# Conflict Detection in Hardware Transactional Memory

Ricardo Quislant, Eladio Gutierrez, Emilio L. Zapata, and Oscar Plata

Dept. Computer Architecture, University of Malaga, E-29071, Malaga, Spain
{quislant,eladio,zapata,oplata}@uma.es

**Abstract.** This chapter is dedicated to the conflict detection mechanism in the context of hardware transactional memory (HTM) systems. An effective mechanism is needed to detect conflicts amongst transactions, thus ensuring atomicity while allowing concurrency. Together with version management and conflict resolution, the conflict detection mechanism is one of the main design choices in HTM systems.

In this chapter, the two most common ways of detecting conflicts are described: eager and lazy. Then, we discuss the main HTM approaches to conflict detection, from the very first system proposed by Herlihy and Moss in 1993, to the commercial systems delivered by Intel or IBM, amongst others. Finally, a survey on conflict detection virtualization, i.e. support for unbounded transactions, is presented, emphasizing the signature topic.

## 1 Introduction

One of the most important design choices in hardware transactional memory (HTM) systems is how to address the conflict detection problem. Transactions must be perceived by the user as though they were a single, indivisible instruction. That is, the HTM system must ensure the *atomicity* property of transactions. A single global lock is able to provide atomicity, but eliminates concurrency. In order for a HTM system to exploit potential parallelism, an effective mechanism must be designed that keeps track of every memory access issued by transactions to detect conflicts amongst them and preserve atomicity.

Conflict detection mechanisms can be classified into two main categories depending on when data conflicts are detected: *eager* and *lazy*.

### Eager Conflict Detection

When conflicts are detected eagerly, the HTM system has to intercept each memory access so that the conflict is detected just before it occurs. This way of detecting conflicts is *conservative* since it keeps transactions from working with stale data, thus reducing the amount of useless computation.

Eager conflict detection can be combined with either form of version management, also categorized as *eager* and *lazy*, where eager version management updates transactional data directly to memory, and lazy version management isolates transactional writes into a private buffer. We can therefore find eager-eager HTM approaches [4,19] and eager-lazy ones [2,15,26].

R. Guerraoui and P. Romano (Eds.): Transactional Memory, LNCS 8913, pp. 127–149, 2015.

Most HTM systems rely on the cache coherence mechanism to detect conflicts early [2,15,19,27]. Note that having transactional state together with each cache block allows the conflict detection mechanism to check for conflicts in an effective way. Although the cache coherence protocol is a critical element of a multicore processor, which is difficult to specify and verify [31], this is a common way to address eager conflict detection implementation.

As far as conflict resolution is concerned, eager conflict detection enables another alternative than aborting transactions, which is *stalling* them [19]. As conflicts are detected just before happening, the HTM system might as well delay the resolution of the conflicting memory access until it is not a conflict anymore. The involved transaction is then stalled and its work preserved unless the transaction has to be aborted eventually, either to ensure forward progress or to avoid performance pathologies [5].

**Lazy Conflict Detection**

With lazy conflict detection the HTM system allows transactions to access shared data concurrently regardless of conflicts, whose detection is deferred to commit time. This kind of conflict detection is *optimistic*, as it encourages parallelism. However, the lazy approach could readily serialize execution when conflicts are often encountered, because the only conflict resolution possibility is to abort one of the conflicting transactions. In general, stalling is impossible with lazy conflict detection. The increased level of speculation translates into an increased amount of discarded computation in case of conflict. On the other hand, anti-dependencies (WAR) and output dependencies (WAW) can be filtered out as the instructions of the committing transaction are considered sequentially earlier than the instructions of the other transactions that have not yet committed.

Unlike eager conflict detection, lazy conflict detection can be coupled only with lazy version management. Otherwise, the isolation property of transactions would not be enforced as conflicts are not detected until commit time. We can find several lazy-lazy HTM systems in the literature [7,14].

The lazy conflict detection mechanism can simplify the implementation of the HTM system by minimizing added complexity to the cache coherence protocol and primary caches. Private buffers are often used to keep new versions of the data accessed by a transaction. But the system must deal with an increased interconnection network traffic as the transactional state of a transaction has to be broadcast in order for the rest of processors to detect conflicts.

A third form of conflict detection can be considered for those systems that allow validations inside transactions. *Conflict validation* consists of checking that the data accessed by a transaction have not been updated by other transactions, and it can be thought of as a way to attain a trade-off between eager and lazy conflict detection, since it can be performed at any point in the transaction. Although validation is more frequent in software transactional memory (STM) systems, we can also find it in HTM systems [15], where the conflict is detected eagerly but users can be notified whenever they ask for validation.

HTM system proposals can also be classified by the amount of transactional accesses they are able to track. This can be determined by either the conflict detection or the version management mechanism. Depending on whether or not HTM systems can cope with transactions of any duration and size, they can be classified into *unbounded* and *bounded* HTM systems. Bounded systems, also known as best-effort HTM systems, are able to deal with transactions that do not overflow their hardware resources or do not survive operating system events. Some of them burden programmers with the task of handling overflow events, which defeats a key TM motivation: reducing the difficulty of parallel programming. Conversely, unbounded systems deploy mechanisms to tackle transactions of any size and duration, thus facilitating the task of transactional programming.

In this chapter, we focus on hardware conflict detection from the point of view of bounded and unbounded HTM systems. Section 2 discusses the main HTM proposals with bounded conflict detection mechanisms. The recent approaches of main hardware manufacturers are also surveyed. Section 3 is devoted to the unbounded HTM system proposals and their conflict detection mechanisms, with special interest in signatures as the means to effective conflict detection virtualization. Finally, Section 4 draws the conclusions.

## 2 Bounded Conflict Detection

This section discusses several HTM systems that implements bounded conflict detection mechanisms. These *best-effort* systems execute transactions properly as long as certain events do not occur during the execution.

The main events that can abort transactions in a bounded HTM system, apart from conflicts, are those coming from the operating system (OS) and the ones caused by hardware overflow. In regard to OS events, virtual memory paging can cause the relocation of pages that contain transactional data, which means that the physical address of the data has changed and the conflict detection mechanism loses track of the locations accessed by a transaction. Also, context switches caused by descheduling or thread migration are difficult to manage if the transactional information is not visible to the OS. As far as hardware overflow is concerned, bounded HTM systems are not able to execute transactions whose data set (DS) is larger than the hardware structures used to hold it. Usually, the response to these events is to abort the transaction in the hope that they do not happen again. This can work in case of OS events. However, overflow events are likely to recur, thus risking livelock whenever a fall-back solution is not provided.

The remainder of this section deals with bounded conflict detection in bounded HTM systems that use the cache coherence protocol to enforce atomicity (Section 2.1). We also discuss those systems that use alternative methods to implement the conflict detection mechanism (Section 2.2). We then review the main approaches taken by hardware manufacturers (Section 2.3).

### 2.1 Leveraging the Cache Coherence Protocol

There are several bounded HTM system proposals in the literature that modify the cache coherence protocol to implement the conflict detection mechanism [2,15,19,27].

They usually implement eager conflict detection. Next, the most relevant characteristics of them are described.

Herlihy and Moss [15] were the first to propose a HTM implementation leveraging the cache coherence protocol to detect conflicts amongst transactions and ensure atomicity. Figure 1 depicts the hardware needed to implement their proposal. They use a *transactional cache* besides the private primary cache to keep track of the data accessed by transactions, both old and new values. New transactional states are added to those of the coherence protocol to indicate whether the entry is old or new. Data updates are performed over the new version. Old versions are discarded (invalidated) on commits, and new versions are invalidated on aborts. The transactional cache is fully associative and has additional logic to perform commits and aborts in a single cache cycle, as they assume a few entries are needed per transaction. The primary cache and the transactional cache are exclusive, so a location can only be in one of them at a time, and the coherence protocol probes both caches in parallel.

Herlihy and Moss modify an invalidation-based snoopy coherence protocol [13] to add three more messages related to transactional accesses. One of the new messages requests a location needed by a transactional load, the second one is for requesting a location needed by a transactional exclusive load or a transactional store, and the third message signals a conflict for requested transactional locations (busy message). When a transaction loads a location, its transactional cache is searched just in case the location was previously written by the same transaction. In case of a miss, a transactional load message is broadcast to check all the transactional caches. This is done in one cycle, as transactional caches are fully associative. If there is at least one hit in the transactional caches, a busy message is sent to the requesting core. Then, the requesting core sets a flag to false (aborted), indicating that the transaction conflicted and must abort. Subsequent transactional loads and stores of the conflicting transaction do not cause network traffic and may return arbitrary values. Therefore, conflict detection can be said to be eager, although the conflict does not resolve until the program executes a commit/abort/validate instruction that checks the abort flag.

