SCIN-Cache: Fast Speculative Versioning in Multithreaded Cores

ANURAG NEGI and RUBEN TITOS-GIL, Chalmers University of Technology

This article describes cache designs for efficiently supporting speculative techniques like transactional memory on chip multiprocessors with multithreaded cores. On-demand allocation and prompt freeing of speculative cache space in the design reduces the burden on nonspeculative execution. Quick access to both clean and speculative versions of data for multiple contexts provides flexibility and greater design freedom to HTM architects. Performance analysis shows the designs stand up well against other HTM design proposals, with potential performance gains in high contention applications with small transactions.

Categories and Subject Descriptors: B.3.0 [Memory Structures]: General; C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors)

General Terms: Design, Performance

Additional Key Words and Phrases: Cache design, speculation support, hardware transactional memory

ACM Reference Format:

Negi, A. and Titos-Gil, R. 2013. SCIN-Cache: Fast speculative versioning in multithreaded cores. *ACM Trans. Architec. Code Optim.* 9, 4, Article 58 (January 2013), 26 pages.

DOI = 10.1145/2400682.2400717 <http://doi.acm.org/10.1145/2400682.2400717>

1. INTRODUCTION

Speculative execution has proven an effective means of extracting more performance from codes where some form of parallelism—instruction level (ILP), memory level (MLP), or thread level (TLP)—is likely but occasionally gets hindered by dependencies. This concept has formed the basis of numerous microarchitectural advancements that have attempted to increase both single-core and multi-core performance, ranging from branch prediction to hardware support for transactional memory (HTM). At the core of these techniques lies the idea of isolating speculative modifications until such time as the speculation is verified, at which point this isolation may be released or the modifications discarded. Most designs use structures closely coupled to the processor pipeline, like the reorder buffer (ROB), load-store queue (LSQ) [Park et al. 2003], and store buffers, for isolation which are usually large enough for shallow speculation scenarios like branch prediction. Designs like the kilo-instruction processor [Cristal et al. 2005] and Rock [Chaudhry et al. 2009] have extended the use of such structures to enhance run-ahead execution and provide rudimentary HTM functionality. However, it has proven difficult to scale these up as doing so directly impacts the speed at which the pipeline can function. Integrating ad hoc structures like speculation buffers is inefficient in terms of area, power, and design complexity. Furthermore, resource utilization is low, being limited to periods when the need for speculation exists.

This work has been supported by the Swedish Foundation for Strategic Research under grant RIT10-0033 as well as the HiPEAC Network of Excellence under contract EU FP7/ICT 287759.

Authors' addresses: A. Negi (corresponding author), R. Titos-Gil, Department of Computer Science and Engineering, Chalmers University of Technology, SE-41296 Gothenburg, Sweden; email: negi@chalmers.se. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 1544-3566/2013/01-ART58 \$15.00

DOI 10.1145/2400682.2400717 <http://doi.acm.org/10.1145/2400682.2400717>

This article attempts to alleviate bottlenecks that restrict the effectiveness of speculative data management at the level of private data caches. Our contributions in this work would be better appreciated if one recognizes the importance of multithreading (fine-grained or simultaneous) as a technique to exploit memory-level parallelism. Many recent high-performance chip multiprocessor (CMP) designs have adopted this approach, which allows many threads of execution to share core resources (functional units, instruction and data caches, etc.). The wide performance gap between the core and the memory system results in the need for a large number of independently schedulable instructions to keep the pipeline busy in the event of long latency memory operations. Since ILP in single threads of execution is restricted, only a small fraction of memory access latency can be hidden on more traditional cores. Moreover, this gain in performance comes at a considerable cost in terms of area and power. Therefore, it becomes prudent to support multiple threads on one core to increase utilization and throughput while remaining within a reasonable power budget. This has led to the introduction of designs like Niagara and AMD Bulldozer that trade off single-threaded performance to varying extents for increased throughput. However, the desire for single-application performance scaling remains strong as ever, resulting in developments in programming models that allow easy construction of parallel codes while maintaining good performance scalability. Transactional Memory (TM) [Herlihy and Moss 1993], which forms the basis for investigating performance in this article, is one such technique that exploits optimistic concurrency to improve performance.

Transactional Memory is an attempt to attain two key goals: increased programmer productivity and improved performance. A multithreaded application splits up a problem among several threads that work concurrently, thereby solving it faster. All threads access one address space, allowing quick inter-thread communication through updates to shared memory locations. However, safe shared memory communication must avoid data races between concurrent threads. With traditional lock-based approaches, designing safe, bug-free algorithms is a complex task. However, TM allows programmers to write safe code much more easily [Rossbach et al. 2010] by delegating, to an underlying TM system, the responsibility of achieving properties of atomicity and isolation (and thereby, mutual exclusion), when executing special sections of code called transactions. The TM system tracks accesses by concurrent transactions and manages these execution flows such that a clear global order exists among transactions that manage to complete successfully. Though TM systems have been proposed in both software (STM) and hardware (HTM), the large performance gap between the two approaches favors supporting the common case of bounded transactions (both in duration and space) in hardware.

The discussion above brings us to the focus of this article: efficient support for HTM on CMPs with multithreaded cores. When a transaction updates a memory location, it does so speculatively, i.e. the change is only made visible to other threads if the transaction commits successfully. The transaction may abort, i.e. discard any speculative changes it has made, due to conflicts with other concurrent transactions or resource limitations. Thus, when multiple threads (or contexts) running on a core speculate, each may generate speculative updates which must be managed efficiently by the HTM design. Several HTM proposals exist for single-threaded cores, by confining speculative values in private caches and leveraging higher levels of the memory hierarchy [Hammond et al. 2004] or thread-local virtual memory buffers [Yen et al. 2007] to buffer old values in case speculative values must be discarded on aborts. However, on a multithreaded core the cache private to the core is shared among multiple, possibly speculative contexts running on the core. This poses the problem of devising an effective means of storing both speculative and clean versions for several execution contexts within one data cache, extracting the correct version from it at speeds which do not

hinder ordinary processor execution and performing fast context commits, squashes, and conflict detection (among both local and remote contexts). The contributions of this article lie here. We present two cache designs which are able to seamlessly support variable number of speculative contexts. The design allocates resources to these contexts on demand. The first cache design allows management of both speculative and clean data for multiple contexts in distinct ways of the same set-associative cache, without any performance penalties when providing data to the core. With decoupled metadata, single-cycle gang operations allow speculative state to be committed or discarded. The second design using additional metadata allows multiple speculative versions of the same cache line to exist in cache, allowing two conflicting local speculative contexts to run lazily (i.e., in a committer wins mode). Lazy resolution of conflicts has been typically associated with better exploitation of available parallelism.

The article also contributes with a study of the related design space, measuring application performance on several configurations differing in the number of contexts supporting locally at each core. The article then analyzes the effect of thread placement to answer the question: is it better to keep cooperating threads close together on one core than to spread them out across the chip? In one case the threads can benefit from inter-thread locality and possibility of fast conflict resolution. In the other case the application can benefit from access to effectively larger private caches.

The rest of this article is organized as follows. Section 2 develops data cache designs for speculation and applies them for the purpose of supporting HTM on a multicore. Section 4 includes results from our study of the impact of multithreaded cores on HTM performance and performance implications of the cache designs. Section 5 presents an overview of existing literature on related topics, placing our contributions in context of prior work. Finally Section 6 summarizes the article and highlights some promising avenues for further study.

2. DESIGN

In this section we develop two cache designs. The first design leverages data cache associativity for speculation. It does so without imposing any penalty on cache access times. The second design builds on the first and adds new metadata to relax the constraint of restricted mapping of speculative contexts to ways. This allows not only greater freedom in placement of speculative data but also the possibility of lazy resolution of conflicts among local contexts.

2.1. Design 1: Leveraging Cache Associativity for Speculation

This design, called the SCIN-cache (read *skin*-cache), has been developed to achieve the following goals:

- on-demand allocation of speculative buffering space;
- dynamic allocation of resources to speculative contexts;
- single-cycle gang operations on metadata;
- negligible impact on cache access latency;
- entire cache is available for placement of nonspeculative data for any context.

The SCIN-cache architecture closely resembles caches found in most modern microprocessors. A SCIN-cache, however, must have associativity greater than 1. The SCIN-cache does not introduce any circuit-level changes in L1 SRAM cells and keeps architectural changes simple and easily verifiable.

