Balancing ILP and TLP in SMT Architectures through Out-of-Order Instruction Dispatch

Joseph Sharkey and Dmitry Ponomarev
Department of Computer Science
State University of New York at Binghamton
{jsharke, dima}@cs.binghamton.edu

Abstract

Simultaneous Multi-threading (SMT) architectures open up new avenues for datapath optimizations due to the presence of thread-level parallelism (TLP). One recent proposal for exploiting such parallelism is the 2OP_BLOCK scheduler design, which completely avoids the dispatch of instructions with two non-ready source operands into the issue queue. This technique reduces the scheduler complexity and also provides performance benefits for workloads with sufficient TLP, as the issue queue is more efficiently utilized.

In this paper we first revisit the 2OP_BLOCK scheduler and show that this design actually results in performance losses for workloads with a limited number of threads because the constraints imposed on the exploitable ILP within each thread outweigh its advantages. To balance the ILP and TLP in SMT processors supporting such schedulers, we propose out-of-order dispatch of instructions within each thread. This simple augmentation naturally allows the 2OP_BLOCK scheduler to perform well even when the number of threads is small. Furthermore, for environments with a larger number of threads, the out-of-order dispatch mechanism improves the performance of the original proposal by up to 15% on the average across simulated multithreaded mixes of SPEC 2000 benchmarks.

1. Introduction and Motivation

As traditional techniques attempting to increase processor performance through the extraction of Instruction-Level Parallelism (ILP) within applications have reached the point of diminishing returns, the research focus has shifted towards the designs that exploit the parallelism across multiple threads of control, or Thread-Level Parallelism (TLP). Current designs realize TLP either through the use of Simultaneous Multithreading (SMT), where multiple threads execute together on a slightly enhanced superscalar core and share its key resources, or through the use of Chip Multiprocessing (CMP) when multiple processor cores, along with the cache memory subsystem and the interconnection fabric, are placed on a single die. Often, the two paradigms are combined such that each core of a CMP is also multithreaded. For example, IBM Power 5 is dual-core CMP, with each core being 2-way SMT [20]. The Intel Pentium Extreme Edition is also a dual-core processor supporting up to two simultaneous threads per core [19], and Intel's Montecito processor is a dual-core, dual-thread implementation of the Itanium Processor [18].

As the number of transistors available on a chip will continue to increase in future technologies, it is likely that a higher degree of multithreading will be supported within each core of future CMPs or within single-core SMT processors [22]. Therefore, it is important to consider techniques for increasing the efficiency of SMT-enabled cores, either in single or multi-core designs.

The SMT processors trade the ILP that can be momentarily extracted from within each thread for the larger amounts of TLP that are harvested across all threads. An optimized SMT design should carefully balance both the ILP and the TLP in order to achieve the best performance. For example, previous studies show that the techniques mostly relying on TLP (by disallowing speculative execution or limiting the instruction issue within each thread to be strictly in-order [21]) significantly underperform the techniques which support more complex scheduling mechanisms. On the other hand, an attempt to extract too much ILP from one thread can result in the monopolization of shared resources (such as the issue queue) by instructions from that one thread, thus denying these valuable resources to the instructions from other threads and also degrading the overall throughout.

The complex trade-offs between ILP and TLP in SMT processors are traditionally managed by the instruction fetch policies, which control the instruction delivery to the dynamic scheduling logic. Various fetching mechanisms have been proposed in the literature with the goal of avoiding the resource monopolization by any one thread. The dynamic instruction scheduling logic can then simply extract the available ILP from within each thread using the instructions supplied by the fetch mechanism. For example, the I-Count fetching policy [16] gives priority to threads with fewer not-yet-executed instructions that are already in the pipeline. Some optimizations to the I-Count policy that further increase the efficiency of the issue queue (IQ) usage have also been proposed. Fundamentally, these solutions attempt to avoid clogging the queue with instructions that reside there for a large number of cycles before being issued. For example, FLUSH [15], FLUSH++ [3] and the Data Miss Gating technique of [4] 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.

A recent study [13] has shown that augmenting these fetch policies with another level of control at the time of instruction dispatch, taking into account the register status information available after register renaming, can result in a more effective usage of shared datapath resources in SMT and higher performance in some configurations. The study of [13] showed that for an ISA with at most two source operands for each instruction, the instructions with two non-ready source operands at the time of dispatch 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. The authors of [13] then capitalized on this observation by proposing the 2OP_BLOCK mechanism – a design that prevents the instructions with two non-ready sources at the time of dispatch from entering the IQ until one of these sources becomes available. Such instructions, along with all subsequent instructions from the same thread are instead stalled.