The use of a fully-associative cache is an important implementation constraint due to its higher hardware requirements and the slower access time compared with other implementations. Also, a bus-based coherence protocol limits scalability.

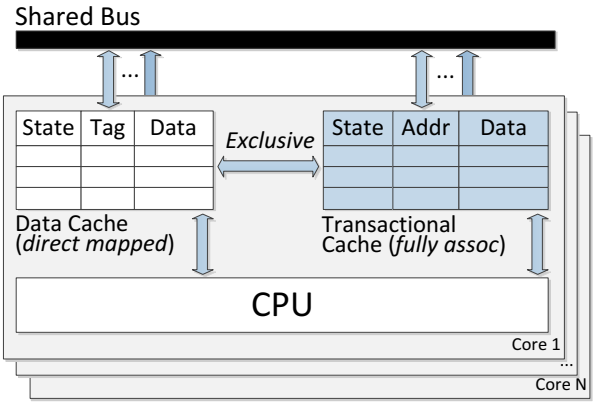


**Fig. 1.** Herlihy and Moss HTM implementation. Transactional data are stored in an exclusive transactional cache apart from the private primary cache. The snooping coherence protocol probes both caches in parallel on request. The cache coherence protocol is modified to detect conflicts eagerly.

Another approach to bounded conflict detection is that of Moore et al. [19]. They propose LogTM, a HTM system implementation devised to provide better scalability and to support larger transactions than the Herlihy and Moss proposal. LogTM virtualizes version management by using the concept of *before-image log* taken from the data base field, where old values are saved in a per-thread log in cacheable virtual memory. New values are kept in place and isolation is ensured by the coherence protocol. The log allows unbounded version management, unlike the fully-associative cache of Herlihy and Moss, but it has the disadvantage of slow aborts, as the log has to be *undone* to restore old versions. Nevertheless, LogTM is bounded because of the conflict detection mechanism.

Figure 2 shows the hardware needed to implement conflict detection in LogTM. Private caches are augmented with transactional read (R) and write (W) bits per cache block. These bits are set whenever a location is read or written by a transaction to keep track of its *read set* (RS) and *write set* (WS). The cache must support flash clear of these bits to reset them when a transaction commits. The system implements a directory that holds a bit vector of sharers per memory block so that the coherence protocol knows which cores share the block (multiple readers) or which one owns it (one writer). When a transaction running in core X requests a block that has been written by another transaction running in core A, the coherence protocol (by means of the directory) forwards the request to core A, which checks the block against its private caches. If the check is a hit and the W bit is set, then core A accessed the block transactionally. Therefore, core A sends a NACK message (like Herlihy's busy message) to core X, which has to manage the conflict.

LogTM deals with transaction overflow in a peculiar manner. A novel "sticky" state is defined for those blocks in the directory that were evicted from the second level cache (L2) during the execution of a transaction. Hence, subsequent requests for those blocks are still forwarded to that core. The core should check its caches on a forwarded request for an evicted block. However, the block is not in the caches anymore, so the core should respond with an ACK, which would result in an atomicity violation. The Overflow bit is used to avoid this situation. The bit is set whenever a transactional block is evicted, and

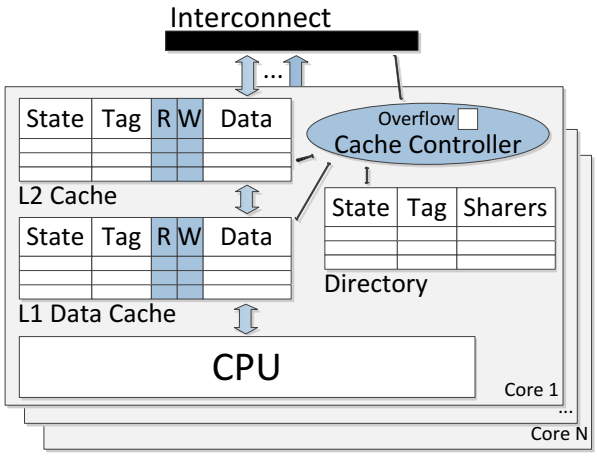


**Fig. 2.** LogTM implementation. Transactional state is supported by R/W bits per cache block. The directory is not updated when a cache block is evicted. Instead, the block state is changed to "sticky", and requests are still forwarded to the core ensuring atomicity. The cache controller's Overflow flag is set when transactional data are evicted.

the controller checks the bit when a block is not in the caches. If the Overflow bit is set, the core conservatively NACKs the requester. Sticky states are cleaned lazily once the transaction has committed. A forwarded request due to a stale sticky state is responded with a message to clean the sticky state as long as the Overflow bit of the core is cleared. Otherwise, false conflicts can arise because of overlapping of stale sticky states from an earlier transactions with an overflowed current one.

Although LogTM can cope with larger transactions it is still a bounded HTM system. OS events cannot be survived because conflict detection information is not persistent since it cannot be saved in a context change. Furthermore, LogTM does not resolve replacements of sticky blocks in the directory.

Having transactional bits or tags to label cache blocks that have been accessed by transactions, along with the modification of the cache coherence protocol to maintain atomicity are the main techniques when it comes to implementing the conflict detection mechanism in these bounded systems. Some approaches before LogTM have used these same techniques although with certain subtleties. Speculative lock elision (SLE) [27] associates an Access bit with each cache block that interacts with the coherence protocol. The difference is that SLE accepts code with locks as input, elides the lock and speculatively executes the critical section enclosed by the lock as though it is a transaction. In case of repeated speculation failure because of conflicts, the lock is acquired and progress guaranteed. Large transactional memory (LTM) [2] has a T bit per cache block to label transactional data. The cache coherence protocol uses NACK messages to hint conflicts. LTM is different from LogTM in the way it deals with cache overflows. Each cache set is extended with an overflow bit that is set when the block is evicted. Then, the block moves into an uncached hash table in memory that has to be traversed by the core on each request from other cores.

### 2.2 Alternatives to Cache Coherence Protocol Modification

Adding complexity to an already complex mechanism like the cache coherence protocol or to a fine-tuned structure like a cache memory could bring implementation issues. Some alternative implementations have been proposed to manage conflict detection without having to make major changes to the cache hierarchy. They are usually based on lazy conflict detection [7,14].

Transaction coherency and consistency (TCC) [14] is proposed to ease the design of chip multiprocessors by defining consistency and coherence at the granularity of transactions. Regarding consistency, all memory accesses from a core that commits earlier happen before the memory accesses of cores that commit later, regardless of if such accesses actually interleaved each other. The coherence protocol is also simplified since "shared" and "exclusive" states are not needed anymore. A block can be unmodified or modified in different cores at the same time, and coherency is enforced at transaction boundaries.

The TCC's conflict detection mechanism needs the deployment of the hardware structures shown in Figure 3. Although no extra complexity is added to the coherence protocol, the core private caches are modified to include transactional read (R) and modified (M) bits. Also, a write buffer is required to store modified data addresses. Thus, a transactional write stores the new value in the cache, sets its M bit and stores the address

in the write buffer. Transactional loads simply set the R bit. Once the transaction comes to commit, the core broadcasts the write buffer for the other cores to check it against their R bits (notice that WAR and WAW dependencies are filtered out). If a core finds a conflict, it invalidates its modified data by transferring (through NOT gates) the W bits to the valid bits (V) of the cache, thus keeping caches coherent. An alternative implementation broadcasts addresses and data to update other caches instead of invalidating their copies of the data. TCC also suggests an optimization by means of the Rename bit field that avoids false conflicts because of false sharing of cache blocks. It extends the M bit to each word or byte in the cache block.

Hammond et al.'s TCC conflict detection mechanism is bounded by the size of the cache and the write buffer. If these structures overflow, they propose to request commit permission, which ensures that all earlier transactions have committed and no other has begun, so there is no need to track transactional information anymore. However, this can be severely detrimental to the system performance. Also, the commit phase may suppose a bottleneck to scalability as addresses have to be broadcast one by one, or in packets. The network bandwidth requirements could increase dramatically, specially if data are transferred as well.