The basic idea involves defining *speculation states* or *SCIN-states* that indicate the versioning status of each line in the data cache. This information is used during cache lookup to affect the result of tag comparison in such a way that, based on the context, the right version of data is matched. The basic design presented in this article also

Fig. 1. 4-way SCIN-cache with 2 contexts.

flags potential dependency violations at the granularity of cache lines, when a speculative write has already occurred. Section 2.1.7 discusses how other potential atomicity violations (when a speculative context writes to a line already read by another speculative context) can be detected. SCIN-states are stored in a special structure called the *Metadata Store*. This makes it possible to fine-tune associated logic independently for speed and implementation of gang operations required for committing or discarding speculative state en masse. The Metadata Store does not require tags. It is indexed in parallel with the tag RAMs to generate a 1-bit value for each way of the cache which is used along with the result of the TAG compare operation to determine if data should indeed be returned from a particular way. Data cache access times will not be affected since the Metadata Store can be easily designed to produce these values before the results of tag compare operation become available.

2.1.1. SCIN-Cache Architecture: A Bird's Eye View. Figure 1 shows the key components of a 4-way SCIN-cache with 2 contexts (Section 2.1.3 will discuss context management). The index calculated from the lookup address is used to access the tag RAMs and the Metadata Store in parallel. The Metadata Store has as many entries as there are sets in the cache. This yields a modest 128 entries for a 4-way 32KB cache with 64-byte cache lines. Each entry stores 2 bits for every cache line in the set identifying its SCIN-state. The Metadata Store also contains simple logic for implementation of gang operations (discussed in Section 2.1.5). *NoMatchLogic* is a simple combinatorial logic circuit that computes a *NoMatch* signal for each way of the cache using the SCIN-state of the cache block, extracted during the indexing operation. Other inputs to the

Fig. 2. SCIN speculation protocol.

NoMatchLogic will be discussed in the sections that follow. If the NoMatch signal is set high for a particular way of the cache then any positive tag compare result in that way should be suppressed. We do not explicitly suppress the tag compare result based on NoMatch, as a cache may want to use it for other purposes (one such case is discussed in Section 2.1.7). We forward both NoMatch and TagMatch signals to associated data select logic and the cache controller.

2.1.2. SCIN-States. The SCIN speculation protocol defines four states—*Speculative (S)*, *Clean (C)*, *Invalid (I)*, and *Normal (N)*—assigning one of these to every cache line. We alert the reader that these states are distinct from coherence states. When no speculative state exists, each line in the cache is either Normal containing valid nonspeculative data (N-line) or Invalid (I-line). However, when a speculative update targets a Normal line, two versions of the line are created in the cache: *Speculative (S-line)* and *Clean (C-line)*. Thenceforth, these states are maintained until a gang operation—*Commit*, *Restore*, or *Clear*—causes transitions to *Normal (N)* or *Invalid (I)* states. C and S states are coeval, i.e. they exist together, and the existence of a line in one state implies the existence of a corresponding version in the other state. The S state corresponds to the speculatively modified version of the line, while the C state corresponds to the original nonspeculative version. Figure 2 shows transitions among SCIN-states and events that trigger such transitions.

2.1.3. Support for Multiple Contexts. The SCIN-cache utilizes cache associativity to support speculative contexts. An n -way associative SCIN-cache can support upto n speculative contexts. Such a scenario can arise in modern simultaneous multithreading (SMT) processors when multiple execution contexts speculate. Each context is assigned a set of speculative and clean ways, hereafter referred to as *s-ways* and *c-ways*. For a given context, any S-lines generated on speculative updates sit in one of the *s-ways* assigned to it, while the corresponding C-lines sit in the *c-ways*. This separation of clean and speculative data is key to understanding SCIN-cache operation.

We note here that this separation does *not* imply that the cache space is dedicated for such purpose. These ways may still contain N-lines belonging to any execution context. The creation of C and S states occurs only on demand (when a speculative update targets an N-line). Moreover, to maximize utilization the *s-ways* for one context may overlap with the *c-ways* for another. This is also the reason the SCIN-cache can support as many contexts as there are ways. However, to ensure correct lookups and state transitions under gang operations (explained later) *s-ways* of different contexts cannot overlap. The same holds true for *c-ways*. Table I shows how cache ways may be allocated in two example scenarios: (a) 4-way associative cache with 2 contexts and (b) 8-way associative cache with 4 contexts. Additionally, we would like to point out that

Table I. C-Ways and S-Ways

	(a) 4-way, 2 Contexts		(b) 8-way, 4 Contexts			
	CTX 0	CTX 1	CTX 0	CTX 1	CTX 2	CTX 3
c-ways	0,1	2,3	0,1	2,3	4,5	6,7
s-ways	2,3	0,1	2,3	4,5	6,7	0,1

Table II. Way Allocation Registers: SWAR and CWAR

		Ways								
		7	6	5	4	3	2	1	0	
(a)	SWAR					0	0	1	1	1 bit/way context ID SWAR and CWAR are 4-bits wide
	CWAR					1	1	0	0	
(b)	SWAR	2	2	1	1	0	0	3	3	2 bits/way context ID SWAR and CWAR are 16-bits wide
	CWAR	3	3	2	2	1	1	0	0

Fig. 3. NoMatchLogic.

the distribution of ways among contexts can be modified when required, for example, when a context requests for speculation resources, rather than being frozen statically at design time. This is managed by maintaining way-to-context mappings in two hardware registers, *SWAR* (*S-Way Allocation Register*) and *CWAR* (*C-Way Allocation Register*) that can be configured by a speculative resource management entity in the hardware or in the operating system. The sizes of these registers depend on the associativity of the cache and the maximum number of simultaneous contexts possible. Figure 1 shows how these registers act as inputs to the NoMatchLogic. Table II shows the contents of the two registers for cases (a) and (b).

2.1.4. SCIN-Cache Lookups. We now describe the derivation of NoMatch signals based on SCIN-states, the contents of the two way allocation registers, the issuing context identifier (see the CTX signal in Figures 1 and 3) the *NSRO* (*NonSpeculative ReadOut*) signal. Figure 4 depicts actions taken at the NoMatchLogic and the cache controller to provide the desired control over the tag compare operation. We discuss four cases.

(a) *Nonspeculative Data Extraction.* Nonspeculative data in the cache may correspond to N or C states. Thus, we need to preclude tag matches for S-lines and I-lines in the cache. This is achieved by raising the NSRO signal during lookup, which takes precedence over the other three mutually exclusive scenarios (b, c, and d) described shortly. The NoMatchLogic components (see Figure 3) ensure correct handling of

Fig. 4. SCIN-cache lookup logic flowchart.

this case for each way. The cache controller signals a conflict alert (a hardware signal) if a match occurs with a C-line, which is indicative of a potential conflict between two contexts. The conflict may not exist if the matching way is part of the c-way for the context which issued the conflict. This is simple to verify by comparing against the appropriate field in CWAR. The need to extract nonspeculative data from the cache may arise when responding to remote coherence requests for data. It might also be required when a main, high-priority thread wishes to access a clean version of data that might have been modified by a speculative runahead thread. SCIN-cache aims to provide basic primitives and enough information to any hardware mechanisms employed to enable these techniques.

- (b) *C-Way Lookups.* For c-ways belonging to the execution context that issued the access (determined using CWAR), only N-lines should be potentially matched. Because of the coeval nature of C and S states, a tag match for a C-line would also imply a tag match with the corresponding S-line in one of the s-ways corresponding to the context. Since the context is speculative (as a C-line exists in its c-ways), it wants to access the corresponding S-line which would contain the latest data from the perspective of the context. Thus, c-way lookups set NoMatch to 0 only for N-lines (and not C-lines).
- (c) *S-Way Lookups.* For s-ways belonging to the execution context that issued the access (determined using SWAR), lines in N, C, or S states could be matched. It is safe to match S-lines because they belong to the correct execution context and contain the latest data from the perspective of the execution context. C-lines contain clean data, but a match in one of the s-ways indicates a potential conflict with another context. This is so because such a C-line corresponds to a version created by the context that has been allocated the way as a c-way. The cache returns C-line data with a conflict alert (a hardware signal) indicating a potential conflict. The conflicting context can be easily identified using the CWAR. Designers can use this alert to trigger safety mechanisms like speculation squashes. Conflicting speculative stores are not completed.
- (d) *Lookups in Other Ways.* These ways have not been allocated to current execution context for speculation. Hence we can only match N- and C-lines, with a conflict alert being raised if a C-line was matched for similar reasons as in (c) given before. Conflicting speculative stores are not completed. While this scenario cannot arise in

Table III. Gang Operations

	s-way transition	c-way transition
Commit	$S \rightarrow N$	$C \rightarrow I$
Restore	$S \rightarrow I$	$C \rightarrow N$
Clear	$S \rightarrow I$	$C \rightarrow I$

the 4-way 2-context arrangement shown earlier, it can occur in the 8-way 4-context case. Furthermore, it is worth noting that it is not possible for two lines in a cache set to match simultaneously. Either the tags will not match or NoMatch will be high for one of the lines. Figure 4 describes the functioning of the NoMatchLogic component pictorially. The logic is designed to be simple. Way-to-context mappings are already available from SWAR and CWAR registers. NSRO and the identifier of the context which issued the access are also typically available early. The *NoMatch* signal can then be generated quickly once the SCIN-state becomes available from the indexing operation on the Metadata Store.

