Synchronizing Store Sets (SSS): Balancing the Benefits and Risks of Inter-thread Load Speculation[*]

Sam S. Stone, Kevin M. Woley, Kshitiz Malik,
Mayank Agarwal, Vikram Dhar and Matthew I. Frank
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801

November 17, 2006

Abstract

Speculative parallelization (SP) enables a processor to extract multiple threads from a single sequential thread 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.

Synchronizing predicted inter-thread dependences enables aggressive load speculation while minimizing the risk of misspeculation. In this paper, we present store set synchronization, a complexity-effective technique for speculating on inter-thread data dependences. The store set synchronizer (SSS) predicts store-load dependences using store sets and enforces those predicted dependences using recently proposed techniques for dynamic register synchronization. The key insight behind store set synchronization is that predicted dependences carried through store sets can be treated exactly like the dependences carried through architectural registers.

By balancing the benefits and risks of load speculation, the SSS increases performance, conserves power, and reduces complexity. On integer benchmarks the SSS increases performance by as much as 56% and by 20% on average. The SSS also reduces the average rate of dependence violations by 80%, which conserves power and dramatically decreases the number of threads squashed due to dependence violations. Furthermore, the low rate of dependence violations mitigates the need for costly disambiguation hardware such as per-thread load queues. We show that replacing the associative load queues with filtered load re-execution in an SSS-equipped system decreases performance by just 3%.

1 Introduction

With chip budgets exceeding 1 billion transistors and superscalar pipelines scaling poorly, microprocessor manufacturers have recently focused much attention on multithreaded and multicore processors. The multicore architecture is inherently scalable and achieves dramatic speedups on explicitly parallel workloads, but does not increase the performance of a single, sequential thread. To leverage the substantial resources

[*]University of Illinois at Urbana-Champaign Center for Reliable and High-Performance Computing Technical Report number CRHC-06-14.

of multithreaded and multicore architectures without incurring the significant costs of hand-parallelizing sequential codes, some mechanism for automatically parallelizing sequential threads is needed.

Speculative parallelization (SP) is a technique whereby a processor extracts multiple, possibly dependent threads from a single sequential thread and executes them in parallel. Examples of the speculative parallelization paradigm include the Multiscalar [35], thread-level speculation (TLS) [12, 39, 15], and speculative multithreading [1, 18, 27], among others. By exploiting not only the local instruction-level parallelism but also the global thread-level parallelism within a sequential thread, SP has the potential to dramatically increase a thread's performance.

For speculative parallelization to achieve high performance on integer programs, loads must speculate on the data dependences among threads. Otherwise, ambiguous inter-thread data dependences render many integer codes difficult, if not impossible, to parallelize. Techniques for speculating on inter-thread data dependences have a first-order impact on the performance, power, and complexity of architectures that support speculative parallelization.

Synchronizing predicted inter-thread memory dependences enables aggressive speculative parallelization while minimizing the risks of load misspeculation. In this paper we describe store set synchronization, a complexity-effective technique for speculating on inter-thread data dependences. The store set synchronizer (SSS) predicts store-load dependences using store sets [7] and enforces those predicted dependences using recently proposed techniques for dynamic register synchronization [17]. The key insight behind store set synchronization is that predicted dependences carried through store sets can be treated exactly like the dependences carried through architectural registers.

We evaluate the store set synchronizer in an SP architecture that dynamically synchronizes inter-thread register dependences. Our results show that the SSS increases performance, conserves power, and reduces complexity. On integer benchmarks *the SSS increases performance by as much as 56% and by 20% on average. The SSS also reduces the average rate of dependence violations by 80%*, which conserves power and dramatically decreases the number of threads squashed due to dependence violations. Furthermore, *the low rate of dependence violations mitigates the need for costly disambiguation hardware such as per-thread load queues.* We show that replacing the associative load queues with filtered load re-execution in an SSS-equipped system decreases performance by just 3%.

The remainder of this paper is organized as follows. Section 2 discussed background and related works. Section 3 develops store set synchronization in the context of a superscalar and extends the SSS to an SP

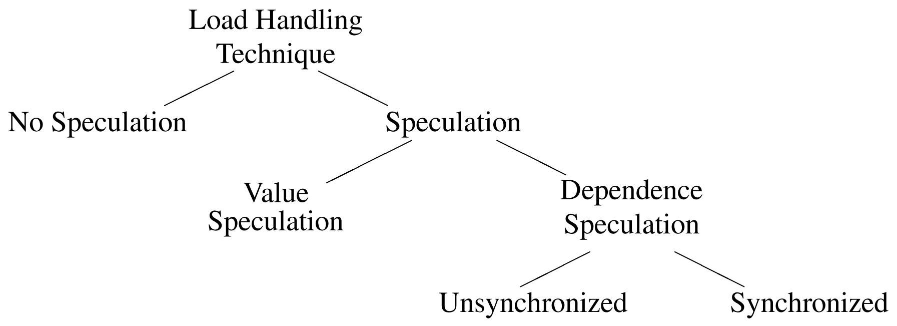

Figure 1: A taxonomy for load speculation. Value speculation encompasses load address prediction and load value prediction. Dependence speculation may be synchronized or unsynchronized. With synchronized dependence speculation, the system speculates that a load depends on some set of unresolved stores (possibly empty) and delays the load only until that set of stores resolves. With unsynchronized dependence speculation, the system either speculates that a load has no dependence on any unresolved store or does not speculate the load at all.

architecture. Sections 4 and 5 present methodology and results, and Section 6 concludes.

2 Background and related work

Figure 1 describes a taxonomy of techniques for load speculation, with principal classes defined as *no speculation*, *value speculation*, *unsynchronized dependence speculation*, and *synchronized dependence speculation*. Each technique strikes a different balance between the benefits of successful speculation and the costs of misspeculation. In this section, we review applications of these techniques that have been proposed for SP systems, with reference to related proposals for ILP systems [1] when appropriate. As this paper focuses on synchronized dependence speculation, we limit our discussion of the other load speculation techniques.

The remainder of this section is organized as follows. Section 2.1 discusses parallelizing compilers that do not permit speculation on inter-thread data dependences. Section 2.2 reviews the mechanisms used to detect dependence violations in systems that permit load speculation. Sections 2.3, 2.4, and 2.5 discuss value speculation, unsynchronized dependence speculation, and synchronized dependence speculation, respectively.

[1]ILP systems: architectures that exploit ILP but do not speculatively parallelize threads, such as superscalars and VLIW's.

2.1 No speculation

Early parallelizing compilers did not permit speculation on inter-thread data dependences. These compilers allow threads to execute in parallel only if the compiler can prove that the threads are not data dependent [26] or explicitly synchronize all suspected dependences [2, 36]. Given the abundance of ambiguous data dependences in integer codes, this technique extracts little SP from integer programs.