Qian et al. present OmniOrder [21], a lazy-lazy HTM system that keeps the cache coherence protocol untouched and conflict-serializes transactions to avoid unnecessary aborts. The history of transactional stores to a memory block is maintained in a per-processor fully associative buffer called speculative version buffer (SVB). The SVB's information for a memory block is moved piggybacked on coherence messages on each block's transition to the M state in a directory-based MESI coherence protocol [30]. From these transitions, each core figures out the processors that are executing predecessor and successor transactions to the one it is executing, and stores that information into bitmask registers. Thus, if processor P1 updates block B, the coherence protocol brings B to P1's cache an set the B's state to M. The new value of B is also stored in the P1's SVB. When another processor, P2, updates B, the unmodified coherence protocol moves B to P2's cache and invalidates the block in the P1's cache. P1 piggybacks the SVB's entry for B in the coherence message and P2 is now responsible for it. Also,

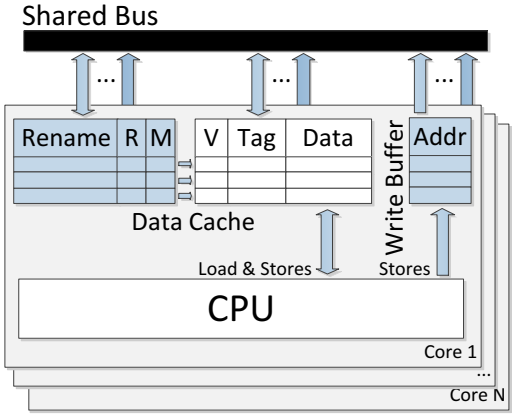


**Fig. 3.** TCC implementation. A write buffer is added to store the addresses of transactional modified data. Such a buffer is broadcast on commit for other cores to check it against their transactional read bits (R). In case of conflict, their modified data (M) is invalidated (V). Alternatively, the write buffer can be broadcast together with the modified values in order to update instead of invalidating.

P1 is marked as predecessor of P2 so that P2 must commit after P1 filtering out the WAW output dependence. In case of a cycle where a group of processors are both in the predecessor and the successor list of each other, one transaction must be aborted to break the cycle. On L1 cache evictions, OmniOrder aborts the transaction and restarts it in a conventional transactional mode that does not expose its transactional state to other transactions.

### 2.3 Hardware Manufacturers' Approaches

Hardware manufacturers include HTM support in their multiprocessors that is bounded and based on the cache hierarchy. Below we describe the main HTM extensions focusing on the implementation of the conflict detection mechanism.

Sun Microsystems' Rock multicore processor was the first production processor to include HTM support [8], although it was never distributed commercially as a result of Sun acquisition by Oracle. Each Rock core has hardware support to run two threads simultaneously. Rock implements a form of speculative threading that uses the second thread to execute the code whose data is not yet available because of long-latency instructions. Rock leverages the speculative threading hardware to support HTM. In addition, two new instructions have been added to the instruction set: `checkpoint fail-pc` to denote the beginning of a transaction, which accepts a pointer to compensating action code used in case of abort, and `commit` to denote the end of the transaction. Also, cache lines include a bit to mark lines as transactional. Stores within the transaction are placed in the store queue and sent to the L2 cache, which tracks conflicts with loads and stores from other threads. If the L2 cache detects a conflict, it reports the conflict to the core, which aborts the transaction. When the commit instruction begins, the L2 cache locks all lines being written by the transaction. Locked lines cannot be read or written by any other threads, thus ensuring atomicity. Rock's TM supports efficient execution of moderately sized transactions that fit within the hardware resources. However, a wide variety of events may abort a transaction: invalidation or replacement of cache lines marked as transactional, interrupts and processor exceptions, TLB misses, context switches, divide instructions, etc. These constraints make it difficult to predict and reason about why transactions abort, thus complicating parallel programming.

AMD Advanced Synchronization Facility (ASF) [11] is proposed as an eager-lazy AMD64 extension. ASF adds two bits per L1 cache line to mark read and written data inside a transaction. Besides, two queues are used to hold transactional loads and stores to guarantee a higher minimum transaction length. This is because a 4-way set-associative L1 cache implies a minimum transaction size of 4 different cache blocks, since a mapping miss in a set might cause a transaction to abort. With this design choice ASF reduces the unpredictable nature of transactions, unlike Rock's HTM. The AMD cache coherence protocol detects conflicts by checking the cache transactional bits on each forwarded coherence request. On commit, the cache bits are flash-cleared and the L1 cache is update with the data in the store queue. ASF is designed to coexist with an out-of-order processor design and it allows a transaction to survive branch mispredictions and TLB misses. Last but not least, programmers need to write software fallback code to deal with capacity overflows.

Intel has released its Transactional Synchronization Extensions (TSX) [28] on the multicore processor code-named *Haswell*. TSX provides two interfaces to denote transactional code. The first one is known as Hardware Lock Elision (HLE — similar to SLE described in Section 2.1), and involves two prefixes for instructions: `XACQUIRE` and `XRELEASE`. HLE is compatible with the conventional lock-based programming model. So, software written using the HLE prefixes can run on both legacy hardware without TSX and new hardware with TSX, since the prefixes correspond to the `REPNE`/`REPE` IA-32 prefixes which are ignored on the instructions where `XACQUIRE` and `XRELEASE` are valid. Thus, the programmer uses the `XACQUIRE` prefix in front of the instruction that is used to acquire the lock which is protecting the critical section. The processor treats the indication as a hint to elide the write associated with the lock acquire operation, and a transaction is started instead. If the transaction aborts, the processor will roll back the execution and then resume it non-transactionally. In case of a processor not supporting TSX, the lock is acquired normally, and the execution is serialized. The second interface provided by TSX is known as Restricted Transactional Memory (RTM) and allows more flexibility in transaction declaration than HLE. RTM adds three new instruction to the ISA: `XBEGIN`, `XEND` and `XABORT`. Intel does not provide implementation details of TSX, but gives some hints which suggest that TSX is a *best effort* approach to HTM, like Sun's Rock and AMD's ASF. That is, they do not guarantee successful execution of transactions of any size and duration, and they abort transactions that exceed on-chip resources for HTM, or encounter certain events like page faults, cache misses or interrupts. Thus, Intel enumerates a list of runtime events that may cause transactional execution to abort, namely, synchronous and asynchronous exceptions, memory operations other than write-back cacheable type operations, executing self-modifying code, excessive sizes for transactional regions, non-transactional requests to a cache line accessed within a transaction (*strong atomicity* [17] is ensured), and so on.

IBM BlueGene/Q hardware support for TM [33] leverages the L2 cache to implement the main transactional mechanisms. The shared L2 cache is 16-way set-associative and it is *multi-versioned*, to allow multiple versions (old and new speculative data) of the same physical memory block. Each L2 cache set guarantees up to 10 ways to be used for transactional writes, so it can handle a maximum transaction size of up to 20MB, out of the 32MB shared L2 cache. However, a transaction might be aborted with just 11 speculative blocks due to mapping misses. The L2 directory maintains read, write, and speculative bits per block of the cache, and it also holds a transaction ID to identify the transaction that read or wrote the block. There are 128 transaction IDs and a scrubbing process is executed every 132 cycles to recover IDs of aborted or committed transactions. The conflict detection mechanism uses the read/write bits of the directory to detect RAW, WAR and WAW conflicts among transactions through the cache coherence protocol. Also, a conflict is detected when non-transactional code writes a memory location that was previously accessed by a transaction (BlueGene/Q ensures strong atomicity). Transactional threads involved in a conflict are hardware interrupted and the conflict handler resolves the conflict. A special conflict register is set to indicate the cause of the conflict.

BlueGene/Q extends a pre-existing core design and therefore private L1 caches are not modified. To ensure forward progress without bothering the programmer, BlueGene/Q uses *irrevocability* [34], a special transactional mode that, once engaged, ensures transaction commit with the impossibility of being aborted. With the irrevocable mode, transactions can handle I/O irreversible operations, hardware overflows and other events. A runtime algorithm can make a transaction irrevocable after being aborted a fixed number of times. Also, if the aborting ratio for that transaction surpasses a threshold, subsequent executions will be performed in irrevocable mode after only one abort.

