Adaptive Memory Synchronization (AMS): Balancing the Risks and Benefits of Inter-thread Load Speculation

Kshitiz Malik, Mayank Agarwal,Matthew I. Frank
{kmalik1,magarwa2,mif}@uiuc.edu
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract

Speculative parallelization (SP) enables a processor to extract multiple threads from a sequential instruction stream, and execute them in parallel. For speculative parallelization to achieve high performance on integer programs, loads must speculate on the data dependences among threads. Techniques for speculating on inter-thread data dependences have a first-order impact on the performance, power, and complexity of SP architectures.

This paper proposes a store-set based synchronization mechanism for SP architectures that leverages previous work on inter-thread register synchronization. The mechanism is simple to implement, requires no selective re-execution or global store broadcast, and uses existing hardware structures that perform register synchronization. Compared to an SP system that speculates across memory dependences, the store-set based synchronizer improves performance by up to 33%, and reduces load violation rate by 80% on average.

However, we find that the store-set synchronizer does not improve performance for all benchmarks: speculating across dependences instead of synchronizing them is a better approach for some applications. Based on this observation, we extend the store-set synchronizer with a likelihood-of-violation predictor that dynamically adapts to application behavior, and finds the right balance between synchronization and speculation. Adaptive memory synchronization (AMS) further improves the performance of the store-set synchronizer by up to 27%, without a significant degradation in violation rate.

1 Introduction

With transistor budgets exceeding one billion transistors, and conventional pipelines scaling poorly, microprocessor manufacturers have recently focussed much attention on multicore processors. Multicore processors are attractive because they dramatically increase instruction throughput on multiprogrammed or parallel workloads without a large increase in design complexity and validation cost.

However, these processors do not improve the performance of single-threaded benchmarks, which are important for a number of application domains. Speculative Parallelization (SP) enables single-threaded programs to take advantage of multicore processors without the need for hand-parallelization. SP is a technique whereby a processor extracts multiple, possibly dependent threads from a sequential instruction stream and executes them in parallel. Speculative Parallelization is specially relevant for hard to parallelize control-intensive integer benchmarks, like the Spec2000 integer suite. Examples of Speculative Parallelization architecture include the original Multiscalar [20] proposal, Stampede [22], TLS [9] and Speculative Multithreading [15], among others.

For Speculative Parallelization to achieve high performance on integer programs, loads must speculate on data dependences among threads. Otherwise, ambiguous inter-thread dependences may render many codes difficult, if not impossible to synchronize. However, when speculation fails, it causes thread squashes that reduce the performance of an SP system.

We propose a store-set based synchronization scheme for SP architectures that extends hardware mechanisms for register synchronization to also perform memory synchronization. We find that the store-set synchronizer reduces dependence violations substantially, resulting in improved performance and higher efficiency (fewer squashed instructions). **While the violation rate reduces by 80% on average, performance increases by up to 33%.**

The store-set synchronizer is a complexity-effective mechanism for performing memory synchronization in a Speculative Parallelization system. Unlike previous proposals, **the store-set based inter-thread synchronizer doesn't require selective re-execution or global broadcast of stores. It uses already existing hardware for register-value synchronization, and thus, adds little to the overall complexity of the system.** The synchronization mechanism also allows an SP architecture to dispatch dependent loads and their forward slices into a FIFO queue, instead of in the scheduler. Thus, the SP processor is able to extract parallelism from a large, effective instruction window.

Perhaps the most important contribution of this paper is an illustration of the fundamental trade-off between speculation and synchronization in an SP architecture. While most proposals for Speculative Parallelization either speculate on all dependences, or synchronize all likely dependences, we find that neither of these techniques work well for all benchmarks. Instead,

the processor should aim to strike the right balance between synchronization and speculation. **The store-set synchronization mechanism was extended with a likelihood-of-violation predictor that dynamically decides for each load instruction, whether the load should be synchronized or whether it should speculate. This adaptive memory synchronization mechanism improves the performance of the store-set synchronizer further by up to 27%, without significantly affecting the power consumption.** Compared to a system that performs blind speculation, adaptive memory synchronization can improve performance by up to 67%.

1.1 Road map

The rest of the paper is organized as follows. Section 2 discusses Speculative Parallelization architecture and terminology. We describe the data-dependence problem in SP architectures at an abstract level, and discuss previous work in register synchronization. In Section 3, we describe how to extend the register synchronization mechanism to also perform inter-thread memory dependence prediction and synchronization by using store-sets. We discuss the need for adaptivity in memory synchronization in Section 3.6.

Section 4 evaluates the performance of our store-set based synchronization mechanism. In Section 5, we discuss an example from vpr.route where adaptivity in the store-set synchronizer is necessary for high performance, following which we evaluate the performance of our adaptive synchronization system. Finally, we summarize our results in Section 6.

2 Background

Speculative Parallelization is an important technique to improve single-threaded program performance on a multi-core processor without cumbersome hand-parallelization. Data dependences between are a first-order determinant of the design complexity and performance of an SP system. This section gives a brief overview of dependence synchronization mechanisms for Speculative Parallelization.

2.1 Speculative Parallelization

The basic premise behind Speculative Parallelization is to identify code that is very likely to be executed sometime in the future. Such code can then be concurrently executed in a separate thread. For example, recent proposals for Speculative Parallelization use profile-based [14] or compiler-based [1] post-dominance analysis to identify future code that is very likely or guaranteed to be executed, respectively.

Figure 1 shows an example CFG, where block A is postdominated by block E. In other words, if block A is executed, then block E is also guaranteed to be executed. A Speculative Parallelization system could exploit this fact upon reaching block A by *spawning* a new thread that starts fetching and (possibly) executing instructions from block E. The original thread is called the spawner thread, while the newly created thread is called the spawned thread. The next section describes the spawn process in more detail. Note that the spawner thread will eventually arrive at block E, and will *reconnect* with the spawned thread. At this point, the spawned thread represents work that the spawner thread is about to execute. Therefore, the spawner thread stops fetching instructions upon reconnecting to the spawned thread.

The first PC of block A is called the spawner PC, while the first PC of block E is called the reconnection PC. Instructions between the spawner and the reconnection PCs are called *skipped instructions*. The thread that is earliest in program order is called the non-speculative thread, while other threads are called speculative threads.

2.2 Data Dependences in Speculative Parallelization

Since a speculative thread execute future work, the machine needs to identify instructions that are dependent on data produced in the *skipped region*. In this section, we present a brief overview of how data dependences are handled in Speculative Parallelization systems. We focus multi-core machines, where each thread executes on a different core, although the techniques described in this paper are applicable to, and were first implemented for a Simultaneously Multithreaded processor.