Without some form of load speculation, executing parallel threads extracted from an integer program is often difficult if not impossible. Even with extensive compiler analysis, ambiguous data dependences often exist between regions of code that are otherwise independent [9]. By breaking those ambiguous data dependences, load speculation allows a SP system to execute parallel threads that are likely to be independent most of the time, even if they are not provably independent or not always independent. In this manner, load speculation dramatically increases the parallelism exposed to a SP system.

2.2 Detecting dependence violations

Because loads that issue speculatively may violate true data dependences, a system that performs load speculation must detect dependence violations. *Disambiguation* is the process by which a system detects data dependence violations. Before discussing load speculation, we describe the mechanisms proposed for disambiguation in SP systems.

Static disambiguation. Speculation can be performed by the compiler, in software, even on processors with little or no hardware support for memory disambiguation. In [28], the compiler identifies loads that may execute speculatively and inserts code to test for dependence violations and to repair the program state if a violation occurs.

Caches and coherence protocols. In [12, 39, 38, 37], a multiprocessor uses the L1 caches and a snoopy cache coherence protocol to disambiguate inter-thread memory references. Although the details of the proposed schemes differ, the general frameworks are similar. The L1 cache line is extended with a "speculatively loaded" bit that is set when a load from a speculative thread accesses the cache line. When a thread commits a store to its L1 cache, the coherence protocol sends an invalidation to the other caches. If a thread that is later in program order has speculatively loaded the cache line, the system detects a dependence violation. The speculative versioning cache (SVC) implements a similar disambiguation scheme for the Multiscalar [11].

The memory disambiguation table (MDT) of [15] manages inter-thread disambiguation for a directory-based coherence protocol. In [5], bit vectors external to the caches (*signatures*) are used to perform bulk disambiguation, thereby decreasing the complexity of the caches and the coherence protocol.

Load queues. ILP processors that speculate loads typically use a load queue to perform disambiguation [14]. The load queue is an age-ordered, fully-associative buffer containing all in-flight loads. When a store completes, it searches the load queue for a later, completed load that matches the store's address. If such a load is found, a dependence violation has occurred.

Two SP systems based on simultaneous multithreaded (SMT) cores, DMT [1] and IMT [27], use per-context load queues to perform inter-thread disambiguation. In these systems, each hardware context contains a load queue. When a store completes, it not only searches the load queue in its own context to detect intra-thread dependence violations, but also searches the load queues of contexts running later threads to detect inter-thread violations. Skipper [6] and Ginger [13] also use a load queue to perform disambiguation.

The Multiscalar uses an address resolution buffer (ARB) to disambiguate in-flight stores and loads [8, 35]. The ARB is a centralized buffer of address-indexed banks shared by all of the Multiscalar's processing units. Each address-indexed bank contains the age-ordered set of loads and stores that are in-flight and that access the address associated with that bank. In that sense, each ARB bank is essentially a load-store queue. If a load and store to the same address execute out of program order, the ARB detects a dependence violation.

Filtered load re-execution. To reduce the complexity of disambiguation in ILP systems, [4] and [30] proposed filtered load re-execution as an alternative to the fully-associative load queue. With filtered load re-execution, a small percentage of completed loads access the cache just prior to retirement. If the value obtained non-speculatively from the cache does not match the value obtained during speculative execution, then the load has violated a true dependence. Roth also proposed the use of filtered load re-execution in Ginger [13], an architecture that uses out-of-order fetch to exploit control independence.

2.3 Value speculation

With load value speculation, a processor speculatively executes a load and its dependents based on a prediction of the load's address or value. Address prediction breaks the dependence between a load and the instructions that produce the load's address, while value prediction breaks the dependence between the load and the store that produces the load's value. Speculative execution based on load value prediction was originally proposed in [16].

SP systems use load value speculation to break inter-thread dependences, thereby allowing dependent threads to execute in parallel. Although the benefits of successful value speculation are significant, misspeculations are frequent. The clustered speculative multithreaded processor (clustered SM) and its predecessors perform inter-thread load value speculation by predicting the addresses of a thread's load instructions and by predicting the values of a thread's live-out registers, including the live-outs defined by load instructions [18, 22, 19, 21, 20]. Likewise, the dynamic multithreading processor (DMT) speculates on load addresses by predicting the values of a thread's live-in registers, including the live-ins used to compute load addresses [1]. In both clustered SM and DMT, misspeculations are handled by selective re-execution. Finally, Gonzalez [10] and Steffan [38] studied the benefits of load value speculation in TLS systems.

2.4 Unsynchronized dependence speculation

With unsynchronized dependence speculation, the system predicts whether a load has a dependence on any earlier unresolved store, but does not identify the set of stores on which the load may depend. Loads with predicted dependences may not execute speculatively; all other loads may execute speculatively. By speculating only loads that are not likely to have dependences on in-flight stores, unsynchronized dependence speculation reduces the rate of misspeculation, at the expense of exposing less parallelism.

The compiler (perhaps using information from a profiler, or from the user) can make a static decision to speculate across ambiguous memory dependences. This is the approach taken by, for example, Rauchwerger and Padua in the LRPD test for the Polaris parallelizing compiler [28]. When performing such speculation, the compiler adds extra code to check the speculation and to roll back program state after any violations.

ILP processors with dynamic, unsynchronized speculation use a *load wait table* to identify loads that have violated dependences in the past [14, 29, 41]. Loads that hit in the wait table are not permitted to issue speculatively. For SP systems, Steffan proposed that a static load that has violated an inter-thread dependence in the past should not issue until its thread becomes non-speculative [38].

2.5 Synchronized dependence speculation

With synchronized dependence speculation, a *memory dependence predictor* identifies a (possibly empty) set of in-flight stores upon which a load is likely to depend, and the scheduler delays the load until the stores in that set resolve their addresses. That is, the memory dependence predictor predicts store-load dependences, and the scheduler enforces those predicted dependences. The goal of synchronized dependence speculation is to delay a given load just long enough to avoid violating the load's true data dependence. Thus, synchronized dependence speculation balances the benefits of speculation against the costs of misspeculation by speculating on dependences that are predicted to be ambiguous and enforcing dependences that are predicted to be true.

ILP systems. Several memory dependence predictors (MDP's) have been proposed for ILP processors. Moshovos proposed the first MDP, leveraging the observation that past store-load violations are a good predictor of future store-load dependences [23, 25]. Moshovos' MDP predicts that a dynamic load depends on a dynamic store if (1) the corresponding static load and store have violated a data dependence in the past, and (2) the *dependence distance* between the dynamic load and store matches the dependence distance of the earlier violation. Moshovos' MDP includes two large, fully-associative tables: a prediction table (which predicts store-load dependences) and a synchronization table (which synchronizes predicted dependences).