A different implementation approach to HTM has been used in IBM's System z mainframe computers with the microprocessor generation zEC12 [16]. Each IBM zEC12 chip has 6 cores with 2 levels of private caches that share a 3rd-level cache. Six of these Central Processing (CP) chips are connected to an off-chip 4th-level cache, thus forming a multi-chip module (MCM) with 36 cores. Up to 4 MCM's can form a coherent SMP system with up to 144 cores. Coherency is managed with a MESI protocol variant.

Unlike BlueGene/Q, System z chips implement HTM by leveraging the L1 private cache instead of the shared one. Figure 4 depicts the core organization with the transactional state highlighted. The L1 cache directory is augmented with two transactional state bits per cache line (tx-read and tx-dirty bits) with flash-clear support to reset all bits in one cycle on transaction commit. Also, tx-dirty bits are connected to the valid bits so that every transactional store can be flash-invalidated on aborts. L1 and L2 caches are store-through caches, so every store causes an L3 access. To hide L1 and L2 store miss latencies, the core has a store queue and a store cache respectively. Both buffers are augmented with a tx-dirty bit and are probed in parallel with the caches by the coherence protocol. In case of conflict, that is, an exclusive or demote (from exclusive to shared) coherence request is received, then the core rejects the request back to the sender which will repeat the coherence request. This mechanism, called *stiff-arming* [16] or stall [19], gives more time to the requested core in the hope of finishing its transaction. The number of rejects is determined by a counter that triggers a transaction abort when a threshold is exceeded. Thereby, deadlock is prevented.

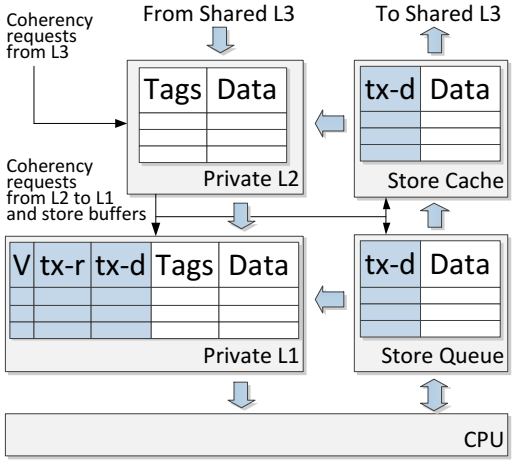


**Fig. 4.** IBM System z HTM implementation. The L1 cache and the store buffers (both the store queue and the store cache are used to hide the store miss latency) maintain the transactional state that comprises a tx-read bit (tx-r) and a tx-dirty bit (tx-d). The valid bit is tied to the tx-dirty bit to flush-invalidate cache entries in case of abort. The L1 cache and the buffers are probed in parallel on a coherency request.

IBM zEC12 processor's L1 cache is a 96KB cache organized in 64 sets with 6 ways and 256 byte lines. Its latency is 4 cycles. On the other hand, the private L2 cache is a 1MB 8-way associative cache with a 7 cycle L1 miss penalty. On abort, the tx-dirty lines in L1 are invalidated (new values), and the old values are very close in L2 at 7 cycle L1 miss penalty. In order for transactions not to be limited by L1 size and associativity, a special bit per L1 set is asserted whenever a transactional line is evicted from L1. Thus, transactional footprint capability is extended to L2 size and associativity without modifying L2, and to the store cache size, at the cost of false positives. The special eviction bits in the L1 cache do not store address information, so every coherence request for an address that maps to a set whose eviction bit is set will abort the transaction regardless of whether the address was transactional or not. Therefore, a false positive might occur. However, the system can track much larger transactions, specially on the read set. The write set is limited to the size of the store cache (64 x 128 bytes).

Finally, IBM has added a HTM facility to the POWER8 processor [1,6] from which few implementation-specific details have been revealed. Each POWER8 core has two data private caches, L1 and L2, and one bank of a larger shared L3 cache. Unlike BlueGene/Q and System z, the POWER8 processor keeps track of transactional state in the private L2 cache [1]. When the transaction commits, the new values stored in L2 are committed to the memory sub-system. POWER8 introduces the concept of *suspended transactional mode* [6] to allow for transactions to survive interruptions (context switches, hypervisor, debuggers,...). In this mode, memory accesses are performed non-transactionally and cannot be undone if the transaction eventually aborts. The initiation of a new transaction is prevented, and the hardware tracks conflicts with the transactional data of the suspended transaction. Stores to memory locations that were transactionally accessed by the same thread will abort the suspended transaction.

## 3 Unbounded Conflict Detection

Programming a bounded HTM system might become a difficult task if the hardware is overflowed persistently, and it can happen more frequently than expected. Table 1 shows the number of overflowed transactions and the average number of evicted blocks for the STAMP benchmark suite [18]. Those figures have been obtained from an *implicitly transactional* system, where only the boundaries of transactions have to be defined and all memory accesses within them are tracked[1], and 32KB L1D caches. As a result, none of the benchmarks would have been able to complete in a bounded HTM system that uses the primary cache to provide transactional support.

Increasing the size of caches does not always guarantee that the HTM system can handle larger transactions, since an eviction can happen because of mapping misses regardless of whether the cache is full or empty. Bounded HTM systems usually provide a fallback mechanism to tackle overflowing situations, which might involve the programmer. However, next we describe several unbounded HTM proposals that are able to handle transactions of arbitrary size and duration, even in the presence of OS events, without further programming effort.

[1] Conversely, *explicitly transactional* systems urge the programmer to explicitly identify transactional memory accesses.

**Table 1.** Number of transactions that overflow the L1D cache and the number of cache blocks replaced on average, both read and written within a transaction

| Benchmark | Overflowed Transactions | Average Number of Block Evictions | |
|---|---|---|---|
| | | Read | Written |
| Bayes | 102 | 68.2 | 100.8 |
| Genome | 447 | 78.7 | 1.8 |
| Intruder | 4511 | 2.1 | 0.1 |
| Kmeans | 387 | 1.0 | 0 |
| Labyrinth | 48 | 62.9 | 76.8 |
| Vacation | 2710 | 7 | 0.1 |
| Yada | 816 | 117.2 | 73.2 |

### 3.1 Persistent Meta-Data Systems

The unbounded HTM systems described in this section hold transactional meta-data (the information needed to perform conflict detection and version management) in virtual memory that persists hardware overflows and OS events.

Unbounded transactional memory (UTM) [2] holds, in virtual memory, a structure called XSTATE that represents the state of all transactions running in the system. Besides, each memory block is augmented with a transactional read/write bit and a pointer to the old value of the block that resides in an entry of the XSTATE structure. Such an entry of the XSTATE structure, in turn, has a pointer to the memory block. So, the XSTATE structure holds a linked list of memory blocks whose transactional read/write bits are set. Conflict detection is carried out eagerly, so every memory access operation must check the pointer and bits of the memory block to detect any conflict. The access to the XSTATE and memory block meta-data is done by means of several hardware registers that hold pointers to their base and bounds. For non-overflowed transactions, UTM implements a conventional cache-based HTM to accelerate execution, called LTM (see Section 2.1).

Virtual transactional memory (VTM) [26] assigns each transaction a status word (XSW), which is used to commit or abort the transaction by modifying it atomically with a CAS instruction. VTM also defines a transaction address data table (XADT), which is the shared log for holding overflowed transactional data. Both structures reside in the application's virtual address space. However, they are invisible to the user. The VTM system, implemented in either hardware or microcode, manages these structures by means of new registers added to each thread context that point to them and are initialized by the application. When a transaction issues a memory operation that is a cache miss, it must be checked against overflowed addresses by traversing the XADT. Traversing the XADT might be too slow, so VTM provides two mechanisms for not interfering with transactions that do not overflow. First, an XADT overflow counter records the number of overflowed entries. If it is set to zero, no traffic is needed as it is locally cached at each processor. Second, an XADT filter (XF), implemented as a software counting Bloom filter [12] that allows deletions, provides fast detection of conflicts. A miss in the filter guarantees that the address does not conflict, and a hit triggers an XADT walk.