When the *spawner PC* A, is fetched by a core and another core is idle, a new thread is *spawned* on the idle core. As part of the spawn process, the spawned PC E is communicated to the idle core. To handle data-dependences, a Speculative Parallelization system needs to perform three tasks:

- Identify the set of registers and memory addresses produced in the *skipped region*, often called the *create-set*. For example, in Figure 1, the register r1 would be in the *create-set* since it is written in blocks B and C, while register r8 would not be in the *create-set*.

- Identify instructions in the spawned thread that consume registers or memory addresses that are in the *create-set*. Such instructions are called *waitsFor* instructions, because they must wait for their input data operands to be produced. For example, the instruction in block G will be marked *waitsFor* because it reads a register (r1) that is in the create set. However, the instruction in block E will not be marked *waitsFor*, since r8 is not in the *create-set*. Note that the instruction in block H will not be a *waitsFor* either, because it reads a locally defined value of r1.

- Synchronize waitsFor instructions with their corresponding producers in the spawner thread. In particular, the execution of *waitsFor* instructions must be delayed till after their producer instructions have been executed. Instructions in the spawned thread that are not marked *waitsFor* can execute in the usual manner.

The next two sections look at how Speculative Parallelization systems perform these three steps for register and memory dependences

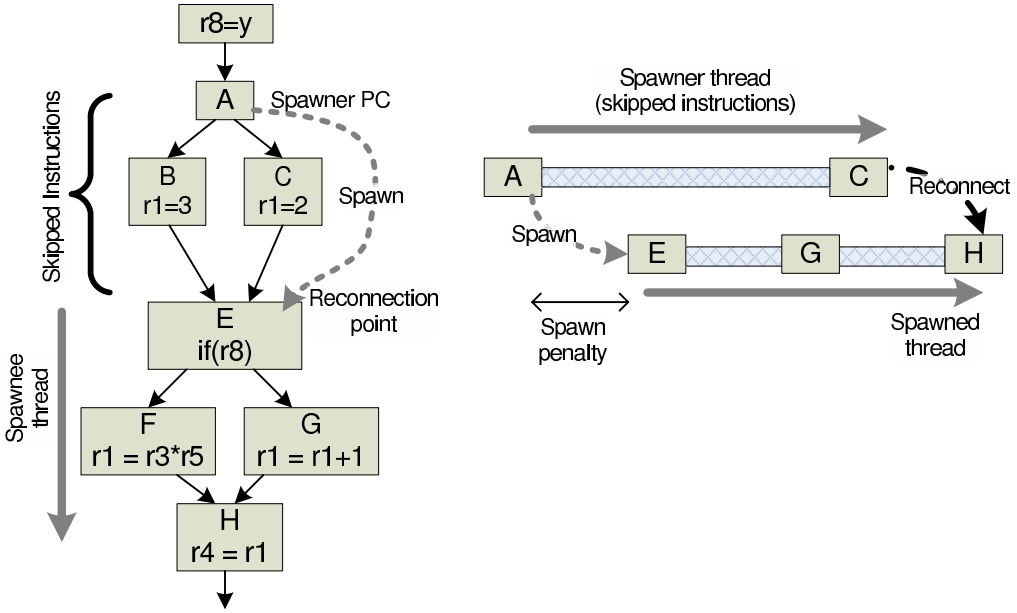

Figure 1. Terminology used in this section. The diagram on the left shows a static control flow graph. The diagram on the right shows a particular dynamic instantiation of the same code, where the Speculative Parallelization system has decided to create a new thread starting at the beginning of block E.

2.3 Synchronizing Register Dependences

Register dependences are relatively easier to handle, because all information about registers is available at compile time.

Identifying the create-set for registers is a relatively trivial process. Multiscalar [24] and Stampede [25] perform a compiler analysis which produces a conservative create-set, taking into account all control-flow paths through the skipped region. However, the synchronization process in these two proposals is quite different: Multiscalar uses a special bus in the processor to perform synchronization and transfer register values between cores. On the other hand, Stampede converts register dependences into loads and stores from memory, which can then be synchronized using full-empty bits.

In this paper, we use a dynamic register synchronization mechanism called RSync [11]. Like Multiscalar, RSync uses a register bus for synchronization; however, it is quite different from Multiscalar's synchronization in other aspects. First, it doesn't require recompilation: the *create-set* and *waitsFor* instructions are both identified dynamically, by using a small predictor. Since the *create-set* is just a prediction, RSync can be less conservative in estimating it: rarely executed control-flow paths can be ignored while constructing the *create-set*. However, to guarantee correctness, RSync requires a bit-vector (as many bits as the number of architectural registers) to identify cases where the prediction was inaccurate. Perhaps the most important difference is that unlike all other Speculative Multithreading systems, RSync dispatches *waitsFor* instructions (and the instructions on their forward slices) into a FIFO buffer called the *divert queue*, instead of dispatching them into the dynamic scheduler. Since *waitsFor* instructions are usually long-latency instructions, *diverting* these and their dependents into a FIFO queue allows RSync to use the scheduler for instructions that have only local (intra-thread) dataflow, resulting in a large effective window.

2.4 Synchronizing memory dependences

While register synchronization is a well-understood area, with simple, complexity effective, and high-performance hardware solutions, synchronizing memory dependences in Speculative Multithreading is a much harder problem. Some amount of dynamic prediction or profiling is required for memory synchronization, since memory addresses cannot be resolved at compile-time. Even inter-procedural alias analysis tends to be quite conservative for real benchmarks [7].

Speculative Parallelization systems have the choice of either predicting and synchronizing memory dependences, or speculatively executing loads assuming no inter-thread dependences. Since accurately predicting memory dependences is a hard problem, there is an inherent trade-off between speculation and synchronization. While synchronization reduces thread squashes from dependence violations, false dependences often arise causing unnecessary serialization. On the other hand, speculation removes the serialization bottleneck, but often causes thread squashes that degrade performance. While other Speculative Parallelization systems are on either end of this trade-off, we pursue a balanced approach in this paper. Our goal is to find the right balance between speculation and synchronization, and we develop a simple predictor towards this purpose.

The memory synchronization mechanism that we propose uses modest hardware. We leverage existing register synchronization mechanisms and extend them to handle memory de-

pendences. As a result, the memory synchronization system uses copies of simple hardware structures that have been well-researched for register synchronization. Moreover, by using RSync, we can build large effective instruction windows by *diverting* dependent loads and their forward slices into a FIFO queue.

Before we present our solution, we describe previous approaches to memory synchronization in Speculative Parallelization.