Handling Speculative Writes. When a speculative write is issued we take certain actions based on the current state of the line in the cache. If the line exists in N state, then it must be versioned. To do so the cache controller allocates space for two versions: one in the context's c-way and the other in its s-way. The N-line may already be located in either the c-way or the s-way, in which case only one allocation is required in the other way. Allocations may result in evictions which are handled as usual, unless they require eviction of a C- or S-line when a resource conflict alert is issued. Designers can then use a safety net (like LogTM's virtualized buffering or squashing one of the contexts in question). Updates that target S-lines belonging to the context are handled as normal updates. Other out-of-context matches (against C-lines as described previously) will raise conflict alerts.

2.1.5. Gang Operations: Concepts and Implementation. Now, we describe a way in which gang operations may be efficiently implemented in the Metadata Store. We define three gang operations: Commit, Restore, and Clear. These operations are handled independently for each speculative context and when invoked are applied to c-ways and s-ways for the context. s-ways for different contexts do not overlap and neither do the c-ways. Thus, gang operations can be safely implemented for each context.

- Commit.* This involves converting S-lines in s-ways for the context to N, while converting C-lines in the c-ways for the context to I. Other states remain unchanged. No action is required for ways not allocated to the context for speculation.
- Restore.* This involves converting C-lines in c-ways for the context to N while converting S-lines in the s-ways for the context to I. Other states remain unchanged. No action is required for ways not allocated to the context for speculation.
- Clear.* This involves converting both C-lines and S-lines for the context to I. This operation should be invoked only when it's been made sure that dirty values have been written back to upper levels of the cache hierarchy prior to speculation.

Table III lists transitions for the three gang operations. We now describe a possible way of implementing these operations in the Metadata Store. As a first step we define 2-bit encodings (denoted by bits E1 and E0) for the four SCIN-states, as shown in Table IV. With these encodings we can now define gang operation logic. We want to apply the same operation to all lines in the s-ways or the c-ways of the set, irrespective of the state the lines are currently in (no operational selectivity). The operations are chosen such that encodings of states that must be retained remain invariant under

Table IV. SCIN-State Encoding

Encoding		State Name
E1	E0	
0	0	Normal(N)
0	1	Speculative(S)
1	0	Clean(C)
1	1	Invalid(I)

Table V. Gang Operation Logic

	s-way logic	c-way logic
Commit	$E1^+ \leftarrow E1$	$E1^+ \leftarrow E1$
	$E0^+ \leftarrow E1.AND.E0$	$E0^+ \leftarrow E1.OR.E0$
Restore	$E1^+ \leftarrow E1.OR.E0$	$E1^+ \leftarrow E1.AND.E0$
	$E0^+ \leftarrow E0$	$E0^+ \leftarrow E0$
Clear	$E1^+ \leftarrow E1.OR.E0$	$E1^+ \leftarrow E1$
	$E0^+ \leftarrow E0$	$E0^+ \leftarrow E1.OR.E0$

Table VI. Basic SCIN-State Manipulation Operations

	$E1^+$	$E0^+$
Hold-AND	$E1$	$E1.AND.E2$
Hold-OR	$E1$	$E1.OR.E2$
AND-Hold	$E1.AND.E2$	$E0$
OR-Hold	$E1.OR.E2$	$E0$

the applied operation. Table V shows how this can be achieved. $E1^+$ and $E0^+$ indicate new values for bits E1 and E0 respectively.

From Table V we can infer four basic operations that can be applied to the SCIN-states of all lines in an entire way of the cache. Table VI lists these four basic operations, indicating the manner in which new values for the SCIN-state bits are computed. The logic is simple: either a state bit is retained as is (Hold) or it is combined using basic boolean operators—AND/OR—with the other state bit. These four base operations can be used to implement Commit, Restore, and Clear, as shown in Table V. The base operation to be applied is determined using the identifier for the context that invoked the gang operation, whether a Commit, Restore, or Clear has been invoked and the contents of the CWAR and SWAR registers. Figure 5 shows a simple way to implement these in a flip-flop-based Metadata Store. The cache controller controls *GangOp* (a 2-bit signal indicating the base operation to be performed) and *GangEn* (a signal that enables the cell to store in new state) independently for each way. The figure also shows how the cache controller can load contents into each row of the Metadata Store using the Row Load signal and Ein lines. Although the Metadata Store is a small structure, we note that to save power the Metadata Store can be clock-gated most of the time (as gang operations/state updates are expected to be relatively infrequent).

Figure 6 gives an example of SCIN-cache operation. It depicts an occurrence of a conflict between two speculative contexts on the same core. The contents of SWAR and CWAR indicate that CTX0 uses ways 0 and 1 as its c-ways and ways 2 and 3 as its s-ways. Similarly, CTX1 uses ways 0 and 1 as its s-ways and ways 2 and 3 as its c-ways. Two cache lines, A and B are shown, which map to different sets in the cache. Subscripts after names A and B denote the SCIN-state of the lines. Both lines exist in the Normal (N) state initially. The two speculative contexts, CTX0 and CTX1, then update lines A and B respectively, causing the creation of (S,C) pairs for both A and B. This is followed by a conflicting access (a speculative read to line A) from CTX1. The conflict resolution

Fig. 5. A possible way to implement gang operations.

Fig. 6. SCIN-cache operation.

Table VII. Hardware for a 32KB 4-Way 2-Context SCIN-Cache

Component	Logic	Muxes	Flip Flops
Metadata Store	512 AND; 1024 OR	1024 (4-to-1); 1024 (2-to-1) and Indexing	1024 bits
NoMatchLogic	8 XNORs; 4 ANDs; 4ORs; 8 NOTs	4 (4-to-1)	None

mechanism, in this example, chooses to squash CTX1. A Restore operation on CTX1 is issued to the cache, resulting in the invalidation of the speculative copy of line B while the clean copy reverts to the Normal state. Note that the downward pointing arrows in the figure indicate gang operations occurring on the particular ways. CTX0 then commits resulting in the speculative copy of A transitioning to the Normal state with the clean copy being invalidated.

2.1.6. Hardware Requirements. As is evident from Figure 1, a SCIN-cache requires very modest hardware resources. The basic data storage and lookup elements of the cache remain unchanged. A Metadata Store with associated logic and cache controller support is required. Table VII shows hardware requirements, estimated using simple calculations, at the Metadata Store and the NoMatchLogic for supporting a 32KB 4-way 2-context SCIN-cache with 64-byte cache lines.

2.1.7. Tracking Speculative Reads. During speculative execution reads must be tracked in order to detect potential dependency violations when a speculative context updates a line that has been read by another speculative context. Since the SCIN-cache design just provides support for versioning, a separate mechanism must be provided to track reads. A simple solution is to add a set of per-context metadata bit for each cache line which is set when a speculative read occurs and is gang cleared on commits and restores. It can be implemented as part of the Metadata Store. Bloom filters can be used to track such lines that are evicted from the cache, greatly increasing the allowable read-set size. This idea has been used in several prior proposals [Ceze et al. 2007; Yen et al. 2007] using signature-based conflict resolution to support transactions and other forms of speculative threads. Based on the application in mind the architect can choose the mechanism for tracking speculative accesses.

2.1.8. Interaction with Coherence Mechanisms. The SCIN speculation protocol does not rely upon any coherence mechanisms that might exist in multiprocessor architectures. This permits the use of the protocol for uniprocessor speculation scenarios as well (for example, a microprocessor with just one multithreaded core running contexts that may speculate and possibly conflict). In many-core systems, memory system designers can choose to condition SCIN-state transitions on acquisition of appropriate permissions over the cache block under consideration. This would imply waiting for exclusive permissions over a *Normal* (*N*)-line targeted by a speculative update before creation of the S-C coeval pair. Note that it is also possible for the S-C pair to be generated ahead of time, if opportunity for doing so exists (i.e., the cache line is not invalid), so long as requisite coherence permissions are eventually acquired or a restore occurs.