TokenTM [4] is an unbounded, eager conflict detection HTM system that augments each memory block with transactional meta-data. As depicted in Figure 5, the meta-data consist of a Token and a thread identifier, TID. The system must comply with the following invariant for each memory block: a block can be non-transactional, part of the RS of one or more transactions, or part of the WS of only one transaction. Therefore, a block that is non-transactional will have 0 tokens, and the TID is not necessary. A block read by one transaction will have 1 token and the TID of the thread executing such a transaction. A block read by *n* transactions will have *n* tokens, and the TID is not necessary. And a block written by one transaction will have all the tokens, *T*, and the TID of the thread that issued the transactional write. So, if a transaction reads a block with (Token=T,TID=X) and the TID does not match its own TID (the TID is stored in a CPU register, see Figure 5) a conflict is detected with the thread X. Also, if a transaction writes a block with (1,Y), a conflict is detected with the thread Y. However, if a transaction writes a block with (n,-), the conflict is quickly detected, but the resolution can be costly, as the TIDs of the *n* sharers cannot be stored in the TID field. In this case, if all shared copies of the block were in the cache hierarchy, the coherence protocol would provide the TIDs of the conflicting transactions. Otherwise, the system has to traverse the thread logs, that hold old data versions and precise meta-data, in order to find that information. The coherence protocol is not modified except for piggybacking the transactional meta-data (Token, TID) in each coherence message. Then, to maintain meta-data coherency, as multiple copies of a block can coexist in the cache hierarchy, TokenTM defines simple rules to fission and fusion transactional meta-data.

As transactional meta-data is attached to each memory block, transactions can overflow the caches without losing transactional state. Also, conflict detection suffers no false conflicts unlike other unbounded proposals (see Section 3.2). TokenTM handles paging and context switches easily by initializing, saving, restoring meta-data, and flash-clearing/ORing meta-data in L1 cache. Finally, by means of a Fast Release (this CPU bit is set when none of the locations in the WS have been evicted, so Fast Release is safe), small transactions that fit in the cache can commit at full hardware speed, just by flash-clearing their tokens. Larger transactions must walk the log to reset all their tokens on commit.

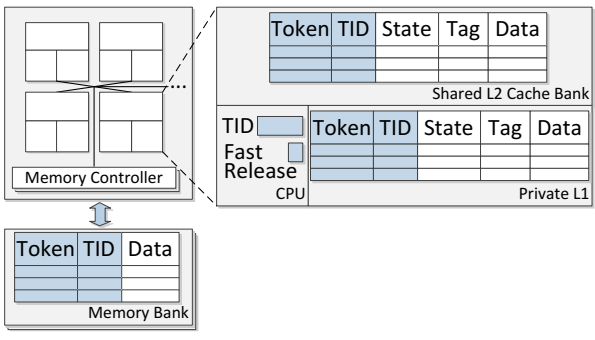


**Fig. 5.** TokenTM implementation. Each memory block is augmented with a field holding a token number, and another field for the thread ID, TID, of the transaction. Caches are also modified to hold such meta-data, but the coherence protocol is not modified. Meta-data is piggybacked on coherence messages.

### 3.2 Signature-Based Systems

Ceze et al. [7] propose Bulk as a mechanism to detach conflict detection from caches, and they manage to implement an unbounded conflict detection mechanism based on *signatures* that is able to track an indeterminate number of addresses and survive certain OS events like context switches.

Bulk is a lazy-lazy HTM system that presumes an invalidation-based coherence protocol that works unmodified when dealing with non-transactional data, and puts off invalidations until commit time when dealing with transactional data. Bulk is similar to TCC (Section 2.2), but the latter only supports transactional data, thus simplifying the cache coherence protocol specification. Unlike TCC, Bulk does not hold transactional state in primary caches. Instead, a Bulk Disambiguation Module (BDM) is defined per core which supports unbounded conflict detection.

Figure 6 shows the architecture of the BDM. The main part of the module consists of a set of signatures. *Signatures* are defined as Bloom filters (see Section 3.2.1), time and space-efficient hash structures that are implemented as a bit array and a set of hash functions. Such functions are a surjection between a larger set of addresses, the memory space, and a smaller set of indexes, the bit array, so the signature represents a superset of the real RS and WS of transactions. Hence, aliases or false positives can arise that do not compromise correctness but can hurt system performance as transactions get larger.

Bulk broadcasts fixed-sized signatures on commit for the other cores to invalidate stale data, just as TCC does, but with the difference that addresses are compacted in the signature instead of having a write set with individual addresses. The Bulk Functional Units implement operations, like signature intersection, to quickly perform the disambiguation of addresses. Thus, when a core receives the WS signature from other core that attempts to commit, the former intersects the received signature with its RS signature. If the result is not empty the conflict has to be resolved, so the receiver invalidates its modified data. To invalidate the speculatively written data, the BDM could walk the cache sets, retrieve the tags of valid entries and perform a membership query to its WS signature. This could be very inefficient if the number of valid lines is small. Instead, the BDM has a bit array (Current W Cache Sets) of length the number of sets in the cache, that holds the valid written sets of the cache and is calculated from the

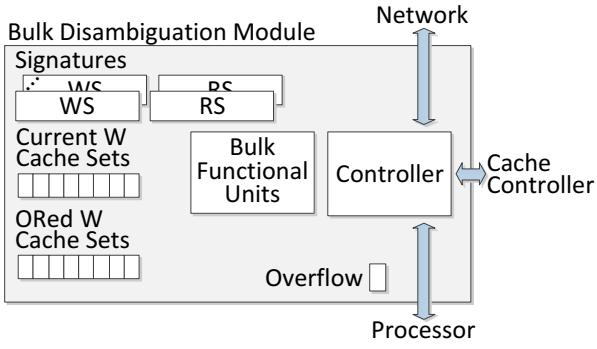


**Fig. 6.** Bulk Disambiguation Module (BDM) implementation. Bulk detaches transactional state from caches and defines the BDM to implement an unbounded conflict detection mechanism. It is based on signatures, time and space-efficient hash structures that are able to store an indefinite number of addresses at the cost of false positives.

WS signature with a decoding operation of the Bulk Functional Units. Invalidations are done sequentially, regardless of that optimization.

The BDM has a set of signatures to support context switches and to keep on detecting conflicts with a transaction that has been preempted. In case that the private caches evict a transactional block, the overflow bit is set. Checking for conflicts with evicted cache blocks does not necessarily imply traversing an overflow memory space as the information is in the signature. However, if the module runs out of signatures, the signature of one thread is moved to the overflow memory space and conflict detection is carried out like in VTM (see Section 3.1) until one transaction commits and clears one signature. There is another bit array, the ORed W Cache Sets, that stores the union of each written cache sets of every signature managed by the BDM, both current and preempted. These bit arrays also help to maintain the *set restriction* property introduced by Bulk, by which each cache set must only contain transactional or non-transactional blocks.

Although Bulk can be considered as unbounded from the conflict detection mechanism point of view, it does not clarify what happens on a page relocation or a thread migration.

LogTM-SE [35] is the unbounded extension of LogTM. Unlike Bulk, LogTM-SE is an eager-eager HTM system with an architecture that fully supports unbounded transactions that can survive thread migration, paging, context switches and transactions of indeterminate size. LogTM-SE stands for LogTM Signature Edition, so it is based on Bulk signatures, although the eager nature of the conflict detection mechanism simplifies the implementation.

The Bulk's BDM implements Bulk Functional Units that provide intersect, decode, and other operations to deal with address disambiguation. The BDM holds the cache sets of transactionally modified blocks and implements a finite state machine to invalidate those blocks on abort, as lazy conflict detection implies bulk disambiguation at commit time. However, LogTM-SE does not need such a complex hardware surrounding the signature since addresses are disambiguated individually and eagerly by the coherence protocol. When a core, A, reads a block within a transaction, the cache coherence protocol forwards the request to the owner of the block, B. Core B checks its WS signature and responds with an ACK or a NACK message depending on whether it was a miss or a hit in the signature. If the block is not in the cache hierarchy, it is fetched from main memory and a signature check is broadcast for the directory to rebuild the block state in cache. If a core hits its signature, its bit in the bit vector of sharers is set, and a conflict is signaled if the owner was not core A.