The 2OP_BLOCK mechanism reduces the complexity of the IQ, because only the capability to support instructions with at most 1 non-ready register source operand is needed. Consequently, the access delay and the power consumption of the IQ are reduced. Additionally, the throughput IPCs also improve for the 4-threaded workloads for some IQ configurations, as substantial TLP available from 4 thread contexts can be exploited.

In this paper we revisit the 2OP_BLOCK design and show that it results in significant performance losses if the number of threads is limited (i.e. less than 4). We then propose a modification to the basic 2OP_BLOCK scheme that allows the instructions from each thread to be dispatched into the IQ out-of-order, while still maintaining the in-order register renaming process.

In our modified design, the renamed instructions which are piled up behind the one with 2 non-ready register sources are still allowed to enter the IQ, thus exposing deeper ILP from this thread to the scheduling logic. Although this technique shows very significant performance gains all across the board, the improvements are especially remarkable in the environments with a limited amount of exploitable TLP (i.e. when only a few threads are available). The out-of-order dispatch technique proposed in this paper achieves the following key results:

- For 4-threaded workloads, the out-of-order dispatch mechanism results in the additional 15% IPC improvement over 2OP_BLOCK for 64-entry schedulers and outperforms both 2OP_BLOCK and traditional designs for larger scheduler sizes as well. In contrast, the basic 2OP_BLOCK scheme only outperforms the traditional designs for the scheduler sizes of up to 64 entries.
- For 2-threaded workloads, the 2OP_BLOCK design exhibits consistently lower performance than the baseline machine (due to the lack of TLP), while the out-of-order dispatch mechanism outperforms both 2OP_BLOCK and the traditional scheduler for the IQ sizes of up to 64-entries (by exploiting deeper ILP within each thread). For 64-entry IQs, the out-of-order dispatch improves the performance over 2OP_BLOCK by 22% and over traditional scheduler by 2%.
- The behavior of 3-threaded workloads shows more complicated trends, combining the features observed in 2-threaded and 4-threaded mixes. For 64-entry IQs, 2OP_BLOCK again results in lower IPCs compared to the baseline machine, while the out-of-order dispatch outperforms the baseline case by 9%. The performance of out-of-order dispatch and traditional case roughly even out at 96-entry schedulers.

The rest of the paper is organized as follows. Our simulation methodology is described in Section 2. Section 3 examines the effectiveness of the previously proposed 2OP_BLOCK mechanism on workloads with a limited number of threads. The out-of-order dispatch mechanism is introduced in Section 4. Section 5 presents the evaluation of the results, the related work is described in Section 6, and our concluding remarks are offered in Section 7.

2. Simulation Methodology

For estimating the performance impact of the schemes described in this paper, we used M-Sim [12]: a significantly modified version of the Simplescalar 3.0d simulator [1] that supports the SMT processor model. M-Sim implements separate models for the key pipeline structures such as the IQ, the reorder buffer, and the physical register file; it also explicitly models register renaming. 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. 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 [16] was implemented and fetching was limited to two threads per cycle.

We simulated the full set of SPEC 2000 integer and floating point benchmarks [6], using the precompiled Alpha binaries available from the Simplescalar website [1]. We skipped the initialization part of each benchmark using the procedure prescribed by the Simpoints tool [14] 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.

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.

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, 96–entry ROB per thread
Function Units and Lat (total/issue)	8 Int Add (1/1), 4 Int Mult (3/1) / Div (20/19), 4 Load/Store (2/1), 8 FP Add (2), 4 FP Mult (4/1) / Div (12/12) / Sqrt (24/24)
Physical Registers	256 integer + 256 floating-point physical registers
L1 I–cache	64 KB, 2–way set–associative, 128 byte line
L1 D–cache	32 KB, 4–way set–associative, 256 byte line
L2 Cache unified	2 MB, 8–way set–associative, 512 byte line, 10 cycles hit time
BTB	2048 entry, 2–way set–associative
Branch Predictor	Per thread 2K entry gShare with 10-bit global history
Pipeline Structure	5-stage front-end (fetch-dispatch), scheduling, 2 stages for register file access, execution, writeback, commit.
Memory	64 bit wide, 150 cycles access 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*

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*

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 [8, 16]. Therefore, we also present the "fairness" metric of "harmonic mean of weighted IPCs" [8, 16], which accounts for individual per-thread performance.

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*

3. 2OP_BLOCK Scheduler Design for SMT and its Limitations

Several instruction scheduler designs with a reduced number of tag comparators per entry have been proposed in the recent literature [5,11,13]. Of those, the design of [13] specifically targets the SMT processors. The scheduler proposed in [13], called 2OP_BLOCK, capitalizes on the observation that instructions which enter the IQ with two non-ready sources typically wait for a much larger number of cycles before being issued compared to all other instructions. Therefore, if an abundant supply of instructions from multiple threads is available for dispatch, then it is advantageous for performance to avoid dispatching the instructions with two non-ready source operands into the IQ and instead make such instructions (and all subsequent instructions from the same thread) wait in the dispatch stage until at least one of the source operands becomes available. Such a dispatch mechanism results in more efficient use of the IQ, as the same IQ entry can be reused multiple times by different instructions instead of being hogged for a long time by an instruction entering the queue with two non-ready source operands. Consequently, both the throughput IPC and the fairness metric can be improved. At the same time, this design also results in a less complex, more power-efficient and faster IQ, as each IQ entry only needs one tag comparator.