Prior to invoking a Commit/Restore/Clear operation, designers must ensure that any such transitions would not violate guarantees provided by the coherence protocol. This implies ensuring that exclusive ownership is acquired over versioned lines prior to commit and that coherence states of both Speculative and Clean lines are kept synchronized by the cache controller. This is easily done as the cache controller has access to unsuppressed tag compare results and can use those to determine the s-way and c-way for any speculatively versioned line during lookup. Moreover, invalid states in either the SCIN protocol and coherence signify the same concept, that the cache line data under consideration is not usable anymore. Hence, coherence invalidations must also result in a corresponding SCIN-state change to Invalid. Conversely, an Invalid SCIN-state means that the line is invalid irrespective of its coherence state (which can be synchronized when an opportunity arises). Interaction with coherence mechanisms will be discussed further in the context of HTMs in Section 2.3.

2.2. Design 2: Relaxing Constraints in the Previous Approach

The SCIN-cache design described in previous section has very modest hardware requirements and complexity and appears to be suitable for basic HTM support in multithreaded cores. However, it has the following limitations.

- It supports updates by at most one speculative context to a particular cache line.
- Speculative buffering space for a context can only be allocated in its corresponding s-ways.

At the cost of some additional complexity, greater flexibility can be incorporated in the design. To relax the restriction of provisioning space for clean and speculative lines in certain predefined ways of the cache, a one-hot encoded context identifier field (CID) is added to the per-cache line metadata in the Metadata Store. Note that one-hot encoding is needed to provide information necessary for effecting correct state changes for commits and restores. The NoMatch signals are now generated based on

Fig. 7. Relaxing SCIN-cache restrictions using additional metadata.

Table VIII. NoMatch Signal Generation in the New Design

	SCIN State			
	S	C	I	N
NoMatch	0 if CID[CTX]=1	1 if CID[CTX]=1	1	0

both context identifier and the SCIN-state. Gang operations now involve all cells in the Metadata Store rather than involving just the c-ways and s-ways for the context in question. Figure 7 shows how each cell in the modified Metadata Store looks. Each cell includes an *Speculatively Read* (SR) bit-vector (indicating contexts that have read the corresponding line) containing as many bits as the maximum number of supported contexts. The way allocation registers are no longer required in this design. The inputs to the each cell are specified on the left of the figure. The CID field is updated (i.e., the bit corresponding to the modifying context is set) when a speculative store targets the line. The CID field is kept synchronized for the S-C pair. A conflict is detected when the CID field is not all zeroes (the line has been updated by some context) and the bit corresponding to the accessing context (CTX) is not set. A conflict is also detected when a line has been speculatively read (the SR field is not all zeroes) by a context different than the context attempting to modify the line (i.e., $SR[CTX] == 0$).

Table VIII shows how NoMatch signals would be generated by the Metadata Store. Gang operations (Commit/Restore) now cause SCIN-state transitions only for those cells for which the CID field of the cell matches the context identifier provided as input (i.e., only if $CID[CTX]=1$). Thus, the Metadata Store is now somewhat more complex than before. However, since it is a separate entity and very regular, the design can be optimized for speed. NoMatch signals are now generated within the cell and multiplexed out (possibly over a set of tristated wired-or signal lines for each way of the cache) to affect the cache lookup operation. Similarly, conflict signals (including those indicating resource conflicts) and SR states are also made available to the cache controller. The cache controller is responsible for updating the information in the Metadata Store as required as a cache read/write operation completes.

In addition to the changes mentioned earlier certain optimizations can be performed that decrease the likelihood of transaction aborts. A simple optimization that targets circumstances where a remote core steals a line that is quite likely to be updated by a local context is the addition of the AC (actively contended) bit in the Metadata Store. This bit is set when a conflict is detected between an existing local-writer context and a new local-reader context. The bit indicates that the line is probably the target of transactional read-modify-write operations. A remote requester can be stalled if there is a pending read request from a local context or if the line has been speculatively read but has not yet been written locally. This improves performance in operations on highly contented data structures like queues. The bit is reset if a transaction that read the line commits without writing to it.

This mechanism also allows more than one context to proceed when their speculative updates target the same cache line. In such a case, only one copy of the clean line is

retained and it may be also evicted (and written back if dirty) if the HTM protocol can support such evictions. In such a scenario, multiple S-lines for the same context can exist. The CID field in the Metadata Store contains adequate information to clearly distinguish data from different contexts and provides a way to implement operational selectivity when implementing per-cell SCIN-state transitions in *Combinational Logic* entity shown in Figure 7.

This ability to have multiple versions enables lazy conflict resolution for local contexts. Prior studies have shown that lazy conflict resolution is often better able to utilize available parallelism in application. More complex schemes that forward speculative data between contexts and track dependencies may be implemented. When doing so the presence of interacting contexts on the same core permits a far greater level of design freedom. However, an in-depth analysis of such a technique is out of the scope of this article. The reader is suggested to refer to Ramadan et al. [2008] to understand some of the issues involved in constructing a dependency-tracking HTM protocol.

2.3. Using the Cache Designs to Build an HTM System

TM support in hardware is typically bounded by resources available for speculation. However, this support is usually sufficient to permit a large variety of transactional applications to run efficiently. Moreover, proposals already exist [Ananian et al. 2005; Rajwar et al. 2005] that deal with scenarios when such resources run out. In this section we therefore limit our discussion to providing effective bounded support for transactions.

Enabling TM semantics requires hardware mechanisms for three essential tasks: (a) checkpointing processor state, (b) handling speculative updates, and (c) detecting and resolving potential data races. Checkpointing of processor state is a relatively common microarchitectural feature in modern processor designs as the hardware proves useful not just for TM but also for better utilization of core resources through simultaneous multithreading. Most HTM designs proposed so far make use of this feature and we do the same to develop our use-case for SCIN-cache.

When speculative contexts are supported on a CMP comprising several multithreaded cores, an additional dimension is available for design optimization. Threads running on the same core can benefit from improved locality if they belong to the same application. However, working set size limitations may demand spreading threads across cores to take advantage of increased effective cache space. Further, when threads belonging to the same application run well together on the same core, one may choose to handle local conflicts and remote conflicts differently. HTM flavors (differing in version management and conflict resolution policies) could be chosen independently for the two cases: interactions among remote contexts and interactions among local contexts. Communication between local contexts can be done with fewer and simpler design complexity trade-offs than inter-core communication. This aspect, to the best of our knowledge, has not yet been investigated in TM research so far.

We borrow the inter-core conflict resolution mechanism used by LogTM [Yen et al. 2007]. It leverages the coherence protocol, introducing minor modifications, to achieve (c). It also ensures forward progress by detecting and resolving potential deadlock scenarios. We suggest the reader to refer to the LogTM paper [Yen et al. 2007] for details regarding how this is achieved in a coherent chip multiprocessor. We note that the eager nature of LogTM allows it to provide TM support with a very minor increase in design complexity when compared to other approaches. The SCIN-cache provides a bounded-resource mechanism for versioning and for conflict detection within several contexts on a core. Note that our choice of an eager conflict resolution protocol like LogTM is *not* indicative of limited applicability of the SCIN-cache approach. The design can be used with equal ease in an HTM which uses lazy inter-core conflict

Fig. 8. Using SCIN-caches to enable HTM.

resolution. However, in the basic SCIN-cache approach (Design 1 in Section 2) at most one speculative version of a given line can exist per data cache. Thus, intra-core conflict resolution must necessarily be eager. This is not a performance issue. On the contrary, low intra-core communication costs between contexts would favor eager designs, as core resources would not be utilized by contexts when a speculation failure is likely.

However, the basic LogTM design suffers from some performance issues that become apparent in workloads with moderate to high contention. The reliance on software abort handlers results in performance problems when they need to be repeatedly invoked. Although one of the aims of LogTM designers was to have fast efficient commits leading to increased performance, they tend to lose ground in terms of performance to their lazy counterparts due to contention. This is so because, unlike lazy systems where only a commit of one transaction can lead to abort of others, in eager systems one can have aborts (and stalls) arising from conflicts detected between any pair of concurrently running transactions. Thus, aborts/stalls tend to be more numerous in eager designs. Furthermore, the need to detect conflicts eagerly puts pressure on inter-core communication and results in significant portions of this latency adding to the execution time of a transaction.

