An L2-Miss-Driven Early Register Deallocation for SMT Processors

Joseph Sharkey Dmitry Ponomarev

Department of Computer Science,
State University of New York,
Binghamton, NY 13902-6000
{jsharke, dima }@cs.binghamton.edu

ABSTRACT

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 paper, we propose a novel technique to early deallocate physical registers allocated to threads that 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 re-broadcasts, register re-mappings and/or additional associative searches.

Categories and Subject Descriptors

C.1 [**Processor Architectures**]: Other Architecture Styles –*Pipeline processors.*

General Terms: Performance, Design

Keywords: Simultaneous Multithreading, Register Files

1. INTRODUCTION

Simultaneous Multithreading (SMT) is an important architectural paradigm for increasing the processor throughput in an area-efficient manner by sharing the key datapath resources among the instructions from multiple threads [18,25]. 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 state for each thread as well as to provide a sufficient number of additional registers for renaming, 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 4-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 [3].

An alternative to building large RFs is to use a smaller number of registers in a more efficient fashion. Higher efficiency of register utilization 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 in order to provide a larger supply 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 [25], STALL [24], FLUSH [24], DCRA [5], and Hill-Climbing [7] 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 which 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 [24], but it also incurs non-trivial overheads, because the squashed instructions have to always be re-fetched, rescheduled, re-executed 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.

ICS'07, June 16–20, 2007, Seattle, Washington, USA.

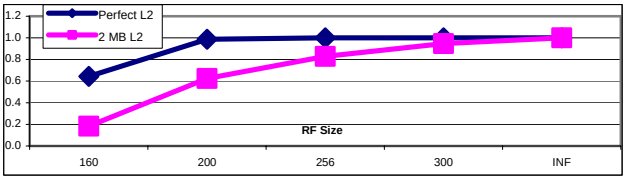

Figure 1. Throughput IPC as a function of RF size

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 [10,15], early deallocation [12,14,15,16] and register sharing [2]. 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 modifications to the thread-specific resources are required. In particular, these techniques often require wakeup tag re-broadcasts, 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 paper, 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 4-threaded SMT machine with perfect L2 cache to one with a realistic L2 cache of 2MB for various RF sizes. As seen from 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 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. Alternatively, a considerably higher performance would be realized using 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.

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 [19]. 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 re-fetch and re-execute all flushed instructions, even those that are independent of the long-latency load.

We propose a mechanism to aggressively release physical registers allocated to threads which experience L2 cache misses by speculatively committing the load-independent instructions and deallocating early 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 instructions within the same thread (to support higher Memory-Level Parallelism (MLP) by overlapping multiple cache misses), as well as to 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. In order to maintain the precise state on interrupts and exceptions, the values of the early released registers are saved directly within the ROB entries of the instructions that triggered corresponding early deallocations – this mechanism avoids the need for full register state checkpointing. 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.

The early register deallocation mechanism proposed in this paper has the following key characteristics:

- It does not incur tag re-broadcasts, register re-mappings, associative searches, rename table modifications or register file checkpoints, does not require per register consumer counters and requires 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 within the existing ROB space for precise state reconstruction.
- While 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 distinct and 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 [5] and Hill-

Climbing [7]. While 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 rest of the paper 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

While most of the related work was briefly described in Section 1, this section provides 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 some 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 [25], FLUSH [24] and STALL [24]) were discussed in Section 1. FLUSH++ [6] 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 [8] avoids fetching from threads that experience an L1 data miss. In [5], 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 [7] 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, [10], [15]. 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 non-trivial increases in the datapath complexity due to the need to: (a) support several levels of register mapping tables, (b) perform various associative searches on the rename table and issue queue after the reassignment of mappings and (c) avoid potential deadlocks. The second set of techniques aim at reducing the register file pressure by using the early deallocation of physical registers [12], [14], [15], [16], [9], [31], [32]. While 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 [12] or register state checkpointing support [9,14]. 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 [16] deallocates a register after commitment of its last consumer, [9] deallocates a register right after commitment of the instruction itself, and [12] and [14] attempt to deallocate registers after the result is produced. [32] completely avoids register allocations to instructions producing transient values. None of these mechanisms directly exploit the L2 cache misses to optimize register file. While 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 paper 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 the results section. The third set of solutions reduces the number of registers through the use of register sharing [2]. To address the RF scalability in a multi-context environment, 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 in [30]. The scheme of [30] requires modifications to the rename stage of the pipeline and the insertion of additional load and store instructions to implement fills and spills.