The store set predictor of [7] groups static stores and loads into *store sets*, such that a load's store set contains a particular store if the load has violated a dependence with respect to that store in the past. The store set predictor consists of two direct-mapped tables, the store set identifier table (SSIT) and the last fetched store table (LFST). The SSIT maps the program counters of load and store instructions to their corresponding store set identifiers (SSID's). The LFST maps the SSID of a load (or store) to the instruction number of the most recently fetched store in that store set, thereby allowing the scheduler to create a dependence between the load (or store) and the most recently fetched store in the store set. The scheduler then ensures that no load or store in a given store set issues until all earlier stores in the same store set have resolved their addresses.

The MDP proposed in [41] uses *store vectors* to predict store-load dependences by tracking the relative ages of stores on which a load has depended in the past. Because the load's predicted dependences are easily encoded in an age-ordered bit vector, the store vectors MDP is amenable to matrix scheduling.

In [31, 32], Sha proposes MDP's that are variants of the store set predictor. The chief distinction is that the store set predictor associates each load with an integer SSID that represents a set of stores, while Sha's MDP's associate each load with a finite set of store instructions PC's or store instruction sequence numbers.

Memory renaming is a type of synchronized dependence speculation in which the system speculates that a particular dynamic store produces the value of a particular dynamic load. The system then passes the value directly from the store to the load (or directly from the producer of the store's value to the consumer of the load's value) without accessing the memory system. Several variants of memory renaming have been proposed for ILP systems [43, 24, 33, 32, 40]. We are not aware of any proposals for memory renaming in SP systems.

SP systems. In [23, 25], Moshovos adapts his MDP to support synchronized dependence speculation on the Multiscalar [35]. A centralized implementation achieves average speedups within a few percent of a perfect MDP on integer benchmarks. However, the centralized implementation uses large, fully associative prediction and synchronization tables. A distributed implementation organized in a manner similar to the store set predictor achieves speedups within 10% of a perfect MDP on integer benchmarks. In the distributed implementation, the prediction and synchronization tables are two-way set associative, and each thread must broadcast a message to all other threads whenever it decodes a store instruction. IMT [27] also uses Moshovos' MDP to synchronize inter-thread dependences.

In [18, 21, 20], the clustered speculative multithreaded processor and its predecessors use load/store address prediction to perform synchronized dependence speculation. 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. Skipper [6] and Ginger [13], two systems that fetch instructions out-of-order to exploit control independence, use Moshovos' MDP and a variant of the store set predictor, respectively.

3 Store set synchronization

In this section we describe *store set synchronization*, a complexity-effective technique for speculating on inter-thread data dependences. The store set synchronizer (SSS) predicts store-load dependences using store sets [7] and enforces those predicted dependences using recently proposed techniques for dynamic register synchronization [17]. The key insight behind store set synchronization is that predicted dependences carried through store sets can be treated exactly like the dependences carried through architectural registers. The SSS increases the performance of an SP system by enabling aggressive load speculation, while conserving power by reducing costly violations. Furthermore, the store set synchronizer's low violation rate mitigates the need for costly disambiguation hardware (such as per-thread load queues) and recovery techniques (such

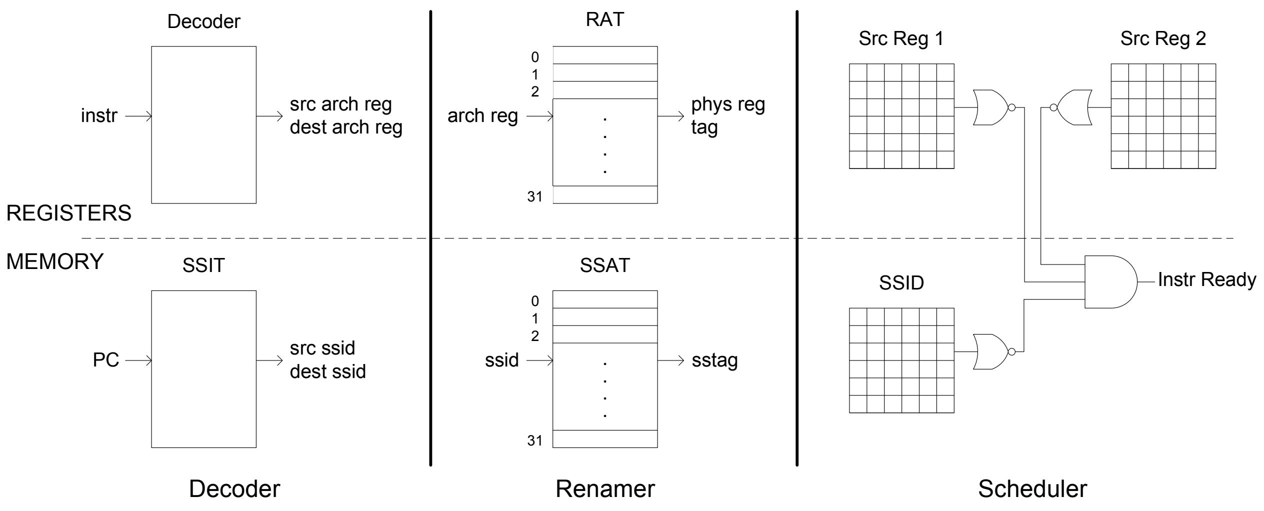

Figure 2: Register and store set synchronization in a superscalar.

as selective re-execution).

The remainder of this section is organized as follows. Section 3.1 introduces store set synchronization in the context of a superscalar. Section 3.2 extends the SSS to an SP system and demonstrates synchronization of inter-thread memory dependences by way of an example.

3.1 Store set synchronization in a superscalar

Store set synchronization enables a superscalar to enforce register dependences and predicted memory dependences using identical structures and techniques. The store set synchronizer is modeled after the store set predictor [7], but uses conventional renaming and scheduling techniques to enforce the predicted dependences. As Figure 2 shows, enforcement of predicted memory dependences is analogous to enforcement of register dependences.

First consider register dependences. In the decoder, the processor learns an instruction's architectural source and destination registers. To eliminate false dependences, the processor renames the instruction's architectural registers. In the renamer, the register alias table (RAT) maps the architectural source registers to the corresponding physical register tags. The instruction obtains a physical register tag from the free list and updates the RAT with the destination register's new architectural-to-physical register mapping. Finally, the instruction dispatches to the scheduler. In Figure 2 the superscalar uses a matrix scheduler to enforce the true register dependences [3].