2.5 Related Work

The simplest approaches towards memory-dependence synchronization are being very aggressive/speculative (assuming no inter-thread memory dependences), or being very conservative (considering all loads as *waitsFor* instructions, unless the compiler can prove independence). The very conservative approach, where loads was taken by some initial work in automatic parallelization [17] [21], however, the amount of parallelization that this technique exposes tends to be quite low.

Some Speculative Parallelization systems like TLS [9], Posh [10], and DMT [2] take the blind speculation approach, which is usually much better than the completely conservative approach in terms of performance. However, since misspeculations are quite common, these architectures often have support for selectively re-executing the misspeculated load and its dependents [19] [2]. Unfortunately, selective re-execution is quite expensive from a power and implementation complexity viewpoint, and does not scale very well either [8].

Among systems that dynamically predict and synchronize memory dependences, Moshovos proposed a memory dependence predictor for Multiscalar that identifies loads that have inter-thread dependences [16]. A centralized implementation achieves average speedups that are within a few percent of a perfect dependence predictor. However, the centralized implementation uses large, fully associative prediction and synchronization tables. A distributed implementation is also presented, but its performance is only marginally better than blind speculation. In the distributed implementation, each thread must broadcast a *wakeup* message to all other threads whenever a store is decoded. This message could potentially cause loads sitting in the schedulers of other threads to be issued. Such fine-grained inter-thread synchronization may not be very attractive from a complexity viewpoint, specially when the performance improvement over aggressive speculation is not very significant.

The clustered speculative multithreaded(SM) processor [13] [12] uses load/store address prediction to perform memory synchronization. When spawning a thread, the clustered SM processor predicts the *addresses* to which the thread will store and from which the thread will load. When the address predictions imply an inter-thread dependence, the clustered SM delays execution of the load in the later thread until the matching store in the earlier thread has executed. Like the Moshovos' predictor, this requires broadcasting of executing stores to all cores on the chip. Moreover, since the misspeculation rate is still quite high, the memory synchronization mechanism is backed up by selective re-execution.

In the next section, we present our proposal for memory synchronization, and describe how it is different from previous work.

3 Store-set based memory synchronization

In this paper, we propose a memory synchronization mechanism for Speculative Parallelization that treats store-sets IDs (SSIDS) as architectural registers [23], and uses well-understood register synchronization mechanisms called RSync [11] to perform memory synchronization. Our proposal does not require selective re-execution, nor does it broadcast executing stores to all cores on the chip. However we find that always synchronizing dependences doesn't work very well. Instead, the architecture needs to be adaptive as different applications behave differently: while some perform well with synchronization, others perform better with speculation. We extend the store-set mechanism with a simple likelihood-of-violation predictor that finds the right balance between synchronization and speculation.

3.1 Store Sets in a conventional processor

The store-set predictor [5] can be used to synchronize loads and stores in an out-of-order processor. This predictor groups static loads and stores into store-sets, such that a load's store-set contains all stores that the load has been found depend upon in the past. The simplest means to learn about load-store dependences is by monitoring dependence violations. The store-set predictor consists of two direct-mapped tables. The store-set index table (SSIT) maps static PCs to store-sets, while the SSAT (store-set alias table) [1] identifies the most recently decoded store for each store-set. Conceptually, the store-set ID of a load is similar to the architectural register of an instruction. Analogously, the SSAT is similar to the Register Alias Table(RAT), in that it maps each SSID to the tag of the most recent store from that SSID. As shown in Figure 2, the scheduler can use the store-set tag of a load instruction to enforce a dependence with a store instruction with the same tag, using a matrix-scheduling mechanism [4] similar to the one used for registers. [2]

We extend the conventional store-set mechanism to synchronize inter-thread register dependences a register synchronization mechanism called RSync. While a complete description of RSync is beyond the scope of this paper, this section provides a brief overview.

3.2 RSync Register Synchronization

Broadly, dynamic inter-thread register synchronization consists of five distinct tasks:

[1] Sometimes called the last-fetched store table(LFST)

[2] A store is considered as both a consumer and a producer of its SSID. While this implies that stores in the same store-set issue in order, it also ensures that a load does not issue until all stores in its store-set have been issued [5].

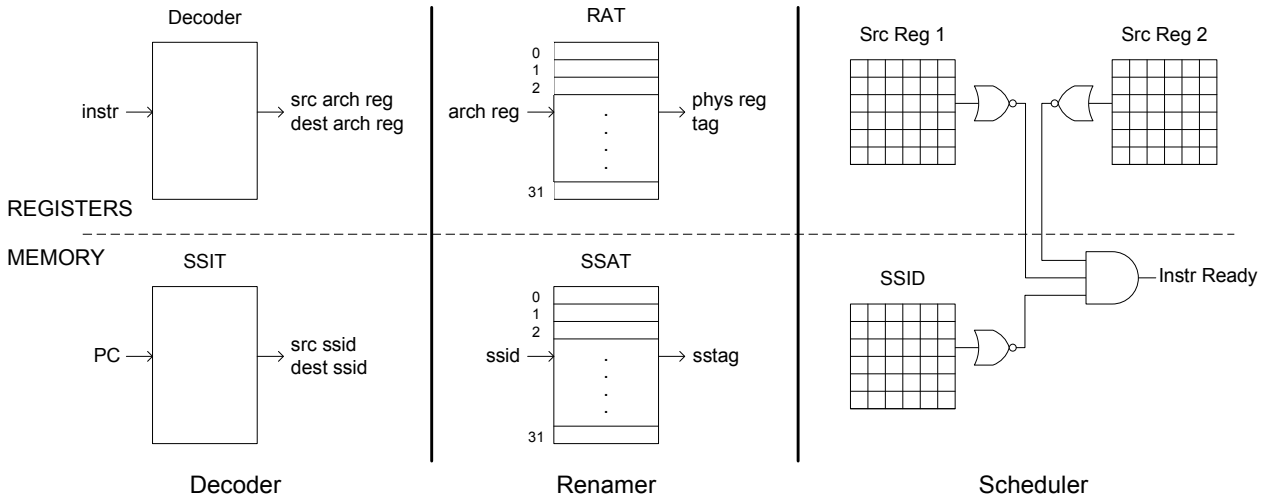

Figure 2. Register and store set synchronization in a superscalar.

- For each spawn, RSync learns the *create-set*: the set of registers that are written in the skipped region. For this purpose RSync uses a simple, 2-way associative, 2KB predictor indexed by the spawnerPC and spawnedPC pair that observes the retiring stream of instructions. For each spawn, the predictor outputs a 32-bit value, representing the *create-set*.