Checkpointed Processor Architectures. Although we do not rely on creating checkpoints in this work, we draw some inspiration from several recent proposals which use register file checkpointing and exploit load-independence to accelerate the execution of single-threaded applications [11,17,19]. There are some principal differences between our proposal and those prior works and it is important to highlight them. For example, Runahead Execution [17] unblocks the ROB when the missing load reaches the ROB head, creates a full register file checkpoint and then allows the following instructions to pseudo-commit. 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 [11] 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 re-executed 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 re-executed and no resource wastages occur. Most importantly, all pseudo-committed 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 (such as the need to maintain and manage the runahead cache as in [17] and [11]). While our scheme has a somewhat limited utility on a single-threaded superscalar machine as the ROB is likely to quickly fill up and block 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.

A recent study [19] 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 [1,23] 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.

3. METHODOLOGY

For estimating the performance impact of the schemes described in this paper, we used M-Sim [20] - a significantly modified version of the Simplescalar 3.0d simulator [4] that supports the SMT processor model. M-Sim implements separate models for the key pipeline structures such as the issue queue (IQ), the reorder buffer (ROB), 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.

Table 1: Simulated processor configuration.

Parameter	Configuration
Machine width	8-wide fetch, 8-wide issue, 8-wide commit
Window size	64 entry IQ, 48 entry per-thread LSQ, 128-entry per-thread ROB
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)
L1 I–cache	32 KB, 2–way set–associative, 32 byte line, 1 cycles hit time
L1 D–cache	64 KB, 4–way set–associative, 32 byte line, 2 cycles hit time
L2 Cache unified	512 KB, 8–way set–associative, 128 byte line, 12 cycles hit time
BTB	2048 entry, 2–way set–associative
Branch Pred. Per Thread	4K entry gShare, 10-bit global history
Load-latency Predictor	4K entry bimodal predictor
Memory latency	300 cycles
TLB	64 entry (I), 128 entry (D), fully associative

We simulated the full set of SPEC 2000 integer and floating point benchmarks, using the precompiled Alpha binaries available from the Simplescalar website [4]. We skipped the initialization part of each benchmark using the procedure prescribed by the Simpoints tool [21] 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. The specific benchmarks that constituted each of our workloads are available in [29].

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 [13]. 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 paper we refer to this metric as "fairness" to be consistent with some previous work [13].

4. The PROPOSED MECHANISM

We now 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.

	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

Figure 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.

In the example of Figure 2, only 5 out of the 28 instructions which 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, 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 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 instructions from the same thread as well as from other threads. To support precise interrupts, the values of the deallocated registers are directly 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). In order 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 light-weight 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 results section.

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, then 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 (say, 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 the *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 (say 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 [1]. 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).

4.3. Saving the Values of the Deallocated Registers

In order to store the values of the early deallocated registers, three opportunities exist, depending on the specific microarchitecture used and the storage capabilities of the existing structures. Whenever the deallocated values can be squeezed into the already existing datapath components, such opportunities can and should be exploited. The specific choices are as follows:

a) In processors that maintain the full program counter (PC) values within each ROB entry to support precise interrupts [22,26], 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 instead saved within the ROB entry of the instruction that triggered the deallocation, in place of the PC values;

b) 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 [12], the limitation imposed by this restriction does not have a significant impact on performance (as we detail in the results section). The attractive feature of both alternatives (a) and (b) is that no additional storage within the datapath is required. It is also possible that the deallocated values can be stored in some other ROB fields, such as the ones used for storing exception codes. Since the pseudo-committing instruction is guaranteed to be exception-free, the field storing the exception codes can be overwritten.