The SCIN-cache can be used here to help in two ways. The first, and the most obvious, way is by avoiding software rollbacks when the private cache space is sufficient to buffer speculative updates. In that case a simple Restore operation will result in speculative state being discarded. An abort handler need not be called. We compare this to LogTM, where a handler routine walks through a virtualized undo log restoring application state as it was before the aborted transaction started execution. The second, but a no less important, benefit is that the cache can now be used to decouple transactional execution from conflict detection. Coherence actions can now be performed in the background as the core continues to progress with transactional execution, so long as all such actions for the transaction are completed prior to commit. While store-buffers do provide this benefit to a certain extent, having the cache act as a pre-coherence buffer would potentially provide much larger performance benefits. Negi et al. [2011] have studied the impact of such buffering on eager designs and found substantial performance gains could be achieved.

When using an eager HTM design, the transaction commit operation is simple, involving just a SCIN-cache Commit gang operation for the particular context. By the time a transaction attempts to commit, it has acquired exclusive permissions over its entire set of speculatively modified cache blocks (i.e., its *write set*). The SCIN-cache protocol indicates conflicts when remote requests cause lookups in the cache. The design identifies not only conflicting condition but also the conflicting context. The HTM designer can then choose to resolve the conflict by either stalling (“NACKing”)

Table IX. System Parameters and Configurations

Simulator and Configurations	
Simulator	Full system (GEMS, Simics)
Core Settings	
Cores	16, multithreaded (1- to 16-way SMT) in-order, non-memory IPC/context 1
Chip Configurations	
Active cores	Active SMT ways per core
1	16
2	8
4	4
8	2
16	1
Memory and Directory Settings	
L1 I&D caches	Private, 16KB 8-way, 1-cycle latency
L2 cache	Shared, 512KB per tile, unified 16-way, 12 cycle-latency
L2 Directory	MESI, Bit vector, 6-cycle latency
Memory	4GB, 300-cycle latency
CMP NoC Settings	
Topology	2D Mesh
Link latency	1 cycle

the remote access or aborting the context in question. Transactional reads by local contexts are managed separately (as mentioned in Section 2.1.7) and detect potential conflicts when remote writes are observed.

Figure 8 shows a typical organization for an HTM system that uses a SCIN-cache. Information/command communication that might exist between components is shown. Incorporating HTM on an SMT core is simple as well, as shown in the figure. The cache eagerly alerts when conflicting accesses occur. HTM management logic in each core can use these alerts to stall or squash a local context or ask a remote core to take necessary measures to avoid the conflict.

3. METHODOLOGY AND SIMULATION ENVIRONMENT

To validate that the design works and performance is as expected from a performant HTM design, we have analysed a variety of scenarios that include popular HTM proposals.

3.1. Simulation Platform

GEMS [Martin et al. 2005] was used with SIMICS [Magnusson et al. 2002] as a full system simulation platform. Table IX shows various relevant simulator parameters. We model a tiled 16-core CMP configuration with private L1I and L1D caches and a shared, multi-banked L2 cache consisting of 16 banks of 512KB each (one L2 slice per tile). For each workload/configuration pair we gathered average statistics over 10 randomized runs designed to produce different interleavings between threads.

The simulator was modified to allow each core to support a variable number of threads, ranging from 1- to 16-way SMT. We then adapt the number of active cores in each experiment according to the degree of multithreading (number of SMT ways), so that for all configurations the total number of hardware thread contexts presented to the application remains constant and fixed to 16, as can be seen in Table IX under *Chip Configurations*. In this way, the *SMT2* suffix in the plots that follow indicates

a configuration that uses 2-way SMT cores, which implies that only half of the 16 available cores in the CMP are turned on in that case. For the *SMT4* case, 4 out of 16 cores are used, and so on. All evaluated workloads are then executed with 16 threads, binding each thread to a different logical processor.

The key challenge in constructing the desired simulation environment was the incorporation of a reliable and flexible mechanism to the vary number of threads that can be associated to each core. We accomplished this by altering the way SIMICS processing units communicate with the Ruby memory system model in GEMS. The simulator was modified to allow processing of memory requests from a variable number of SIMICS cores by one private data cache.

3.2. HTM Systems

Experiments were performed using the following HTM designs.

- Baseline*. This design uses LogTM-SE [Yen et al. 2007] with perfect signatures for storing transactional read/write sets for each context. We refer to this design point using the prefix *Base* throughout the plots in Section 4. Since LogTM-SE as proposed does not support multiple contexts per cache, we modified the base implementation in order to include an extra *local* conflict detection step amongst all contexts running in the same core. These signature checks are performed before loads/stores are issued to the memory subsystem, ensuring that speculatively modified lines are only accessible by their respective contexts. Note that the non-SMT LogTM setup (16 single-threaded cores, *Base_SMT_16*) only necessitates signature checks upon remote forwarded coherence requests. However, when SMT cores are considered (*Base_SMT_x*), LogTM must check all other local signatures upon every memory access. For this study, we consider that such Bloom filter accesses can be performed at no cost in terms of latency and thus do not affect the critical path of a cache hit. Nonetheless, this assumption clearly favors SMT performance of the baseline compared to the SCIN, cache designs, and should be kept in mind when analyzing the performance levels obtained by our scheme.
- LogTM*. In order to ease the comparison against prior works, we also show results for the original LogTM-SE system, as distributed in the latest GEMS 2.1 release. Unlike the *Baseline* system given before, which is a fine-tuned version of LogTM, this *LogTM* configuration always traps to software on abort, uses a randomized exponential backoff routine inside the abort handler, and relies on a conflict resolution policy that is susceptible to the *starving writer* pathology [Bobba et al. 2007]. On the other hand, *Baseline* can perform single-cycle rollbacks for transactions whose undo log is empty, uses a randomized linear backoff scheme with no software intervention, and prevents writer starvation by using a hybrid resolution policy that allows an elder writer to simultaneously abort a number of younger readers.
- FASTM*. This eager design [Lupon et al. 2009] is built on top of LogTM, and modifies the MESI coherence protocol in order to push clean state to lower levels of the cache hierarchy prior to speculative writes. The FASTM cache allows speculatively modified lines to be gang-invalidated on abort, effectively speeding up the rollback phase and avoiding software handler intervention. However, this approach may slow down the re-execution of aborted transactions due to the encounter of additional L1 misses for such *contaminated* lines [Waliullah and Stenstrom 2011], as the L1 is not able to hold both clean and speculative copies of the data. For the experiments with FASTM, we did not run any SMT configurations, but simply set up the simulation to use 16 single-threaded cores.
- SCIN*. Throughout the evaluation, plots labeled *SCIN_dyn* or simply *SCIN* correspond to the flexible SCIN-cache presented in Section 2.2, while the lower-complexity

Table X. Workloads in the STAMP Benchmark Suite

Name	Parameters
genome	-g256 -s16 -n16384
genome+	-g512 -s32 -n32768
intruder	-a10 -l4 -n2048 -s1
intruder+	-a10 -l16 -n4096 -s1
kmeans-high	-m15 -n15 -t0.05 -i random-n2048-d16-c16
labyrinth	-i random-x32-y32-z3-n96
SSCA2	-s13 -i1.0 -u1.0 -l3 -p3
SSCA2+	-s14 -i1.0 -u1.0 -l9 -p9
vacation-high	-n4 -q60 -u90 -r16384 -t4096
vacation-high+	-n4 -q60 -u90 -r1048576 -t4096
yada	-a20 -i 633.2
yada+	-a10 -i ttimeu10000.2

yet more restrictive design described in Section 2.1 is referred to as *SCIN_sta*. These SCIN-based designs perform read and write set tracking using perfect signatures. Signatures are only required for book-keeping lines that overflow the cache. In the absence of evictions, the SCIN-cache internal logic is capable of simultaneously tracking read and write sets for all contexts. Data races amongst local SMT contexts are detected upon cache access and the conflict signal propagated to the cache controller along with the data retrieved. To provide a fair comparison against the previous log-based designs when running transactions with large foot-prints, we have also incorporated logging capabilities to both SCIN-cache-based designs. Since S/C-line pairs are pinned down to the cache, these designs dedicate at most *cache_associativity* – 2 ways of each set to accommodate such pairs. If a subsequent write targets a *full S/C set* and results in a write miss, the SCIN-cache signals this event and the HTM system triggers the logging circuitry. The new speculative line is eventually allocated in the set, but it is marked as Normal (N) and thus can be safely evicted if necessity arises. Aborting an *overflowed* transaction thus involves restoring the SCIN-cache to its clean state, prior to the execution of the log unroll by the abort handler. Note that, unlike *FASTM*, the SCIN-cache approach does not need any modifications to the coherence protocol, as lines held in exclusive ownership cannot simply disappear from the owner’s private cache on abort.

