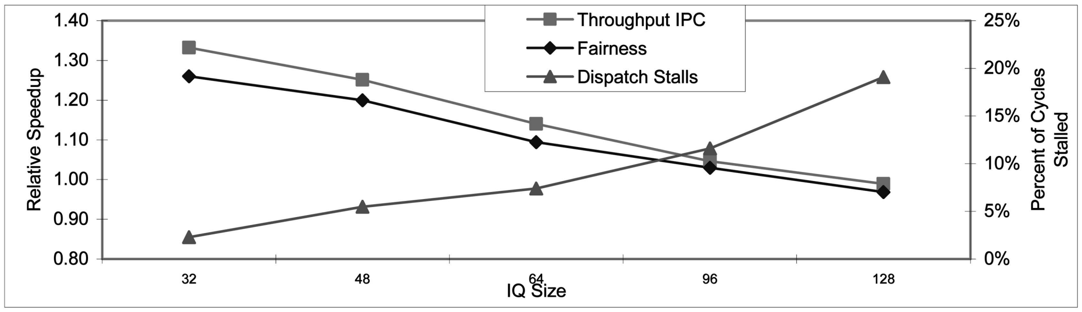


Fig. 9. Speedups of the *N-2OP_BLOCK* scheduler over the $N\text{-}traditional$ scheduler in terms of both throughput IPC and harmonic mean of weighted IPC (axis on the left) and the percentage of cycles in which the dispatch is stalled for all threads (axis on the right) for schedulers of various sizes on the 4-way SMT machine.

stalled, dispatch from other threads may continue. Our results show that 2OP_BLOCK is effective when the percentage of cycles in which the dispatch of all threads is stalled due to the presence of instructions with two nonready sources is rather small. In these cases, the advantages of the 2OP_BLOCK scheme outweigh its potential limitations.

Fig. 9 presents the scalability analysis of the 2OP_BLOCK scheduling logic. Results show the relative performance increase of the 2OP_BLOCK scheduler compared to the traditional scheduler of the same total capacity for various sizes of the IQ and are presented in terms of both throughput IPC and the fairness metric (y-axis on the left) and the percentage of cycles in which the dispatch was stalled because the oldest not-yet-dispatched instructions from all threads had both of their source operands in the nonready state (y-axis on the right). As seen from the graph, the 2OP_BLOCK design outperforms the traditional queue for up to 96-entry schedulers. Note that the gains are especially high for smaller schedulers: For example, the 32-2OP_BLOCK IQ outperforms the 32-entry traditional IQ by as much as 33 percent on average across all simulated mixes according to the throughput IPC metric and by 27 percent according to the fairness metric. As the size of the traditional queue increases beyond 96 entries, the dispatch stalls introduced by 2OP_BLOCK start to dominate its advantages and the resulting performance is actually lower than that of the traditional designs. In particular, the percentage of cycles during which the supply of instructions completely stops because all threads have instructions with two nonready source operands increases dramatically as the IQ size is increased beyond 96-entries. These stalls prevent the 2OP_BLOCK design from fully utilizing the entries for very large IQs, resulting in a performance degradation compared to the traditional queue of the same capacity. Essentially, at large IQ sizes, the extent to which 2OP_BLOCK relieves the IQ pressure is minimal (because the IQ itself is less of a bottleneck), but, instead, additional dispatch stalls are introduced. The percentage of cycles when all threads are simultaneously blocked at dispatch due to the conditions imposed by 2OP_BLOCK increases significantly with larger scheduling windows, as seen in Fig. 9. This is because the larger number of nonexecuted instructions buffered in the scheduler increases the number of nonready registers.

We now evaluate the performance of 2OP_BLOCK on 3-threaded and 2-threaded workloads. The respective results, in terms of both throughput IPC and fairness, are presented in Figs. 10 and 11. For 2-threaded workloads (Fig. 10), 2OP_BLOCK consistently underperforms the baseline machine for all examined IQ sizes. This is not surprising as the amount of thread-level parallelism is limited and the percentage of cycles when dispatch stalls due to the 2OP_BLOCK-induced conditions increases. On 3-threaded workloads, 2OP_BLOCK outperforms the baseline scheduler for small schedulers (up to 64 entries), but, for larger schedulers, 2OP_BLOCK still shows some performance degradations, although the extent of this degradation is much smaller than for 2-threaded workloads. Specifically, 2OP_BLOCK exhibits performance gains of 18.2 percent, 12.1 percent, and 3.7 percent in terms of throughput IPC and 11.2 percent, 7.0 percent, and 0.6 percent in terms of the fairness metric for the IQ sizes of 32, 48, and 64-entries, respectively. For the scheduler sizes of 96 and 128 entries, the 2OP_BLOCK