c) If no storage within the ROB can be exploited or the exploitable storage is too narrow to capture a reasonable percentage of register values, then the additional structure has to be introduced for storing the deallocated values either partially (with the other part stored in the ROB) or in their entirety. In this case, the additional storage can be viewed as a back-up register file, similar to what is described in [27]. Without loss of generality and in the interest of space, we present our evaluations for the first two options. If a separate structure (backup RF) is needed, then the performance advantages will be similar to variation (a), if sufficient number of registers in the backup storage is provided. As the backup RF will be located off the critical path and will not be accessed in the normal course of operations, multi-cycle accesses to this RF can be easily tolerated. Specific microarchitectural implementation of case (c) can be trivially derived from the explanations of cases (a) and (b) below. Furthermore, the backup RF can be implemented by adding some bits to each ROB entry (if the PC bits are not already available), making the implementation of this case similar to that of cases (a) and (b) above.

Here, we only describe the implementation that utilizes the ROB for storing the values of the deallocated registers. In this case, each ROB entry has a separate field for storing the PC (program counter) value of the instruction allocated to this entry, so that if the instruction generates an exception, or the branch misprediction occurs, the address of the offending instruction is immediately available [22, 26]. While 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 ([28] - section 1.14) that suggests that the AMD K8 processors maintain the full PC addresses within the ROB slots. In such case, 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. Neither 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.

While 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. For example, the Alpha 21464 architecture features 64-bit registers and uses 48-bit physical addresses. The more recent implementations of x86-64 ISA from Intel and AMD use 36-bit and 40-bit physical addresses respectively. In all of these cases, the combined width of the PC and exception code (usually a few bits) fields in the ROB is insufficient for storing the full 64-bit value of an early deallocated register. 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 (36

bits or 40 bits), i.e., is a narrow-width value. Otherwise, the deallocation opportunity is forfeited. Previous research [12] showed that a large number of values, both integer and floating point, are narrow-width. For integer values, a large number of upper-order bits are often the same as the sign bit, and for floating-point values, a large number of lower-order bits (i.e. least significant bits of mantissa) are either all zeroes or all ones. Therefore, forfeiting some opportunities to early deallocate registers based on the width of the result is not expected to seriously limit the performance. In the results section, we evaluate the performance sensitivity of the proposed scheme to the available width of the PC field within the ROB, but the main result is that limiting early deallocation to only the registers which hold the values fitting into 36 bits of storage provides performance within 1% of the width-unconstrained deallocation. 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 considered further simplifications to the logic by spreading these activities over several cycles in a non-pipelined manner. The general conclusion from those experiments 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 (non-pipelined 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 multi-cycle non-pipelined implementation of the early deallocation logic.

4.4 Reading 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. 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 instructions within the same thread or to 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 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 not to 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 non-pipelined fashion without impacting the performance (Section 4.3), 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 multi-threaded 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.

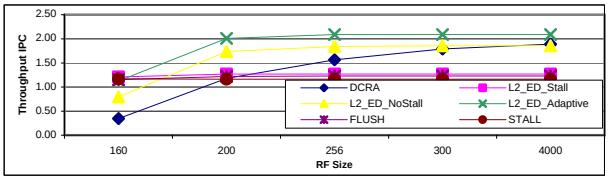

Figure 3: Throughput IPC for various register file sizes.

Figure 3 depicts the throughput IPC as a function of the RF size for various register management/distribution schemes on a 4-way SMT. The results are presented for the following schemes: baseline SMT implementing DCRA resource distribution policy (Base_DCRA), FLUSH mechanism of [24], STALL mechanism of [24] and the three variations of the L2_ED mechanism proposed in this paper. 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.