Enforcement of predicted memory dependences follows naturally. In the decoder, each load or store

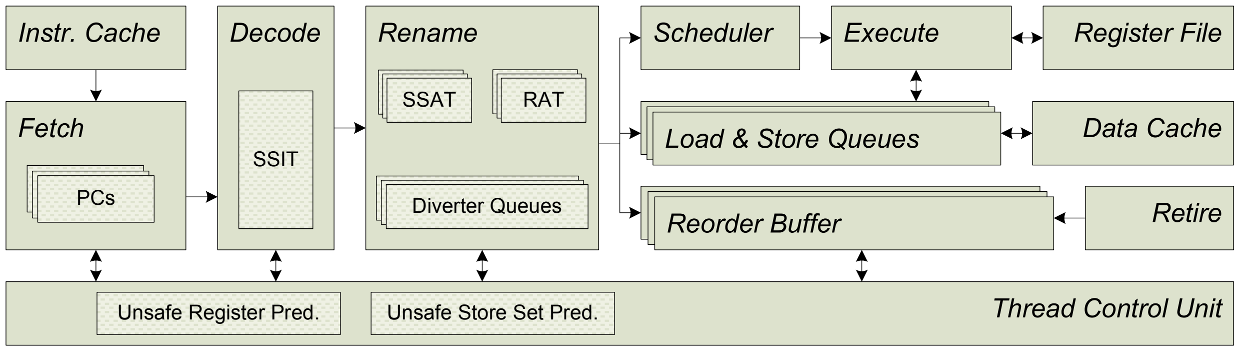

Figure 3: An overview of a SyncMT pipeline. The SyncMT extends an underlying SMT core to support speculative parallelization. The thread control unit monitors the instruction stream and, when possible, spawns speculative threads. The register synchronization logic includes the register alias tables (RAT's), diverter queues, and unsafe register predictor. The store set synchronizer includes the store set identifier table (SSIT), store set alias tables (SSAT's), diverter queues, and unsafe store set predictor.

instruction accesses the store set identifier table (SSIT) and obtains the identifier (SSID) of the store set that contains the instruction, if any. The SSS regards each store as both a consumer and a producer of the dependence carried through its SSID, while each load is simply a consumer. This policy ensures that a load with a valid SSID does not issue until all earlier stores in its store set have issued, at the expense of issuing stores in the same store set in order[2].

In the renamer, the store set alias table (SSAT) maps each load or store instruction's SSID to the corresponding store set dependence tag (SSTAG). Each store also obtains a dependence tag from the free list and updates the SSAT with the new mapping from SSID to SSTAG. Finally, the load or store dispatches into the scheduler, which uses an additional matrix to enforce the predicted memory dependences.

3.2 Store set synchronization in an SP system

While synchronization of inter-thread register and memory dependences is a general technique that may be applied to any SP system, we discuss synchronization in the context of the synchronized multithreading processor (SyncMT) of Figure 3. The SyncMT consists of a simultaneous multithreaded (SMT) core [42] with support for speculative parallelization. The SyncMT handles inter-thread register and memory dependences by predicting and synchronizing them. In the following paragraphs, we briefly describe the SyncMT's architecture.

The SyncMT's underlying SMT core contains multiple hardware contexts that are capable of executing

[2]As discussed in [7], a load should not issue until all earlier stores in its store set have issued because the load is likely to depend on some earlier store in its store set, but the precise identity of the matching store is not predicted.

independent threads in parallel. Each hardware context has its own program counter, RAT, SSAT, and reorder buffer. Threads dynamically share other pipeline resources, including the scheduler, caches, and physical registers.

To enable speculative multithreading with synchronization of inter-thread dependences, the SyncMT augments this SMT core with a thread control unit (TCU), a register synchronizer, and a store set synchronizer. They synchronizers are described below. The thread control unit manages the speculative threads. As instructions are fetched, the TCU identifies control-independent points at which speculative threads may be created (*spawned*). If a candidate thread seems profitable, the TCU spawns the thread on one of the processor's idle contexts.

3.2.1 Dependence synchronization

In [17], Malik proposes a novel technique for synchronizing register dependences in an SP system. Referring to Figure 3, the register synchronization components include the SyncMT's per-context RAT's, per-context diverter queues, shared scheduler, and unsafe register predictor. These components are described below.

When the system spawns a new thread, it copies the RAT from the spawner's context to the spawnee's context. Thus, the spawnee's RAT initially contains the correct architectural-to-physical register mappings for all architectural registers that are not written by the unfetched instructions between the *spawn point* (the instruction that initiated the spawn) and the *reconnection point* (the first instruction in the spawnee). Such architectural registers are referred to as *safe*. Likewise, any architectural register that is written by an instruction in the spawnee becomes safe after that local definition. Clearly, the spawnee can rename and dispatch any instruction that reads only safe source registers without violating any inter-thread register dependences.

How does a thread differentiate between those registers that are initially safe and those that are initially unsafe? When the system spawns a thread, the *unsafe register predictor* supplies a bit vector that indicates which architectural registers are predicted to be unsafe. The set of registers predicted to be unsafe is simply the set of registers that have ever been written between the spawn point and the reconnection point.

The spawnee does not rename or dispatch any instruction that reads a register predicted to be unsafe, because that instruction is likely to obtain an incorrect architectural-to-physical register mapping from the RAT. To avoid violating an inter-thread register dependence, the renamer places any instruction that depends directly on an unsafe register in the context's *diverter queue*. Likewise, the renamer places any instruction

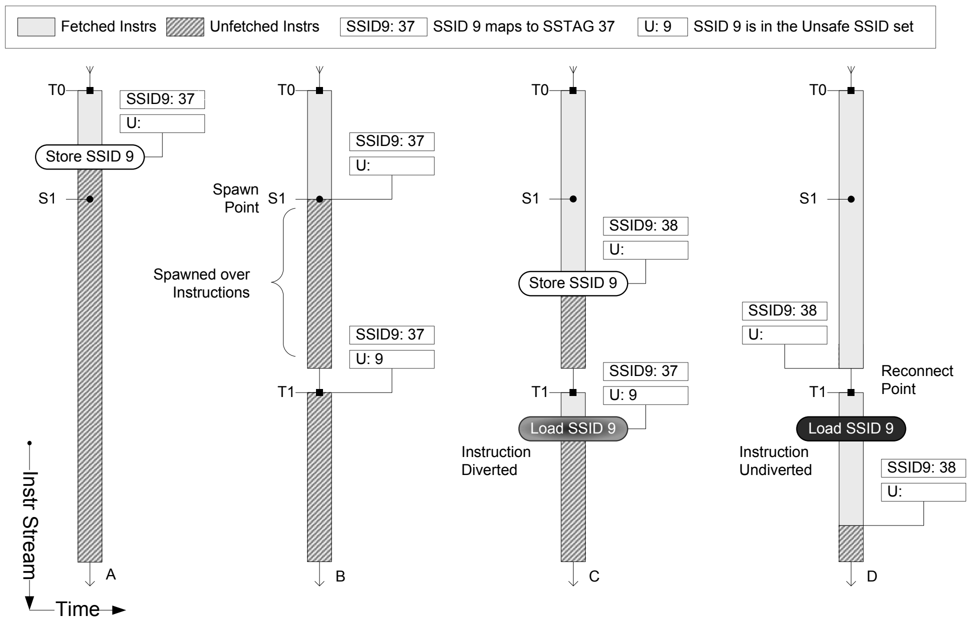

Figure 4: An example of synchronizing inter-thread memory dependences.

that depends on a diverted instruction in the queue. These transitively diverted instructions can actually obtain correct architectural-to-physical register mappings from the RAT, but as these instructions are known to depend on diverted instructions, dispatching them would only waste valuable slots in the scheduler.

