Reducing Register Pressure in SMT Processors through L2-Miss-Driven Early Register Release

JOSEPH J. SHARKEY
Assured Information Security Inc.
and
JASON LOEW and DMITRY V. PONOMAREV
State University of New York, Binghamton

The register file is one of the most critical datapath components limiting the number of threads that can be supported on a simultaneous multithreading (SMT) processor. To allow the use of smaller register files without degrading performance, techniques that maximize the efficiency of using registers through aggressive register allocation/deallocation can be considered. In this article, we propose a novel technique to early deallocate physical registers allocated to threads which experience L2 cache misses. This is accomplished by speculatively committing the load-independent instructions and deallocating the registers corresponding to the previous mappings of their destinations, without waiting for the cache miss request to be serviced. The early deallocated registers are then made immediately available for allocation to instructions within the same thread as well as within other threads, thus improving the overall processor throughput. On the average across the simulated mixes of multiprogrammed SPEC 2000 workloads, our technique results in 33% improvement in throughput and 25% improvement in terms of harmonic mean of weighted IPCs over the baseline SMT with the state-of-the-art DCRA policy. This is achieved without creating checkpoints, maintaining per-register counters of pending consumers, performing tag rebroadcasts, register remappings, and/or additional associative searches.

Categories and Subject Descriptors: C.1 [**Processor Architectures**]: Other Architecture Styles—*Pipeline processors*.

General Terms: Performance, Design

Additional Key Words and Phrases: Simultaneous multithreading, register file

This research was supported by NSF through Award CNS 0720811 and by Intel.
Author's addresses: Joseph J. Sharkey, Assured Information Security Inc., Rome NY (this work was done while Joseph Sharkey was a PhD student at SUNY Binghamton); Jason Loew and Dmitry V. Ponomarev, Department of Computer Science, State University of New York, Binghamton, NY 13902-6000; email: jsharke@ainfosec.com, jloew@cs.binghamton.edu, dima@cs.binghamton.edu.

ACM Reference Format:
Sharkey, J. J., Loew, J., and Ponomarev, D. V. 2008. ACM Trans. Architec. Code Optim. 5, 3, Article 13 (November 2008), 28 pages. DOI = 10.1145/1455650.1455652 http://doi.acm.org/10.1145/1455650.1455652

1. INTRODUCTION

Simultaneous multithreading (SMT) is an important architectural paradigm for increasing microprocessor throughput in an area-efficient manner by sharing the key datapath resources among the instructions from multiple threads of control [Marr et al. 2002; Tullsen et al. 1996]. One such shared resource in an SMT datapath is the physical register file (RF), which needs to be sized very generously to support the full architectural register state of each thread as well as to provide a sufficient number of additional registers for renaming, typically all within a common RAM structure. For example, for an ISA with 32 architectural registers, 128 registers are needed to maintain the precise state for a four-threaded SMT, in both integer and floating point RFs. When renaming registers are taken into account, the total number of entries within each RF can reach several hundred. The large access delays, high power consumption, and significant design complexity associated with such RFs are major factors limiting the number of simultaneous threads that can be supported by an SMT machine, especially at high-frequency implementations. Pipelining the access to large RFs over several cycles requires multiple levels of bypass and also lengthens the branch resolution and the load-hit speculation loops [Borch et al. 2002].

An alternative to building large RFs is to use a smaller number of registers in a more efficient fashion. The higher efficiency of using physical registers in an SMT processor can be achieved by addressing two related issues: (1) how to distribute the available registers among the threads, and (2) how to manage these registers to provide a larger supply of them for distribution. We generically refer to these two key aspects as *register distribution* and *register management*.

Register Distribution. This issue is addressed in the recent literature through a series of proposals, such as I-Count [Tullsen et al. 1996], STALL [Tullsen et al. 2001], FLUSH [Tullsen et al. 2001], DCRA [Cazorla et al. 2004], and Hill-Climbing [Choi and Yeung 2006] techniques. I-Count gives fetching (and thus register allocation) priority to threads with fewer not-yet-executed instructions. The FLUSH mechanism completely squashes a thread that experienced a long-latency L2 cache miss, releasing all physical registers allocated to this thread and assigning them to other threads while the cache miss is being serviced. The STALL mechanism simply blocks further resource allocations to such threads, without squashing the in-flight instructions. FLUSH generally provides higher performance than STALL [Tullsen et al. 2001], but it also incurs nontrivial overheads, because the squashed instructions have to always be refetched, rescheduled, reexecuted, and all shared resources have to be reallocated to these instructions again. Consequently, the first allocations, performed prior to the discovery of a cache miss, just waste the resources, even for the load-independent instructions that executed without problems. The DCRA policy takes a different approach and instead allocates more resources to memory-bound threads, attempting to help their performance. The Hill-Climbing mechanism further improves on DCRA by observing the impact of resource distribution decisions at run time and feeding this information back to the front end of the pipeline to guide future allocations.

Register Management. All of the above approaches still work within the traditional register management framework. Traditional register allocation and deallocation mechanisms, both on superscalar and SMT processors, are very conservative—a physical register allocated to the destination of an instruction is released only when the next instruction (from the same thread) writing to the same destination architectural register commits. Several techniques have been proposed in the literature (all in the domain of superscalar processors) to relax these conditions and achieve higher efficiency of register usage. These techniques can be broadly classified into three groups: late allocation mechanisms [Gonzáles et al. 1998; Monreal et al. 2004], early deallocation mechanisms [Lipasti et al. 2004; Martinez et al. 2002; Monreal et al. 2002; 2004] and register-sharing mechanisms [Balakrishan and Sohi 2003]. While it is possible to trivially extend at least some of these approaches to an SMT machine, the complexity of the resulting solution (which is already high in superscalars) could further escalate on SMT if the modifications to the thread-specific resources are required. In particular, these techniques often require wake-up tag rebroadcasts, register remappings, additional associative searches within the issue queue and the rename table, deadlock avoidance techniques and/or periodic creation of full register file checkpoints. In this article, we seek solutions for improving the register-management mechanisms, but without incurring the complications of the aforementioned designs. Moreover, the scheme that we propose can be used in conjunction with all of these techniques, as it exploits different opportunities for optimizing register usage.

Our solution is motivated by the observation that the primary reason for having a large number of physical registers in the RF, especially on SMT processors, is the capability to buffer a sizable number of in-flight instructions following a load that misses into the L2 cache. Figure 1 compares the performance of a four-threaded SMT machine with perfect L2 cache and one with the realistic L2 cache of 2MB for various RF sizes. As shown in the graph, the performance saturates at 200 registers in the former case, and at around 300 registers in the latter case (details of our simulation framework are provided in Section 3). Consequently, if the registers assigned to the instructions waiting for the resolution of the L2 cache miss were not tied up, a much smaller RF would be sufficient to maintain the same performance level. Alternatively, a considerably higher performance would be realized using the similarly sized RFs. As the L2 cache misses are relatively more frequent on SMT (because the caches are shared among multiple threads) and physical registers are relatively scarcer compared to a superscalar, it is important to consider techniques for optimizing the RF usage under the L2 cache misses. A technique to perform such *lazy*

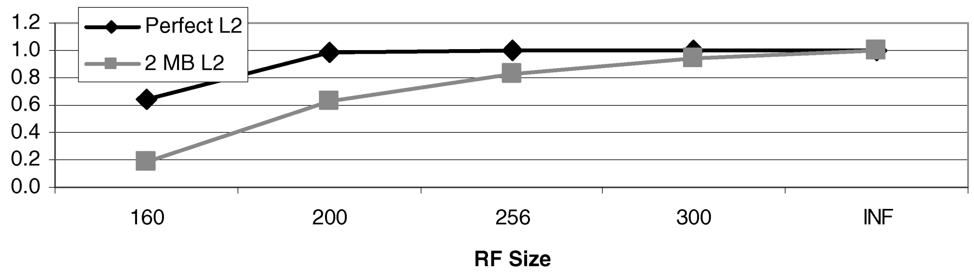


Fig. 1. Throughput IPC relative to the respective machine with infinite number of registers.

deallocation of registers in the shadow of L2 cache misses is the main contribution of this article. As detailed in Section 2, the timing and the conditions for this deallocation are disjoint from those used by the previous schemes; therefore, our technique can be used either by itself or even in conjunction with previous proposals.

The key statistics that provided an inspiration for this work is that the majority of the instructions in the shadow of the long-latency loads are, in fact, load-independent [Karkhanis and Smith 2002; Sarangi et al. 2005]. These load-independent instructions release their issue queue entries fairly fast, but then pile up in the reorder buffer, waiting for the cache miss to be serviced. Therefore, the only shared resources within the SMT datapath that remain allocated to these instructions are the physical registers—those are not released until the cache miss is serviced and the process of instruction commitment resumes. We note that neither STALL nor FLUSH is designed to specifically exploit this behavior. While STALL does not release the already allocated resources, FLUSH does so aggressively, but incurs inefficiencies of having to refetch and reexecute all flushed instructions, even those that are independent of the long-latency load.

We propose a mechanism to release physical registers allocated to threads which experience L2 cache misses by speculatively committing the load-independent instructions and early deallocating the previous mappings of their destination registers. The instructions committed speculatively in this fashion remain in the ROB until the load miss is serviced and the actual commitment occurs. The early released registers can be made available to the instructions within the same thread to support higher memory-level parallelism (MLP) by overlapping multiple cache misses), as well as to the instructions from other threads, thus directly exploiting the thread-level parallelism (TLP) and improving the overall throughput. The specific nature of the register assignments to threads is still dictated by the best-performing underlying resource distribution policies, such as DCRA or Hill-Climbing—our mechanism just provides more registers for the distribution.