- In speculative threads, RSync identifies *waitsFor* instructions, i.e, instructions for which at least one source operand belong to the *create-set*. In this process, RSync also needs to take into account local producer instructions that may rewrite a register in the *create-set*, thus removing the said register from the *create-set*.

- RSync identifies instructions that are on the forward slice of *waitsFor*(WF) instructions, called *transitive waitsFor* (TWF) instructions. Both WF and TWF instructions are dispatched into the divert queue.

- Since the *create-set* is a prediction, RSync needs to identify cases where an incorrect prediction led to a dependence violation. Such an instance causes a machine flush, similar to a branch misprediction. When a thread reconnects with its successor, the actual *create-set* is compared with the predicted *create-set*. If a register was missing from the predicted *create-set* and was upwards-exposed in the successor thread (was read before it was written), a violation is signaled.

- If the above step does not signal a violation, *waitsFor* instructions in the successor thread are *undiverted*: dispatched into the scheduler. During this process, the front end of the pipeline is stalled.

For these five tasks, RSync stores 4 extra bits for each architectural register in the register alias table. The next section describes how to extend RSync to perform memory synchronization.

3.3 Memory Synchronization using RSync

To perform inter-thread memory synchronization, store-set IDs are treated like registers. A small (2KB, 2-way associative) predictor learns the SSID *create-set* for each spawn point. RSync identifies load instructions in speculative threads that should be marked *waitsFor* because their SSID belongs to the *create-set*. Just like for registers, RSync takes into account local stores that may cause their respective SSIDs to be removed from the working *create-set*.

There are two important differences between synchronizing registers and synchronizing memory. First of all, unlike registers, violation detection for memory operations is not performed by the synchronization mechanism. Since store-sets are only predictions of dependences, the processor needs an address-based violation detection mechanism. We consider the detection mechanism and memory synchronization to be somewhat orthogonal issues. The focus of this paper is on the memory synchronization aspect.

Secondly, while register synchronization requires a special bus to transfer register values, no such bus is required for memory synchronization. Loads simply read values from memory (or the cache): it is the jobs of the synchronization mechanism to ensure that the corresponding producer instructions have already stored the value in memory.

Using the RSync mechanism for memory synchronization has a number of advantages. First of all, the processor can 'slice-out' *waitsFor* load instructions and their transitive dependents into the divert queue, resulting in larger effective instruction windows. Since the slicing mechanism already exists for registers, extending it to memory operations is trivial. Moreover, the hardware structures used to perform memory synchronization (a predictor to identify the *create-set* for SSIDs, mechanisms to identify *waitsFor* instructions given the *create-set*) are simply copies of structures that were already being used to synchronize registers. Thus, treating registers and memory in an analogous fashion reduces design complexity.

3.4 Coarse-grained Synchronization

The complexity of inter-thread memory synchronization can be reduced significantly if loads and stores are synchronized at a coarse granularity. In previous proposals for the Multiscalar [16] and SM processors [13], stores executing in any core are broadcasted to all other cores, and could possibly 'wakeup' loads in other cores. Such fine-grained synchronization may complicate implementation. Instead, we use a coarse grained approach, where inter-thread loads and stores are synchronized only on thread reconnection. When a thread reconnects with its successor, it sends a 32-bit value representing the SSIDs that it wrote to. Using this bit-vector, the successor thread wakes up the corresponding loads in bulk. Such coarse-grain synchronization reduces the bandwidth requirements and implementation complexity of Speculative Parallelization systems.

Using store-set IDs to perform memory synchronization in a multi-core processor requires that the store-set predictors in all cores are kept consistent. To achieve this, memory misspeculations on any core are made visible to all cores. Since memory misspeculations are quite rare (less than 0.5% of loads violate, on average), this doesn't require high bandwidth. Note that other memory synchronization mechanisms, like the ones used in Multiscalar [16] and Clustered Speculative Multithreaded processors [13] also broadcast information about violations to all cores.

An important observation that we make is that loads and stores that belong to the same store-set are not always dependent on each other. Thus, always synchronizing memory dependences can cause unnecessary serialization, as explained in the next section.

3.5 False Dependences in the store-set predictor

As shown in Section 4.3, using the vanilla store-set predictor for memory synchronization in Speculative Parallelization does not improve performance appreciably. While the predictor significantly reduces thread squashes for all benchmarks, only two benchmarks show a net improvement in performance. For some benchmarks, performance actually decreases, which can be attributed to false dependences introduced by the store-set predictor. We find that the store-set predictor occasionally errs on the conservative side when predicting dependences. Enforcing these false dependences does not cause a substantial performance drop in conventional superscalar processors, because the instruction window is relatively small (few tens of instructions). The probability of both the consumer load and the producer store of a false dependence to be in the window at the same time tends to be small, and therefore, these false dependences rarely manifest as serialization.

However, in Speculative Parallelization, the effective window can span hundreds of instructions, specially when divert queues are used. In such a case, false dependences introduced by the store-set predictor can reduce performance substantially by causing unnecessary serialization of loads and stores. We discuss the issue of false-dependences in more detail using an example from the benchmark vpr.route in Section 5.

3.6 AMS: Adaptive Memory Synchronization

To alleviate the problem of unnecessary serialization, we propose a simple predictor which decides, for each static load, whether the load should be synchronized or speculated. The observed inter-thread violation rate for a load is used to decide whether it should be synchronized with stores in other threads that belong to the same store-set, or whether the load should simply be speculated. We show in Section 5that this simple predictor finds the right balance between synchronization and speculation, which directly results in improved performance.

In the next section, we present a simple example that illustrates the process of diversion and undiversion in a store-set synchronizer.

3.7 An Example

Figure 3 shows four successive snapshots of a program executing on a Speculative Parallelization system that uses store-set based memory synchronization. Each snapshot represents active threads as instruction streams, with already fetched (and possibly executed) instructions at the top, and not-yet-fetched instructions at the bottom. Time increases from left to right. Frame A depicts a single thread T0 with a potential spawn point S1 that is yet to be fetched. Since T0 is the non-speculative thread, no SSIDs are in its *create-set*.

In Frame B, thread 0 spawns thread 1. T1 consults the *create-set* predictor, and obtains a single SSID, 9, as its *create-set*. This prediction implies that T0 is expected to have a store with SSID 9 among skipped instructions.

In Frame C, T0 has fetched a store with SSID 9, as was predicted. T1 has fetched a load with SSID 9, and since the *create-set* of T1 contains 9, this load dispatched into the divert queue.