When the spawner renames the instruction preceding the reconnection point, the spawner and spawnee *reconnect*. During reconnection, the spawnee obtains the correct architectural-to-physical mappings for registers that were predicted to be unsafe, then renames and dispatches the instructions in the diverter queue. Furthermore, if the spawnee inadvertently read an unsafe register, this inter-thread dependence violation is discovered during reconnection, and the misspeculated instructions are canceled. Finally, at reconnection the SyncMT transmits the reconnection information not only to the successor thread but to any threads to which the successor thread has already reconnected; we refer to this transmission as *limited broadcast*. For further details on the bitset manipulations used to divert and undivert instructions and to detect inter-thread register dependence violations, refer to [17].

3.2.2 An example

Because inter-thread register synchronization and inter-thread store set synchronization are analogous, we forgo a formal discussion of store set synchronization and instead offer an example. Figure 4 depicts successive snapshots of execution on a SyncMT processor. Each frame represents the active threads as instruction streams, with fetched 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 in the unfetched portion of its instruction stream. The thread has just renamed a store with SSID 9. Thus, in the thread's store set alias table (SSAT), store set identifier (SSID) 9 is mapped to store set tag (SSTAG) 37. No SSID's in T0 are marked unsafe.

In Frame B, thread 0 spawns thread 1. T1 copies T0's SSAT, thereby obtaining the mapping from SSID 9 to SSTAG 37. T1 also consults the unsafe store set predictor; the predicted unsafe set includes SSID 9. Thus, the store set synchronizer predicts that the unfetched portion of thread 0 contains a store in store set 9.

In Frame C, thread 0 fetches a store in store set 9, while thread 1 fetches a load in store set 9. When T0 renames the store, it obtains a free store set tag (38) and updates the SSAT so that SSID 9 is mapped to SSTAG 38. When T1 attempts to rename the load, it discovers that the load's SSID is unsafe and places the load in the diverter. Subsequent instructions that depend on the load are also placed in the diverter.

Finally, in Frame D, thread 0 renames its last instruction and reaches the reconnection point. During reconnection, thread 1 obtains the correct mapping for SSID 9, marks SSID 9 as safe, and undiverts the load that was diverted in Frame C, along with its transitive dependents. The load accesses the SSAT, obtains the correct mapping for SSID 9, and dispatches. Assuming that the second store in thread 0 and the load in thread 1 actually access the same address, the store set synchronizer has successfully prevented the load from violating an inter-thread memory dependence.

Note that diversion is not required to synchronize all inter-thread dependences. For example, if the store between the spawn point and the reconnection point in Figure 4 were not in store set 9, then the unsafe SSID predictor would probably not predict SSID 9 to be unsafe. In that case, thread 1 would not divert the load with SSID 9. Instead, the load would access T1's SSAT, obtain the (correct) mapping from SSID 9 to SSTAG 37, and dispatch into the scheduler. The scheduler would then enforce the predicted inter-thread dependence between the first store in T0 and the load in T1.

3.2.3 Disambiguation

Enforcement of inter-thread memory dependences is precisely analogous to enforcement of inter-thread register dependences, with two exceptions. First, the unsafe store set predictor predicts an SSID to be unsafe if it was written between the spawn point and the reconnection point the last time those instructions retired.

Second, whereas the register synchronizer uses bitsets in the spawner and spawnee to detect violations of inter-thread register dependences, the store set synchronizer uses per-context load queues to detect violations of inter-thread memory dependences. The store sets are, after all, only an approximation of the true memory dependences that are actually carried through addresses. Thus, some form of memory disambiguation is necessary, and the SyncMT processor of Figure 3 uses the load queues for that purpose.

Even per-context load queues are not sufficient to detect all inter-thread memory dependence violations, because the SyncMT permits inter-thread store-load forwarding from its store queues. Thus, when a load in one thread obtains its value from a store in an earlier thread, and that forwarding store is subsequently canceled by a branch misprediction, the SyncMT will not necessarily squash the dependent load's thread.

Clearly, the SyncMT needs some mechanism to detect dependence violations caused by inter-thread forwarding from wrong-path stores to right-path loads. The processor uses a variant of the store vulnerability window (SVW) for this purpose [30]. SVW is a form of filtered load re-execution; see Section 2 for an overview of this disambiguation technique.

The SyncMT uses the following algorithm to identify retiring loads that should re-execute. The decode stage assigns a sequence number to each store. At any time, the sequence numbers represent a total ordering on all stores in all threads in the processor. All stores and loads access a direct-mapped, address-indexed table (the filter) as they retire. Each store writes its sequence number in the corresponding entry. If a retiring load obtained its value from the store queue, it compares the sequence number of the forwarding store to the sequence number in the corresponding filter entry. If the sequence numbers do not match, the load re-executes. We refer to this algorithm as limited SVW (L-SVW), and it is sufficient to detect all cases of inter-thread forwarding from wrong-path stores to right-path loads.