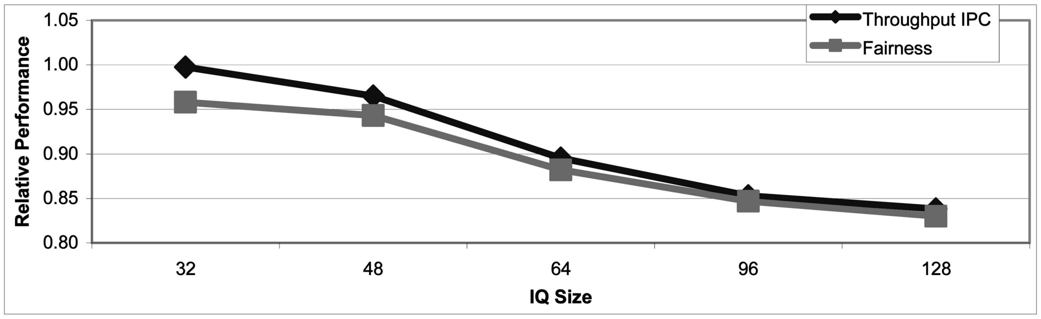


Fig. 10. Performance of 2-threaded workloads with 2OP_BLOCK compared to the baseline machine.

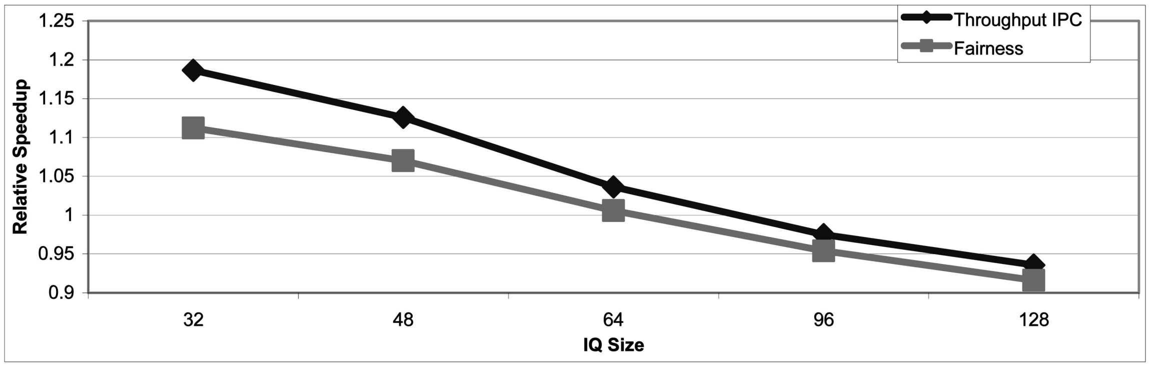


Fig. 11. Performance of 3-threaded workloads with 2OP_BLOCK compared to the baseline machine.

design degrades performance by 3 percent and 7 percent, respectively.

To mitigate the performance degradations experienced by 2OP_BLOCK for workloads without sufficient parallelism, we now present the results for the hybrid scheme that was introduced in Section 5. Figs. 12, 13, and 14 present the throughput IPCs for 4-threaded, 3-threaded, and 2-threaded workloads, respectively, for the various scheduling schemes relative to the traditional scheduler of the corresponding size. Results are presented for packing, 2OP_BLOCK, and the hybrid scheme that dynamically switches between the two. Similar trends were observed for the fairness metric; we do not present those results due to space limitations.

For 2-threaded workloads, instruction packing always performs better than 2OP_BLOCK and performance of the hybrid scheme comes very close to the performance of packing. Specifically, for 64-entry schedulers, the performance of the hybrid scheme comes within 0.4 percent of that of the instruction packing, which represents a 3.2 percent gain over the 2OP_BLOCK design of the same size. The opposite is true for the 4-threaded workloads, where the performance of the hybrid scheme is almost identical to that of 2OP_BLOCK, and both significantly outperform packing. For the schedulers with the capacity to hold up to 64-instructions, the performance of the hybrid scheme comes within 2 percent of that of the 2OP_BLOCK scheduler. The most interesting situation happens for 3-threaded workloads (Fig. 14). Here, the hybrid scheme actually outperforms both packing and 2OP_BLOCK on average. This is because, for 3-threaded workloads, packing outperforms 2OP_BLOCK on some mixes, while 2OP_BLOCK outperforms packing on others. To clarify, the hybrid scheme does not provide performance that is better than both instruction packing and 2OP_BLOCK for any one *individual* workload, but is able to improve the *average* performance by selecting the best for each workload (which may be instruction packing for some and 2OP_BLOCK for others). In this manner, the hybrid scheme is able to harness both the benefits of instruction packing and 2OP_BLOCK to provide a scheduler design that is scalable with both the IQ size and the number of threads.

Figs. 15, 16, and 17 present the detailed, per-workload IPC improvements for various scheduler configurations. The bars in these graphs depict the performance of the instruction packing design, the hybrid technique from Section 5, and the basic 2OP_BLOCK design. Also, for comparison purposes, we present the performance of the Tag Elimination design proposed in [15]. In that design, some IQ entries have two comparators, others have just one comparator, and yet others have zero comparators. For this comparison, the 64-entries of the Tag Elimination scheduler were partitioned according to the percentage of instructions with zero, one, and two nonready operands presented in Fig. 3b. Specifically, this scheduler has eight entries with two tag comparators, 28 entries with one tag comparator, and 28 entries with no tag comparators. All the results in Figs. 15, 16, and 17 are presented as relative speedups (slowdowns) with respect to a traditional 64-entry scheduler.