Frame D represents the point in time when thread T0 has *arrived* [3] at the reconnection point. At this point, T0 sends a bit-vector over to T1 which indicates that SSID 9 was written by T0. [4] T1 removes SSID 9 from its *create-set*, and also undiverts the load instruction. During the undiversion process, the front end pipeline stages are stalled till all instructions from the divert queue have been dispatched into the scheduler.

4 Performance Evaluation

4.1 Machine Architecture

We model a distributed architecture with 4 cores, where each core is a 4-wide, out-of-order superscalar processor . Important processor parameters are given in Table 4. Each core has local L1 instruction and data caches and branch predictors. The L1 data caches are kept coherent using an update-based protocol. Note that in Speculative Parallelization systems, at any point in time, only one core is retiring instructions and committing stores to memory.

[3] A thread is said to arrive at the reconnection point when it has fetched and executed all branches up to the reconnection PC.

[4] A two-thread example cannot illustrate why T0 needs to explicitly specify that SSID 9 was written. This functionality is only important for three of more threads.

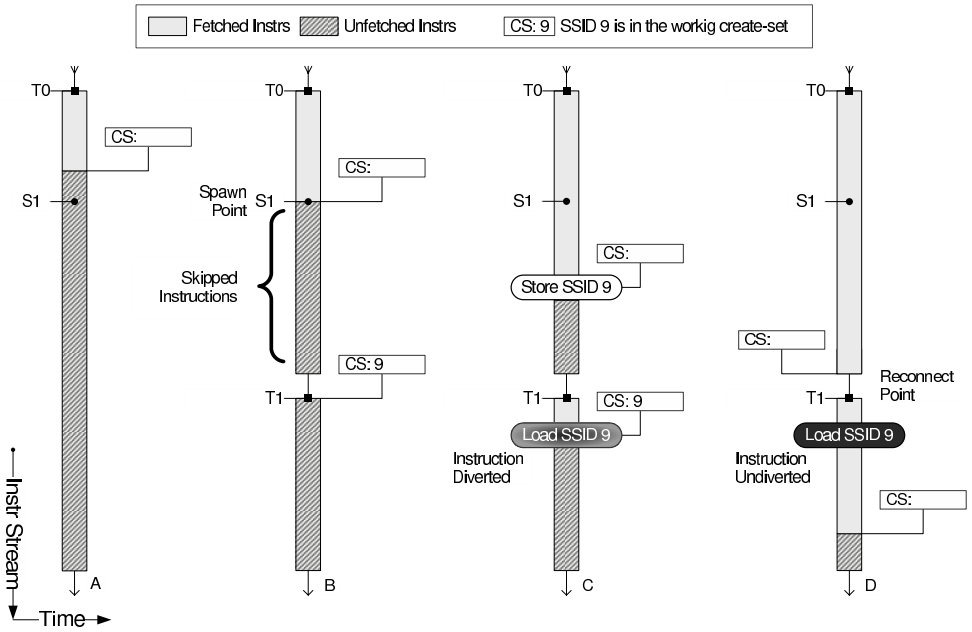

Figure 3. Inter-thread synchronization using store-sets: an example.

When a core (say P) fetches an instruction which is specified as a spawner, it sends to its neighbor (say Q) the program counter of the spawnee. We model a 4 cycle latency for this process. Inter-core register communication happens on a register-value bus, which has a latency of 4 cycles. In each core, *waits-For* instructions are are stored in a FIFO buffer called the *Divert Queue*, similar to the structure used by Al-Zawawi et al [3]. Instructions that have only local (intra-thread) dataflow are sent to the scheduler.

In this paper, violation detection for memory operations is performed by stores snooping the load-queues of other threads. A 4 cycle latency is assumed for this purpose. We are working on implementing a load re-execution based mechanism to replace inter-thread load-queue searches, which would significantly reduce the design complexity of our Speculative Parallelization system. However, the issue of violation detection is somewhat orthogonal to the issue of synchronization.

For inter-thread forwarding, all stores write their value to a single, chip-level speculative cache upon execution, as proposed by Garg et al[6]. About 4% of dynamic loads receive their data from this cache, which takes an extra 4 cycles. In this paper, we don't model the effects of badpath stores forwarding data to loads.

4.2 Simulation Methodology

Our experimental evaluation was performed on a fully execution-driven simulator running a variant of the 64-bit MIPS instruction set ISA. The ISA does *not* have any special instructions to support multithreading. It not only simulates timing, but also *executes* instructions out-of-order in the back end, writing results to the register file out of program order. When an instruction is retired, its results are compared against an architectural simulator, and an error is signaled if the results don't match. When a branch mispredict is discovered, the simulator

Parameter	Value
Pipeline Width	4 instrs/cycle (retire 8 instrs/cycle)
Branch Predictor	192KB Combined, 64KB gshare, 64KB bimodal 64KB selector 18 bits of history
Misprediction Penalty	10 cycles
Reorder Buffer	512 entries
Scheduler	64 entries
Functional Units	4 identical general purpose units
L1 I-Cache	32Kbytes, 4-way set assoc., 128 byte lines, 10 cycle miss
L1 D-Cache	32Kbytes, 4-way set assoc., 64 byte lines, 10 cycle miss
L2 Cache	512Kbytes, 8-way set assoc., 128 byte lines, 100 cycle miss
Diverter Queue	128 entries
Spawn Latency	4 cycles
Inter-core Register Comm. Latency	4 cycles
Number of Store Sets	32
Register Dependence Predictor	2KB, 2-way assoc
Memory Dependence Predictor	2KB, 2-way assoc

Figure 4. Pipeline parameters.

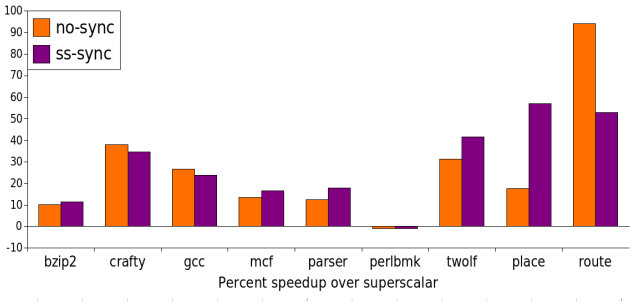

Figure 5. Speedup over superscalar, blind speculation(no-sync) versus full synchronization(ss-sync). The bars place and route refer to the benchmark vpr. While the performance of twolf and vpr.place increases from store-set based synchronization, the performance of vpr.route decreases substantially.

immediately reclaims back end resources (ROB and scheduler entries, etc.) and recovers using a rename checkpoint.