In the Section 5, we evaluate store set synchronization on a SyncMT that uses per-context load queues and L-SVW to disambiguate memory dependences. We also evaluate the SSS on a SyncMT that has no load queues. Eliminating the load queues altogether changes the filtering algorithm slightly. As described in [30], each load obtains a store sequence number (SSN) when it executes. If a load does not get its value from the store queues, then the load's SSN is set to the SSN of the most recently retired store. Each retiring load *that did not obtain its value from the store queues* compares its SSN to the SSN in the corresponding filter entry. If the load's SSN is less than the SSN in the filter entry, the load must re-execute. However, as described above, loads that actually obtained their values from the store queues face a more stringent test: if the load's SSN is not equal to the SSN in the filter entry, the load must-re-execute. We refer to this algorithm

Contexts	8
Pipeline width	8 instructions
Functional units	8 identical, fully pipelined
ROB size	512 instructions
Scheduler size	128 instructions
Diverter size	128 instructions
Branch predictor	16 kbit gshare with 8 bits of global history
I Cache	8KB, 2-way, 10-cycle miss latency
D Cache	16KB, 4-way, 10-cycle miss latency
L2 Cache	512KB, 8-way, 100-cycle miss latency
SSIT	Shared, 16K entries, direct-mapped
SSAT	1 per context, 32 entries, direct-mapped, not checkpointed

Figure 5: SyncMT system resources.

benchmark	Input
bzip2	lgred.source, 128KB
crafty	lgred.in
gap	lgred.in
gcc	mdred.rtlanal.i
gzip	lgred.log, 1MB
mcf	mdred.in
parser	2.1.dict, mdred.in
perlbmk	mdred.makerand.pl
twolf	mdred
vortex	mdred.raw
vpr route	lgred.net, small.arch
vpr place	lgred.net, small.arch, lgred.place

Figure 6: SPECINT2K benchmarks and inputs.

as extended SVW (E-SVW), and it is sufficient to detect all memory dependence violations.

4 Methodology

We use execution-driven, cycle-level simulation of the SyncMT architecture to evaluate the store set synchronizer. For an overview of the SyncMT architecture, refer to Section 3.2. The SyncMT simulator executes a variant of the 64-bit MIPS ISA that has no special instructions to support speculative multithreading. Inter-thread dependences are predicted and enforced using the register and store set synchronizers described in the previous section, The simulator faithfully models all effects of mispredictions and dependence violations, including execution of misspeculated instructions in the simulator's back end. After a misprediction or dependence violation resolves, the simulator immediately reclaims the resources allocated to canceled instructions (e.g. rob slots, physical registers) and repairs the states of the affected RAT's.

We model an 8-wide pipeline with 8 hardware contexts. Each hardware context has its own diverter and reorder buffer (ROB). Like the ROB of the POWER5 [34], the SyncMT's ROB and diverter are dynamically shared among contexts. We model zero latency for undiversion; when threads reconnect, the undiverted instructions are immediately renamed and inserted into the scheduler. The load and store queues are fixed-sized structures that are large enough to never cause pipeline stalls, and the scheduler is compacting. We model 32 SSID's and sufficient SSTAG's to avoid stalls. Other pertinent pipeline parameters are listed in Figure 5.

With regards to the register and store set synchronizers, the register synchronizer predicts that a register is unsafe if that register has ever been written between the specified spawn and reconnection points. The store set synchronizer predicts that an SSID is unsafe if that register was written between the spawn and reconnection points the last time those points retired, as described in the previous section. We do not model conflict or capacity misses in the unsafe register and unsafe store set predictors, but we do model cold misses. We do not model the latency of training the unsafe register predictor (this policy is conceptually similar to not modeling the latency of training a branch predictor).

With regards to the thread control unit, the spawn points are obtained from a control-independence analysis of program traces and are loaded into a spawn hint cache at the beginning of execution. We do not model misses in the hint cache. Also, the TCU uses information obtained from a dynamic trace to avoid spawning over fewer than 5 instructions or more than 128 instructions.

We evaluate the store set synchronizer using the SPECINT2K benchmarks with the inputs listed in Figure 6; we lack runtime library support for eon. The SPECINT2K benchmarks are difficult to parallelize because potential threads exhibit many inter-thread register and data dependences. Thus, these benchmarks should stress the store set synchronizer. We compile the benchmarks with gcc -O3 and link them with our own libraries, which are largely derived from FreeBSD.

5 Results

In this section we evaluate the performance of the store set synchronizer in the SyncMT simulator. We compare the performance of the SSS to a scheme that synchronizes only intra-thread memory dependences, and to an oracle synchronizer that delays the issue of any load just until the store that produces its value has issued. We evaluate the three synchronizers both in a SyncMT system that has per-context load queues and in a SyncMT that uses only filtered load re-execution to detect dependence violations. With regards to filtered load re-execution, we do not model contention for data cache ports between executing loads and re-executing loads, and stores commit to the cache-memory hierarchy as soon as they retire. In the figures, IPC is normalized to the IPC of a superscalar with the same resources as the SyncMT; the IPC's achieved

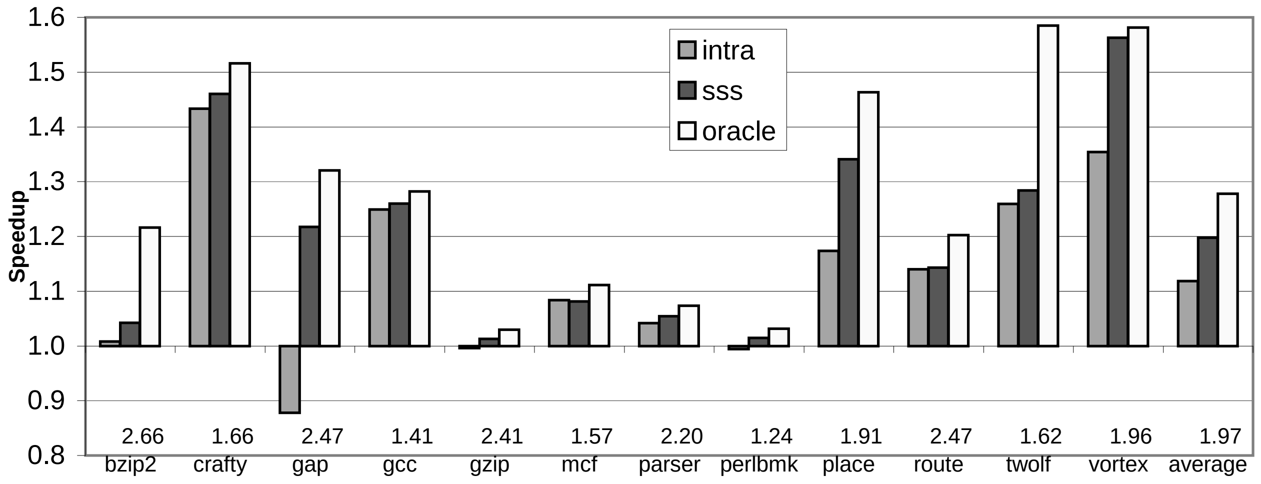

Figure 7: SyncMT performance with per-context load queues. Results are normalized to the performance of a superscalar with the same resources as the SyncMT. The left bar (intra) represents SyncMT's performance when only intra-thread dependences are synchronized. This configuration achieves an average speedup of 12% over the superscalar, which demonstrates the SyncMT architecture effectively extracts thread-level parallelism from the benchmarks. The middle bar (sss) represents SyncMT's performance when the SSS synchronizes intra- and inter-thread store-load dependences. This configuration obtains an additional speedup of 8% over intra, which demonstrates that the SSS effectively identifies and enforces inter-thread dependences. The right bar (oracle) represents SyncMT's performance with an oracle synchronizer.

by the superscalar are listed in Figure 7.