The key novel features behind our design include the following:

- The physical registers are reclaimed *lazily*, only in the shadow of long last-level cache misses. (Although the technique can in principle be generalized to deallocate registers under any long-latency event, in this article we only focus on the L2 cache misses). Barring interrupts and exceptions, load-independent instructions executed in the shadow of the missing load are properly committed (and the physical registers are released) after the cache miss is serviced. Recognizing that exceptions and interrupts are rare, we propose to pseudo-commit such load-independent instructions *and perform respective register deallocation activities* earlier, as soon as the cache miss is detected.

- The register reclamation process is only performed along the correctly resolved control flow paths (i.e., we never early deallocate a register if the corresponding instruction may be subject to a branch misprediction). In particular, if an unresolved (unexecuted) conditional branch instruction is encountered in the course of examining the instructions in program order for possible register deallocation, the entire process stops. Such approach allows us to decouple the mechanism for branch misprediction and exception handling and allow for a much higher overhead in restoring the values of the deallocated registers (as the restoration occurs only on exceptions or interrupts).
- In order to correctly execute the load-dependent instructions, neither the previous instances of their destinations nor the previous instances of their sources are early released.

To maintain the precise state on interrupts and exceptions, the values of the early released registers can either be saved and restored through periodic register file checkpointing or they can be saved directly within the ROB entries of the instructions that triggered corresponding early deallocations—this mechanism avoids the need for checkpointing and we used it for quantifying our results.

The key characteristics and performance implications of the proposed mechanism are as follows:

- It does not incur tag rebroadcasts, register remappings, associative searches and rename table modifications, does not require per register consumer counters. Instead, it relies on a simple off-the-critical-path logic at the back end of the pipeline to identify the early deallocation opportunities and save the values of the early deallocated registers for precise state reconstruction.
- Although it provides some performance benefits even in a single-threaded execution environment (5% IPC gains for 64-entry RFs), the real advantages come in SMT processors with multiple threads due to the more-frequent L2 cache misses and higher pressure on the register file. For a 4-way SMT machine with 256 integer and 256 floating point registers, the proposed mechanism results in 25% improvements in fairness compared to the baseline processor with DCRA resource distribution policy.
- It is complementary to all existing late register allocation and early register deallocation schemes and can be used in conjunction with those mechanisms, as it exploits previously unexplored early register deallocation opportunity. However, even by itself, it outperforms some previously proposed early deallocation schemes (e.g., physical register inlining) on SMT, although results in slightly lower gains during single-threaded execution.

- It works synergistically with recently proposed resource allocation policies for SMT, such as DCRA [Burger and Austin 1997] and Hill-Climbing [Cazorla et al. 2003]. Although those policies control the distribution of available resources among threads, our technique effectively provides more resources (physical registers) to be used by these policies. As a result, we achieve additional 25% gains on top of DCRA, and 26% gains on top of Hill-Climbing for 256-entry RFs in terms of the fairness metric.

The early register deallocation mechanism described here was first proposed in the conference version of Sharkey and Ponomarev [2007]. In this article, we extend the work of Sharkey and Ponomarev [2007] by presenting the following set of additional results:

- Comparison of various resource allocation and register management policies in terms of fairness metric (Figure 4).
- Detailed per-benchmark results of various schemes in terms of fairness metric (Figure 5)
- Statistics about the number of registers released during each pass through the ROB during the early deallocation stage (Figure 7).
- Performance sensitivity to the ROB size (Figure 8(b)), L2 cache size (Figure 10(a)) and memory latency (Figure 10(b)).
- Performance sensitivity to the degree of datapath superscalarity and also the sizes of the key datapath resources (Figure 12).

The rest of the article is organized as follows. Section 2 reviews the related work. Our simulation methodology is presented in Section 3. We motivate the proposed technique and present its details in Section 4. Section 5 presents the results and we conclude in Section 6.

2. RELATED WORK

Although most of the related work was briefly described in Section 1, this section provides a more-detailed analysis of some prior efforts in the areas of SMT resource distribution, aggressive register management, and checkpointed processor architectures, all of which provided an inspiration for this work.

SMT Resource Distribution. The use of shared as well as partitioned resources in an SMT processor can be indirectly controlled by instruction fetching mechanisms. Several such mechanisms (I-Count [Tullsen et al. 1996], FLUSH and STALL [Tullsen, et al. 2001]) were discussed in Section 1. FLUSH++ [Cazorla et al. 2003] 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 El-Moursy and Albonesi [2003] avoids fetching from threads that experience an L1 data miss. In Cazorla et al. [2004], a resource distribution policy (called DCRA) exercising a more fine grained dynamic control over the shared SMT resources (such as physical registers) was proposed. DCRA 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. The Hill-Climbing approach

[Choi and Yeung 2006] observes the impact of resource distribution decisions on the performance and uses this information to improve future decisions. We apply the proposed mechanism on top of both DCRA and Hill-Climbing and show that serious additional performance gains can be realized in either case. In fact, our mechanism works synergistically with the best-performing resource distribution techniques: it does not take away their advantages, but instead supplies more physical registers to enable more effective register distribution.

Register File Optimizations. Researchers have exploited the inefficiencies in register usage to reduce the number of registers in three major ways. One set of solutions delays the actual allocation of physical registers until the time that the result is written back [Gonzáles et al. 1998; Monreal et al. 2004]. These schemes avoid tying up destination physical registers between the time of instruction dispatch and instruction writeback by allocating a physical register only at the time of writeback and using separate tags to satisfy the data dependencies. The major drawback of the late allocation schemes is in the form of nontrivial increases in the datapath complexity due to the need to: (1) support several levels of register mapping tables, (1) perform various associative searches on the rename table and issue queue after the reassignment of mappings and (3) avoid potential deadlocks. The second set of techniques aim at reducing the register file pressure by using the early deallocation of physical registers [Lipasti et al. 2004; Martinez et al. 2002; Monreal et al. 2002, 2004; Ergin et al. 2004]. Although these mechanisms differ in the timing and manner of register deallocation, the additional logic needed to support precise state reconstruction and guarantee correctness of the execution is fairly complex, sometimes requiring additional accesses to the rename table [Lipasti et al. 2004] or register state checkpointing support [Ergin et al. 2004; Martinez et al. 2002]. Furthermore, these schemes need to maintain accurate counters of pending consumers to ensure that deallocation occurs only after all such consumers are issued. The technique of Monreal et al. [2004] deallocates a register after commitment of its last consumer, Ergin et al. [2004] deallocates a register right after commitment of the instruction itself, and Lipasti et al. [2004] and Martinez et al. [2002] attempt to deallocate registers after the result is produced. None of these mechanisms directly exploit the L2 cache misses to optimize register file. Although it is possible to implement some of these techniques on SMT to reduce the number of registers, the complexity of the resulting solution will be at least as high as on a superscalar and in some cases (when modifications to the thread-specific resources are involved) even higher. The solution to register file optimization proposed in this article takes advantage of a previously unexploited opportunity for early register deallocation. Because of this, our technique works synergistically with the previous schemes for register optimization (including those that deallocate registers early) and can be used in conjunction with those mechanisms to provide additional performance advantages, especially in SMT machines. A detailed performance analysis of such synergies is presented in Section 5. Finally, the third set of solutions reduces the number of registers through the use of register sharing [Balakrishan and Sohi 2003].

To the best of our knowledge, only one prior effort specifically addressed the issue of RF scalability in a multicontext environment [Oehmkr, et al. 2005]. In that work, a technique to virtualize logical register contexts by automatically saving the values of unused registers to memory and restoring them back on demand was proposed. The scheme of Oehmkr et al. [2005] requires modifications to the rename stage of the pipeline and also results in the insertion of additional load and store instructions into the pipeline to implement fills and spills and has a few other implications on the complexity of the datapath design. The work of Lo et al. [1999] investigated compiler support for efficient register management in SMT.

Checkpointed Processor Architectures. Although the proposed mechanism does not necessarily rely on creating checkpoints, we draw some inspiration from several recent proposals that use register file checkpointing and exploit load-independence to accelerate the execution of *single-threaded* applications [Kirman et al. 2005; Mutlu et al. 2003; Sarangi et al. 2005]. There are some principal differences between our proposal and those prior works and it is important to highlight them. For example, runahead execution [Mutlu et al. 2003] unblocks the ROB when the missing load reaches the ROB head, creates a full register file checkpoint and then allows the following instructions to pseudocommit. After the load is serviced, the execution resumes from the checkpoint. The net effect is that some subsequent cache misses are generated earlier, effectively resulting in cache prefetching and improved performance. The technique of Kirman et al. [2005] introduced load value prediction on top of the runahead execution and augmented the benefits of runahead execution with direct performance advantages on the correct predictions. Unless the value prediction is used and it is correct, all instructions executed in the runahead mode still need to be reexecuted when the cache miss is serviced. In addition, the complexities and overheads of maintaining the register state checkpoints, buffering the results of the speculative store instructions, and possibly supporting value prediction/verification logic are considerable, especially in SMT processors.