In order to support context switches and migrations, LogTM-SE proposes to add an additional hardware *summary* signature per thread context that holds the union of the signatures from all descheduled threads. In addition, the signatures are saved to the transaction's log header to be reinstalled in the normal signature when the thread is rescheduled. The summary signature is maintained by the OS in software, which is in charge of interrupting all threads of the same process to set their hardware summary signature to the global software summary signature. The hardware summary signature is checked before loads and stores reach the primary cache, so coherence requests do not have to check the summary signature because that early checking filters out conflicts

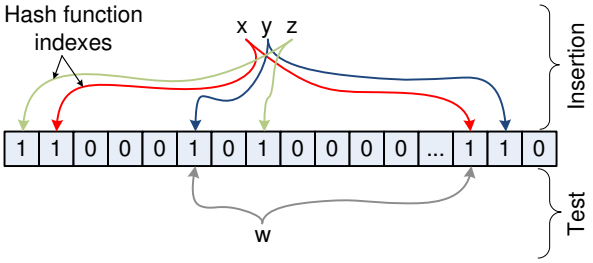


**Fig. 7.** Design of a Bloom filter. A false positive scenario.

with descheduled transactions. When a conflict is detected in the summary signature, the OS traps to a conflict handler since normal hardware conflict resolution is not valid as one of the conflicting transactions is not running in the system.

With virtual memory paging the problem lies in the fact that signatures operates on physical memory addresses, so if a page is relocated to a different address space, false negatives can arise risking incorrect conflict detection. LogTM-SE proposes to interrupt all threads to update their signatures whenever a page relocation occurs. This signature update consists in decoding the signature to get the addresses inserted in it and check if those addresses belonged to the old page. If so, the addresses are reinserted with the new page address.

LogTM-SE support for virtualization is costly and can be very detrimental to performance if large transactions become the norm. Otherwise, if these OS events are not frequent, the described mechanisms prove to be an effective solution.

### 3.2.1 Signature Implementation Enhancements

Signatures have been proved an effective mechanism to implement unbounded conflict detection. They are implemented as Bloom filters [3], proposed by Burton H. Bloom in 1970 as a time and space-efficient hash coding method with allowable errors. Figure 7 shows the design of a Bloom filter. It comprises a bit array of $2^m$ bits and $k$ different hash functions that map elements into $k$ randomly distributed bits of the array. Such an array is initially set to 0, and inserting an element into the filter consists in setting to 1 the $k$ bits indexed by the hash functions. Test for membership consists in checking that those $k$ bits are asserted. As the array is fixed-sized there exists the possibility of errors of testing, called *false positives*. For instance, in Figure 7, elements x, y and z are inserted in the filter and the bits indexed by the hash functions ($k = 2$ in this case) are set to 1. When we test for element w, it happens to be mapped into bits that have already been set to 1, so the test is a false positive. However, false negatives are not possible.

The probability of false positives rises as signature fills, and it might cause substantial performance degradation because of false conflicts or false contention. Figure 8 shows the probability of false positives for a signature implemented as a Bloom filter [3] with a 1Kbit array and different number of hash functions. The false positive rate is given by the equation:

$$p_{FP}(M,n,k) = \left(1 - \left(1 - \frac{1}{M}\right)^{nk}\right)^k \approx \left(1 - e^{-\frac{nk}{M}}\right)^k, \quad (1)$$

where $M$ is the signature size, $n$ the number of insertions and $k$ the number of hash functions. And $p_{FP}$ can be simplified by using the Taylor series expansion of the exponential function, $e^x = \sum_{n=0}^{\infty} \frac{1}{n!} x^n$ [29]. We can see that better false positive probability is expected for low populated filters and a high number of hash functions ($k \in \{4, 8\}$). However, the more hash functions the Bloom filter has, the earlier the filter populates and the higher the false positive probability is expected for high populated filters.

We can find manifold signature implementation proposals in the literature that try to enhance signature performance by reducing both the false positive rate and the hardware budget as well.

Bloom filter signatures can be implemented as a $k$-ported SRAM in its regular version. However, Sanchez et al. [29] proposed *parallel signatures* as an alternative hardware-efficient implementation to regular Bloom filter signatures. Multiported SRAMs require much hardware as they grow quadratically with the number of ports. Figure 9 shows the implementation of both regular and parallel filters. Whereas the regular filter is implemented as a $k$-ported SRAM, the parallel one consists of $k$ subfilters implemented as single-ported SRAMs, yielding the same or better false positive rate.

Cuckoo-Bloom signatures are also proposed in [29]. They are intended to perform like high-$k$ Bloom filters for small transactions, while yielding the false positive rate of Bloom filters with few hash functions when transactions are large, i.e. Cuckoo-Bloom signatures try to get the lowest false positive rate in each situation. Cuckoo-Bloom filters act like a hash table at the beginning of the transaction. Addresses are stored as if in a set-associative cache, where tags and data are the result of hashing the address with two independent hash functions, and sets are indexed by other hash function. When a set is full, the filter executes a sequence of evictions and re-insertions to store the incoming address. If such a sequence takes too long, the set is converted into a regular Bloom filter with low $k$, after storing the addresses into a separate storage space. Then such addresses are hashed into the newly converted Bloom filter. Lookups are fast, but insertions are more complicate, and the filter needs certain control logic, additional storage, a bit array to signal whether a set has been converted into a Bloom filter or not, and other structures (comparators,...) that complicate the design and might rise the hardware budget.

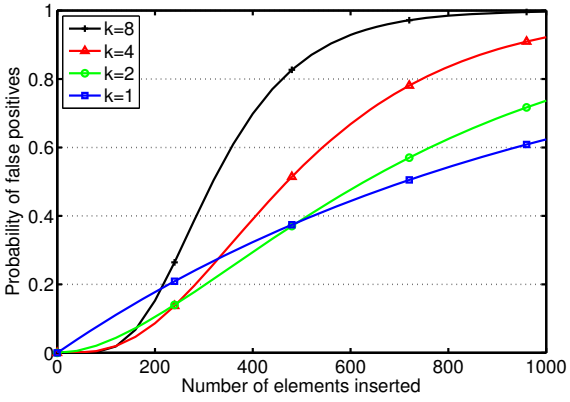


**Fig. 8.** False positive probability of signatures implemented as regular Bloom filters. The signature's bit array is 1024bit length and the number of hash functions $k \in \{1, 2, 4, 8\}$.

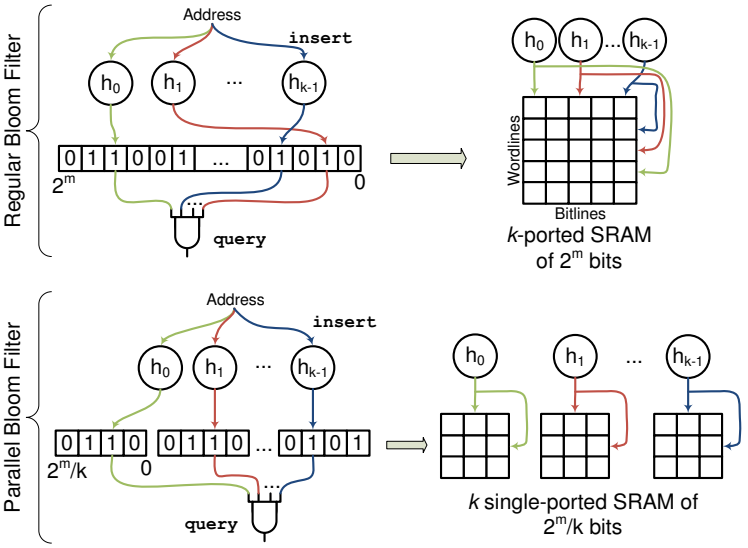


**Fig. 9.** Regular Bloom filter vs. parallel Bloom filter design and implementation. The bit array is implemented as a bidimensional SRAM where the most significant bits of the hash index select the wordline and the less significant bits select the bitline within the word.

An alternative hardware-efficient implementation of hash functions, Page-Block-XOR hashing (PBX), is proposed in [36]. They use the concept of entropy to find the highest randomness bits of the address, to allow reducing the hardware complexity of hash functions, that are usually implemented as a tree of XOR gates per hash index bit. PBX compacts those trees to a single XOR gate per hash bit, although it requires a profiling of the application to find the most random address bits. Notary [36] also proposes a technique to reduce the number of asserted bits in the signature. Their approach is based on segregating addresses into private and shared sets. Then, only the shared addresses are recorded. This solution requires support at the compiler, runtime/library and operating system levels. In addition, the programmer must define which objects are private or shared, which might be a difficult and error-prone task.