When one thread is blocked at the dispatch stage waiting for one of the source operands of its oldest non-dispatched instruction to become available, the other threads can continue processing through the front end as long as they do not encounter instructions with two non-ready sources. Since typically the thread processing is split in the front end (e.g. each thread uses its own rename table), it is easy to block only the progress of one specific thread. Every cycle when the instructions from this particular thread are

considered for dispatching, the ready bits associated with the source operand registers of the blocked instruction are re-examined. If one of these registers becomes ready, the thread is unblocked and further fetches, renames, and dispatches from that thread resume. Such checks are not unique to this scheme; they are routinely performed in the baseline machine to determine the status of the source register operands before the instruction is moved into the IQ.

In essence, the 2OP_BLOCK design attempts to maximize the exploitation of TLP at the expense of temporarily limiting the amount the ILP that can be extracted from the individual threads, even compared to the baseline SMT. While indeed providing performance improvement in some configurations, the 2OP_BLOCK design can result in significant performance losses when the limitations on ILP (e.g. percentage of cycles when dispatch stalls due to all threads having instructions with two non-ready sources) outweighs the potential benefits attributed to higher TLP (e.g. more efficient usage of the queue). This is especially true if the number of simultaneous threads is limited.

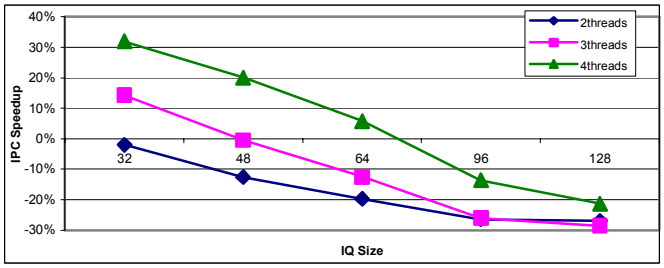

Figure 1: IPC speedup (harmonic mean across all mixes) of the 2OP_BLOCK scheduler compared to the traditional IQ of the same capacity for various IQ sizes.

The work of [13] only considered 4-threaded workloads, and it is therefore not surprising to see positive results reported in that paper. However, the current commercial implementations of SMT rarely support more than two threads [18,19,20]. We analyzed the performance of the 2OP_BLOCK scheduler for 2-threaded, 3-threaded, and 4-threaded workloads for various IQ sizes and arrived at some interesting conclusions. The main results of these simulations are summarized in Figure 1. In each case, we compare the performance of the 2OP_BLOCK schedulers against traditional schedulers of the same overall capacity. For the 4-threaded workloads, the 2OP_BLOCK scheduler provides significant speedups over the traditional queue for schedulers up to 64-entries – these results are in line with those presented in [13]. For schedulers larger than 64 entries, however, even the presence of 4 concurrent threads does not provide sufficient TLP to sustain the performance, and therefore the 2OP_BLOCK scheduler performs worse than the traditional IQ by 14% and 21% for 96-entry and 128-entry schedulers, respectively. Furthermore, the workloads with 2 threads experience performance degradations compared to the traditional scheduler for all sizes of the IQ – by as much as 19% on the average for the 64-entry schedulers. The situation with 3-threaded workloads is somewhere in between, with 2OP_BLOCK outperforming the traditional queue of the same capacity for 32-entry queues and breaking even for 48-entry queue. After that point, the performance degrades.

To understand these trends, it is useful to examine the statistics pertaining to the percentage of cycles when the dispatch of all threads stalls due to the conditions imposed by 2OP_BLOCK scheduling. For example, for a 64-entry queue, the average percentage of such stalled cycles is 7% for 4-threaded workloads, it increases to 17% for 3-threaded workloads, and balloons to 43% for 2-threaded workloads. This significant hindering of each thread's ILP is the main reason for the performance losses in various configurations examined in Figure 1. Therefore, we conclude that the 2OP_BLOCK scheduler design as proposed in [13] does not scale well with either the number of threads, or the size of the IQ. To address this deficiency, in the next section we explore a technique to balance ILP and TLP and therefore provide a more attractive scheduling solution in the presence of the IQ with a reduced number of tag comparators.

4. Augmenting 2OP_BLOCK with Out-of-Order Dispatch

As shown in the previous section, 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 to choose the instructions from, such a limitation can have a huge impact on the performance (as shown in Figure 1).