Spawn points that we use are obtained from a control-independence analysis performed on the program binary, along with profiling to identify the best spawn points. For integer benchmarks, branch mispredictions one the most important impediments to high performance. We have found that for these benchmarks, it is most beneficial to spawn across the postdominators of low-confidence branches, which is the spawn strategy used in this paper. As proposed by Agarwal et al [1], a spawn cache can be used to store these postdominators. Since the total number of distinct postdominators are less than 100 for all applications other than gcc (415), we don't model capacity or size constraints for the spawn cache.

We present results from running 9 SPEC2000 integer benchmarks. Our tool chain is incapable of compiling eon and gap. We don't simulate vortex and gzip as these benchmarks have very low mispredict rates (0.58% and 2.01% respectively) on the aggressive tournament predictor that we use, and thus spawn very rarely. The simulator fast forwards through the initialization phase of all benchmarks, and executes 50 million instructions. All the graphs that we present show the speedup of different Speculative Parallelization configurations over a superscalar with the same configuration as a single core in the Speculative Parallelization configuration. Note that intra-thread dependence synchronization is performed using store-sets for both the superscalar and Speculative Parallelization processors. The SSIT is cleared every half million cycles.

4.3 Blind Speculation versus Full Synchronization

Figure 5 shows the speedup of two Speculative Parallelization configurations. The first bar represents an aggressive configuration where memory dependences are not synchronized, called no-sync. The second bar shows the performance of the store-set based memory synchronization mechanism, called ss-sync. Figure 6 shows the corresponding load misspeculation rates for these two configurations, as well as for the original superscalar.

As can be seen, while the load violation rate for the no-sync

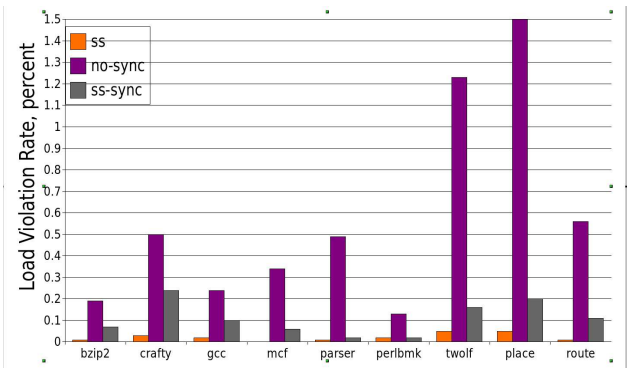

Figure 6. Load violation rate in percentages for superscalar, blind speculation(no-sync) and full synchronization(ss-sync) configurations.

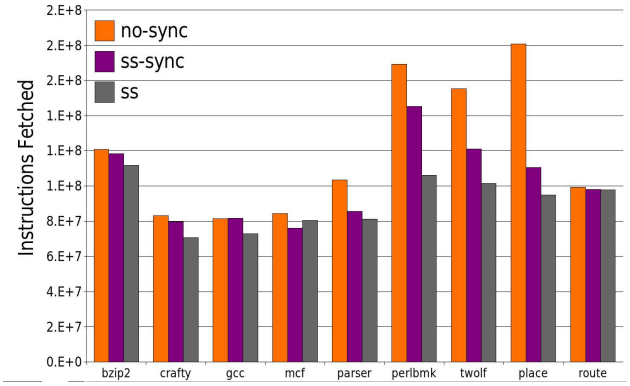

Figure 7. Total number of instructions fetched when 50 million instructions are executed, for superscalar, blind speculation(no-sync) and full synchronization(ss-sync) configurations.

configuration is quite high, the store-set synchronizer reduces the violation rate significantly. However, fewer violations result in a net performance gain for only two benchmarks,twolf and vpr.place. The performance of vpr.route and gcc *decreases* when loads are synchronized with stores from the same SSID that belong to different threads. In the next section, we investigate this phenomenon in more detail.

Figure 7 shows the total number of instructions fetched for the no-sync and ss-sync configurations. As can be seen, ss-sync fetches fewer instructions than no-sync, which is a direct result of fewer squashes. Reducing the number of instructions fetched should reduce the overall power consumption of the Speculative Parallelization system. Thus, synchronizing memory dependences is beneficial from a power standpoint. While we do not measure power consumption directly , we expect that the number of instructions fetched by the processor should be a first order determinant of the overall power.

The two configurations examined here, no-sync and ss-sync are on extreme ends of the synchronization-speculation spectrum, and thus, neither of these strategies works well across the board. In the next section, we describe an adaptive memory synchronizer that finds the right balance between synchro-

nization and speculation. Adaptive memory synchronization achieves high performance, while retaining most of the power advantages of full synchronization.

5 Adaptive synchronization

Section 4.3 showed that full memory synchronization sometimes performs worse than blind speculation. We analyze this phenomenon in more detail in this section.

5.1 Store Sets and dependences

The store-set predictor assigns SSIDs to loads and stores that have participated in dependence violations. Loads and stores that belong to the same store-set are serialized. However, a load and a store that have the same SSID are not necessarily dependent on each other. In other words, the store-set predictor sometimes enforces serialization even when there is no dependence. Note that this characteristic is not unique to the store-set predictor: all memory dependence predictors occasionally enforce false dependence.

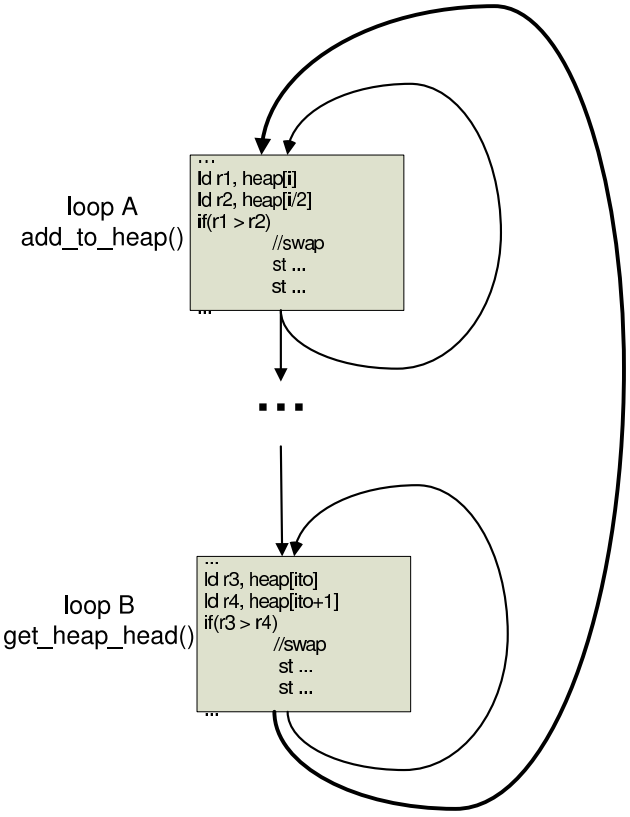

Figure 8. A snippet of code from the benchmark vpr.route. Loop A adds an element to the bottom of the heap and bubbles it up, loop B removes the head of the heap by replacing it with the last element in the heap, which then bubbles down.