Titos et al. [32] propose a directory-based scheme for detection of conflicts in HTM. They detach conflict detection from the L1 caches and shift it to the directory level. This approach optimizes eager conflict detection HTM systems with an unordered and scalable network, when running applications with high number of conflicts. The network traffic is reduced up to 30% since the directory does not have to send signature check messages to the cores. Furthermore, transactional bookkeeping is more efficient since transactional information is globally encoded into the directory instead of having a local signature per core. Transactions usually access the same shared data which is not kept redundantly into the directory. The main disadvantage of this proposal is that hits on private caches do not go through the directory. A message has to be sent out to notify the directory of transactional loads and stores that hit private caches. The problem is that the critical path of a private cache cannot be slowed down by an access to

the directory, so the communication between cache and directory is set asynchronously, thus introducing races in conflict detection.

Orosa et al. [20] propose *FlexSig* as a flexible hardware signature implementation to change dynamically the amount of signatures per core according to system requirements. FlexSig groups all signatures in the system into a pool of signatures and assigns them to the cores on demand. It relies in the fact that all cores are not always running transactional code at the same time. Thus, if there are only two transactions running in the system, they will use half of the signature pool each. If other cores start a transaction, they demand signature allocation to the pool and it is repartitioned to meet the necessities of all the cores running transactions in the system, without incurring false positives.

Choi and Draper [9] propose adaptive grain signatures, that keep the history of transaction aborts and dynamically changes the input bit range to the hash functions on the abort history. The aim of this design is to reduce the number of false positives that harm the execution performance.

Quislant et al. [22,25] propose locality-sensitive signatures, LS-Sig, that exploit the spatial locality property of memory references to reduce the probability of false conflicts. LS-Sig defines new maps for hash functions to reduce the number of bits inserted in the filter (occupancy) for those addresses with spatial locality. That is, nearby memory locations share some bits of the Bloom filter. As a result, false conflicts are significantly reduced in transactions that exhibit spatial locality in their read or write sets, but the false conflict rate remains unalterable for transactions that do not exhibit locality at all. This is favorable particularly for large transactions that usually present a significant amount of spatial locality. In addition, as the proposal is based on new locality-aware hash maps, its implementation does not require extra hardware.

The probability of false positives for LS-Sig can be expressed as follows:

$$p_{FP}^{local}(M,n,k) = \left(1 - \left(1 - \frac{1}{M}\right)^{n\sum_{t=1}^{k} t f_t}\right)^{k}, \tag{2}$$

where the exponent $nk$ of Equation 1, which stands for the number of bits of the array that are set after $n$ insertions, is replaced by $n\sum_{t=1}^{k} t f_t$. Now, inserting an address in the filter does not necessarily set $k$ bits as fewer bits can be set depending on locality. $f_t$ is the probability that an insertion only sets $t$ bits in the filter because a nearby address was already inserted.

Figure 10 shows the analytical evaluation of false positive probability for the generic Bloom filter given by Equation 1 with several $k$ values, and the proposed LS-Sig scheme (Equation 2) for $k = 4$. To parameterize the evaluation, $f = \sum_{t=1}^{k-1} f_t$ was introduced as the probability of an address being near to some inserted address. Consequently, $1 - f = f_k$ is the probability of being far from those already in the filter. With a generic Bloom filter low values of $k$ are advantageous for large transactions and high values of $k$ for small ones. However, it can be inferred from Figure 10 that the LS-Sig scheme can achieve the benefits of both situations if the address sequence exhibits medium/high spatial locality.

Unified [10], Multiset and Asymmetric [23,24] signatures are proposed to deal with asymmetry in transactional data sets. Read and write signatures are usually

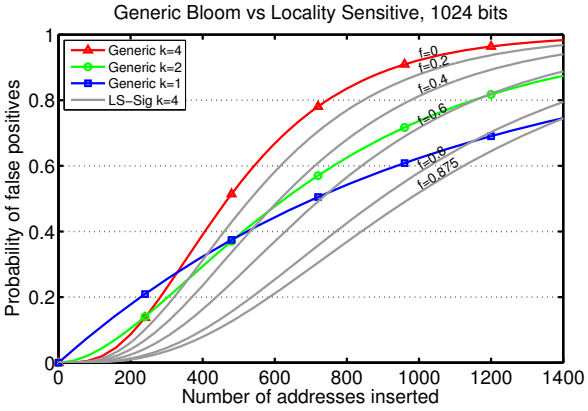


**Fig. 10.** Probability of false positives of generic and LS-Sig varying the parameter $f = \sum_{t=1}^{3} f_t$ (the higher the $f$, the more the locality).

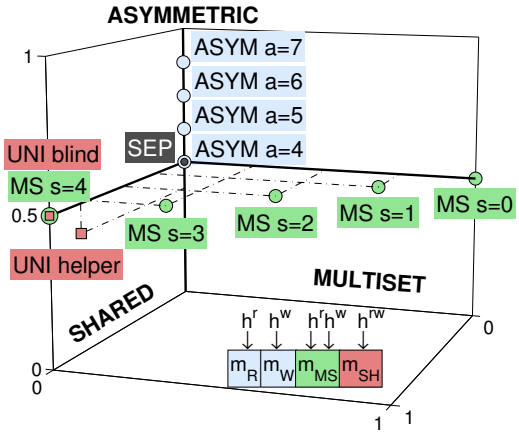


**Fig. 11.** Unified (UNI), Multiset Shared (MS s) and Asymmetric (ASYM) signature configurations studied in [10,23]. UNI blind is the same as MS s=4 (the number of hash functions is set to 4 and s=4 means that all hashes are shared between the RS and the WS). ASYM a=7 devotes 7 subfilters to the RS and 1 to the WS. SEP is the conventional separate parallel signature proposed in [29].

implemented as two separate, same-sized Bloom filters. In contrast, transactions frequently exhibit read and write sets of uneven cardinality. In addition, both sets are not disjoint, as data can be read and also written. This mismatch between data sets and hardware storage introduces inefficiencies in the use of signatures that have some impact on performance, as, for example, read signatures may populate earlier than write ones, increasing the expected false positive rate.

Figure 11 shows all the signature configurations explored in [10,23]. There are three orthogonal axes: asymmetric, shared or unified, and multiset. Asymmetric signatures are implemented using parallel Bloom filters, where the number of subfilters devoted to the RS and the WS can be configured via a reconfiguration register that can be set by a new instruction of the ISA or by the HTM system. A profiled RS to WS ratio can be computed for each application to configure the asymmetric signature. Multiset signatures merge RS and WS bit arrays into a common array while keeping their hash functions separate one another. However, sharing/unifying the hash functions of the RS and the WS is also proposed and it proves to be a good and general solution to the problem of asymmetry in data sets. Shared/Unified signatures have the problem of

introducing read-read dependencies, since they share all hash functions so they cannot distinguish between read and written locations. In [10], it is proposed to augment the signature with an extra register to filter out read-read dependencies, called the helper register, where only writes are stored. The same helper register effect is achieved with multiset shared signatures by segregating one hash function per set while sharing the rest (the number of hash functions is assumed greater than one). Last but not least, a study of the different combinations of multiset shared signatures with LS-Sig is carried out in [23].

## 4 Conclusions

The conflict detection mechanism is a key element in the design and implementation of a HTM system, as it is the means to attain atomicity while providing optimistic parallelism. In this chapter, we have surveyed the main approaches to hardware conflict detection implementation and we have classified them into two big groups: bounded and unbounded.

Whereas unbounded HTM conflict detection mechanisms release the programmers from worrying about HTM limitations and restrictions, they may require a significant multicore architecture modification that could compromise overall system performance. Signature-based proposals try to keep the hardware simple but suffers from false-positives on conflict detection that can be detrimental for the performance.

On the other hand, bounded HTM conflict detection is more feasible from the point of view of the hardware design and implementation. Several approaches have been explored that either leverage the cache hierarchy or use alternative implementation solutions. Hardware manufactures have adopted this bounded approach and some of them are releasing commercial processors with bounded HTM support. However, these HTM extensions could fail to comply with one of the main features that transactional memory systems claim to deliver, i.e. simplifying concurrent programming. Thus, effective unbounded HTM systems, and unbounded conflict detection in particular, could help to ease multicore processor programming so that transactional memory becomes the paradigm to use.