The trends are similar in terms of fairness metric. In this case, 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% respectively for 300 registers. In terms of the performance of individual mixes, never stalling a thread with L2_ED_NoStall mechanism performs better than the adaptive scheme on some workloads, but on the average, the adaptive scheme still provides the best performance. Our simulations also showed that at least one of the variations of the L2_ED scheme outperforms the baseline machine with DCRA on all examined workloads. In the rest of this section, we report the results only for the L2_ED_Adaptive mechanism (due to the space constraints).

Figure 4 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). These statistics are presented for each 4-threaded benchmark mix (described in [29]) 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.

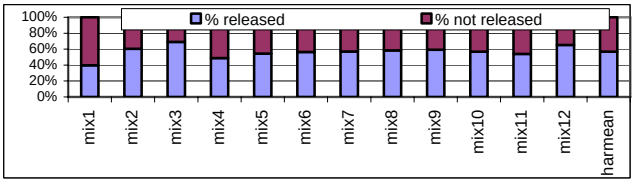

Figure 4. Percentage of early released registers

If we consider the distribution of the early released registers across the passes through the ROB during the ERD phase, then for most of the workloads, between 2 and 3 registers are released 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. Therefore, the number of passes through the ROB can be limited by a fairly small number without forfeiting the opportunity to release additional registers.

As described in Section 4.6, the early deallocation of registers occurs off the critical path. Figure 5 examines the impact of implementing this logic in a non-pipelined fashion over 1, 2 and 3 cycles. As seen from 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 1 cycle and 3 cycles is only 4% on the average – still providing more than 20% speedup over the baseline machine with the DCRA policy.

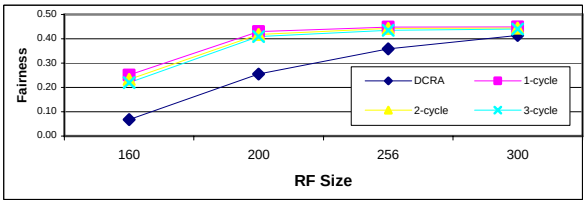

Figure 5. Performance sensitivity (in terms of Fairness metric) to the multi-cycle delays in savings the values of early deallocated registers.

In all previous and all forthcoming results, we assume that the PC field within the ROB is 36-bits long, so that only the registers storing the values fitting within 36 bits can be early deallocated. We also performed the sensitivity analysis of the proposed mechanism to this parameter. Our results indicated that a somewhat lower performance is attained by a few workloads when the PC storage within the ROB is only limited to 32 bits, but the losses in terms of the harmonic mean across all workloads are still tolerable even in that case (9%). When 36 bits of PC storage are available, the performance achieved by the L2_ED scheme is within 1% of the case where the full 64-bits PC storage within the ROB is used (i.e. all data values fit into the PC field, lifting the width-related restrictions on early register deallocations).

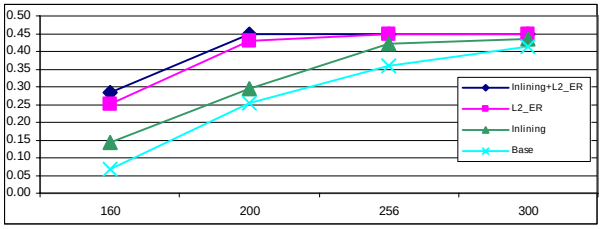

Figure 6. Comparison with Physical Register Inlining for 4-threaded workloads. Results are presented in terms of fairness metric.

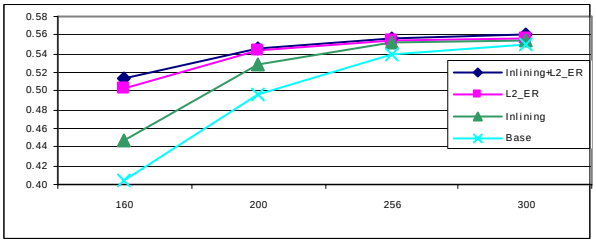

Figure 7. Performance Comparison with Physical Register Inlining for 3-threaded workloads. Results are presented in terms of fairness metric.