Figure 7 shows that in a SyncMT with per-context load queues, store set synchronization achieves an average speedup of 20% over the superscalar, while intra-thread synchronization achieves only a 12% speedup. Furthermore, the average speedup of SSS is within 8% of the average speedup obtained by the oracle. This result compares favorably with those reported in [25], in which Moshovos' distributed, 2-way associative dependence predictor achieves performance within 11% of an oracle.

On three benchmarks (gap, place, and vortex), the SSS improves performance by at least 15% compared to the intra-thread synchronizer; Figure 8 shows that these benchmarks exhibit high rates of inter-thread dependence violations. By contrast, the SSS and the intra-thread synchronizer both achieve performance within a few percent of the oracle on gcc, gzip, mcf, parser, and perlbmk. None of these benchmarks experience high dependence violation rates, and the latter four are clearly constrained by factors other than memory dependence violations. On the remaining benchmarks (bzip2, crafty, route, and twolf), the SSS neither achieves significant speedup over the intra-thread synchronizer nor approaches the performance of the oracle. In all five cases, the SSS is most likely diverting loads unnecessarily. Adding path-sensitivity to the unsafe store set predictor (by hashing the global history with the spawner's PC) might enable the SSS to

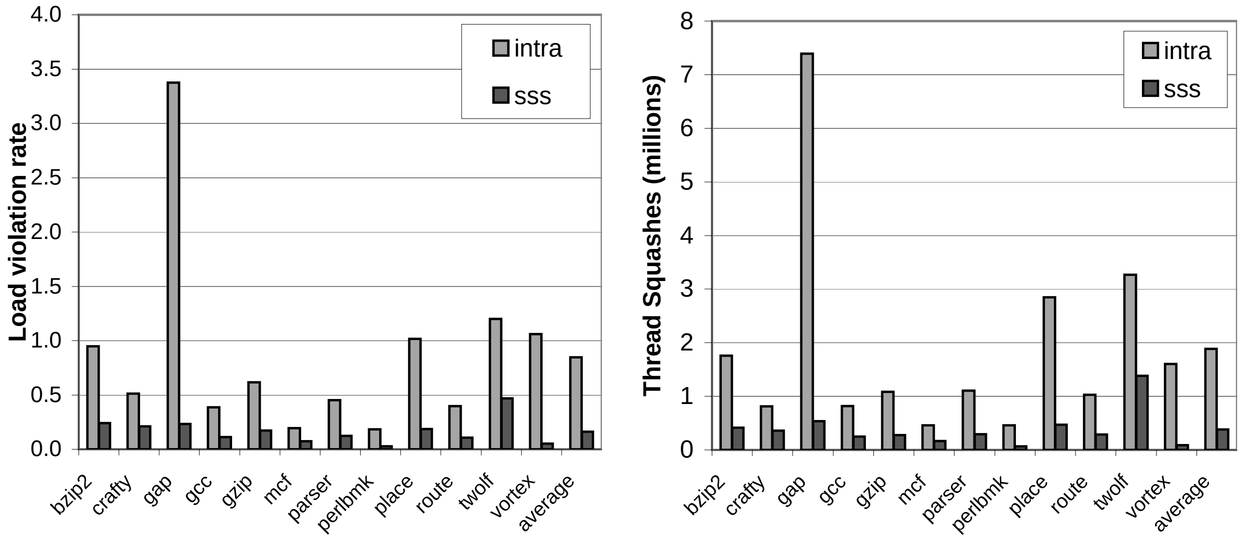

Figure 8: Load misspeculation rates. When inter-thread store-load dependences are not synchronized, load speculation incurs 8.5 inter-thread dependence violations for every 1000 retired loads. When inter-thread dependences are synchronized, the average dependence violation rate decreases to 1.6 violations per 1000 retired loads, a reduction of 80%. The decrease in inter-thread violations translates directly to an 80% reduction in the number of threads squashed due to inter-thread violations.

divert fewer loads unnecessarily.

Figure 9 shows that in a SyncMT that uses filtered load re-execution to detect dependence violations, the SSS achieves an average speedup of 17% over the superscalar, while intra-thread synchronization and the oracle achieve speedups of 5% and 27%, respectively. The performance of the SSS-equipped SyncMT decreases by only 3% when the load queues are removed. Comparing the complexity and power consumption of the load queues to the complexity and power consumption of filtered load re-execution, this trade-off seems quite favorable.

Why does the performance of intra-thread synchronization suffer more than the performance of the SSS when the load queues are removed? As Figure 8 shows, synchronizing only intra-thread dependences yields an average inter-thread dependence violation rate of 8.5 violations per 1000 retired loads, while the SSS suffers only 1.6 violations per 1000 retired loads - a reduction of roughly 80%. Furthermore, the number of threads squashed due to inter-thread dependence violations is strongly correlated with the inter-thread dependence violation rate. The SSS reduces the average number of squashed threads from 1.9M to 0.4M - again, a reduction of nearly 80%. By synchronizing inter-thread dependences, the SSS not only increases performance, but also conserves the power associated with costly misspeculations.

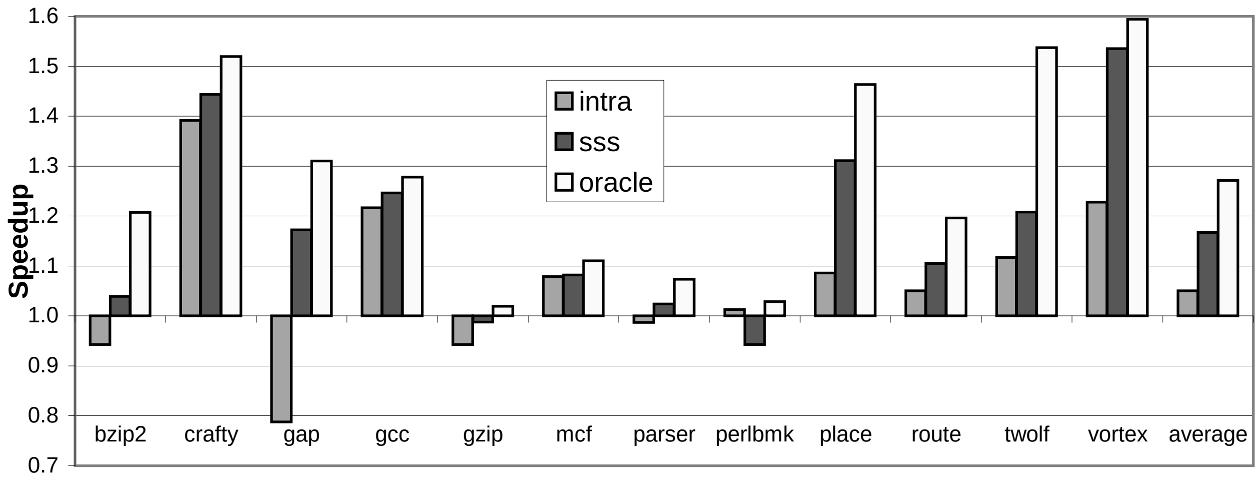

Figure 9: SyncMT performance with filtered load re-execution. The baseline and the SyncMT configurations are identical to those in Figure 7, with the exception that the SyncMT now uses filtered load re-execution rather than per-context load queues. While sss maintains a speedup of 17% over the superscalar (only 3% lower than the speedup achieved with load queues), intra's speedup decreases from 12% to 5%. In short, when inter-thread dependences are synchronized, the per-context load queues are not necessary.