Figure 8 shows an example from the benchmark vpr.route. The main loop of vpr route contains two inner loops that operate on a large heap. Loop A, in the function add_to_heap, adds an element to the bottom of a large heap. The element *bubbles up* till the heap property is restored. Loop B, in function get_heap_head removes the top of the heap, which is done by bringing the element at the bottom of the heap to the top, and allowing it to *bubble down* till the heap property is restored. These two loops both have a number of load and store instructions that perform a conditional swap between heap elements. The loads from loop A are frequently dependent on the stores in loop A, so all of these are in the same store-set. Similarly, the loads and stores in loop B are also in the same store set. Also, occasionally, the stores in loop A (which add an element to the heap) touch the same memory addresses as the loads in loop B. For example, if an element that is larger than the top of the heap is added to the heap, it will bubble up to the top in loop A. Loop B will then remove this very element from the heap.

Since the heap is very large, loops A and B *usually* operate on different address sets. However, they operate on at least one common address, and hence, a dependence violation is possible. When such a violation happens, the store-set predictor assigns the same SSID to all loads and stores in loops A and B. As a result, loops A and B are sequentialized.

This serialization does not affect performance significantly on a superscalar processor. Loops A and B are usually long in terms of dynamic instruction counts, and since the scheduler of a superscalar processor is much smaller, A and B rarely execute together. Thus, sequentializing them does not affect performance. Moreover, both of these loops have a number of branch mispredicts: branch instructions that decide whether to swap two heap elements mispredict quite frequently. These mispredictions cause the two loops to be serialized even without taking into account the effects of the store-set predictor.

Unlike in a superscalar, the serialization of loops A and B can be quite detrimental to performance in a Speculative Parallelization system, because these two loops are often active concurrently on different cores. Since branch mispredicts in a core typically don't affect instructions in other cores, the serialization from memory synchronization is the only effect preventing these loops to be execute with a high degree of concurrency.

These two loops are a classical example of a low-probability memory dependence where speculation is better than synchronization, even though speculation occasionally results in violation of the dependence.

Note that the loads in loop A still need to be synchronized with the loads in the same loop, we just need to somehow prevent loads in loop A from being synchronized with the stores in loop B. We take the simple approach of adding a likelihood-of-violation filter to inter-thread synchronization. The idea is that intra-thread store-set synchronization works quite well, but inter-thread synchronization should be performed only for loads that suffer a high number of inter-thread dependence violations. Such an adaptive mechanism will speculatively execute a load assuming no inter-thread dependences, unless the load frequently violates because of stores in a different thread. For a frequently-violating load, synchronization is performed using SSIDs, as explained before. The resulting memory synchronization scheme is called AMS, or adaptive memory syn-

chronization.

The AMS predictor adds a likelihood-of-violation field to each SSIT entry, which contains information about inter-thread violations seen by that store-set. A load is not marked *waitsFor* unless the likelihood-of-violation value of its SSIT entry is greater than 20%. As shown by Riley [18], a predictor to estimate this likelihood can be built with just a one bit per entry, by using probabilistic updates. For the results in this paper, we use two 16-bit integers to count violations and successful retirements for each SSIT entry.

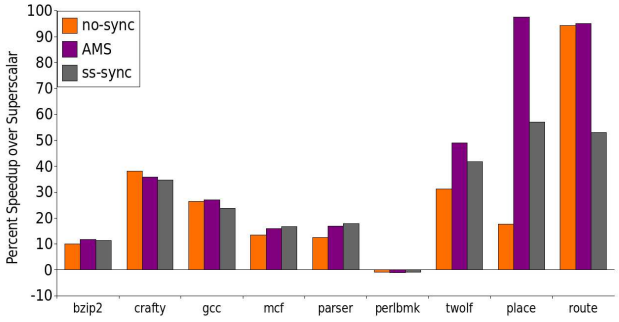

Figure 9. Speedup over superscalar for the blind speculation(no-sync), adaptive synchronization (AMS) and full synchronization(ss-sync) configurations. Adaptive Synchronization finds the right balance between speculation and synchronization.

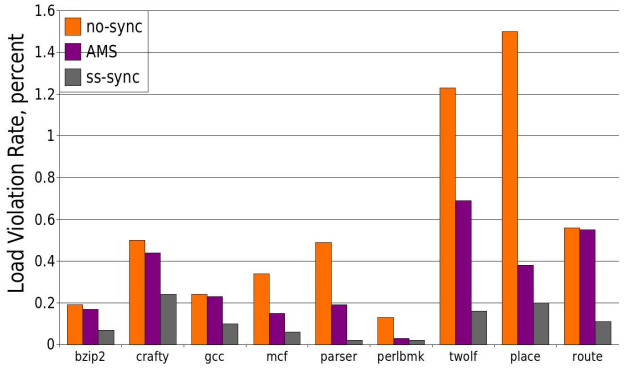

Figure 10. Load violation rate in percentages for blind speculation(no-sync), adaptive synchronization (AMS) and full synchronization(ss-sync) configurations.

Figure 9 shows the performance of our Speculative Parallelization system using AMS. The bars for no-sync and ss-sync from the previous section are also shown for reference. We see that AMS finds the right balance between synchronization and speculation for most benchmarks. For some benchmarks (like vpr.route), it performs as well as the best among no-sync and ss-sync. For some other benchmarks, vpr.place in particular, AMS actually performs better substantially better than both no-sync and ss-sync. This is because AMS can choose, on a per-load granularity, whether to synchronize or to speculate. In no-sync and ss-sync, either all loads are synchronized, or all loads are speculated.

Figure 10 shows the load violation rate for different memory

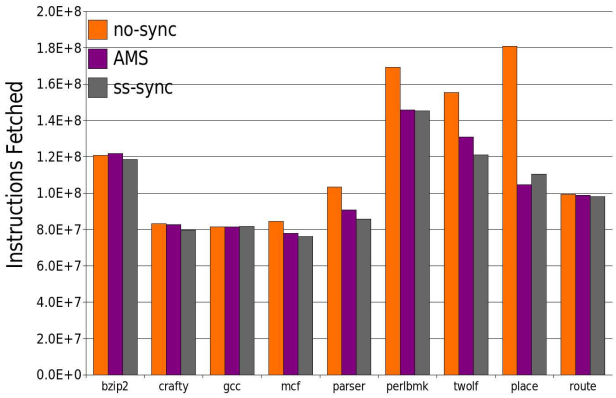

Figure 11. Total number of instructions fetched for blind speculation(no-sync), adaptive synchronization(AMS) and full synchronization(ss-sync) configurations.

synchronization configurations. Figure 11 indicates the total instructions fetched for these configurations. As can be seen, AMS usually preserves the power reductions expected from ss-sync, while improving performance.