The results for the 2-threaded workloads are presented in Fig. 15. As seen from the graph, instruction packing nearly always outperforms the 2OP_BLOCK design. The performance of the hybrid scheme is close to that of instruction packing and both noticeably outperform the Tag Elimination mechanism of [15]. For 3-threaded workloads (Fig. 16), the 2OP_BLOCK design outperforms packing on some mixes (mixes 1, 2, 4, 7, 8, 9, 10, and 12) and underperforms packing on all other mixes. Consequently, the hybrid scheme, which dynamically adapts to the better

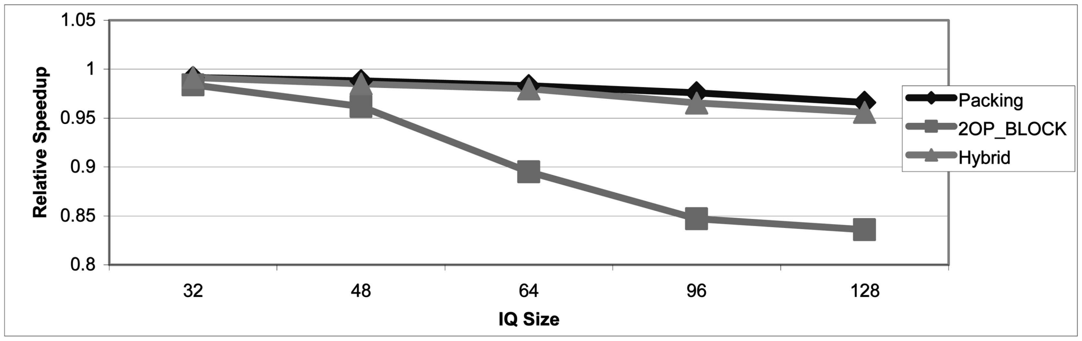


Fig. 12. Throughput IPC speedup of 2-threaded workloads for various scheduling schemes compared to the baseline machine.

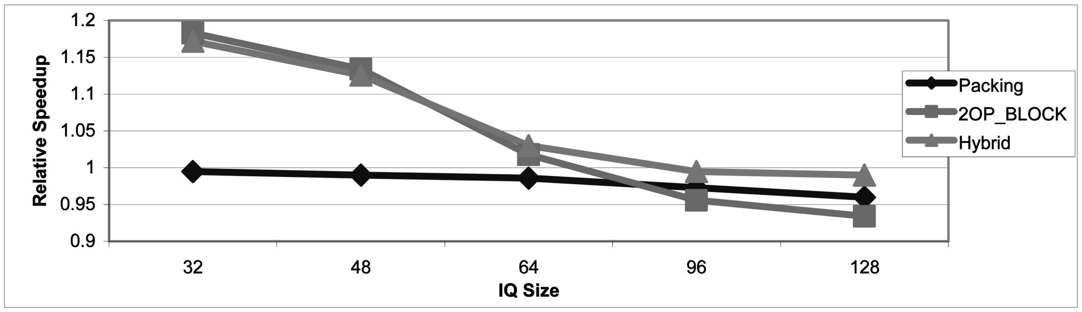


Fig. 13. Throughput IPC speedup of 3-threaded workloads for various scheduling schemes compared to the baseline machine.

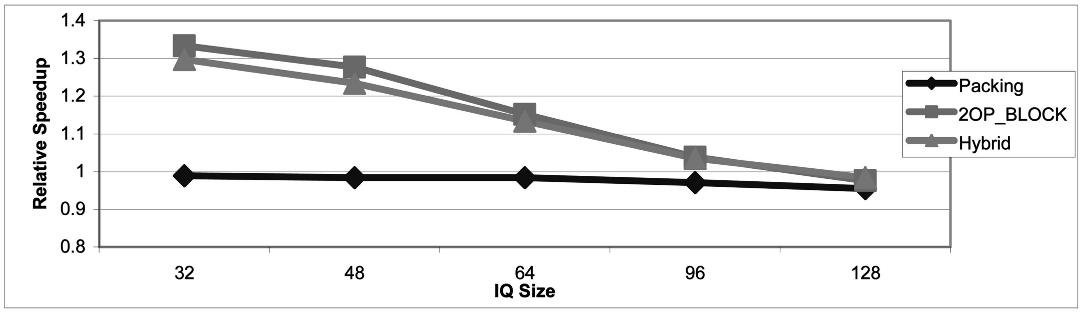


Fig. 14. Throughput IPC speedup of 4-threaded workloads for various scheduling schemes compared to the baseline machine.

of the two, outperforms both packing and 2OP_BLOCK on average. For 4-threaded workloads (Fig. 17), 2OP_BLOCK outperforms packing for all mixes except mixes 3 and 8.