In contrast, the registers that are early deallocated by our scheme are always used by the instructions from the same thread as well as from other threads for the *actual execution*. Instructions are never reexecuted and no resource wastages occur. Most importantly, *all pseudocommitted instructions still remain in the ROB* (in contrast to the above mentioned schemes); thus, no speculative updates of the memory state by the store instructions ever occur and need to be addressed, avoiding the associated complexities (like the need to maintain and manage the runahead cache as in Mutlu et al. [2003] and Kirman et al. [2005]). Although our scheme has a somewhat limited utility if applied to a single-threaded superscalar machine as the ROB is likely to quickly fill up and block the forward progress (although we show in the results section that performance gains can be realized even in that case), the presence of explicit TLP within a multithreaded processor allows for the progress of other threads to be sustained and fueled by the early deallocated registers even if the ROB of the thread that experienced the L2 cache miss becomes full.

Table I. Configuration of the Simulated Processor

Parameter	Configuration
Machine width	8-wide fetch, 8-wide issue, 8-wide commit
Window size	64-entry issue queue, 48 entry load/store queue, 128–entry ROB per thread
Function Units and Lat (total/issue)	8 Int Add (1/1), 4 Int Mult (3/1) / Div (20/19), 8 4 Load/Store (2/1), 8 FP Add (2), 4 FP Mult (4/1) / Div (12/12) / Sqrt (24/24)
Physical Registers	Separate integer and floating point register files, size as specified
L1 I–cache	32KB, 2-way set-associative, 64-byte line
L1 D–cache	64KB, 2-way set-associative, 64-byte line
L2 Cache unified	512KB, 8-way set–associative, 128-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, 300 cycles access latency

Recent studies by Karkhanis and Smith [2002] and Sarangi et al. [2005] showed that most of the instructions fitting into the instruction window following an L2 miss are load-independent. This result was corroborated by our experiments and we exploit these statistics directly. Other proposals [Akkary et al. 2003; Srinivasan et al. 2004] completely eliminate the ROB and instead use periodic checkpoints, effectively allowing out-of-order commit and more aggressive use of registers. In contrast, our technique builds on top of traditional ROB-based architectures, keeping the datapath changes to the minimum. In addition, efficiently implementing checkpointing on SMT processors requires additional considerations, as per-thread checkpointing is needed to avoid the simultaneous rollback of all threads if a global register and memory state checkpoint is used.

3. SIMULATION METHODOLOGY

For estimating the performance impact of the schemes described in this article, we used M-Sim [Sharkey 2005]: a significantly modified version of the Simplescalar 3.0d simulator [Burger and Austin 1997] 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 I.

We simulated the full set of SPEC 2000 integer and floating point benchmarks, using the precompiled Alpha binaries available from the Simplescalar website [Burger and Austin 1997]. We skipped the initialization part of each benchmark using the procedure prescribed by the Simpoints tool [Sherwood et al. 2002] 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 II. Simulated Three-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*

Table III. Simulated Two-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*

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 four-threaded workloads, 12 three-threaded workloads and 12 two-threaded workloads. All workloads were created by mixing the benchmarks with different ILP levels in various ways. Tables II, III, and IV depict the specific benchmarks that constituted each of our workloads. The ILP level of each benchmark is also shown.

We used several metrics for evaluating the performance of the multithreaded workloads throughout this article. 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 [Luo et al. 2001]. Therefore, we also present the result using another metric, "harmonic mean of weighted IPCs." Although the latter metric is really a performance metric that represents a balance between performance and fairness, in the rest of the article, we refer to this metric as "fairness" to be consistent with some previous work [Luo et al. 2001].

Table IV. Simulated Four-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*

4. L2-MISS-DRIVEN EARLY REGISTER DEALLOCATION

In this section, we describe the details of the proposed mechanism.

4.1 Motivation and Overview

To motivate our technique, we begin by examining a short code fragment obtained from the execution of the *equake* benchmark from the SPEC 2000 suite. Consider the sequence of instructions depicted in Figure 2, the oldest of which is a load (*ldt* instruction on line 1 that missed into the L2 cache. For each register-to-register instruction, the destination register is shown last. Both the original and the renamed version of each instruction are shown, as well as the previous mappings of the destination architectural registers. In the normal course of operations (i.e., without early register deallocation), when an instruction commits, it deallocates the previous mapping of its destination, using the information that is available from the commit-time rename table.

In the example of Figure 2, only 5 out of the 28 instructions that follow the missing load are load-dependent—those are the shaded instructions shown on lines 8, 12, 13, 14, and 15. The execution of all other instructions will complete well before the L2 cache miss triggered by the *ldt* instruction from line 1 is serviced. In the absence of branch mispredictions, exceptions and interrupts, the commitment of the load-independent instructions and the corresponding release of their previous mappings are only delayed because of the long latency to service the L2 cache miss. When the miss is finally serviced and the ldt instruction on line 1 commits, all subsequent instructions will also commit and the previous commit-time mappings of the corresponding destination architectural registers will be deallocated. For example, when the *ldq* instruction on line 2 commits, physical register pr0 is deallocated, and when *addq* on line 3 commits, the register *pr5* is deallocated, and so on. The inefficiency of this approach is that both *pr0* and *pr5* will be deallocated only after the L2 cache miss is serviced—possibly hundreds of cycles after the new instances of their corresponding architectural registers, the results of the *ldq* and *addq* instructions are produced. In many situations, the registers like *pr0* and *pr5* in the

	Source Code	Renamed Code	Old mapping
1	ldt **f13**,0(r27)	ldt **pf32**,0(pr27)	pf13
2	ldq r0,0(r0)	ldq pr32,0(pr0)	pr0
3	addq r5,r11,r5	addq pr5,pr11,pr33	pr5
4	ldt f8,0(r5)	ldt pf33,0(pr33)	pf8
5	addq r17,r11,r17	addq pr17,pr11,pr34	pr17
6	addq r6,r11,r6	addq pr6,pr11,pr35	pr6
7	mult f24,f2,f24	mult pf24,pf2,pf34	pf24
8	addt **f13,f13,f15**	addt **pf32,pf32,pf35**	pf15
9	ldt f11,0(r17)	ldt pf36,0(pr34)	pf11
10	ldt f17,0(r6)	ldt pf37,0(pr35)	pf17
11	addq r0,r11,r13	addq pr32,pr11,pr36	pr13
12	subt **f13**,f24,**f13**	subt **pf32**,pf34,**pf38**	**pf32**
13	mult **f15**,f11,**f11**	mult **pf35**,pf36,**pf39**	pf36
14	mult **f13**,f17,**f13**	mult **pf38**,***pf37***,**pf40**	**pf38**
15	subt **f11,f13,f6**	subt **pf39,pf40,pf41**	**pf41**
16	bsr r26,0x120012520	bsr pr26,0x120012520	---
17	lda r30,-16(r30)	lda pr37,-16(pr30)	pr30
18	ldq_u r31,0(r30)	ldq_u pr38,0(pr37)	pr31
19	ldah r28,-8193(r29)	ldah pr39,-8193(pr29)	pr28
20	stq r26,0(r30)	stq pr40,0(r37)	pr26
21	bis r31,r31,r31	bis pr38,pr38,pr41	pr38
22	ldt f0,-16792(r29)	ldt pf42,-16792(pr29)	pf0
23	ldt f17,-2680(r28)	ldt pf43,-2680(pr39)	***pf37***
24	cmptle f16,f0,f1	cmptle pf16,pf42,pf44	pf1
25	fbeq f1,0x120012580	fbeq pf44,0x120012580	---
26	ldq r26,0(r30)	ldq pr42,0(pr37)	pr40
27	cpys f31,f31,f0	cpys pf31,pf31,pf46	pf42
28	lda r30,16(r30)	lda pr43,16(pr30)	pr37
29	ret r31,(r26)	ret pr44,(pr42)	pr41

Fig. 2. Example code sequence from the *equake* benchmark. The **ldt** instruction on **line 1** is a load that missed into the L2 cache. Load-dependent instructions are shown in the shaded boxes. For each register-to-register instruction, the destination register is shown last. Both original and renamed instructions are shown.

above example will remain allocated for a large number of cycles only to support recovery to a precise state in case of very infrequent interrupts and/or exceptions—all their consumers would have read the values and all potentially intervening branches would have been resolved.

We propose to deallocate the previous mappings of the destinations of load-independent instructions (such as registers *pr0* and *pr5*) without waiting for the long-latency load to commit, and make these registers immediately available for the allocations to the instructions from the same thread as well as from other threads. To support precise interrupts, the values of the deallocated registers are instead saved within the ROB entries of the instructions that trigger their early deallocations using the storage for the instruction address (or possibly elsewhere, as detailed later). To guarantee the correct execution of load-dependent instruction in the presence of early register deallocation, we ensure that neither the destinations nor the sources of the load-dependent instructions are early released. For example, the *subt* instruction on line 12 requires registers *pf32* and *pf34* as the sources and it writes the result into register *pf38*. Our technique thus guarantees that none of these registers will be early released, even if such opportunity was available. For example, we do not early release the previous mapping (register *pf37*) of the load-independent instruction *ldt* on line 23, because *pf37* is used as a source by the load-dependent instruction *mult* on line 14. We accomplish this by using a lightweight off-the-critical-path logic at the back end of the pipeline, and the execution core remains almost unaffected. We now describe the implementation details of this scheme.

4.2 Identifying Registers for Early Deallocation

When a load instruction that missed into the L2 cache from any thread reaches the head of the corresponding ROB, the process of early register deallocation (ERD) from that thread begins. When a thread enters the ERD phase, further fetches from that thread can continue or they can be blocked to ensure that the early deallocated registers are only distributed to other threads. Our results showed that continuing fetching achieves significantly higher performance, as it also exploits the memory-level parallelism by allocating more registers to the thread that experienced L2 cache miss in addition to speeding up the execution of other threads. Again, the specific policy for allocating these registers is dictated by the existing resource distribution mechanisms, such as DCRA.