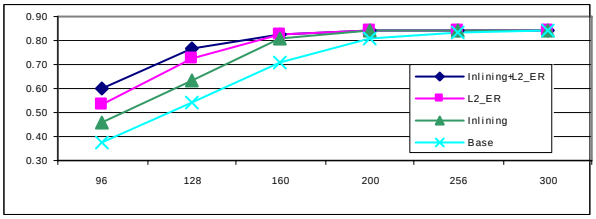

Figure 8. Comparison with Physical Register Inlining for 2-threaded workloads. Results are presented in terms of fairness metric.

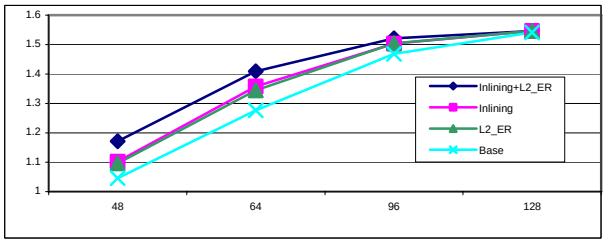

Figure 9. Comparison with Physical Register Inlining for single-threaded workloads. Results are presented as commit IPCs.

As described in Section 2, a number of previous proposals for early register deallocation that were developed for superscalars can, in principle, be applied to SMT processors. The following four figures present the direct comparison between the L2_ED_Adaptive technique proposed in this paper 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 [12]. We implemented both schemes and applied them to the SMT machines with 4, 3, and 2 threads, as well as to a single-threaded superscalar machine. The quantitative results of our comparisons (in terms of fairness metric) are presented in Figures 6, 7, 8, and 9. For the PRI scheme, we assumed that the additional tag buses are allocated to perform the re-broadcasts of the new register 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_Adaptive scheme (referred to simply as L2_ED) proposed in this paper, and finally, the topmost curves show the impact of combining PRI and L2_ED (recall that these techniques are synergistic 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 3-, 2-, and 1-threaded workloads in Figures 7, 8, and 9, 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.

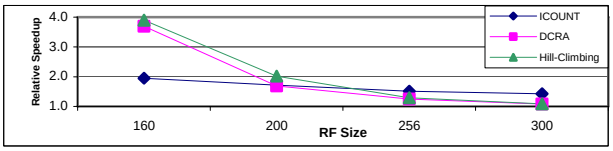

Figure 10: Fairness improvement of the L2_ED mechanism over various fetching and resource distribution polices for a 4-way SMT processor.

Finally, Figure 10 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 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).

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 paper, 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: a) increasing memory access latencies and, b) relatively higher number of L2 cache misses due to cache sharing effects.

The early register deallocation scheme proposed in this paper 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 [5] and Hill-Climbing [7].
- Applied to a 4-threaded 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 does not incur tag re-broadcasts, register re-mappings, associative searches, rename table modifications or register file checkpoints, does not require per register consumer counters and requires 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.

7. ACKNOWLEDGEMENTS

This work was supported in part by NSF grant No. CNS 0454298 and by Intel. We would like to thank Kanad Ghose, Aneesh Aggarwal, Deniz Balkan, Matt Yourst and Hui Zeng for their comments on previous versions of this paper.

8. REFERENCES

[1] H. Akkary, et. al, “Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors”, International Symposium on Microarchitecture, 2003.

[2] S. Balakrishnan, G. Sohi, “Exploiting Value Locality in Physical Register Files”, International Symposium on Microarchitecture 2003.

[3] E. Borch, et al., "Loose Loops Sink Chips", in Proc. Intl Symp. on High Performance Computer Architecture, 2002.

[4] 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.

[5] F. Cazorla, et al. “Dynamically Controlled Resource Allocation in SMT Processors”, Int’l Symposium on Microarchitecture 2004

[6] F. Cazorla, et al. “Improving Memory Latency Aware Fetch Policies for SMT Processors,” HiPC, 2003.