Table 5 shows the percentage of the execution cycles that the hybrid scheme spends in the 2OP_BLOCK mode for the various IQ sizes and various number of simultaneous threads. The percentage is higher for the workloads with larger number of threads and smaller IQ sizes, as, in those configurations, the pressure on the IQ and the efficiency of the 2OP_BLOCK design is higher. For 2-threaded workloads (particularly with the larger IQ sizes), most of the cycles are spent in the packing mode, as expected.

Finally, we briefly discuss the implications of the proposed designs on the power consumption. It has been shown in [34], [35] that instruction packing achieves a significant reduction in the power dissipation of the scheduling logic for superscalar machines—in fact, power reduction was the primary goal of those works. Naturally, these results still hold true for SMT if we compare similarly sized instruction packing and traditional schedulers. The 2OP_BLOCK scheduler results in slightly more power reduction even compared to the instruction packing scheduler. This is because some logic (such as the AND gate, the allocation bit vectors, etc.) is eliminated. However, this power reduction is expected to be small (within a few percentage points) because the scheduling power is dominated by the dissipations expended in the course of wake-up tag broadcasts as well as within the logic of the selection tree. None of these components are affected by the 2OP_BLOCK scheduler compared to instruction packing. For more detailed power-related comparisons between the instruction packing and the traditional scheduler, we refer the readers to [34], [35].

7 Supporting Speculative Scheduling and Instruction Replays

For the sake of clarity in presenting the concepts of the SMT scheduler designs, our discussions in the previous sections ignored the issues that arise when the scheduling logic with a reduced number of tags (such as instruction packing, 2OP_BLOCK, or tag elimination technique of [15]) is incorporated into the data path that supports speculative scheduling based

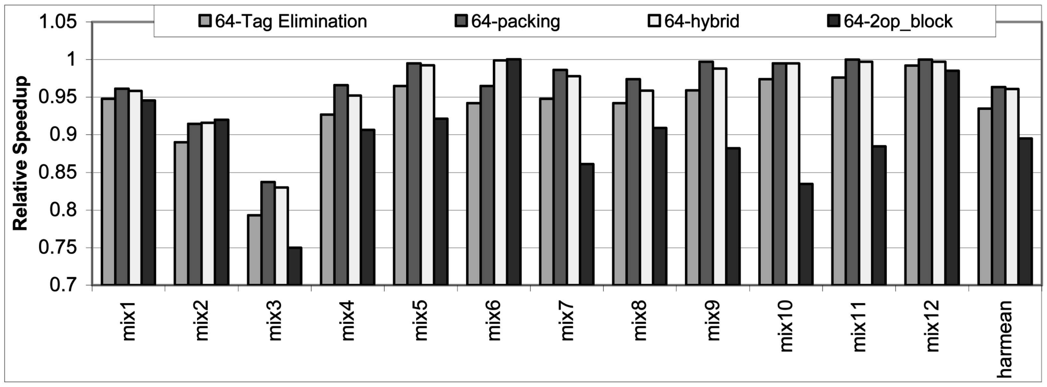


Fig. 15. Throughput IPC speedup of the 2-way SMT machine for various scheduler designs relative to the 64-entry traditional IQ.

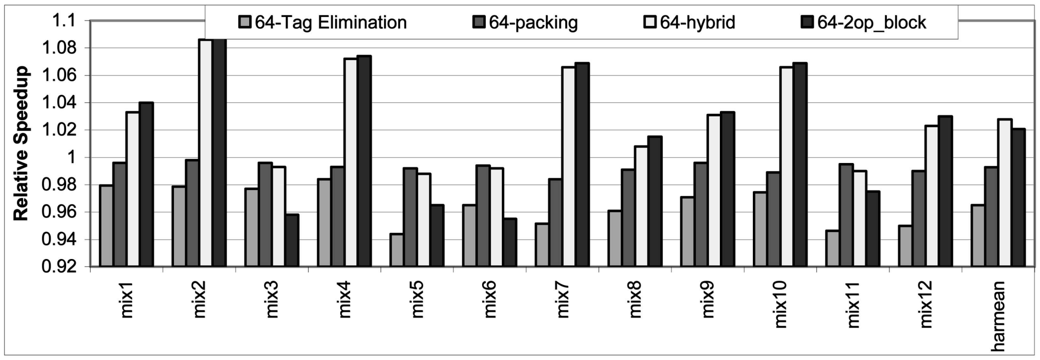


Fig. 16. Throughput IPC speedup of the 3-way SMT machine for various scheduler designs relative to the 64-entry traditional IQ.