3.3. Workloads

STAMP [Cao Minh et al. 2008], a suite of transactional workloads, was used to validate the design and estimate performance metrics. Table X lists applications in the suite that have been used for evaluation. The suffix “+” indicates a larger working set. The suffix “-high” indicates the use of high contention parameters. We have excluded *bayes*, an application from the suite, because of large variations in execution times from run to run. Such behavior is also noticeable, though to a much lesser extent, in the application *labyrinth*, which has been included in the study.

4. EVALUATION

The data caches presented in this article have been designed with the aim to allow speculation on multi-way threaded cores, while maintaining a level of performance similar to that of popular optimized HTM design points. Therefore, major deviations in overall performance from other design points are not expected.

Fig. 9. Relative performance of SCIN-based systems against other HTM proposals in a non-SMT configuration.

Table XI. Execution Time: Breakdown Components

Component	Description
barrier	wait at barriers
non_txnal	non-transactional execution
tx_useful	successful transactional execution
tx_aborted	aborted transactional execution
stall_useful	stalls followed by a commit
stall_aborted	stalls followed by an abort
backoff	idle cycles after an abort
rollback	cycles spent in sw log unroll

4.1. Experiment 1: Impact of Simultaneous Multiversioning on TM Overheads

We first evaluate the performance implications of the use of the SCIN-cache in a non-SMT environment. Figure 9 presents normalized execution times for each of the five HTM systems described in the previous section: the original LogTM-SE, our own fine-tuned flavor of it (*Base*), the FASTM system, and finally the two SCIN-cache-based designs. Results are normalized to *Base*. We represent parallel execution time divided into disjoint components, each one corresponding to sum of the cycles spent by each thread in a given state during its execution. The categories in which execution time is divided into are explained in Table XI. This breakdown serves as an intuitive picture about the characteristics of the transactional application, and it also gives us some insights into the behavior of the HTM system.

A key observation that can be drawn from Figure 9 is that the contribution of the rollback mechanism to overall execution time is small. Thus, optimizations that attempt to minimize these overheads, like those that form the basis of the FASTM design and that are present in the SCIN-cache as well, can only have a limited influence of performance. However, in a high contention application like *intruder*, improvement in abort rate due to such optimizations result in a significantly higher throughput via

Table XII. Cache Performance and Log Utilization for Base, FASTM and SCIN Designs

	Misses per transaction				Writebacks		Main TID	Log size (entries)		
	Base	FASTM	SCIN_sta	SCIN_dyn	FASTM	SCIN		Base/FASTM	SCIN_sta	SCIN_dyn
genome	7,8	9,0	7,8	7,8	12	0,3	2	2,8 (3,5)	0 (0)	0 (0)
genome+	9,4	10,0	9,6	9,6	52	1,2	2	3,0 (3,4)	0 (0)	0 (0)
intruder	4,3	10,8	4,3	4,3	16	0,5	1	2,9 (6,1)	0 (0)	0 (0)
intruder+	5,3	10,4	5,3	5,3	116	10	1	2,7 (5,7)	0 (0)	0 (0)
kmeans-high	3,4	3,5	3,3	3,3	3	3	0	1,7 (2)	0 (0)	0 (0)
labyrinth	277,2	748,9	365,7	175,0	148	200 (51)	1	94,0 (216)	51 (121)	3,5 (9)
ssca2	2,6	2,6	2,7	2,7	87	31	2	2 (2)	0 (0)	0 (0)
ssca2+	2,4	2,4	2,5	2,5	218	88	2	2 (2)	0 (0)	0 (0)
vacation-high	41,8	42,3	41,4	41,4	37	4,5	0	6,7 (6,8)	0 (0)	0 (0)
vacation-high+	70,9	70,9	70,4	70,4	36	5,8	0	6,6 (6,7)	0 (0)	0 (0)
yada	28,1	41,9	27,2	26,7	92	15 (7)	2	11,3 (61,7)	0,9 (5,1)	0 (0,01)
yada+	38,3	46,8	38,2	37,5	332	59 (30)	2	14,0 (70,8)	1,2 (6,5)	0 (0,01)

synergistic effects which yield substantial improvement in back-off times and wasted work. A large part of the performance improvement in FASTM relative to LogTM arises primarily from the choice of the back-off policy and avoidance of a substantial number of instances of software rollbacks in workloads with small to moderate sized transactions. The Base configuration, which essentially applies simple optimizations to the LogTM design to mitigate these overheads (like avoiding trapping to software when the undo log is empty) performs fairly competitively to the three better performing configurations. The SCIN-cache designs perform noticeably better than FASTM in high contention, due to their improved cache performance: By retaining a clean copy of speculatively updated lines in L1 cache, they avoid contamination misses [Waliullah and Stenstrom 2011] when transactions reexecute. This is visible in Table XII, whose first four data columns show the average number of cache misses per transaction for the systems plotted in Figure 9. The relaxed SCIN-cache configuration, SCIN_dyn, performs better in workloads with large transactions (large read- and write-sets) due to greater flexibility in data placement within the cache. One can also notice rather large barrier times for LogTM when running genome and vacation-high. This is primarily due to the susceptibility of the LogTM design to the *starving writer* pathology, which delays writer threads due to an abundance of reader transactions in the system.

Table XII presents statistics that show how the two SCIN-cache designs outperform both LogTM and FASTM. Software abort handlers in LogTM restore cache state upon an abort, but the overhead of trapping to software imposes a high performance penalty. FASTM is able to avoid the penalty of trapping to a software abort routine by invalidating speculative data in the private cache on abort. FASTM provides correct operation by ensuring that upper levels of the cache hierarchy contain a clean version of every speculatively updated lines. However, invalidations result in cache misses upon re-execution (contamination misses), which in turn slow down the application. The SCIN-cache designs are able to combine fast hardware restoration of cache state and absence of contamination misses. We see in Table XII that the number of misses per transaction in SCIN_sta is similar to Base—except for labyrinth—, also indicating that the static allocation of S/C pairs does not generally cause extra conflict misses. Only labyrinth’s huge write set makes SCIN_sta suffer, but SCIN_dyn overcomes this limitation and significantly reduces the number of misses in labyrinth (well under the levels of Base), while also requiring fewer writebacks in workloads with large transaction foot-prints (labyrinth, yada), when compared to FASTM. Furthermore, FASTM constantly needs to push consistent values to the L2 cache before creating a new

Fig. 10. Impact of mapping an increasing number of threads per core, to a decreasing number of cores, for the baseline system. Note: Total number of threads remains fixed.

speculative version in L1, hence putting more pressure on the interconnect. This can be seen in Table XII (data columns 5 and 6), which shows the number of L1 writebacks—in thousands of messages—for FASTM, SCIN_sta, and SCIN_dyn (the latter appears in brackets when different from SCIN_sta).

The four right-most columns of Table XII show the number of speculative spills seen in the SCIN designs due to insufficient space for a new S/C pair in the cache set. Along with average figures across all transactions, specific log sizes are shown in brackets for the main atomic block of each benchmark, whose identifier (*TID*) is shown in data column 7. Overall, we can observe that for the majority of the applications, the SCIN_dyn design rarely relies on safety nets, and even for an extreme case such as labyrinth, the number of speculative spills is very modest.

4.2. Experiment 2 : Impact of Multithreaded Cores

We now estimate the impact of mapping more than one thread of an application to each multithreaded core. We run upto 16 contexts on each core, keeping L1 cache size fixed at 16KB, to see whether effects due to locality allow some applications to perform better than others. The choice of a 16KB cache was driven by a desire to investigate effects of increasing the number of contexts on workload performance. STAMP benchmarks are quite cache-friendly. Thus, with an intent to observe performance impact when working set sizes approach available cache space, we reduced the cache space to 16KB. Figure 10 shows how the performance of the baseline system varies when we map an increasing number of threads to a core. We keep the number of contexts constant. We believe that by doing so a fair comparison of relative merits of varying SMT resources is achieved. Note that the data cache size for each core remains fixed across configurations. Therefore, amount of private cache space per context decreases as a larger number of threads are mapped to each core. This allows expected performance trends due to various thread placement strategies to be inferred. For some applications, like intruder and kmeans, it is beneficial to co-locate several threads on the same core. Transactions in intruder are small and highly contended. Thus, applications benefit from increased locality and better chances of finding migratory lines in cache when transactions attempt to update them. For other applications like yada and labyrinth

Fig. 11. Performance of various SCIN-cache SMT configurations relative to a non-SMT baseline, and impact of mapping more SMT threads per core.

the reduction in available private cache space results in increasing performance penalties as more threads are mapped to each core.