To address the issues of 2OP_BLOCK performance in the environments with a limited number of threads, we propose the use of out-of-order dispatching of instructions within each thread, i.e. opening up the opportunities to dispatch instructions with some of their operands ready, which would have otherwise piled up behind the blocked instructions with 2 non-ready sources. Such out-of-order dispatching naturally increases the ability to exploit deeper ILP within each thread.

To support the discussions in the rest of the paper, we introduce the term *Dispatchable Instruction (DI)* – which refers to an instruction that is considered for dispatch in a given cycle and for which an appropriate IQ entry (one containing the necessary number of tag comparators required by this particular instruction) is also available. In general, the number of dispatchable instructions may be equal to the number of instructions considered for dispatch (when there is sufficient number of entries available in the IQ), or it may be less than the number of instructions considered for dispatch (i.e. due to the presence of instructions with more non-ready source operands than the number of tag comparators in the available IQ entries). For example, for the 2OP_BLOCK scheduler, only instructions with at most one non-ready source operand can be dispatchable. An instruction that is considered for dispatch in a given cycle, but for which an appropriate IQ entry is not available is termed *Non-Dispatchable Instruction (NDI)* (for the 2OP_BLOCK scheduler, all instructions with 2 non-ready sources are NDIs).

In a superscalar machine (with or without reduced-tag schedulers), instruction dispatch operates in program order each cycle until either *W* instructions have been dispatched

(where *W* is the dispatch rate of the machine), or a non-dispatchable instruction is encountered. The same semantics hold true for the instructions within each thread of an SMT processor - dispatch occurs in-order within each thread, although it can occur out of fetch order for instructions between different threads. Thus, a single non-dispatchable instruction within a thread will stall dispatch of the *entire thread* until it becomes dispatchable (i.e. when one of its source operands becomes available). The 2OP_BLOCK scheduler is able to sustain the rate of instruction dispatching *unless* all threads encounter a non-dispatchable instruction simultaneously, in which case the supply of instruction to the out-of-order core of the SMT pipeline comes to a halt. With a limited number of threads, however, such situations are very frequent and they impact the performance drastically, as seen in Figure 1.

We now observe that, while a thread may contain one NDI in a given cycle, there may also be several DIs behind it in program order that are otherwise eligible for dispatch. In the basic 2OP_BLOCK design, the dispatch of this entire thread will stall, missing the opportunity to bring these DIs into the scheduling widow. These instructions are in a way hidden from the scheduler as an artifact of the in-order dispatch policy. In the rest of the paper, we call such instructions *Hidden Dispatchable Instructions*, or HDIs. In fact, we observe that almost 90% of instructions piled up behind the NDIs, can be classified as HDIs.

Further, we observe that both the NDI and HDI instructions piled up behind it in the 2OP_BLOCK case would be considered DIs with regular schedulers (e.g., schedulers without reduced number of tag comparators), and could be dispatched into the IQ all together in the same cycle (at least in the absence of other instructions considered for dispatch from other threads). Therefore, all necessary connections and write ports to the IQ to allow the dispatch of the HDI instructions already exist – it is simply that the 2OP_BLOCK scheduler does not make use of this hardware in an effort to increase the scheduler efficiency by exploiting TLP. The solution that we propose is to allow the dispatching of all HDIs into the IQ, effectively introducing out-of-order instruction dispatch from each thread. Notice that the register renaming, as well as the allocation of ROB and load/store queue entries, are still performed in program order within each thread, thus guaranteeing that all true data dependencies are still enforced correctly. In other words, the HDIs are dispatched from a buffer, which contains the instructions already in the renamed form.

Figure 2 clarifies this classification through an example of specific code segment. In terms of the example of Figure 2, both instructions I3 and I4 will be dispatched into the IQ before the instruction I2. Notice that while I4 is dependent on I2 and I3 is not dependent on it, both I3 and I4 are still dispatched prior to I2. While it could be more efficient to only dispatch the NDI-independent instructions out-of-order, the logic to perform such filtering would be complicated and would almost certainly impact the cycle time. At the same time, our simulation results actually showed that even under the idealized assumption of perfect and zero-overhead filtering, the potential to further boost the IPC is very limited if such filtering is implemented – the IPCs only improved by about 1.2% on the average. This is because only about 10% of all HDIs which entered the IQ out of program order were directly or indirectly dependent on a prior NDI. Therefore, the performance impact of foregoing the filtering opportunity and blindly dispatching all HDIs into the IQ is minimal.