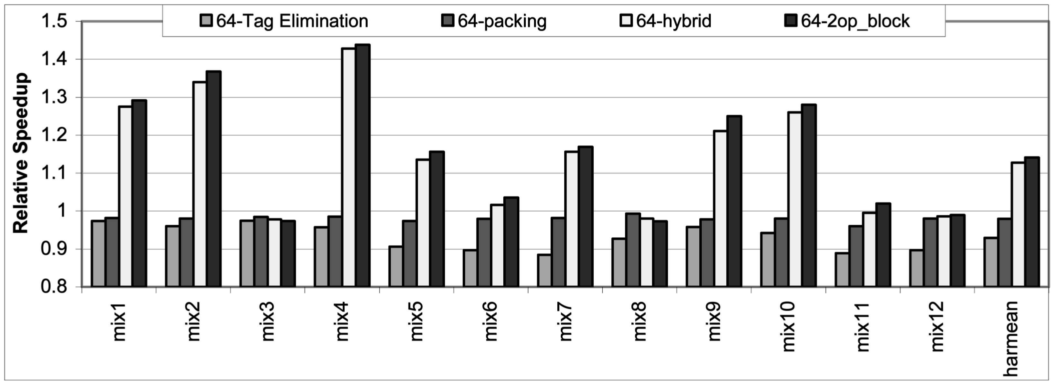


Fig. 17. Throughput IPC speedup of the 4-way SMT machine for various scheduler designs relative to the 64-entry traditional IQ.

on load latency prediction. In such designs, the instructions dependent on a load (and possibly their dependents as well) are scheduled speculatively, assuming that the load will hit into the L1 D-cache or relying on a more elaborate hit/miss prediction information. Upon a misprediction, the prematurely scheduled instructions need to be reexecuted (replayed). A comprehensive treatment of such scheduling mechanisms and various associated replay schemes is presented in [45]. As the number of stages between issue and execution increases in deeply pipelined high clock rate designs, it is virtually imperative that scheduling techniques support instruction replaying.

TABLE 5
Percentage of Cycles Spent in the 2OP_BLOCK Mode for the Hybrid Scheduler of Various Capacities

	IQ Size				
	32-entries	48-entries	64-entries	96-entries	128-entries
4threaded	98.8%	97.2%	90.3%	61.2%	43.5%
3threaded	99.1%	98.3%	68.4%	43.8%	33.6%
2threaded	52.1%	39.4%	5.8%	5.6%	5.2%

Associative instruction schedulers with a reduced number of tags, such as the ones described in this paper, present complications in this respect because the source operand tags of some instructions do not have the associated comparators and cannot participate in the wake-up process during a replay following a load-latency misprediction [45]. The problem arises when these instructions need to be replayed following a load latency misprediction and the source operand that was (speculatively) determined to be ready in the course of the initial instruction dispatching is no longer

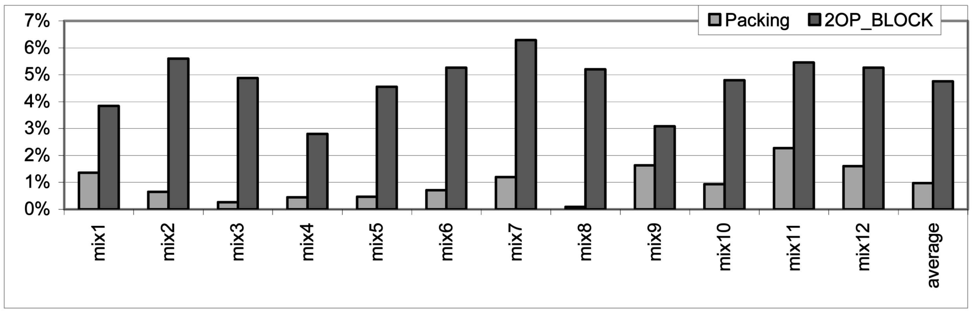


Fig. 18. IPC impact of using speculative versus nonspeculative ready bits for instruction packing and 2OP_BLOCK on the 4-way SMT machine.

ready after the misprediction. This occurs when the source operand in question depends, directly or indirectly, on the mispredicted load. In such a situation, the wake-up buses need to be monitored during replay to determine the readiness of the source in a nonspeculative manner. However, since the comparator is not available, such monitoring is impossible. The Tag Elimination mechanism of [15] has a similar problem.

Following a load-latency misprediction, all affected instructions typically get replayed, either from the main IQ or from a separate buffer [45], [46]. In [46], Ernst and Austin describe how such a replay mechanism can be used with tag elimination. The schemes proposed in this paper can trivially support instruction replays using a similar buffer. Due to the space limitation, we not discuss this particular implementation of replay mechanism further in detail and instead refer the readers to [45], [46] for more detailed discussions and the associated power/performance trade-offs.

For solutions where both regular and replaying instructions are issued out of the same IQ, we use the nonspeculative version of the "register ready" bit-vector to make the decision regarding the IQ allocation actions. A nonspeculative ready bit associated with a physical register is only set when the instruction producing that register value is no longer subject to the load-latency related replays. Basically, the speculative and the nonspeculative ready bits will be set a few cycles apart from each other and that difference is determined by the number of stages between instruction issue and execution. If the nonspeculative versions of the ready bits are checked to make the IQ allocation decisions, then instruction packing and 2OP_BLOCK can trivially support speculative scheduling and various forms of replays (such as squash recovery [45] or selective recovery [45], [46]) without any problems.