4.3. Experiment 3: Impact of Using the SCIN-Cache

Figure 11 compares the performance of a number of SCIN-cache (the relaxed design, SCIN_dyn) designs with an increasing number of contexts per core to that of Base (the baseline configuration) running one context per core, using a 16KB data cache. Using a smaller data cache allows cache space limitations to have a more pronounced effect. We only present SCIN_dyn numbers from this point on, since they do not differ significantly from the SCIN_sta results for all benchmarks except labyrinth, for the reasons explained earlier. By looking at Figure 10, we observed how some applications benefit with more SMT ways but less cores (due to locality), while others suffer (due to competition for L1 cache resources). Comparing each base-SMT configuration in Figure 10 with its corresponding SCIN-SMT pair (i.e., same number of SMT ways) in Figure 11 we get a clear picture of the value that the SCIN-cache adds in an SMT environment. The key result to note from the plot is that, for all the applications with the exception of labyrinth, it is possible to find a SCIN-cache configuration (i.e., number of contexts per core) which tracks the best observed performance. For example, intruder performs the best with 8 contexts per core, while 2 contexts per core appear to be optimal for genome and SSCA2. In the case of intruder, the ability to support a variable number of contexts allows the SCIN-cache to achieve substantial gains in performance.

4.4. Experiment 4: Lightweight Locking and Penalty on L1 Hit Time

Figure 12 compares the two-context per core SCIN-cache design (SCIN_dyn) to an approach which supports lightweight locking between two contexts. Results are normalized to the same base as Figures 10 and 11. Also shown in the figure is a LogTM configuration which supports two contexts, with the necessary conflict detection filter checks imposing a penalty on 2 cycles. The lightweight locking approach serializes transactions on each core. However, the approach permits parallelism between cores.

It can be seen for high contention workloads like intruder and intruder+ that lightweight locking provides benefits over the basic LogTM configuration. However,

Fig. 12. Relative performance of 2-way SMT configurations: Base, SCIN, and locking.

the ability of the SCIN-cache to make better use of any available parallelism results in a substantial performance advantage over the lock-based scheme. For labyrinth, an application with very limited scalability, giant write-sets cause performance degradation when using the SCIN-cache design. In vacation, one can see how optimistic concurrency mechanisms (LogTM and SCIN-cache) outperform a pessimistic (although partially) approach (lightweight locking). A similar effect is visible in genome as well.

5. RELATED WORK

In this section we look at related work done in the following two categories .

Speculative Versioning in Data Caches. The use of data caches for versioning has been explored in several studies in the past. Most studies have concentrated on providing versioning support in a multiprocessor environment with each data cache capable of storing one speculative version. Address Resolution Buffer (ARB) [Franklin and Sohi 1996] use special book-keeping structures to manage speculative versions across many private caches in processors. This can be done in a centralized or distributed fashion. However, extra cycles are required to access these structures. Speculative Versioning Cache (SVC) [Gopal et al. 1998], an approach to enable speculative multithreading, provides support for multiple speculative versions spread across various data caches in a multiprocessor and has special protocols for dependency management. The architecture envisioned by SVC involves several cores onto which speculative tasks are mapped. Each core has a speculative data cache that can contain one set of speculative data (i.e., that produced by the task mapped to the core). Note that this differs from SCIN-cache, which provides speculative versioning for multiple contexts in the *same* data cache. The SVC design requires conditional gang invalidation, an uncommon architectural feature which requires circuit-level changes (additional transistors) to the SRAM cell. In a similar vein, BulkSMT [Qian et al. 2012] describes a design for supporting simultaneous speculative contexts on a multithreaded core using cache line annotations requiring conditional gang clearing logic based on comparison between multi-bit fields. The Memory Disambiguation Table (MDT) design [Krishnan and Torrellas 1999] requires caches that support both a speculative write-back mode and a nonspeculative write-through mode. Speculative updates are performed in the

write-back mode and require a flush of dirty lines to the L2 cache, so that the cache can safely transition to nonspeculative write-through operation. Marcuello and González [1999] propose per-context private caches and one shared cache in SMT processing element. This would quite likely result in underutilization of cache resources. Smart Memories [Mai et al. 2000] proposes architectural support that is similar to but simpler than SVC for multiprocessors. Colohan et al. [2006] use associativity at the L2 for the purpose of storing multiple speculative copies of a particular cache block. However, the work, done in the context of thread-level speculation (TLS), does not deal with the problem of efficient context management and extraction of data. The authors would like to point the reader to the study by Garzarán et al. [2005] which presents a taxonomy and a study of trade-offs in approaches to speculative buffering in TLS.

A study by Packirisamy et al. [2006] looks at ways for supporting two or four speculative threads on an SMT core. Commits in the design are slow and complex involving write-backs and merging of lines from multiple contexts. Their key limitation, however, is that the design fails to address the issue of extraction of correct speculative data versions in response to accesses from the core, assuming that such a mechanism exists. This problem of data extraction lies at the heart of the SCIN-cache approach. Moreover, their design attempts to support multiple speculative contexts updating the same cache line, a case which in our opinion is indicative of limited concurrency and also usually a relatively easy target for compiler optimizations and shallow speculation techniques involving the LSQ or a store buffer. HTM proposals [Hammond et al. 2004; Yen et al. 2007] have typically used SVC-like annotations to mark speculatively read or written lines in the cache. Most rely in part on the ability to gang-clear SRAM bits. We find that there is room for a new cache architecture with support for multiple contexts that provides versatile primitives to enable various speculative techniques. We attempt to provide such an architecture in this work.

Data Cache Usage in HTM. In the domain of HTM prior techniques exist for speeding up transactional commits and aborts. Lazy HTM designs [Hammond et al. 2004; Chafi et al. 2007; Tomic et al. 2009] and techniques like FASTM [Lupon et al. 2009] for eager designs push clean state to lower levels of the cache hierarchy and invalidate speculative lines on abort. This trades off performance for correctness in high contention scenarios (where a large number of aborts may occur) by causing L1 misses for such lines upon re-execution of aborted transactions. Although aborts are now fast, re-executions slow down, lowering effective performance. Moreover, additional complexity introduced into the already complicated coherence protocol by adding transactional states increases design time and verification costs. In eager systems [Yen et al. 2007] that employ stalls and backoffs, high contention scenarios can turn pathological degrading performance severely. This can be noticed for certain benchmarks like *intruder* in numerous studies that use the STAMP benchmark suite [Cao Minh et al. 2008] for evaluation. An attempt to tackle this problem has been made by Armejach et al. [2011], proposing major changes to the highly optimized SRAM cell architectures in the cache. Each SRAM cell in the RDC is designed to store two bits of information with exchange circuits allowing the value of one bit to be copied to the other. Thus, both speculative and consistent versions of a cache line can be held close to the core. Although this is an interesting exploratory design, RDC imposes severe penalties on TM usage by dedicating half the cache for speculative versioning for workloads that might use transactions. Thus, in addition to major design changes, applications that have nontrivial nontransactional phases would suffer greatly from unavailable cache resources. We would also like to compare our approach to TM support in IBM BlueGene/Q processor [Ohmacht 2011]. BlueGene/Q provides TM support to threads running on the same chip, with the L2 providing centralized versioning support. This requires extensive hardware

support, limits design scalability, and imposes latency costs of communicating with the L2 for versioning.

Lightweight Synchronization on SMT Processors. Simultaneous contexts on one multithreaded core can have fast inter-context communication and easy thread halt/restart mechanisms. Lightweight locking [Tullsen et al. 1999] leverages these properties to provide inexpensive synchronization amongst local contexts for lock-based programs. However, such an approach is inherently pessimistic. Under light or variable contention performance losses can occur, as nonconflicting threads get unnecessarily serialized. While full per-context versioning on an SMT core may not be necessary to provide TM guarantees of atomicity, isolation, and consistency, a mechanism for conflict management is necessary to allow nonconflicting threads to proceed without hindrance from each other and, thereby, realize the full potential of optimistic concurrency.