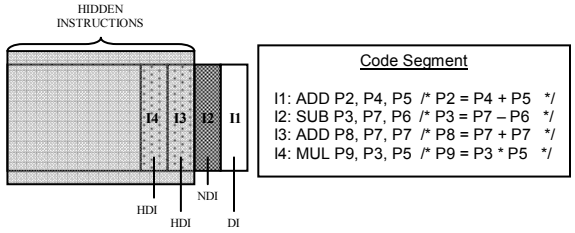

Figure 2: Example

One potential problem with the out-of-order instruction dispatching is the possibility of a deadlock. Consider the situation where the oldest instruction in the ROB is blocked at dispatch because there are no available IQ entries and that all younger instructions from that thread that have already been dispatched out-of-order into the IQ are directly or indirectly dependent on this oldest instruction. If this scenario transpires simultaneously for all the threads, then the processor comes to a deadlock state and no instruction can be committed or dispatched. While such an occurrence would be extremely rare, it is necessary to provide a mechanism that either avoids such deadlocks, or detects and recovers from them.

Several solutions are possible to address the deadlock-related issues arising with the out-of-order dispatch. One alternative is to rely on a simple *watchdog timer*, which is a counter that counts down the number of cycles since the last instruction was dispatched. The timer can be initialized to a value exceeding the largest expected delay between consecutive dispatches in the course of normal execution (something in the order of 2 to 3 times more than the main memory access latency). This timer is decremented every cycle when no dispatches take place and it is reset back to its maximal value when a dispatch of an instruction occurs. When the value of the watchdog timer reaches zero, the PCs of all threads are reset to the addresses of the oldest instructions in the corresponding ROBs and the pipeline is flushed. While the implementation of this mechanism requires little additional hardware (only a small counter is needed; the capability to flush the pipeline is already present for handling branch mispredictions, exceptions, and interrupts), the performance penalty due to the pipeline flushes could be non-negligible. Therefore, in our evaluations we used a more elegant technique that does not require pipeline flushing when deadlocks are detected (as in watchdog timer design), but instead avoids the occurrence of deadlocks in the first place.

This alternative method relies on the use of a small deadlock-avoidance buffer. This buffer is only used when a free IQ entry can not be allocated for an instruction that is the oldest in the ROB at the time of dispatch. In these cases, the instruction is placed in the deadlock avoidance buffer and will issue from there. Note that, since this instruction is the oldest in the ROB, it has all source operands ready by definition. Therefore, this deadlock-avoidance buffer is a

simple RAM structure and does not require any CAM wakeup-logic. Instructions in this buffer can either arbitrate for selection with the instructions in the IQ, or can simply take precedence over the instructions in the IQ, in which case selection from the IQ is disabled when there are instructions present in the deadlock-avoidance buffer. Note that this latter alternative does not significantly impact performance since it is very unlikely that instructions will be able to issue out of the IQ anyway (especially in the deadlock cases). We find that a simple one-entry deadlock-avoidance buffer is sufficient to prevent deadlocks with a minimal impact on performance.

In summary, in terms of hardware implementation the changes to the basic 2OP_BLOCK design amount to the removal of the logic that enforces in-order dispatch within the threads and the addition of the logic to implement the deadlock avoidance mechanism.

5. Results

In this section, we present the results of the out-of-order dispatch mechanism for various sizes of the IQ and for the workloads with various numbers of threads. The results are presented both in terms of the overall processor throughput IPC and the fairness metric. All results are shown as harmonic means across the simulated multithreaded mixes.

Figure 3 presents the speedup in throughput IPC for the traditional scheduler, the 2OP_BLOCK scheduler, and the 2OP_BLOCK scheduler augmented with out-of-order instruction dispatch for various sizes of the IQ on the workloads with 2 threads. As seen from the graph, the use of out-of-order instruction dispatch increases the performance compared to the basic 2OP_BLOCK scheduler significantly for all IQ sizes – the gains are 12% for 32-entry schedulers, 19% for 48-entry IQs, and 22% for 64-entry IQs. The large performance differences observed here are not surprising because the 2-threaded environment does not contain sufficient TLP to be harnessed by the 2OP_BLOCK design, and thus results in a significant number of cycles in which the dispatch is blocked due to the presence of instructions with 2 non-ready operands from both threads – this happens in 43% of the cycles, on the average. On the other hand, the ability to extract the deeper ILP within each thread afforded by the out-of-order dispatch mechanism almost eliminates the cycles in which dispatch is stalled for both threads simultaneously – this percentage drops to only 0.2% of the cycles on the average. Therefore, the out-of-order dispatch policy allows for the sustained performance even in the environment with a limited amount of TLP. In other words, where TLP can not be exploited, the scheduler focuses on harvesting the ILP within each thread.