When a thread enters the ERD phase, its ROB entries are examined one-by-one in program order (starting at the missing load and examining up to W entries per cycle where W is the commit width from this ROB), and the registers that can be early deallocated are identified. When all instructions in the ROB are examined, the process is repeated because several more load-independent instructions may have completed their execution during the first pass, thus opening up new opportunities for early deallocation which were not available during the first pass. Such passes through the ROB can continue until the cache miss is resolved, but in practice, just a few passes are sufficient to reap most of the benefits of this scheme, as we quantify in the Section 5.

The following activities are incurred during each pass through the ROB in the ERD phase. To identify the registers that can be early released, we maintain a bit-vector called *Don't_Release* with one bit per physical register. The *Don't_Release* bits identify the sources and the destinations of the load-dependent instructions and prevent the early release of their previous instances. When a thread enters the ERD phase, all of these bits are reset to zero. These bits are also cleared in the beginning of every new pass through the ROB in the ERD phase. As instructions in the ROB are examined and an instruction that has not yet completed its execution is encountered, it is assumed that this instruction is dependent on the long-latency load, and the *Don't_Release* bits corresponding to both its source and destination registers are set to one to ensure that all of the source values necessary to execute this instruction as well as the destination register to writeback the result are still available when this instruction executes. This is important because, if a source register A of a load-dependent instruction X is early released, then A may be reallocated to another instruction (e.g., Y), and Y can complete the execution and writeback the result into A before X is scheduled. Since X and Y can belong to two different threads, these interactions are very difficult to control, and, therefore, the mechanism based on *Don't_Release* bits is needed to prevent such instances. Once the *Don't_Release* bits are set for the registers of a not-yet-executed instruction, no further actions are triggered and the next instruction in the ROB is examined.

When a load-independent instruction that executed *without raising an exception* is encountered in the course of examining the ROB during the ERD phase, the *Don't_Release* bit corresponding to the previous mapping of the destination architectural register (e.g., Register X) is examined. The information about the previous mapping of the destination register is often readily available from the ROB itself in processors that support "walk-back" and "walk-forward" branch misprediction recovery methods [Akkary et al. 2003]. If, however, the previous mapping is not available from the ROB (i.e., checkpoint-based recovery is used), then the commit-time rename table can be consulted to obtain that information.

If the *Don't_Release* bit of register X is set to 0, then X is immediately released and the commit-time rename table is updated (as if this instruction were committing as normal). However, before this occurs, the value stored in X will be read from the register file and stored elsewhere so that it can be resurrected later to reconstruct the precise register state on interrupts or exception. To ensure that relatively more frequent branch mispredictions do not require additional handling, *we stop the ROB examination in the course of ERD phase at the first unresolved branch instruction*. When such a branch is encountered, the new pass begins from the head end of the ROB. If a branch is dependent on the load, then the early release of registers will never occur for instructions following this branch, even on subsequent passes through the ROB. However, if the branch is independent of the load and was merely delayed in the issue queue due to other data dependencies, it will eventually resolve and subsequent passes will release the registers for instructions following the branch.

To keep track of the instructions that triggered the early deallocation of physical registers and avoid duplicate deallocations of the same registers during the subsequent passes through the ROB, each ROB entry is augmented with one additional bit, called the *Early_Committed* bit. This bit is reset to 0 when the ROB entry is allocated, and is set to 1 if the instruction residing in the entry triggers early deallocation of the previous mapping of its destination register. When the long-latency load miss is serviced and the load is marked as "ready to commit," then the ERD process stops and the normal commitment process continues. As the normal commitment process resumes, instructions whose *Early_Committed* bit is set simply deallocate their ROB entry and do not update the commit-time rename table or release any registers (as this has already been done).

While the serial examination of the ROB entries in the course of the ERD phase may appear to be complex, these accesses reuse the read/write ports that already exists on the ROB to write the PC values during instruction dispatching and possibly read them during commitment. Therefore, no new ports are added for the purpose of ERD. During most of the ERD phase, the ROB will become full anyway and no other useful activities can be performed on it. Furthermore, the timings of the ERD-related operations can be significantly extended without performance impact, as explained and evaluated later in the article. That can open up additional opportunities for optimizing circuit-level design of this logic.

4.3 Saving the Values of the Deallocated Registers

Since the values of the early deallocated registers are only needed on exceptions or interrupts, and *never on branch mispredictions,* perhaps the simplest way to restore a precise state in these cases (at least in terms of control logic) is to rely on the creation of periodic checkpoints of the architectural registers. As interrupts and exceptions are generally rare, such checkpoints can be created infrequently. Furthermore, the backup storage to implement such checkpoints can be designed to support only high-latency (multicycle) access, as the checkpoints will be rarely needed. The mechanics for creating such checkpoints are rather simple—specifically, the entries of commit-time rename table corresponding to a thread can be periodically examined and physical registers corresponding to all architectural registers can be read (a few at a time) into the back up storage. While this mechanism is simple to implement in terms of control logic and has virtually no performance overhead if the checkpoints are created infrequently, the main overhead lies in the amount of extra storage. For example, for an ISA with 32 integer and 32 floating point registers, 64 registers need to be checkpointed for each thread context. If four threads are executed simultaneously on SMT, then the total number of checkpointed registers is 256 (separate checkpoints have to be maintained for each thread). To eliminate this additional datapath storage, we also propose a new scheme for saving the values of the deallocated registers, which uses the storage capabilities of existing datapath structures. Specifically, we attempt to squeeze the deallocated values within the ROB itself, in place of the PC values which are no longer necessary after the instruction pseudocommits. This approach also reduces the overhead of recovery from exceptions or interrupts, because no instructions have to be reexecuted following exceptions, as would be the case with checkpointing, especially if checkpoints are separated by large number of cycles. The rest of the discussions in this section and evaluations assume that the values of the deallocated registers are stored in the ROB. Due to the port contention, this approach would result in slightly lower performance than the checkpointing-based solution with infrequent checkpoints. All additional delays, including port contention, have been fully modeled and taken into account by our simulations. Microarchitectural details of checkpoint-based mechanism can be trivially derived from the explanations provided.

To store the values of the early deallocated registers within the ROB, three opportunities exist, depending on the specific microarchitecture used and the storage capabilities of the ROB. In processors that maintain full program counter (PC) values within each ROB entry to support precise interrupts [Kucuk et al. 2002; Smith and Pleszkun 1985] and the bit width of the PC field is greater or equal to the bit width of physical registers, the values of the early deallocated registers can be directly saved within the ROB entry of the instruction that triggered the deallocation, in place of the PC values. In processors that either maintain only the portions of the whole PC values within the ROB, or the width of the PC values is smaller than the width of physical registers, the early register deallocation can simply be limited to the cases when the values stored in the early deallocated registers can fit within the number of bits available in the ROB for storing PC. Since previous research showed that a large percentage of the register values are narrow-width [Lipasti et al. 2004], the limitation imposed by this restriction does not have a significant impact on performance (as we detail in the results section).

Although it is difficult to obtain the information as to whether current processor implementations actually store the PC bits within the ROB or somehow reconstruct this value as needed (such details are typically not available publicly), there is at least one recent reference [De Vries 2003] that suggests that the AMD K8 processors maintain the full PC addresses within the ROB slots. In such cases, this field can be used directly to store the values of the early deallocated registers in place of the PC values. In terms of the example of Figure 2, when the instruction *ldq* on line 2 triggers the early deallocation of register *pr0*, the value stored in *pr0* will be read from the register file and saved within the ROB entry of the instruction *ldq*. Likewise, the value of register *pr5* will be stored within the ROB entry of the instruction *addq* on line 3. The PC fields of the ROB entries already have all the necessary write ports that can be directly used for storing the values. Since the PC values of some uncommitted instructions in the ROB will be overwritten with the values of the deallocated registers, the precise state will be available only at the discrete points, associated with the instructions with the intact PC values in the ROB. However, this presents no problems for handling asynchronous interrupts as the specific instruction to associate the precise state with can be chosen rather freely in this case, nor does this present problems for exception handling because the PC field of an instruction's ROB entry is overwritten if and only if that instruction had completed the execution *without* exceptions. Thus, if the PC value is overwritten, the corresponding instruction can never raise an exception. The ROB field used for storing the exception codes can also be used (in conjunction with the PC field) to extend the number of ROB bits available for storing values of the deallocated registers.

Although this scheme does not require any additional storage, the only caveat is that, in some architectures, the combined width of the PC and exception code fields in the ROB may be insufficient to store the full values of the deallocated registers. The simplest solution to address this issue is to early deallocate a register only if the value stored in that register fits within the number of bits available in the ROB. Our experiments showed that limiting early deallocation to only the registers whose values fit into 36 bits of storage provides performance within 1% of the width-unconstrained deallocation; therefore, we assume that 36 bits are available in each ROB entry for storing these values. The checking of the result width can be performed after reading out the register values from the register files. If the check indicated that the result does not fit in the PC field within the ROB, then the register is not deallocated.