Fig. 18 shows the performance difference if the speculative versions of the ready bits are used instead of nonspeculative. For comparison with the idealistic case that uses speculative ready bits, we, of course, used a perfect predictor because, otherwise, speculative scheduling cannot be supported with packing or 2OP_BLOCK. For all other experiments, realistic nonideal predictors were used. On average, there is less than 1 percent IPC difference for instruction packing and less than 5 percent difference for 2OP_BLOCK. Therefore, the use of nonspeculative ready bits can support speculative scheduling with minimal impact on performance even if a separate replay queue is not used. *Note that, in all previously presented graphs, we used the nonspeculative ready bits, therefore the presented results fully support speculative scheduling*. Additional gains could be realized if speculative ready bits were used with some additional smart logic to support replays, but these gains are minimal (Fig. 18).

8 Related Work

Various fetching policies have been proposed in the literature to ensure the supply of instructions for building the most efficient execution schedules. The I-Count [41] gives fetching priority to the threads with fewer not-yet-executed instructions. One deficiency of the I-Count is that it does not effectively handle situations where the instructions pile up in the IQ following an L2 cache miss. As a result, the IQ entries are occupied by such instructions for a long time. Several optimizations of I-Count have been proposed to address this deficiency. STALL [40] prevents the thread from fetching further instructions if it experienced an L2 cache miss. FLUSH [40] extends STALL by squashing the already dispatched instructions from such a thread, thus making the shared IQ resources available for the instructions from other threads. FLUSH++ [11] combines the benefits of STALL and FLUSH and uses the cache behavior of threads to dynamically switch between these two mechanisms. The Data Gating technique of [14] avoids fetching from threads that experience an L1 data miss. Predictive Data Gating goes one step further and avoids fetching from the thread if it is predicted (during the early stage) that a cache miss will soon occur. While all these mechanisms are effective to various degrees, they do not exploit dynamic microarchitectural information about the instructions available after register renaming. In this paper, we show that additional benefits can be realized if such information is considered.

Several works proposed specific optimizations for the SMT processors. El-Moursy and Albonesi [14] explored new front-end policies that reduce the required integer and floating-point issue queue sizes in SMT architectures. Their techniques limit the number of nonready instructions in the queue from each thread and also block further instruction fetching from a thread if that thread experiences an L1 cache miss. As a result, the queue occupancy is reduced significantly (by about 33 percent) for the same level of performance. In [33], a partitioned version of the oldest-first issue policy is proposed, where separate issue queues are used to buffer the instructions from different threads. In [32], the effect of partitioning the data path resources, including the issue queues, across multiple threads is discussed. In [10], a more fine-grained dynamic control over SMT resources is proposed.

Researchers have proposed several ways to reduce the complexity and the power consumption of the issue logic in superscalar processors. Dynamic adaptation techniques [2], [6], [7], [17], [29] partition the queue into multiple segments and deactivate some segments periodically when the applications do not require the full issue queue to sustain the commit IPCs. The issue queues used in the SMT processors are generally less amenable to such optimizations because the occupancy of the queue is typically high as it is shared among multiple threads. Energy-efficient comparators, which dissipate energy predominantly on a tag match, were proposed in [30], [31]. Also in [30], the issue queue power was reduced by using zero-byte encoding and bitline segmentation. All of these techniques can be naturally applied to the SMT processor without any changes. In [20], the associative broadcast is replaced with indexing to enable only a single instruction to wake-up.

The observation that many instructions are dispatched with at least one of their source operands ready is not new —it was used in [15], where the scheduler design with a reduced number of comparators was proposed. In that scheme, some IQ entries have two comparators, others have just one comparator, and yet others have zero comparators. While the work of [15] statically partitions the queue into the groups of entries with various numbers of tag comparators, instruction packing achieves this partitioning dynamically; thus it can better adjust to the characteristics of the executing programs and results in lower performance degradation, as shown in [35].

In [22], the tag buses were categorized into fast buses and slow buses such that the tag broadcast on the slow bus takes one additional cycle. While the technique proposed in [22] can be trivially adapted to SMT, the design proposed in this paper (2OP-BLOCK scheme, in particular) completely eliminates the second set of comparators and, therefore, obviates the need to perform last-tag speculation and maintain fast and slow wake-up buses. The capacitive loading on all tag buses is reduced because half of the comparators are offloaded from every tag bus.

Several techniques have been proposed to pipeline the scheduling logic on a superscalar machine into separate wake-up and selection cycles without commensurate degradation in the IPCs [21], [39]. Other proposals have introduced new scheduling techniques with the goal of designing scalable dynamic schedulers to support a very large number of in-flight instructions [3], [12], [23], [24], [28]. Brown et al. [4] proposed removing the selection logic from the critical path by exploiting the fact that the number of ready instructions in a given cycle is typically smaller than the processor's issue width. The technique of [4] is less likely to be applicable to SMT as the number of ready instructions increases.

Scheduling techniques based on predicting the issue cycle of an instruction [1], [8], [9], [16], [19], [24], [26], [36] remove the wake-up delay from the critical path and remove the CAM logic from instruction wake-up, but need to keep track of the cycle when each physical register will become ready. In [13], the wake-up time prediction occurs in parallel with the instruction fetching. Future research is needed to determine the effectiveness of these techniques on an SMT processor.