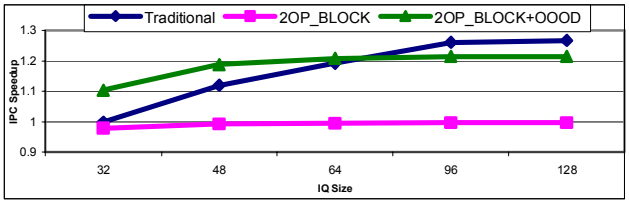

Figure 3: Throughput IPC Speedup for 2-threaded Workloads

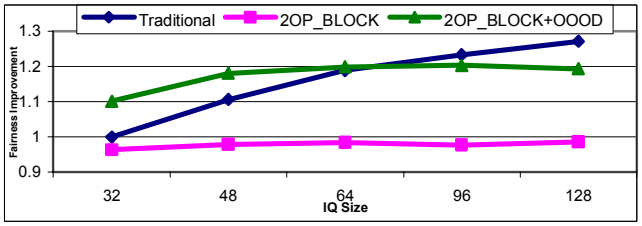

Figure 4: Improvement in Fairness Metric for 2-threaded Workloads

Compared to the traditional schedulers of the same capacity, the 2OP_BLOCK mechanism with out-of-order instruction dispatch increases the performance by 10%, 7%, and 2% for the 32-entry, 48-entry, and 64-entry schedulers, respectively, for 2-threaded workloads. Again, this is not surprising because the benefits of the 2OP_BLOCK mechanism are achieved (i.e., the IQ entries are used more efficiently), but the additional dispatch stalls introduced are minimal. Specifically, for the 64-entry schedulers, the average number of cycles that an instruction spends in the IQ drops from 21 cycles with the traditional scheduler to 15 cycles for the 2OP_BLOCK scheduler with out-of-order instruction dispatch.

However, for the larger scheduler sizes the traditional scheduler outperforms the 2OP_BLOCK with out-of-order dispatch slightly – by 4% for 96-entries and 5% for 128-entries. It should be noted that the schedulers of these sizes (greater than 64 entries) are perhaps too large for 2-threaded workloads, at least in the framework of the simulated processor. Therefore, at these sizes the scheduler efficiency is not a concern even for the baseline design and the issues of delay and power consumption (which are addressed by our scheme directly) are likely to take a central role. Therefore, even in such cases where there is small performance degradation, the savings in delay and power may overshadow this and still provide an attractive design point. For detailed circuit delay and power-related analysis of the 2OP_BLOCK mechanism, we refer the readers to [13].

Figure 4 presents similar results in terms of the improvement in the fairness metric. The trends observed with respect to this metric are very similar to the previous results. Specifically, for the 64-entry schedulers, the out-of-order dispatch mechanism improves the fairness metric compared to the basic 2OP_BLOCK by 21% and over the traditional scheduler by 1%.

We now examine the results for the 3-threaded workloads. Figure 5 presents these statistics in terms of the speedup in throughput IPC. Here, out-of-order dispatch increases the performance compared to the basic 2OP_BLOCK for all scheduler sizes and by as much as 21% for 64-entry schedulers. Compared to the traditional scheduler, the 2OP_BLOCK with out-of-order dispatch increases performance by 20%, 16%, and 9% for the schedulers with 32-entries, 42-entries, and 64-entries, respectively, and degrades performance by only 2% for 96-entries and 4% for 128-entries. Once again, similar trends are observed in terms of the fairness metric, presented in Figure 6, where the improvements for 64-entry schedulers are 17% over the basic 2OP_BLOCK and 6% over the traditional scheduler.