Since the activities involved in saving the values of the early deallocated registers within the ROB are rather lengthy (involving the read from the RF, examination for the result width and write to the ROB), it is unlikely that all these actions can be performed within a single cycle. One solution is to pipeline these activities over several cycles, which has a negligible impact on performance and just requires a few intermediate latches to be used. Furthermore, we also consider further simplifications to the logic by spreading these activities over several cycles in a *nonpipelined* manner. As shown in Figure 8(a), the basic conclusion is that spreading these activities over three cycles and not starting the deallocation of the next group until the registers triggered by the previous group are deallocated (nonpipelined implementation) has almost no impact on performance. In any case, the supply of free registers is replenished at a much higher rate compared to traditional designs and all early deallocatable registers are freed up relatively early in the course of servicing a miss, even with multicycle nonpipelined implementation of the early deallocation logic.

4.4 Reading the Values of the Early Deallocated Registers

Finally, to read the values of the early deallocated registers from the register file for saving them within the ROB, we use the existing register file read ports and perform these reads only when the processor issue width is not fully utilized (i.e., the read ports are not used by the issued instructions). In a sense, the early deallocation logic acts like another functional unit that competes for the use of the register file read ports, but with the lowest priority. These effects have been thoroughly modeled in our simulation environment. Since the issue width is typically grossly underutilized (even on SMT machine), ample opportunities for stealing the register read ports are available. In some cases, when the peak issue width is sustained for a number of cycles, the registers will be deallocated a few cycles later but still significantly sooner than if they had waited for the L2 miss to be serviced.

4.5 Restoring the Precise State

The precise state restoration on exceptions or interrupts is trivial, as the ROB can simply be walked through and correct values can be easily restored. Specifically, the ROB is walked through, starting from the tail end, and the actions related to early deallocation are undone for each instruction by reading the stored value of the previous mapping of the architectural register and writing it to the physical register, which represents the current mapping for that particular architectural register in the commit-time rename table. This mechanism does not require the new register allocations to reconstruct the precise state; it merely ensures that the physical registers representing the current commit-time mappings contain the correct values. The process of precise state restoration, as described, requires the write ports to the register file to restore correct values. The most complexity-effective solution for providing these write ports is to stall the rest of the processor when exception or interrupt occurs and until the precise register state is restored. Since interrupts and exceptions are generally very infrequent events, stalling the pipeline for a few cycles on these occasions will have a negligible impact on performance.

4.6 Distributing Early Released Registers to Threads

In SMT processors, the early released registers can be reallocated either to the instructions within the same thread or to the instructions from different threads. One alternative is to stall the thread that triggered early deallocations after the ERD mode is entered, thus ensuring that the available registers will only be assigned to the instructions from other threads. This is beneficial when most of the instructions from the thread in question are load-dependent and they therefore place significant pressure on the issue queue. Fetching further instructions from this thread in such a scenario is likely to deny the issue resources to other threads, thus limiting the overall performance. However, if most of the instructions in the thread that experienced an L2 cache miss are load-independent, then it is beneficial to continue fetching from this thread to exploit memory-level parallelism. In the results section, we evaluate both of these alternatives.

We also propose an adaptive mechanism to decide whether to stall the thread under L2 miss or not. During each pass through the ROB in the course of the ERD phase, we keep a count of the number of not-executed instructions, assuming that all of them are load-dependent. If the count exceeds the value computed as N/W (where N is the number of issue queue entries and W is the number of threads), then the thread is stalled. Otherwise, the thread is not stalled and further fetches continue. Furthermore, we impose the additional limitation that no more half of the executing threads can be allowed to stall at the same time, in order to not impede the performance. We show in the results section that the adaptive technique provides the best overall performance in the majority of cases. Other than these considerations, the nature of register distribution is controlled by the DCRA policy.

Of course, if a thread executing in the ERD mode is not blocked, then port arbitration within the ROB needs to be performed between the instructions writing the PC values in the course of regular dispatching and the instructions writing the values of the early deallocated registers in the ERD phase. Since the early deallocation process of a group of registers can be spread over several cycles in a nonpipelined fashion without impacting the performance (Section 4.3 and Figure 8), we give the priority for these ROB ports to the newly dispatching instructions. The number of instructions to be written into the ROB is known several cycles in advance (after those instructions are fetched), so appropriate port reservations can be made.

5. RESULTS AND DISCUSSIONS

In this section, we discuss the performance implications of the proposed L2 miss-driven early register deallocation scheme (referred to as L2_ED in the rest of this section) for both single and multithreaded machines. We also report various supporting statistics, perform the sensitivity analysis to the memory latency, the L2 cache size and the ROB size and compare our results with a previously proposed scheme for early deallocation of registers.

Figure 3 depicts the throughput IPC as a function of the RF size for various register management/distribution schemes on a 8-way SMT configured as

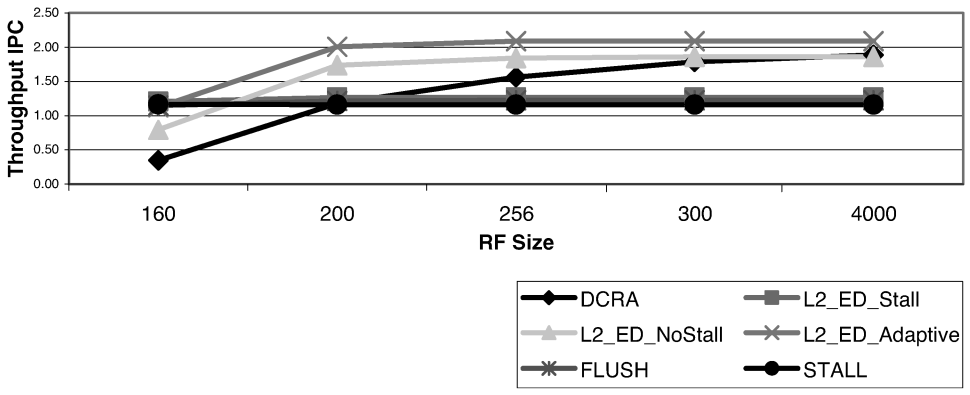


Fig. 3. Throughput IPC for various register file sizes.

shown in Table I. The results are presented for the following schemes: baseline SMT implementing DCRA resource distribution policy (Base_DCRA), FLUSH mechanism of [Sarangi et al. 2005], STALL mechanism of [Sarangi et al. 2005] and the three variations of the L2_ED mechanism proposed in this article. The three variations of the L2_ED scheme differ in whether threads in the ERD phases are stalled (L2_ED_STALL) or not stalled (L2_ED_NoStall) or the stalling decisions are made dynamically based on the pressure that the threads present to the shared issue queue, as described in Section 4.6 (L2_ED_Adaptive). The sizes of the RFs (both integer and floating point) are varied from 160 registers (of which 128 are used to embody the architectural state for 4 contexts) to infinite number of registers. The best performance is achieved by the L2_ED_Adaptive scheme for all register file sizes, except for very small RF of 160 registers, at which size it is better to either use L2_ED_Stall or even existing STALL or FLUSH mechanisms. This is because when the supply of renaming registers is extremely and unreasonably scarce (i.e., only 32 integer + 32 fp renaming registers are available for 4 threads), allocating more registers to a thread that triggered the L2 cache miss to exploit MLP does not justify the limitations imposed on other threads. For 256-entry RFs, the L2_ED_Adaptive scheme outperforms the baseline with DCRA by 33.3% and outperforms FLUSH by 69%. For 200 registers, these percentages are 71% and 64%, respectively.

Figure 4 presents similar trends in terms of fairness metric (or harmonic mean of weighted IPCs). For 256-entry RFs, the L2_ED_Adaptive technique shows 25% improvement over baseline machine with DCRA policy and 88% improvement over FLUSH. These gains are 68% and 80% for 200 registers and 9% and 88% for 300 registers, respectively. In the rest of this section, we report the results only for the L2_ED_Adaptive mechanism (due to the space constraints).

Figure 5 presents detailed per-benchmark statistics for 256 integer and 256 floating point registers. Results are shown in terms of the fairness metric for the same register management schemes that were considered in Figures 3 and 4. Notice that never stalling a thread with L2_ED_NoStall mechanism performs

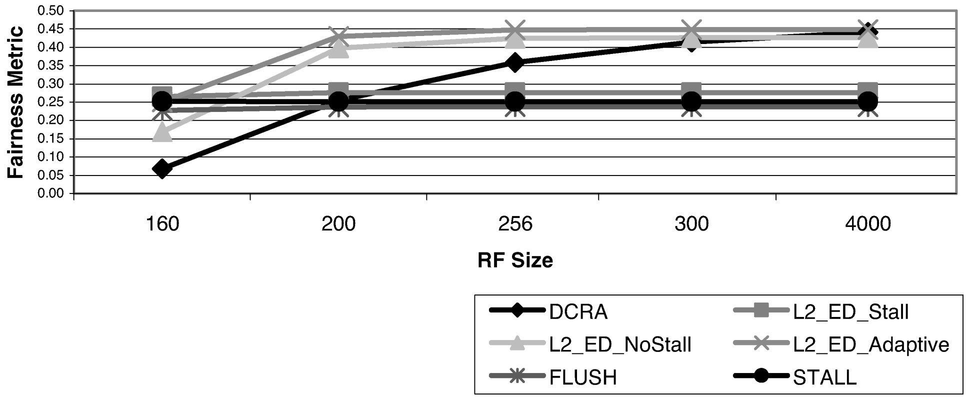


Fig. 4. Fairness metric for various register file sizes.