[7] S. Choi, Yeung, D., “Learning-Based SMT Processor Resource Distribution via Hill-Climbing”, Int’l Symp. Computer Architecture, 2006.

[8] A. El-Moursy, D.Albonesi. “Front-End Policies for Improved Issue Efficiency in SMT Processors”, Int’l Symp High Perf. Comp. Arch. 2003.

[9] O. Ergin, et al.,, “Increasing Processor Performance through Early Register Release”, Int’l Conference on Computer Design, 2004

[10] A. Gonzalez, A., J. Gonzalez, M.Valero, “Virtual-Physical Registers”, Int’l Symp. High Perf. Comp. Arch., 1998.

[11] N..Kirman, et al., “Checkpointed Early Load Retirement”, Proc. Intl Symp on High-Perf. Comp Architecture, 2005.

[12] M. Lipasti, et al., “Physical Register Inlining”, Proc. International Symposium on Computer Architecture, 2004

[13] K. Luo, et al. “Balancing Throughput and Fairness in SMT Processors,” Int’l Symposium Perf Analysis of Systems and Software 2001.

[14] J. Martinez, et. al.., "Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors", International Symposium on Microarchitecture 2002.

[15] T. Monreal, T., et al., “Late Allocation and Early Release of Physical Registers”, IEEE Transactions on Computers, 2004.

[16] T. Monreal, et al., “Hardware Schemes for Early Register Release”, International Conference on Parallel Processing, 2002

[17] O. Mutlu, et.al., “Runahead Execution: An Alternative to Very Large Instruction Windows in Out-of-Order Processors”, in Proc. Int’l Symp on High Performance Computer Architecture, 2003.

[18] D. Marr, et al, “Hyperthreading Technology Architecture and Microarchitecture”, Intel Tech. J., vol. 6, No.1, Feb 2002.

[19] S. Sarangi, et al, “Re-Slice: Selective Re-execution of Long-Retired Misspeculated Instructions Using Forward Slicing”, in Proceedings of the 38th International Symposium on Microarchitecture, 2005.

[20] J. Sharkey. “M-Sim: A Flexible, Multi-threaded Simulation Environment.” http://www.cs.binghamton.edu/~jsharke/m-sim

[21] T. Sherwood, et al. “Automatically Characterizing Large Scale Program Behavior.” Proc. ASPLOS, 2002.

[22] J. Smith, and A. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors", Int’l Symp Comp. Architecture, 1985.

[23] S. Srinivasan et al, “Continual Flow Pipelines”, in Proc. of ASPLOS, 2004.

[24] D. Tullsen, et al. “Handling Long-Latency Loads in a Simultaneous Multi-threaded Processor.”, International Symposium on Microarchitecture 2001.

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

[26] G. Kucuk et al., “Low Complexity Reorder Buffer Architecture”, International Conference on Supercomputing (ICS), 2002.

[27] R. Balasubramoniam, S. Dwarkadas, D. Albonesi, “Reducing the Complexity of the Register File in Dynamic Superscalar Processors”, Intl. Symp. on Microarchitecture (MICRO-34), 2001

[28] H. De Vries, “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

[29] J. Sharkey, et al, “An L2-Miss-Driven Early Register Deallocation for SMT Processors”, Tech Report, SUNY Binghamton, at: http://caps.cs.binghamton.edu/ICS07_benchmarks.html

[30] D. Oehmke, et al., “How to Fake 1000 Registers”, in Proceedings of the 38th International Symposium on Microarchitecture, 2005.

[31] D. Balkan et al., “Selective Writeback: Exploiting Transient Values for Energy-Efficiency and Performance”, Int’l Symp. Low Power Electronics and Design, 2006.

[32] D. Balkan et al, “SPARTAN: Speculative Avoidance of Register Allocations to Transient Values for Performance and Energy-Efficiency”, Int’l Conf Parallel Arch and Compilation Techniques, 2006.