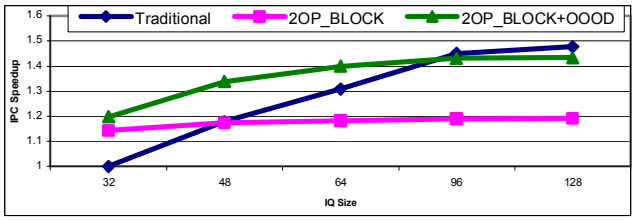

Figure 5: Throughput IPC Speedup for 3-threaded Workloads

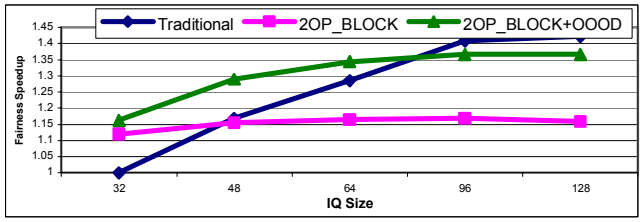

Figure 6: Improvement in Fairness Metric for 3-threaded Workloads

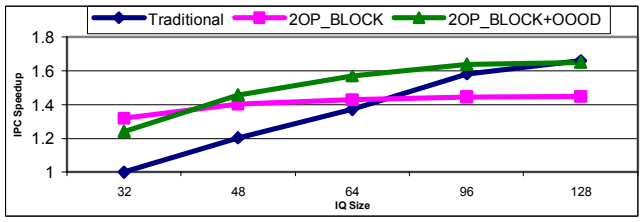

Figure 7: Throughput IPC Speedup for 4-threaded Workloads

Finally, we present the results for the workloads with 4 simultaneous threads. Figure 7 presents these results in terms of the throughput IPC. As see from the graph, the use of out-of-order instruction dispatch increases the performance compared to the basic 2OP_BLOCK scheduler for all IQ sizes larger than 32-entries – this increase is 5% for 48-entry schedulers, 14% for 64-entry IQs, and nearly 20% for both 96 and 128-entry IQs. This shows that, even in the environments with 4-threaded workloads, the basic 2OP_BLOCK design does not sufficiently balance TLP and ILP, resulting in a sub-optimal performance and there are significant opportunities for further improvements. The out-of-order dispatch mechanism, on the other hand, better balances these two forms of parallelism and therefore realizes significant performance gains.

Note that for the 32-entry schedulers, there is a slight performance degradation incurred by the use of out-of-order dispatch compared to the basic 2OP_BLOCK. This is because the amount of TLP available in 4-threaded workloads is quite sufficient to fill a small 32-entry scheduler, without relying on any additional mechanisms. An attempt to extract deeper ILP from within each thread in such situations hinders performance as it reduces the efficiency of 2OP_BLOCK. It is only when the basic 2OP_BLOCK design is incapable of utilizing the IQ, the additional techniques actually pay off. Compared to the machine with the traditional scheduler, the out-of-order dispatch mechanism used with 2OP_BLOCK provides performance gains for all sizes of the IQ. Specifically, gains of 19% are realized for the 64-entry schedulers. The results showing the increase in the fairness metric are presented in Figure 8, where the same trends can be observed. In this case, for the 64-entry schedulers, the improvement over the basic 2OP_BLOCK is 11.6% and the improvement over the traditional scheduler is 13%, on the average.

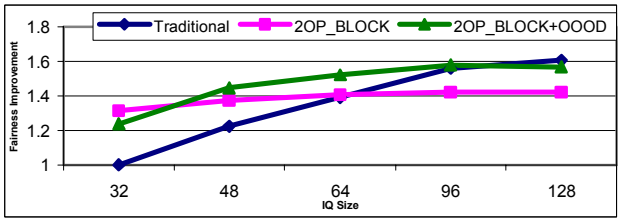

Figure 8: Improvement in Fairness Metric for 4-threaded Workloads

As was observed in Section 3, and can also be seen from the graphs in this section, the basic 2OP_BLOCK scheduler does not scale well with either the number of threads, or the size of the IQ. In contrast, the use of the out-of-order dispatch mechanism improves this scalability for *both* the number of threads and the size of the IQ. Furthermore, the scheduler using both 2OP_BLOCK and out-of-order instruction dispatching scales nearly as well or better than the traditional scheduler in terms of both throughput IPC and fairness with both the size of the IQ and the number of threads.

6. Related Work

The use of shared as well as partitioned resources in an SMT processor can be indirectly controlled by instruction fetching mechanisms. Various fetching policies have been proposed in the literature to provide the best supply of instruction mixes from multiple threads for building the most efficient execution schedules. The I-Count fetching policy [16] gives fetching priority to threads with fewer instructions in decode, rename and the IQ. The goal is to avoid clogging of the IQ with the instructions from one thread. Several optimizations of I-Count have also been proposed in an effort to avoid fetching the instructions that are likely to be stalled in the IQ for a large number of cycles. STALL [15] 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++ [3] 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 [4] avoids fetching from threads that experience an L1 data miss.

Several works proposed specific optimizations for the SMT processors. El-Moursy and Albonesi [4] explored new front-end policies that reduce the required integer and floating point IQ sizes in SMT architectures. Their techniques limit the number of non-ready instructions in the queue from each thread and also block further instruction fetching from a thread if that thread experiences an L1 cache miss. In [10], a partitioned version of the oldest-first issue policy is proposed, where separate IQs are used to buffer the instructions from different threads. In [9], the effect of partitioning the datapath resources, including the IQs, across multiple threads is discussed.

In [2], a more fine grained dynamic control over SMT resources is proposed. The mechanism of [2] first classifies the threads according to their demands for the resources and

based on this classification determines how the resources should be distributed among the threads. In contrast to the previous methods that stall or flush threads which have cache misses, the technique of [2] actually attempts to help these threads by providing more resources to them (if such resources are available) to increase the memory-level parallelism by overlapping multiple cache misses.

The observation that many instructions are dispatched with at least one of their source operands ready is not new – it was used in [5], where the scheduler design with 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 [5] statically partitions the queue into the groups of entries with various numbers of tag comparators, the instruction packing technique proposed in [11], achieves this partitioning dynamically, by sharing one IQ entry between two instructions, each with at most one non-ready source operand at the time of dispatch.