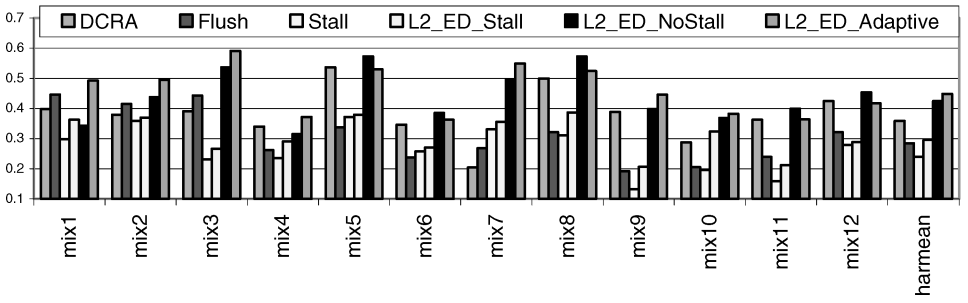


Fig. 5. Per-benchmark performance results for 256-entry RFs on 4-way SMT; results are presented in terms of the "fairness" metric of harmonic mean of weighted IPC.

better than the adaptive scheme on some workloads (namely, mixes 5, 6, 8 and 12; i.e., the mixes dominated by high-ILP benchmarks; these benchmarks do not present a long-term issue queue pressure, therefore it is more beneficial for performance not to stall them during the ERD phases), but on average, the adaptive scheme still provides the best performance. At least one of the variations of the L2_ED scheme outperforms the baseline machine with DCRA on all examined workloads.

Figure 6 presents the amount of early released registers as a percentage of all registers that are examined and considered for early deallocation (i.e., the destination registers of the instructions in the ROB following a load that missed into the L2 cache). Just like in Figure 5, these statistics are presented for each threaded-threaded benchmark mix that we used for 256 integer and 256 floating point registers. Almost 60% of all potentially releasable registers are actually early deallocated if the harmonic mean across all mixes is considered, ranging from 40% for mix 1 to 70% for mix 3. This is not surprising because most of the instructions following the load are load-independent—the scenario presented by the example of Figure 2 is quite typical. Despite the need to maintain the sources and the destinations of load-dependent instructions, the resulting percentage of early released registers is still high.

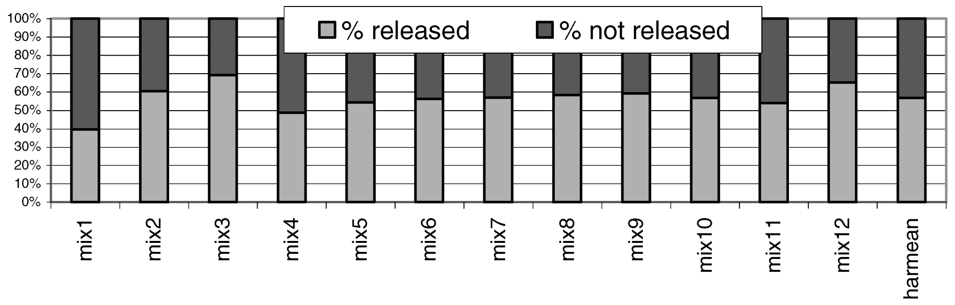


Fig. 6. Proportion of early deallocated registers.

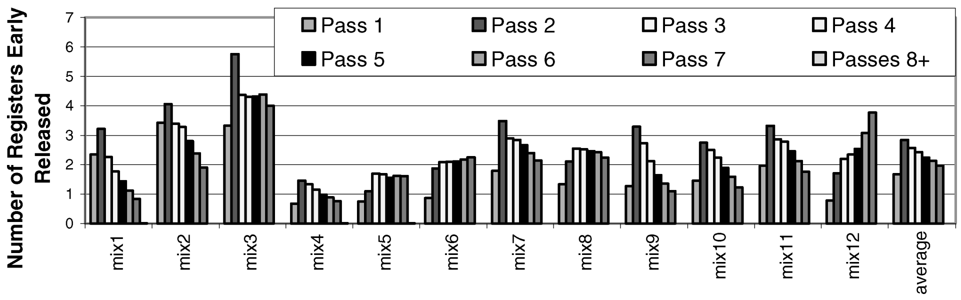
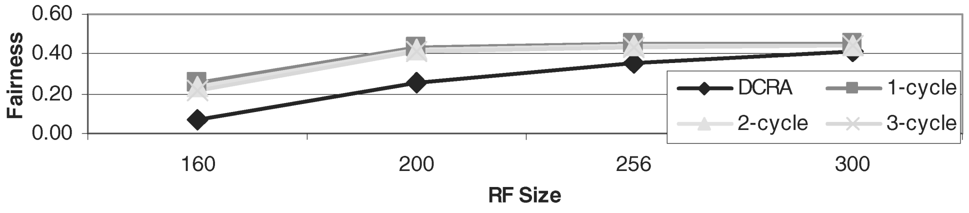


Fig. 7. Number of registers released during each pass through the ROB during the ERD phase.

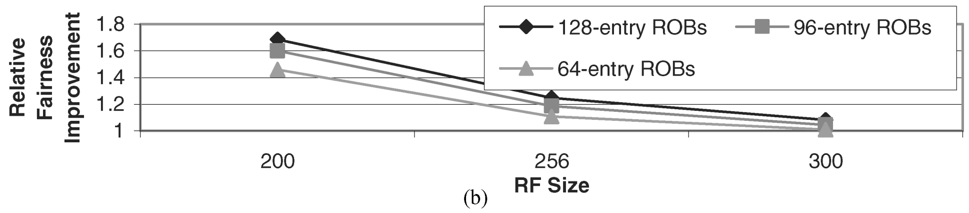
Figure 7 presents the distribution of the released registers across the passes through the ROB during the ERD phase. Each bar corresponds to the number of registers early released during the corresponding pass. For most of the workloads, between two and three registers are leased during each pass. The largest number of registers is released during pass 2. By that time, many load-independent instructions complete their execution and the previous mappings of their destination registers become eligible for early deallocation. Another interesting result is that not many registers are released after pass 7 (the column 8 presents the cumulative number of registers released for passes 8 and beyond). Therefore, it is practical to limit the number of passes through the ROB in the ERD phase to less than 10.

As described in Section 4.6, the early deallocation of registers occurs off the critical path. Figure 8(a) examines the impact of implementing this logic in a nonpipelined fashion over one, two, and three cycles. As shown in the graph, the performance is relatively insensitive to the delay of the early deallocation logic up to three cycles. For example, for the machine with 256-entry register files, the performance difference between the early deallocation delay of one cycle and three cycles is only 4% on the average—still providing more than 20% speedup over the baseline machine with the DCRA policy.

Figure 8(b) presents the sensitivity of the proposed scheme to the size of the per-thread ROBs. Results are presented in terms of the fairness metric; the harmonic mean across all workloads is shown. As expected, with smaller per-thread ROBs the performance gains achieved by the proposed technique are



(a)



(b)

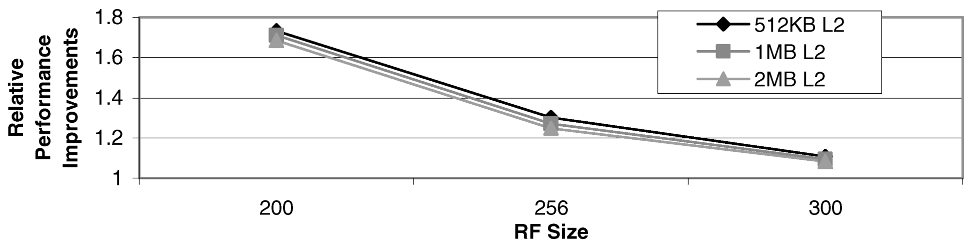
Fig. 8. Performance Sensitivity to (a) multicycle delays in saving the values of deallocated registers and (b) the ROB size.

somewhat reduced. Still, with 256-entry RFs, the performance improvements of almost 10% are achieved even for the small 64-entry ROBs.

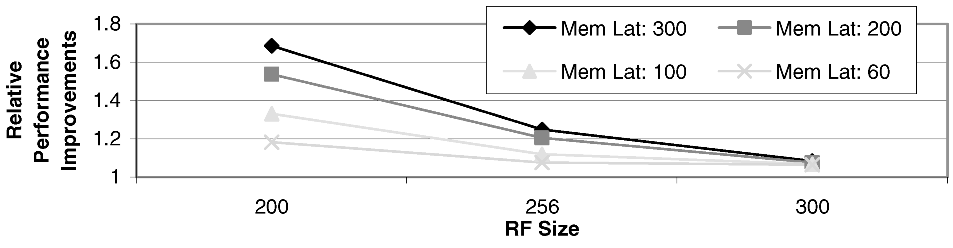
Figure 9(a) shows the performance sensitivity to the L2 cache size. As expected, slightly higher performance gains are obtained for the smaller L2 caches, as more opportunities for early deallocation are presented due to more frequent cache misses. However, the differences between the relative performance improvements of the L2_ED scheme compared to the baseline machine with 512KB and 2MB L2 caches are very small (only 5% for 256-entry RFs). Therefore, even with the large L2 caches, the L2_ED technique still provides significant performance advantages.