**Acknowledgement.** This work has been supported by the Government of Spain with project CICYT TIN2010-16144.

## References

1. Adir, A., Goodman, D., Hershcovich, D., Hershkovitz, O., Hickerson, B., Holtz, K., Kadry, W., Koyfman, A., Ludden, J., Meissner, C., Nahir, A., Pratt, R.R., Schiffli, M., St. Onge, B., Thompto, B., Tsanko, E., Ziv, A.: Verification of Transactional Memory in Power8. In: 51st Ann. Design Automation Conference (DAC 2014), pp. 1–6 (2014)
2. Ananian, C., Asanovic, K., Kuszmaul, B., Leiserson, C., Lie, S.: Unbounded transactional memory. In: 11th Int'l. Symp. on High-Performance Computer Architecture (HPCA 2005), pp. 316–327 (2005)

3. Bloom, B.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (1970)
4. Bobba, J., Goyal, N., Hill, M., Swift, M., Wood, D.: TokenTM: Efficient execution of large transactions with hardware transactional memory. In: 35th Ann. Int'l. Symp. on Computer Architecture (ISCA 2008), pp. 127–138 (2008)
5. Bobba, J., Moore, K.E., Volos, H., Yen, L., Hill, M.D., Swift, M.M., Wood, D.A.: Performance pathologies in hardware transactional memory. In: 34th Ann. Int'l. Symp. on Computer Architecture (ISCA 2007), pp. 81–91 (2007)
6. Cain, H.W., Michael, M.M., Frey, B., May, C., Williams, D., Le, H.: Robust architectural support for transactional memory in the power architecture. In: 40th Ann. Int'l. Symp. on Computer Architecture (ISCA 2013), pp. 225–236 (2013)
7. Ceze, L., Tuck, J., Torrellas, J., Cascaval, C.: Bulk disambiguation of speculative threads in multiprocessors. In: 33th Ann. Int'l. Symp. on Computer Architecture (ISCA 2006), pp. 227–238 (2006)
8. Chaudhry, S., Cypher, R., Ekman, M., Karlsson, M., Landin, A., Yip, S., Zeffer, H., Tremblay, M.: Rock: A high-performance sparc cmt processor. IEEE Micro 29(2), 6–16 (2009)
9. Choi, W., Draper, J.: Locality-aware adaptive grain signatures for transactional memories. In: IEEE Int'l. Symp. on Parallel and Distributed Processing (IPDPS 2010), pp. 1–10 (2010)
10. Choi, W., Draper, J.: Unified signatures for improving performance in transactional memory. In: IEEE Int'l. Parallel Distributed Processing Symp. (IPDPS 2011), pp. 817–827 (May 2011)
11. Chung, J., Yen, L., Diestelhorst, S., Pohlack, M., Hohmuth, M., Christie, D., Grossman, D.: Asf: Amd64 extension for lock-free data structures and transactional memory. In: 43rd Ann. Int'l. Symp. on Microarchitecture (MICRO 43), pp. 39–50 (2010)
12. Fan, L., Cao, P., Almeida, J., Broder, A.: Summary cache: A scalable wide-area web cache sharing protocol. IEEE/ACM Trans. on Networking 8(3), 281–293 (2000)
13. Goodman, J.R.: Using cache memory to reduce processor-memory traffic. In: 10th Ann. Int'l. Symp. on Computer Architecture (ISCA 1983), pp. 124–131 (1983)
14. Hammond, L., Wong, V., Chen, M., Carlstrom, B., Davis, J., Hertzberg, B., Prabhu, M., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional memory coherence and consistency. In: 31th Ann. Int'l. Symp. on Computer Architecture (ISCA 2004), pp. 102–113 (2004)
15. Herlihy, M., Moss, J.: Transactional memory: Architectural support for lock-free data structures. In: 20th Ann. Int'l. Symp. on Computer Architecture (ISCA 1993), pp. 289–300 (1993)
16. Jacobi, C., Slegel, T., Greiner, D.: Transactional memory architecture and implementation for ibm system z. In: 45th Ann. Int'l Symp. on Microarchitecture (MICRO 45), pp. 25–36 (2012)
17. Martin, M., Blundell, C., Lewis, E.: Subtleties of transactional memory atomicity semantics. IEEE Computer Architecture Letters 5(2), 17–20 (2006)
18. Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing. In: IEEE Int'l Symp. on Workload Characterization (IISWC 2008), pp. 35–46 (2008)
19. Moore, K., Bobba, J., Moravan, M., Hill, M., Wood, D.: LogTM: Log-based transactional memory. In: 12th Int'l. Symp. on High-Performance Computer Architecture (HPCA 2006), pp. 254–265 (2006)
20. Orosa, L., Antelo, E., Bruguera, J.: FlexSig: Implementing flexible hardware signatures. ACM Trans. on Architecture and Code Optimization 8(4), 30:1–30:20 (2012)
21. Qian, X., Sahelices, B., Torrellas, J.: Omniorder: Directory-based conflict serialization of transactions. In: 41th Ann. Int'l. Symp. on Computer Architecture (ISCA 2014) (2014)
22. Quislant, R., Gutierrez, E., Plata, O., Zapata, E.L.: Improving signatures by locality exploitation for transactional memory. In: Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT 2009), pp. 303–312 (2009)

23. Quislant, R., Gutierrez, E., Plata, O., Zapata, E.L.: Hardware signature designs to deal with asymmetry in transactional data sets. IEEE Trans. on Parallel and Distributed Systems 24(3), 506–519 (2013)
24. Quislant, R., Gutierrez, E., Plata, O., Zapata, E.L.: Multiset signatures for transactional memory. In: Int'l. Conf. on Supercomputing (ICS 2011), pp. 43–52 (2011)
25. Quislant, R., Gutierrez, E., Plata, O., Zapata, E.L.: LS-Sig: Locality-sensitive signatures for transactional memory. IEEE Trans. on Computers 62(2), 322–335 (2013)
26. Rajwar, R., Herlihy, M., Lai, K.: Virtualizing transactional memory. In: 32th Ann. Int'l. Symp. on Computer Architecture (ISCA 2005), pp. 494–505 (2005)
27. Rajwar, R., Goodman, J.R.: Speculative lock elision: Enabling highly concurrent multithreaded execution. In: 34th Ann. Int'l. Symp. on Microarchitecture (MICRO 34), pp. 294–305 (2001)
28. Reinders, J.: Transactional synchronization in Haswell. Intel's software blogs (2012), http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/
29. Sanchez, D., Yen, L., Hill, M., Sankaralingam, K.: Implementing signatures for transactional memory. In: 40th Ann. Int'l Symp. on Microarchitecture (MICRO 2007), pp. 123–133 (2007)
30. Sorin, D.J., Hill, M.D., Wood, D.A.: A Primer on Memory Consistency and Cache Coherence, 1st edn. Morgan & Claypool Publishers (2011)
31. Sorin, D.J., Plakal, M., Condon, A.E., Hill, M.D., Martin, M.M.K., Wood, D.A.: Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. IEEE Trans. Parallel and Distributed Systems 13(6), 556–578 (2002)
32. Titos, R., Acacio, M.E., García, J.M.: Directory-based conflict detection in hardware transactional memory. In: Sadayappan, P., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2008. LNCS, vol. 5374, pp. 541–554. Springer, Heidelberg (2008)
33. Wang, A., Gaudet, M., Wu, P., Amaral, J.N., Ohmacht, M., Barton, C., Silvera, R., Michael, M.: Evaluation of Blue Gene/Q hardware support for transactional memories. In: 21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT 2012), pp. 127–136 (2012)
34. Welc, A., Bratin, S., Adl-Tabatabai, A.R.: Irrevocable transactions and their applications. In: 20th Symp. on Parallelism in Algorithms and Architectures (SPAA 2008), pp. 285–296 (June 2008)
35. Yen, L., Bobba, J., Marty, M., Moore, K., Volos, H., Hill, M., Swift, M., Wood, D.: LogTM-SE: Decoupling hardware transactional memory from caches. In: 13th Int'l. Symp. on High-Performance Computer Architecture (HPCA 2007), pp. 261–272 (2007)
36. Yen, L., Draper, S., Hill, M.: Notary: Hardware techniques to enhance signatures. In: 41st Ann. Int'l Symp. on Microarchitecture (MICRO 2008), pp. 234–245 (2008)