In [7], the tag buses within the IQ 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 [7] can be trivially adapted to SMT, the design proposed in this paper completely eliminates the second set of comparators and therefore obviates the need to perform last-tag speculation and maintain fast and slow wakeup buses. The capacitive loading on all tag buses is reduced, because half of the comparators are offloaded from every tag bus.

7. Concluding Remarks

In this paper, we proposed out-of-order instruction dispatching to supplement the existing scheduling technique (2OP_BLOCK) that disallows the dispatching of instructions with two non-ready register source operands. Our proposal effectively balances the exploitation of ILP within each thread and TLP across multiple threads, resulting in significant performance improvements over the basic 2OP_BLOCK scheme for 2-threaded, 3-threaded and 4-threaded workloads for various issue queue sizes.

In summary, the performance of 2OP_BLOCK with out-of-order dispatch scales much better with both the number of threads and the IQ size compared to either the traditional design or 2OP_BLOCK alone. Our proposed design significantly reduces the complexity, access delay and power consumption of the dynamic scheduling logic in SMT processors, while achieving the same and in many cases significantly better throughput IPC and fairness compared to the baseline machine.

8. References

[1] D. Burger, T. Austin. "The SimpleScalar tool set: Version 2.0." Tech. Report, Dept. of CS, Univ. of Wisconsin-Madison, June 1997 and documentation for all Simplescalar releases.

[2] F. Cazorla, et al. "Dynamically Controlled Resource Allocation in SMT Processors." in Proc International Symposium on Microarchitecture, 2004.

[3] F. Cazorla, et al. "Improving Memory Latency Aware Fetch Policies for SMT Processors." in Proc Intl Symp. on High Perf. Computing, 2003.

[4] A. El-Moursy, D.Albonesi. "Front-End Policies for Improved Issue Efficiency in SMT Processors." in Proc. International Symposium on High-Performance Computer Architecture (HPCA), 2003.

[5] D. Ernst, T. Austin. "Efficient Dynamic Scheduling Through Tag Elimination." in Proc. Int'l Symp on Comp. Architecture (ISCA), 2002.

[6] J. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium", in the IEEE Computer, 33(7):28-35, July 2000.

[7] I.Kim, M.Lipasti. "Half-Price Architecture." in Proc International Symposium on Computer Architecture (ISCA), 2003.

[8] K. Luo, et al. "Balancing Throughput and Fairness in SMT Processors." in Proc Intl. Symp. on Performance Analysis of Systems and Software, 2001.

[9] S. Raasch et al. "The Impact of Resource Partitioning on SMT Processors." in Proc. PACT, 2003.

[10]B. Robatmili et al. "Thread-Sensitive Instruction Issue for SMT Processors." Computer Architecture News, 2004.

[11]J. Sharkey, et al. "Instruction Packing: Reducing Power and Delay of the Dynamic Scheduling Logic." in Proc. of the International Symposium on Low Power Electronics and Design (ISLPED), 2005.

[12]J. Sharkey. "M-Sim: A Flexible, Multi-threaded Simulation Environment." Tech. Report CS-TR-05-DP1, Department of Computer Science, SUNY Binghamton, 2005.

[13]J. Sharkey, D. Ponomarev, "Efficient Instruction Schedulers for SMT Processors", in 12th Intl. Symp. on High Performance Computer Architecture (HPCA), 2006.

[14]T. Sherwood, et al. "Automatically Characterizing Large Scale Program Behavior." Proc. ASPLOS, 2002.

[15]D. Tullsen, et al. "Handling Long-Latency Loads in a Simultaneous Multi-threaded Processor." in Proc of International Symposium on Microarchtiecture, 2001.

[16]D. Tullsen, et al. "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor." in Proc International Symposium on Computer Architecture, 1996.

[17] D. Tullsen, et al. "Simultaneous Multithreading: Maximizing on-chip Parallelism." in Proc of Intl Symposium on Computer Architecture, 1995.

[18] C. McNairy, R. Bhatia. "Montecito: A Dual-Core, Dual-Thread Itanium Processor." IEEE Micro, Volume 25, Issue 2, March-April 2005. Pages 10–20.

[19] "IA-32 Intel Architecture Sofrware Developers Manual: Basic Architecture", Volume 1, January 2006.

[20]R. Kalla, B. Sinharoy, and J. M. Tendler. "IBM Power5 Chip: A Dual Core Multithreaded Processor." IEEE Micro, 24(2):40–47, Mar/Apr 2004.

[21]S. Swanson, et al. "An Evaluation of Speculative Instruction Execution on Simultaneous Multithreaded Processors", IEEE Transactions on Computer Systems, 21(3), 2003.

[22]A. El-Moursy et al, "Partitioning Multi-Threaded Processors with a Large Number of Threads", in Proc. of ISPASS 2005.