At first glance, the results and trends presented in Figure 9(a) may appear somewhat surprising, as larger caches should conceivably provide fewer opportunities for early register deallocation using our technique (due to the higher hit rates). To get additional insight into the issue, Table V presents the L2 cache miss rates for our 12 simulated workloads (combined miss rates are shown) for the L2 cache sizes of 512KB, 1MB, and 2MB. As shown in the Table V, for some mixes, there is almost no difference in terms of the miss rates between 512KB and 2MB caches (e.g., mix 1 and mix 3). Typically, the difference is small for mixes dominated by the low-ILP benchmarks. For those cases, the performance of our scheme would not be impacted by larger cache size. On the other hand, the overall L2 miss rates are lower for the larger cache sizes for the mixes dominated by high-ILP benchmarks. However, those benchmarks typically do not require a large number of physical registers to begin with, as the instructions go through the pipeline smoother. Examination of individual benchmark behavior results in similar observations. Combined, this two factors results in a situation where the performance improvements achieved by our scheme are only 5% lower for 2MB caches than for 512KB caches, as reported earlier.

Figure 9(b) presents the sensitivity of the proposed scheme to the memory latency. Results are presented in terms of fairness improvement relative to the



(a)



(b)

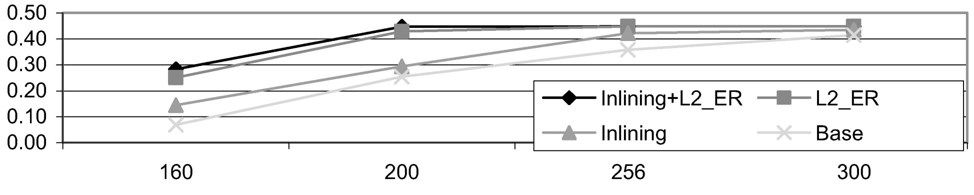
Fig. 9. Performance Sensitivity to (a) L2 cache size and (b) memory latency. Results are shown in terms of improvement in the Fairness metric with respect to the corresponding baseline machine.

Table V. L2 Cache Miss Rates for the Individual Simulated Benchmark Mixes (in percentages)

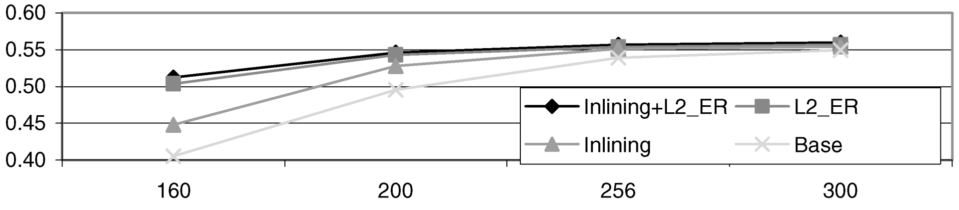
	Mix 1	Mix 2	Mix 3	Mix 4	Mix 5	Mix 6	Mix 7	Mix 8	Mix 9	Mix 10	Mix 11	Mix 12
512KB	57	55	93	29	61	25	29	33	90	46	43	17
1MB	56	47	92	21	61	19	14	17	86	32	30	6
2MB	54	38	90	10	45	16	10	13	23	23	15	5

corresponding baseline case. Again, with smaller memory latency the performance gains of the L2_ED decrease, as expected, although even for reasonably small memory latencies (e.g., 60 cycles) substantial benefits of our scheme are still realized. This is because most early register deallocations occur during the early cycles of the load miss service duration. For 256-entry RFs, the performance improvements on top of DCRA are 25%, 21%, 12%, and 8% for the memory latencies of 300 cycles, 200 cycles, 100 cycles, and 60 cycles, respectively. This indicates that, as the memory/processor gap continues to grow, the relative benefits of our L2_ED technique will commensurately increase and unless the memory latencies are extremely short (i.e., 60 cycles), then substantial performance advantages are still realized.

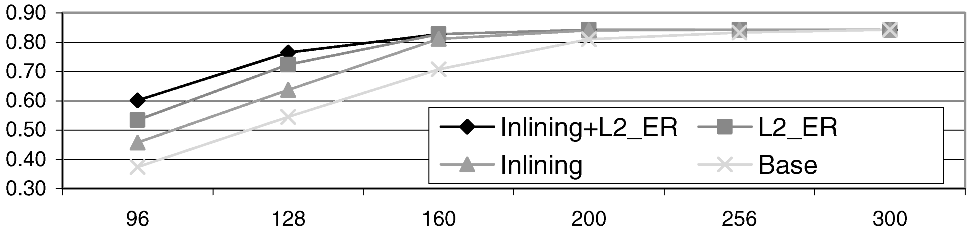
Figure 10 presents the direct comparison between the L2_ED_Adaptive technique proposed in this article and physical register inlining (PRI) scheme for early register deallocation that embeds the narrow-width values directly within the rename table and deallocates the corresponding destination registers earlier. We implemented both schemes and applied them to the SMT machines with four, three, and two threads, as well as the single-thread superscalar machine



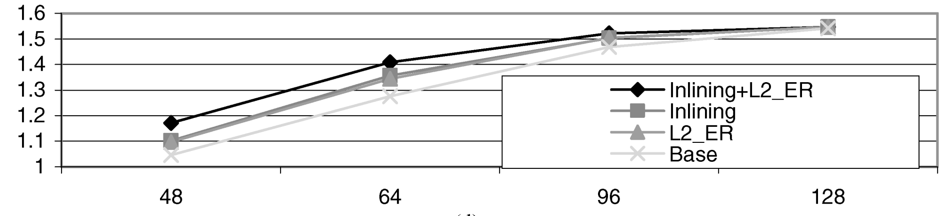
(a)



(b)



(c)



(d)

Fig. 10. Performance Comparison with Physical Register Inlining for (a) four-threaded workloads, (b) three-threaded workloads, (c) two-threaded workloads, and (d) single-threaded (superscalar) workloads. All results are presented for Fairness metric, except for graph 11(d), which uses IPC.

and present these results in Figure 10, respectively. For the PRI scheme, we assume that the additional tag buses are allocated to perform the rebroadcasts of the new mappings, thus not impacting the instruction scheduling. The bottom curve in each figure shows the performance (throughput IPC) of a baseline SMT machine with DCRA resource distribution. The curves right on top of it show the performance of the PRI scheme. The next set of curves show the performance of the L2_ED scheme, and finally, the topmost curves show the impact of combining PRI and L2_ED (recall that these techniques are synergistic

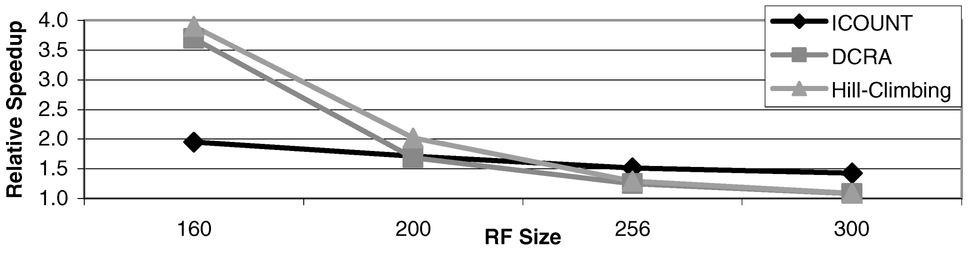
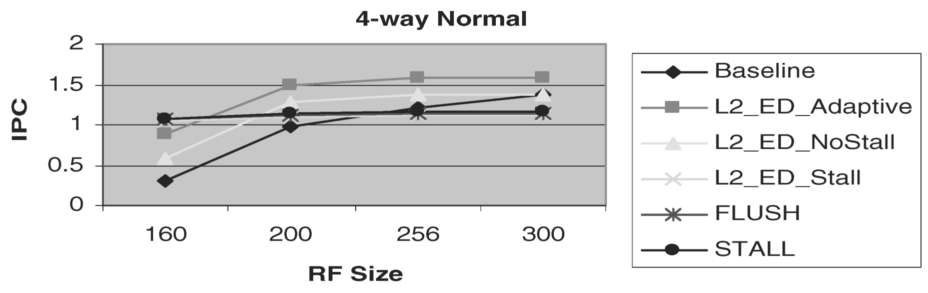


Fig. 11. Fairness improvement of the L2_ED mechanism over various fetching and resource distribution polices for a 4-way SMT processor.

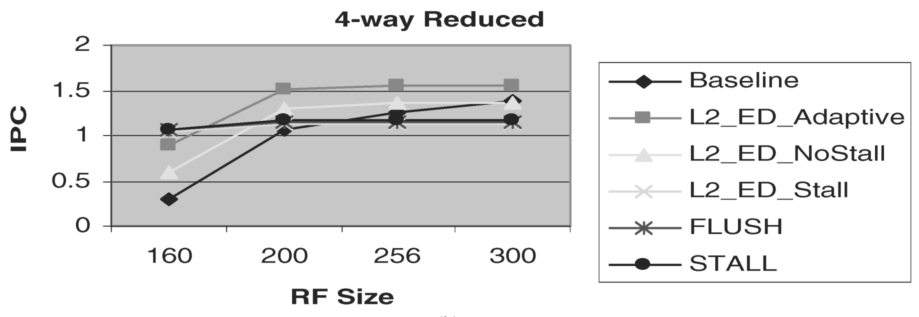
in nature since they exploit different and disjoint early register deallocation opportunities). As seen from the graphs, the L2_ED scheme significantly outperforms PRI on all register file sizes for a 4-way SMT. For example, the gains of L2_ED over PRI are 46% and 8% respectively for 256-entry and 200-entry RFs. Another way to look at this graph is that applying L2_ED on top of PRI still provides significant additional performance advantages, although the opposite is not true.