9 Concluding Remarks

We examined several mechanisms for improving the scalability, reducing the complexity and delays, and increasing the throughput of the instruction schedulers in multithreaded processors. We demonstrated that the instruction packing—a technique to pack multiple instructions into the same issue queue entry—is more effective on an SMT than it is on a superscalar. This is because the percentage of instructions that enter the scheduling window with two nonready register source operands on a 4-way SMT is significantly lower than on a superscalar machine. We then proposed the 2OP_BLOCK scheduler—a scheduling technique that completely disallows the dispatch of instructions with two nonready sources, thus significantly simplifying the IQ logic. This mechanism works well for SMTs because it often allows the reuse of the same IQ entry multiple times for the instructions with no more than one nonready source rather than tying up the entry with an instruction with two nonready sources (which typically spend a longer time in the queue). We also proposed a hybrid scheduling scheme that combines the advantages of packing and 2OP_BLOCK by dynamically switching between these two modes. Such dynamic switching provides good performance, even in the environments with limited number of threads. Finally, we considered the implications of the proposed techniques on a data path that employs speculative instructions scheduling based on load latency prediction. We showed that all the proposed schemes support such speculative scheduling.

Acknowledgments

This paper is an extension of a paper which appeared in the *Proceedings of the International Symposium on High Performance Computer Architecture* (HPCA), 2006.

References

[1] J. Abella and A. Gonzalez, "Low-Complexity Distributed Issue Queue," *Proc. 10th Int'l Symp. High Performance Computer Architecture (HPCA),* 2004.

[2] P. Bose et al., "Early Stage Definition of LPX: A Low Power Issue-Execute Processor," *Proc. Power Aware Computer Systems Workshop (PACS),* 2002.

[3] E. Brekelbaum et al., "Hierarchical Scheduling Windows," *Proc. Int'l Symp. Microarchitecture (MICRO),* 2002.

[4] M. Brown et al., "Select-Free Instruction Scheduling Logic," *Proc. 34th Int'l Symp. Microarchitecture (MICRO),* 2001.

[5] D. Burger and T. Austin, "The SimpleScalar Tool Set: Version 2.0.," technical report, Dept. of Computer Science, Univ. of Wisconsin-Madison, June 1997.

[6] A. Buyuktosunoglu et al., "A Circuit-Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors," *Proc. Great Lakes Symp. VLIS,* 2001.

[7] A. Buyuktosunoglu et al., "Energy-Efficient Co-Adaptive Instruction Fetch and Issue," *Proc. Int'l Symp. Computer Architecture (ISCA),* 2003.

[8] R. Canal and A. Gonzalez, "A Low-Complexity Issue Logic," *Proc. Int'l Conf. Supercomputing (ICS),* 2000.

[9] R. Canal and A. Gonzalez, "Reducing the Complexity of the Issue Logic," *Proc. Int'l Conf. Supercomputing (ICS),* 2001.

[10] F. Cazorla et al., "Dynamically Controlled Resource Allocation in SMT Processors," *Proc. Int'l Symp. Microarchitecture,* 2004.

[11] F. Cazorla et al., "Improving Memory Latency Aware Fetch Policies for SMT Processors," *Proc. Int'l Symp. High Performance Computing,* 2003.

[12] A. Cristal et al., "Out-of-Order Commit Processors," *Proc. Int'l Symp. High Performance Computer Architecture (HPCA),* 2004.