6. CONCLUSIONS AND FUTURE WORK

This article develops fast, easy-to-use cache designs to support speculation in chip multiprocessors (CMP) with multithreaded cores. A detailed description of the novel SCIN-cache architecture, which utilizes cache associativity to manage speculative state for several contexts, is presented. The manner in which multiple contexts are managed and accesses are handled, and the design and implementation of fast gang operations is described in detail. The investigation of the use of SCIN-caches in HTM and performance comparisons with existing design points reveal substantial gains can be achieved when contention is high. Overall, levels of performance similar to optimized HTM designs like FASTM can be maintained. It is also shown that running multiple thread contexts together on the same core impacts locality and can, in some cases, improve performance. With the growing importance of speculative techniques to keep performance scaling alive in the multicore era, there exists an exigent need for fast and simple hardware mechanisms that can support a variety of such techniques. The SCIN-cache design aims to be one of the many architectural innovations required to achieve this end. We hope it will prove to be a useful building block for constructing efficient parallel computing systems. In the future we would like to investigate trade-offs and alternatives in this design space in greater depth. The use of such speculative caches, which permit inexpensive speculative contexts to be generated and removed on-the-fly, for supporting other performance enhancement techniques like speculative parallelization seems to be a promising area of study as well.

ACKNOWLEDGMENTS

The authors thank Magnus Ekman for his early feedback on ideas presented in this work.

REFERENCES

- ANANIAN, C. S., ASANOVIC, K., KUSZMAUL, B. C., LEISERSON, C. E., AND LIE, S. 2005. Unbounded transactional memory. In *Proceedings of the 11th Symposium on High-Performance Computer Architecture*. 316–327.
- ARMEJACH, A., SEYEDI, A., TITOS-GIL, R., HUR, I., UNSAL, O. S., CRISTAL, A., AND VALERO, M. 2011. Using a reconfigurable L1 data cache for efficient version management in hardware transactional memory. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*. 361–371.
- BOBBA, J., MOORE, K. E., YEN, L., VOLOS, H., HILL, M. D., SWIFT, M. M., AND WOOD, D. A. 2007. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th International Symposium on Computer Architecture*. 81–91.
- CAO MINH, C., CHUNG, J., KOZYRAKIS, C., AND OLUKOTUN, K. 2008. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the International Symposium on Workload Characterization*. 35–46.

- CEZE, L., TUCK, J., MONTESINOS, P., AND TORRELLAS, J. 2007. BulkSC: Bulk enforcement of sequential consistency. In *Proceedings of the 34th International Symposium on Computer Architecture*. 278–289.
- CHAFI, H., CASPER, J., CARLSTROM, B. D., McDONALD, A., MINH, C. C., BAEK, W., KOZYRAKIS, C., AND OLUKOTUN, K. 2007. A scalable, non-blocking approach to transactional memory. In *Proceedings of the 13th Symposium on High-Performance Computer Architecture*. 97–108.
- CHAUDHRY, S., CYPHER, R., EKMAN, M., KARLSSON, M., LANDIN, A., YIP, S., ZEFFER, H., AND TREMBLAY, M. 2009. Rock: A high-performance Sparc CMT processor. *IEEE Micro* 29, 2, 6–16.
- COLOHAN, C. B., AILAMAKI, A., STEFFAN, J. G., AND MOWRY, T. C. 2006. Tolerating dependences between large speculative threads via sub-threads. In *Proceedings of the 33rd International Symposium on Computer Architecture*. 216–226.
- CRISTAL, A., SANTANA, O. J., CAZORLA, F., GALLUZZI, M., RAMIREZ, T., PERICAS, M., AND VALERO, M. 2005. Kilo-Instruction processors: Overcoming the memory wall. *IEEE Micro* 25, 48–57.
- FRANKLIN, M. AND SOHI, G. S. 1996. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Trans. Comput.* 45, 552–571.
- GARZARÁN, M. J., PRVULOVIC, M., LLABERÍA, J. M., VIÑALS, V., RAUCHWERGER, L., AND TORRELLAS, J. 2005. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *ACM Trans. Archit. and Code Optim.* 2, 3, 247–279.
- GOPAL, S., JAMES, T. N. V., SMITH, E., AND SOHI, G. S. 1998. Speculative versioning cache. In *Proceedings of the 4th Symposium on High-Performance Computer Architecture*. 195–205.
- HAMMOND, L., CARLSTROM, B. D., WONG, V., CHEN, M., KOZYRAKIS, C., AND OLUKOTUN, K. 2004a. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro* 24, 6, 92–103.
- HAMMOND, L., WONG, V., CHEN, M., CARLSTROM, B. D., DAVIS, J. D., HERTZBERG, B., PRABHU, M. K., WIJAYA, H., KOZYRAKIS, C., AND OLUKOTUN, K. 2004b. Transactional memory coherence and consistency. In *Proceedings of the 31st International Symposium on Computer Architecture*. 102–113.
- HERLIHY, M. AND MOSS, J. E. B. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*. 289–300.
- KRISHNAN, V. AND TORRELLAS, J. 1999. A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. Comput.* 48, 866–880.
- LUPON, M., MAGKLIS, G., AND GONZÁLEZ, A. 2009. FASTM: A log-based hardware transactional memory with fast abort recovery. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*. 293–302.
- MAGNUSSON, P. S., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HALLBERG, G., HOGBERG, J., LARSSON, F., MOEST-EDT, A., AND WERNER, B. 2002. Simics: A full system simulation platform. *IEEE Comput.* 35, 2, 50–58.
- MAI, K., PAASKE, T., JAYASENA, N., HO, R., DALLY, W. J., AND HOROWITZ, M. 2000. Smart memories: A modular reconfigurable architecture. In *Proceedings of the 27th International Symposium on Computer Architecture*. 161–171.
- MARCUCELLO, P. AND GONZÁLEZ, A. 1999. Exploiting speculative thread-level parallelism on a SMT processor. In *Proceedings of the 7th International Conference on High-Performance Computing and Networking*. 754–763.
- MARTIN, M. M., SORIN, D. J., BECKMANN, B. M., MARTY, M. R., XU, M., ALAMELDEEN, A. R., MOORE, K. E., HILL, M. D., AND WOOD, D. A. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Comput. Archit. News*, 92–99.
- NEGI, A., TITOS-GIL, R., ACACIO, M. E., GARCIA, J. M., AND STENSTROM, P. 2011. Eager meets lazy: The impact of write-buffering on hardware transactional memory. In *Proceedings of the 40th International Conference on Parallel Processing*. 73–82.
- OHMACHT, M. 2011. Hardware support for transactional memory and thread-level speculation in the IBM Blue Gene/Q system. In *Proceedings of (WANDS11)*. co-located with PACT-11.
- PACKIRISAMY, V., WANG, S., ZHAI, A., HSU, W.-C., AND YEW, P.-C. 2006. Supporting speculative multithreading on simultaneous multithreaded processors. In *Proceedings of the 12th International Conference on High-Performance Computing*. 148–158.
- PARK, I., FALSAFI, B., AND VIJAYKUMAR, T. N. 2003. Implicitly-Multithreaded processors. In *Proceedings of the 30th International Symposium on Computer Architecture*.
- QIAN, X., SAHELICES, B., AND TORRELLAS, J. 2012. BulkSMT: Designing SMT processors for atomic-block execution. In *Proceedings of the 18th Symposium on High-Performance Computer Architecture (HPCA '12)*. 1–12.
- RAJWAR, R., HERLIHY, M., AND LAI, K. 2005. Virtualizing transactional memory. In *Proceedings of the 32nd International Symposium on Computer Architecture*. 494–505.

- RAMADAN, H. E., ROSSBACH, C. J., HOFMANN, O. S., AND WITCHEL, E. 2008. Dependence-aware transactional memory. In *Proceedings of the 41st International Symposium on Microarchitecture*. 246–257.
- ROSSBACH, C. J., HOFMANN, O. S., AND WITCHEL, E. 2010. Is transactional programming actually easier? In *Proceedings 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 47–56.
- TOMIC, S., PERFUMO, C., KULKARNI, C., ARMEJACH, A., CRISTAL, A., UNSAL, O., HARRIS, T., AND VALERO, M. 2009. EazyHTM: Eager-Lazy hardware transactional memory. In *Proceedings of the 42nd International Symposium on Microarchitecture*. 145–155.
- TULLSEN, D. M., LO, J. L., EGGERS, S. J., AND LEVY, H. M. 1999. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings of the 5th Symposium on High-Performance Computer Architecture*. 54–65.
- WALIULLAH, M. AND STENSTROM, P. 2011. Classification and elimination of conflicts in transactional memory systems. In *Proceedings of the 23rd International Conference on Computer Architecture and High Perf. Computing*. 96–103.
- YEN, L., BOBBA, J., MARTY, M. R., MOORE, K. E., VOLOS, H., HILL, M. D., SWIFT, M. M., AND WOOD, D. A. 2007. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th Symposium on High-Performance Computer Architecture*. 261–272.

Received June 2012; revised November 2012; accepted November 2012