Similar results are presented for the three-, two-, and one-threaded workloads in Figures 10b, 10c, and 10d, respectively. For both 2- and 3-threaded workloads, the L2_ED technique provides significant additional performance gains with the DCRA policy and outperforms the PRI scheme for most sizes of the register files. On the other hand, for the single-threaded processor, the L2_ED technique provides performance that is on par with physical register inlining—about 5% gains for 64-entry RFs. In this case, when both L2_ED and PRI are used, the resulting synergy achieves an 11% performance gain—nearly additive.

Figure 11 summarizes the performance advantages achieved by the L2_ED technique over various fetching and resource distribution polices for a 4-way SMT. Results are presented in terms of harmonic mean for all studied workloads. Notice that the individual performance of ICOUNT, DCRA and Hill-Climbing can not be directly compared against each other using this graph; the only purpose of this graph is to present the additional gains provided by the L2_ED scheme over each of these mechanisms. For 256-entry integer and 256-entry floating point register files, the L2_ED technique provides 51% performance improvement on top of ICOUNT, 33% on top of DCRA, and 39% on top of Hill-Climbing. For 200 registers, these percentages are 71%, 69% and 101%, respectively. Notice that for the very register constrained datapath (toward the left side of the graph) the gains of our scheme compared to DCRA and Hill-Climbing are much higher than they are compared to ICOUNT because both DCRA and Hill-Climbing allocate the early deallocated registers more intelligently by exploiting the memory behavior of the individual constituents of the multithreaded workloads. As such, a synergy is present between these resource allocation techniques and the L2_ED technique that results in very significant gains for the register-constrained datapath. On the other hand, for the datapath with the large register files (towards the right hand side of the



(a)



(b)

Fig. 12. Performance for 4-way machines (a) with full-sized datapath resources and (b) with reduced resources.

graph), the gains over DCRA and Hill-Climbing are small (as the performance of those techniques approaches that of the machine with the infinite number of registers and the RF bottleneck lessens as a result of efficient register distribution by these schemes), but the gains over ICOUNT remain significant (as the register file is still a bottleneck at these sizes with ICOUNT policy).

Finally, we also evaluated our scheme in the context of a less aggressive 4-way superscalar processor. Specifically, we considered two additional configurations: a 4-way machine with the same datapath resource sizes as in Table I (only the number of functional units was reduced appropriately), and a 4-way machine with the number of entries in key datapath queues halved compared to what is shown in Table I. Results are presented in Figure 12(a,b), respectively.

As shown in these graphs, even for 4-way machines, significant performance gains are achieved by our scheme. Specifically, for a 4-way machine with the same resource configurations as in Table I, L2_ED_Adaptive scheme outperforms the DCRA by 29% and FLUSH by 37% for 256-entry register files, and DCRA by 54% and FLUSH by 32% for 200-entry register files. Furthermore, for a 4-way machine with reduced configuration of all datapath queues, L2_ED_Adaptive scheme outperforms the DCRA by 22% and FLUSH by 34% for 256-entry register files, and DCRA by 40% and FLUSH by 30% for 200-entry register files. To summarize, sizable performance improvements are realized even for 4-way processors.

6. CONCLUDING REMARKS

Physical register file is one of the most critical and performance limiting resources in SMT processors that constrains the number of simultaneous threads that can be supported. In this article, we proposed a novel mechanism for early deallocation of physical registers to increase the register file efficiency and provide higher performance for the same number of registers. Our technique specifically exploits two fundamental trends in multithread processor design: (1) increasing memory access latencies and, (2) relatively higher number of L2 cache misses due to cache sharing effects.

The early register deallocation scheme proposed in this article has the following key advantages:

- It works *synergistically* with, and can be applied on top of, existing early register deallocation/late allocation mechanisms as well as existing SMT resource distribution polices such as DCRA [Burger and Austin 1997] and Hill-Climbing [Cazorla et al. 2003].
- Applied to a four-threaded 8-way SMT machine with 256 integer and 256 floating point registers (for the combined 512 registers), it provides additional gains of 33% (25%) on top of DCRA mechanism, 38% (26%) on top of Hill-Climbing technique, and 51% (48%) on top of ICOUNT fetching policy in terms of the throughput IPC (fairness metric).
- It is effective for various memory latencies, L2 cache sizes, ROB sizes, and lower degrees of datapath superscalarity.
- It does not incur tag rebroadcasts, register remappings, associative searches, rename table modifications; does not necessarily require register file checkpoints; does not require per register consumer counters; and involves no additional storage within the datapath. Instead, it relies on a simple off-the-critical-path logic at the back end of the pipeline to identify the early deallocation opportunities and save the values of the early deallocated registers for precise state reconstruction.

REFERENCES

AKKARY, H., ET AL. 2003. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings from the 36th Annual International Symposium on Microarchitecture*.

BALAKRISHNAN, S. AND SOHI, G. 2003. Exploiting value locality in physical register files. In *Proceedings from the 36th Annual International Symposium on Microarchitecture*.

BORCH, E., ET AL. 2002. Loose loops sink chips. In *Proceedings from the 8th International Symposium on High-Performance Computer Architecture*.

BURGER, D. AND AUSTIN, T. 1997. The SimpleScalar tool set: Version 2.0. (Tech. Report, Dept. of CS, Univ. of Wisconsin-Madison, June 1997 and documentation for all Simplescalar releases).

CAZORLA, F., ET AL. 2004. Dynamically controlled resource allocation in SMT processors. In *Proceedings from the 37th Annual International Symposium on Microarchitecture.*

CAZORLA, F., ET AL. 2003. Improving memory latency aware fetch policies for SMT processors. In *Proceedings from the 9th International Conference on High Performance Computing*.

CHOI S. AND YEUNG, D. 2006. Learning-based SMT processor resource distribution via Hill-Climbing. In *Proceedings from the 33rd International Symposium on Computer Architecture*.

DE VRIES, H. Understanding the detailed architecture of AMD's 64-bit core. Available at: http://www.chip-architect.com/news/2003_09_21_Detailed_Architecture_of_AMDs_64bit_Core.html.

El-Moursy, A. and Albonesi, D. 2003. Front-end policies for improved issue efficiency in SMT processors. In *Proceedings from the 9th International Symposium on High-Performance Computer Architecture*.

Ergin, O., et al. 2004. Increasing processor performance through early register release. In *Proceedings from the 22nd IEEE International Conference on Computer Design*.

Gonzáles, A., Gonzáles, J., and Valero, M. 1998. Virtual-physical registers. In *Proceedings from the 4th International Symposium on High-Performance Computer Architecture*.

Karkhanis, T. and Smith, J. 2002. A day in the life of a data cache miss. In *Proceedings from Workshop on Memory Performance Issues*.

Kirman, N., et al. 2005. Checkpointed early load retirement. In *Proceedings from the 11th International Symposium on High-Performance Computer Architecture.*

Kucuk, G., et al. 2002. Low complexity reorder buffer architecture. In *Proceedings from the 16th International Conference on Supercomputing.*

Lipasti, M., et al. 2004. Physical register inlining. In *Proceedings from the 31st International Symposium on Computer Architecture.*

Lo, J. L., Parekh, S. S., Eggers, S. J., Levy, H. M., and Tullsen, D. M. 1999. Software-directed register deallocation for simultaneous multithreaded processors. *IEEE Trans. Parallel and Distributed Systems 10*, 9, 922–933.

Luo, K., et al. 2001. Balancing throughput and fairness in SMT processors. In *Proceedings from the 2001 IEEE International Symposium on Performance Analysis of Systems and Software*.

Martinez, J., et al. 2002. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings from the 35th Annual International Symposium on Microarchitecture.*

Monreal, T., et al. 2004. Late allocation and early release of physical registers. *IEEE Transactions on Computers*.

Monreal, T., et al. 2002. Hardware schemes for early register release. In *Proceedings from the 31st International Conference on Parallel Processing.*

Mutlu, O., et al. 2003. Runahead execution: An alternative to very large instruction windows in out-of-order processors. In *Proceedings from the 9th International Symposium on High-Performance Computer Architecture.*

Marr, D., et al. 2002. Hyperthreading technology architecture and microarchitecture. *Intel Tech. J.* 6, 1.

Oehmkr, D., et al. 2005. How to fake 1000 registers. In *Proceedings from the 38th International Symposium on Microarchitecture.*

Sarangi, S., et al. 2005. Re-slice: Selective re-execution of long-retired misspeculated instructions using forward slicing. In *Proceedings from the 38th International Symposium on Microarchitecture.*

Sharkey, J. M-Sim: A flexible, multi-threaded simulation environment. Available at: http://cs.binghamton.edu/∼jsharke/m-sim

Sharkey, J. and Ponomarev, D. 2007. An L2-miss-driven early register deallocation for SMT processors. In *Proceedings from the 21st International Conference on Supercomputing.*

Sherwood, T., et al. 2002. Automatically characterizing large scale program behavior. In *Proceedings from the 10th International Conference Architectural Support for Programming Languages and Operating Systems.*

Smith, J. and Pleszkun, A. 1985. Implementation of precise interrupts in pipelined processors. In *Proceedings from the 12th International Symposium on Computer Architecture.*

Srinivasan, S., et al. 2004. Continual flow pipelines. In *Proceedings from the 10th International Symposium on Architectural Support for Programming Languages and Operating Systems.*

Tullsen, D., et al. 2001. Handling long-latency loads in a simultaneous multi-threaded processor. In *Proceedings from the 34th Annual International Symposium on Microarchitecture.*

Tullsen, D., et al. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings from the 23rd International Symposium on Computer Architecture.*

Received September 2007; revised March 2008; accepted May 2008