Note that on vpr.place, AMS fetches fewer instructions than ss-sync, even though it has a higher load misspeculation rate. This is because in the no-sync configuration, a number of mispredicted branches are marked *waitsFor* because of conservative synchronization. As a result, no-sync fetches many more instructions on the bad path compared to AMS, increasing the overall fetch count.

Note that the original store-set proposal [5] investigated a 2-bit counter for each SSIT entry that down counts if a store in that SSIT did not have the same address as another load in the same SSIT. A store would not be synchronized with any loads unless the 2 bit counter in its SSIT was greater than 1. However, the authors argued against using this counter, since it did not improve performance significantly, and required address comparators which complicate the design. The two-bit counter would not help in the case outlined above, since multiple loads and stores stores share the same SSIT entry. Moreover, comparing the addresses of loads and stores in different cores would require broadcasting store addresses upon execution, defeating the purpose of coarse-grained synchronization.

6 Conclusion

The mechanism used for predicting and synchronizing memory dependences has a strong first-order effect on the performance and power consumption of a Speculative Parallelization system. We found that a store-set based memory synchronization mechanism that performs coarse-grained synchronization is a simple and complexity-effective technique to reduce thread squashes in Speculative Parallelization. However, different benchmarks respond differently to synchronization, and a single solution does not work well across the board. Instead, the memory synchronization mechanism needs to find the right balance between speculation and synchronization. We observed that extending the store-set mechanism with a sim-

ple likelihood-of-violation predictor allows a Speculative Parallelization system to dynamically adjust its synchronization strategy based on the behavior of individual loads, directly leading to improvements in performance and power consumption.

Acknowledgments

We are grateful to Sam Stone for the implementation of the store-set based memory dependence predictor, and for valuable feedback at numerous points in the research.

References

[1] Mayank Agarwal, Kshitiz Malik, Kevin M. Woley, Sam S. Stone, and Matthew I. Frank. Exploiting postdominance for speculative parallelization. *Int'l Symp. High Performance Comp. Arch.*, (HPCA-13):295–305, 2007.

[2] Haitham Akkary and Michael A. Driscoll. A dynamic multithreading processor. *Int'l Symp. Microarchitecture*, (MICRO-31):226–236, 1998.

[3] Ahmed S. Al-Zawawi, Vimal K. Reddy, Eric Rotenberg, and Haitham H. Akkary. Transparent control independence (TCI). *Int'l Symp Comp Arch*, (ISCA-34), 2007.

[4] Mary D. Brown, Jared Stark, and Yale N. Patt. Select-free instruction scheduling logic. In *MICRO 34*, 2001.

[5] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *25th International Symposium on Computer Architecture (ISCA-25)*, pages 142–153, June 1998.

[6] Alok Garg, M. Wasiur Rashid, and Michael Huang. Slackened memory dependence enforcement: Combining opportunistic forwarding with decoupled verification. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 142–154, Washington, DC, USA, 2006. IEEE Computer Society.

[7] Rakesh Ghiya, Daniel Lavery, and David Sehr. On the importance of points-to analysis and other memory disambiguation methods for c programs. In *PLDI*, 2001.

[8] Ilhyun Kim and Mikko H. Lipasti. Understanding scheduling replay schemes. *hpca*, 00:198, 2004.

[9] Venkata Krishnan and Josep Torrellas. A chip multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 47(9), September 1999.

[10] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: A TLS compiler that exploits program structure. *Principles and Practice of Parallel Programming*, (PPoPP-11):158–167, 2006.

[11] Kshitiz Malik, Kevin M. Woley, Samuel S. Stone, Mayank Agarwal, Vikram Dhar, and Matthew I. Frank. Confidence based out-of-order renaming for speculatively multithreaded processors. Technical Report UILU-ENG-06-2208, University of Illinois, Urbana-Champaign, June 2006.

[12] P. Marcuello and A. González. A Quantitative Assessment of Thread-level Speculation Techniques. *Int'l. Parallel and Distributed Proc. Symp.*, (IPDPS-14):595–604, 2000.

[13] Pedro Marcuello and Antonio González. Clustered speculative multithreaded processors. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 365–372, 1999.

[14] Pedro Marcuello and Antonio González. Thread-spawning schemes for speculative multithreading. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, Washington, DC, USA, 2002. IEEE Computer Society.

[15] Pedro Marcuello, Antonio González, and Jordi Tubella. Speculative multithreaded processors. *Int'l. Conf. Supercomputing*, (ICS-12):77–84, 1998.

[16] Andreas Ioannis Moshovos. *Memory dependence prediction*. PhD thesis, University of Wisconsin-Madison Computer Sciences Department, 1998.

[17] Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, pages 67–78, January 1991.

[18] Nicholas Riley and Craig Zilles. Probabilistic counter updates for predictor hysteresis and bias. *IEEE Comput. Archit. Lett.*, 5(1), 2006.

[19] Smruti R. Sarangi, Wei Liu, Josep Torrellas, and Yuanyuan Zhou. Reslice: Selective re-execution of long-retired misspeculated instructions using forward slicing. In *MICRO*, pages 257–270, 2005.

[20] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. *Int'l Symp Computer Architecture*, (ISCA-22):414–425, 1995.

[21] Bjarne Steensgaard. Sparse functional stores for imperative programs. In *ACM SIGPLAN Workshop on Intermediate Representations*, pages 62–70, 1995.

[22] J. Gregory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *HPCA 4*, pages 2–13, February 1998.

[23] Sam S. Stone, Kevin M. Woley, Kshitiz Malik, Mayank Agarwal, Vikram Dhar, and Matthew I. Frank. Synchronizing store sets (SSS): Balancing the benefits and risks of inter-thread load speculation. Technical Report CRHC-06-14, University of Illinois, Center for Reliable and High-Performance Computing, December 2006.

[24] T. N. Vijaykumar. *Compiling for the Multiscalar Architecture*. PhD thesis, University of Wisconsin-Madison Computer Sciences Department, January 1998.

[25] Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. Compiler optimization of scalar value communication between speculative threads. *Arch. Support Prog. Lang. Operating Sys.*, (ASPLOS-X):171–183, 2002.