[13] T. Ehrhart and S. Patel, "Reducing the Scheduling Critical Cycle Using Wakeup Prediction," *Proc. Int'l Symp. High Performance Computer Architecture (HPCA),* 2004.
[14] A. El-Moursy and D. Albonesi, "Front-End Policies for Improved Issue Efficiency in SMT Processors," *Proc. Int'l Symp. High-Performance Computer Architecture (HPCA),* 2003.
[15] D. Ernst and T. Austin, "Efficient Dynamic Scheduling through Tag Elimination," *Proc. Int'l Symp. Comp. Architecture (ISCA),* 2002.
[16] D. Ernst et al., "Cyclone: A Broadcast-Free Dynamic Instruction Scheduler with Selective Replay," *Proc. Int'l Symp. Computer Architecture (ISCA),* 2003.
[17] D. Folegnani and A. Gonzalez, "Energy-Effective Issue Logic," *Proc. Int'l Symp. Computer Architecture (ISCA),* 2001.
[18] J. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *Computer,* vol. 33, no. 7, pp. 28-35, July 2000.
[19] J. Hu et al., "Exploring Wakeup-Free Instruction Scheduling," *Proc. Int'l Symp. High Performance Computer Architecture (HPCA),* 2004.
[20] M. Huang et al., "Energy-Efficient Hybrid Wakeup Logic," *Proc. Int'l Symp. Low-Power Electronics Design (ISLPED),* 2002.
[21] I. Kim and M. Lipasti, "Macro-Op Scheduling: Relaxing Scheduling Loop Constraints," *Proc. Int'l Symp. Microarchitecture (MICRO),* 2003.
[22] I. Kim and M. Lipasti, "Half-Price Architecture," *Proc. Int'l Symp. Computer Architecture (ISCA),* 2003.
[23] A. Lebeck et al., "A Large, Fast Instruction Window for Tolerating Cache Misses," *Proc. Int'l Symp. Computer Architecture (ISCA),* 2002.
[24] Y. Liu et al., "Scaling the Issue Window with Look-Ahead Latency Prediction," *Proc. Int'l Conf. Supercomputing (ICS),* 2004.
[25] K. Luo et al., "Balancing Throughput and Fairness in SMT Processors," *Proc. Int'l Symp. Performance Analysis of Systems and Software,* 2001.
[26] P. Michaud et al., "Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors," *Proc. Int'l Conf. High Performance Computer Architecture (HPCA),* 2001.
[27] S. Palacharla et al., "Complexity-Effective Superscalar Processors," *Proc. Int'l Symp. Computer Architecture (ISCA),* 1997.
[28] S. Raasch et al., "A Scalable Instruction Queue Design Using Dependence Chains," *Proc. Int'l Symp. Computer Architecture,* 2002.
[29] D. Ponomarev et al., "Reducing Power Requirements of Instruction Scheduling through Dynamic Allocation of Multiple Datapath Resources," *Proc. Int'l Symp. Microarchitecute (MICRO),* 2001.
[30] D. Ponomarev et al., "Energy-Efficient Issue Queue Design," *IEEE Trans. VLSI Systems,* Nov. 2003.
[31] D. Ponomarev et al., "Energy-Efficient Comparators for Superscalar Datapaths," *IEEE Trans. Computers,* vol. 53, no. 7, July 2004.
[32] S. Raasch et al., "The Impact of Resource Partitioning on SMT Processors," *Proc. Parallel Computing Technologies Conf. (PACT),* 2003.
[33] B. Robatmili et al., "Thread-Sensitive Instruction Issue for SMT Processors," *Computer Architecture News,* 2004.
[34] J. Sharkey et al., "Reducing Delay and Power Consumption of the Wakeup Logic through Instruction Packing and Tag Memoization," *Proc. Fourth Workshop Power-Aware Computer Systems,* 2004.
[35] J. Sharkey et al., "Instruction Packing: Reducing Power and Delay of the Dynamic Scheduling Logic," *Proc. Int'l Symp. Low Power Electronics and Design (ISLPED),* 2005.
[36] J. Sharkey and D. Ponomarev, "Instruction Recirculation: Eliminating Counting Logic in Wakeup-Free Schedulers," *Proc. ACM/IEEE Euro-Par Conf.,* 2005.
[37] J. Sharkey, "M-Sim: A Flexible, Multi-Threaded Simulation Environment," Technical Report CS-TR-05-DP1, Dept. of Computer Science, State Univ. of New York Binghamton, 2005.
[38] T. Sherwood et al., "Automatically Characterizing Large Scale Program Behavior," *Proc. Architectural Support for Programming Languages and Operating Systems Conf. (ASPLOS),* 2002.
[39] J. Stark et al., "On Pipelining Dynamic Instruction Scheduling Logic," *Proc. Int'l Symp. Microarchitecture (MICRO),* 2000.
[40] D. Tullsen et al., "Handling Long-Latency Loads in a Simultaneous Multi-Threaded Processor," *Proc. Int'l Symp. Microarchtiecture,* 2001.
[41] D. Tullsen et al., "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," *Proc. Int'l Symp. Computer Architecture,* 1996.
[42] D. Tullsen et al., "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. Int'l Symp. Computer Architecture,* 1995.
[43] J. Sharkey and D. Ponomarev, "Efficient Instruction Schedulers for SMT Processors," *Proc. 12th Int'l Symp. High Performance Computer Architecture (HPCA),* 2006.
[44] D. Marr et al., "Hyperthreading Technology Architecture and Microarchitecture," *Intel Technology J.,* vol. 6, no. 1, Feb. 2002.
[45] I. Kim and M. Lipasti, "Understanding Scheduling Replay Schemes," *Proc. Int'l Symp. High Performance Computer Architecture (HPCA),* 2004.
[46] D. Ernst and T. Austin, "Practical Selective Replay for Reduced-Tag Schedulers," *Proc. Second Ann. Workshop Duplicating, Deconstructing, and Debunking (WDDD-2),* June 2003.

Joseph J. Sharkey received the BS degree in computer science from the State University of New York (SUNY) Binghamton in 2004 and the MS degree in computer science from SUNY Binghamton in 2005. He is currently a PhD student in the Department of Computer Science at SUNY Binghamton. His research interests are in computer architecture, specifically reliable and power-aware microarchitectures. He is a student member of the IEEE, the IEEE Computer Society, and the ACM.

Dmitry V. Ponomarev received the systems engineering degree from the Moscow State Institute of Electronics and Mathematics, Russia, in 1996, the MS degree in computer and information science from the State University of New York (SUNY), Institute of Technology at Utica/Rome, in 1995, and the PhD degree in computer science from SUNY Binghamton in 2003. He is currently an assistant professor in the Department of Computer Science at SUNY Binghamton. His research interests are in computer architecture, particularly in the optimizations of high-end microprocessors for energy efficiency. He is a member of the IEEE, the IEEE Computer Society, and the ACM.