6 Conclusion

By extracting dependent threads from a sequential thread and executing those threads in parallel, speculative parallelization leverages the resources of multithreaded and multicore processors to increase performance. 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 architectures that support speculative parallelization.

Out contributions include the following: (1) Store set synchronization enables aggressive speculative parallelization while minimizing the risks of load misspeculation. The SSS achieves speedups within 10% of an oracle, while decreasing by 80% the rates of dependence violations and thread squashes caused by inter-thread dependence violations. (2) The SSS mitigates the need for costly disambiguation hardware such as per-thread load queues. In particular, a synchronized multithreading processor equipped with SSS loses just 3% performance on average by replacing the load queues with filtered load re-execution.

Acknowledgments

The work reported in this paper was supported in part by the National Science Foundation under grant CCR-0429711 and by the MARCO Gigascale Systems Research Center. Computational resources were supported by an equipment donation from AMD Corp. and the National Science Foundation under grant EIA-0224453, and through the contribution of the use of a computing cluster by the Trusted ILLIAC Center at the Information Trust Institute and Coordinated Science Laboratory at the University of Illinois. This material is based upon work supported under a National Science Foundation Graduate Research Fellowship (Sam Stone). Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Haitham Akkary and Michael A. Driscoll. A dynamic multithreading processor. In *31st Int'l Symp. Microarchitecture*, pages 226–236, November 1998.

[2] Micah Beck, Richard Johnson, and Keshav Pingali. From control flow to dataflow. *J. Parallel and Distributed Computing*, 12:118–129, 1991.

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

[4] Harold W. Cain and Mikko H. Lipasti. Memory ordering: A value-based approach. In *ISCA-31*, pages 90–101, 2004.

[5] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[6] Chen-Yong Cher and T. N. Vijaykumar. Skipper: a microarchitecture for exploiting control-flow independence. In *MICRO 34*, pages 4–15, 2001.

[7] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *ISCA-25*, pages 142–153, June 1998.

[8] Manoj Franklin and Gurindar S. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5):552–571, 1996.

[9] 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.

[10] José González and Antonio González. Limits of instruction level parallelism with data value speculation. In *VECPAR '98: Selected Papers and Invited Talks from the Third International Conference on Vector and Parallel Processing*, pages 452–465, London, UK, 1999. Springer-Verlag.

[11] Sridhar Gopal, T. N. Vijaykumar, James E. Smith, and Gurindar S. Sohi. Speculative versioning cache. In *HPCA*, pages 195–205, 1998.

[12] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS VIII*, pages 58–69, October 1998.

[13] Andrew Hilton and Amir Roth. Ginger: Control independence using parallel tag rewriting. Technical Report TR-CIS-06-16, University of Pennsylvania, November 2006.

[14] R. E. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

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

[16] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 138–147, New York, NY, USA, 1996. ACM Press.

[17] 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.

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

[19] Pedro Marcuello and Antonio González. Exploiting speculative thread-level parallelism on a smt processor. In *HPCN Europe '99: Proceedings of the 7th International Conference on High-Performance Computing and Networking*, pages 754–763, London, UK, 1999. Springer-Verlag.

[20] Pedro Marcuello and Antonio González. A quantitative assessment of thread-level speculation techniques. In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, page 595, Washington, DC, USA, 2000. IEEE Computer Society.

[21] Pedro Marcuello, Antonio González, and Jordi Tubella. Speculative multithreaded processors. In *Int'l Conf. Supercomputing*, pages 77–84, 1998.

[22] Pedro Marcuello, Jordi Tubella, and Antonio González. Value prediction for speculative multithreaded architectures. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 230–236, Washington, DC, USA, 1999. IEEE Computer Society.

[23] Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. In *ISCA-24*, pages 181–193, 1997.

[24] Andreas Moshovos and Gurindar S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *MICRO-30*, 1997.

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

[26] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[27] Il Park, Chong Liang Ooi, and T. N. Vijaykumar. Reducing design complexity of the load/store queue. In *MICRO 36*, pages 411–422, 2003.

[28] Lawrence Rauchwerger and David Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 218–232, La Jolla, CA, June 1995.

[29] Glenn Reinman and Brad Calder. Predictive techniques for aggressive load speculation. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 127–137, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[30] Amir Roth. Store vulnerability window (SVW): Re-execution filtering for enhanced load optimization. In *ISCA 32*, pages 458–468, 2005.

[31] Tingting Sha, Milo M.K. Martin, and Amir Roth. Scalable store-load forwarding via store queue index prediction. In *MICRO-38*, pages 159–170, 2005.

[32] Tingting Sha, Milo M.K. Martin, and Amir Roth. Nosq: Store-load communication without a store queue. In *MICRO*, 2006.

[33] Tingting Sha, Milo M.K. Martin, and Amir Roth. Nosq: Store-load communication without a store queue. In *MICRO 39: Proceedings of the 39th annual ACM/IEEE international symposium on Microarchitecture*, 2006.

[34] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5), 2005.

[35] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA 22*, pages 414–425, June 1995.

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

[37] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *ISCA-27*, pages 1–24, 2000.

[38] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. Improving value communication for thread-level speculation. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 65, Washington, DC, USA, 2002. IEEE Computer Society.

[39] 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.

[40] Samantika Subramaniam and Gabriel H. Loh. Fire-and-forget: Load/store scheduling with no store queue at all. In *MICRO*, 2006.

[41] Samantika Subramaniam and Gabriel H. Loh. Store vectors for memory dependence prediction and scheduling. In *HPCA*, pages 64–75, 2006.

[42] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *ISCA*, pages 191–202, 1996.

[43] Gary S. Tyson and Todd M. Austin. Improving the accuracy and performance of memory communication through renaming. In *MICRO-30*, 1997.