Efficient Eager Management of Conflicts for Scalable Hardware Transactional Memory

Rubén Titos-Gil, *Member*, *IEEE*, Manuel E. Acacio, *Member*, *IEEE*, and José M. García, *Member*, *IEEE*

Abstract—The efficient management of conflicts among concurrent transactions constitutes a key aspect that hardware transactional memory (HTM) systems must achieve. Scalable HTM proposals so far inherit the *cache-based* style of conflict detection typically found in bus-based systems, largely unaware of the interactions between transactions and directory coherence. In this paper, we demonstrate that the traditional approach of detecting conflicts at the private cache levels is inefficient when used in the context of a directory protocol. We find that the use of the directory as a mere router of coherence requests restricts the throughput of conflict detection, and show how it becomes a bottleneck under high contention. This paper proposes a scheme for conflict detection that decouples conflict detection from cache coherence in order to overcome pathological situations that degrade the performance of an eager HTM system. Our scheme places bookkeeping metadata at the directory, introducing it as a separate hardware module that leaves the coherence protocol unmodified. In comparison to a state-of-the-art eager HTM system, our design handles contention more efficiently, minimizes the performance degradation of false positives for signatures of similar hardware cost, and reduces the network traffic generated.

Index Terms—Parallel programming, multicore architectures, cache coherence protocols, transactional memory, conflict detection

1 Introduction

The rise of multicores has brought the problem of concurrent programming to the forefront of computing research, presenting both immense opportunities and enormous challenges. Traditional multithreaded programming models use low-level primitives such as locks to guarantee mutual exclusion and protect shared data. The tradeoff between programming ease and performance imposed by locks remains one of the key challenges to programmers and computer architects of the multicore era. Transactional Memory (TM) [11], [12] has been proposed as a conceptually simpler programming model that can help boost developer productivity by eliminating the complex task of reasoning about the intricacies of fine-grained locking. By using transactions, programmers need not reason about the safety of interleavings or the possibility of deadlocks to write correct code. The underlying TM system attempts to make best use of available concurrency in the application while guaranteeing the properties of atomicity and isolation.

Hardware TM (HTM) systems propose architectural extensions to support TM and leverage existing architectural features like caches and coherence protocols to achieve high performance. The efficient management of conflicts among concurrent transactions is a key aspect, and its policy constitutes a major design dimension of the TM system. The choice of one or other policy—eager or lazy—has broad implications in both performance and implementation costs of adding TM support.

The check for conflicts can be done on each individual memory request [16], [27] or it can be deferred until the end of the transaction [6], [9]. Though the latter approach opens up more opportunities for parallelism, the efficient implementation of lazy commits is far from straightforward [7], [17], [19], [24]. A more evolutionary step toward the adoption of hardware support for TM is to detect and resolve conflicts eagerly. The strategy of booking resources a priori both greatly simplifies commits and leaves the cache controller largely unmodified. Though this makes eager systems inherently less efficient at exploiting parallelism than their lazy opponents, their lower implementation complexity still makes eager solutions an appealing choice.

One common aspect of all eager HTMs proposed so far is that they store the transactional bookkeeping information in structures that are private to the processor running the transaction. The transactional metadata is kept in places that are directly accessible to the private cache controller, which is conveniently modified to use it for detecting data races. This placement makes the most sense when private caches are able to snoop on every memory reference that takes place across the system, as it naturally happens in bus-based systems [10]. Despite the substantially different scenario found in systems with unordered, point-to-point interconnects—such as tiled CMPs—eager conflict detection schemes for directory-based HTMs have so far implicitly inherited the same style of *cache-level* conflict detection [16], [27].

However, the particular characteristics of directory coherence have not been thoroughly analyzed in the context

- *R. Titos-Gil is with the Department of Computer Science and Engineering, Chalmers University of Technology, Rännvägen 6, 41296 Göteborg, Sweden. E-mail: ruben.titos@chalmers.se.*
- *M.E. Acacio and J.M. García are with the Departamento de Ingeniería y Tecnología. de Computadores, Facultad de Informatica, Universidad de Murcia, Campus de Espinardo, 30100 Murcia, Spain. E-mail: {meacacio, jmgarcia}@ditec.um.es.*

Manuscript received 25 Oct. 2011; revised 7 Mar. 2012; accepted 8 Mar. 2012; published online 13 Mar. 2012.
Recommended for acceptance by D. Kaeli.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2011-10-0788.
Digital Object Identifier no. 10.1109/TPDS.2012.103.

of an HTM design. In fact, the literature has not yet addressed a major source of inefficiency that arises as a consequence of the directory's obliviousness to transactions. Using the directory as a mere router that simply forwards messages to the appropriate destinations, has the implicit effect of restricting the throughput of conflict detection to the pace at which the directory can process coherence requests. This means that an HTM system can only resolve conflicts as fast as its directory can process the coherence requests that originate such races. Since the directory acts as the serialization point for the requests to a memory location, it cannot process new requests until it receives confirmation that the previous one has completed. Although this limitation in concurrency affects directory coherency in general, the situation is aggravated to the point of pathological performance when transactions are introduced.

In this work, we propose a novel approach to eager HTM design that extends the directory logic in order to provide fast and efficient conflict management and transactional bookkeeping in tiled CMP architectures. Where previous proposals had combined these two related but distinct roles [16], [27], we propose to decouple conflict detection from cache coherence at the directory level, in order to overcome situations that degrade the performance of an eager HTM system. We demonstrate that traditional *cache-based* conflict detection introduces several sources of inefficiency when used in the context of a directory protocol, and show how under situations of high contention the directory becomes a bottleneck for the conflict detection mechanism. We propose an alternative solution, a *directory-based* scheme, which places transactional bookkeeping at the directory. By comparing our proposal with an HTM system such as LogTM-SE [27], we observe several advantages.

The main contribution of our design is that it substantially increases the conflict resolution throughput of directory-based HTMs, greatly improving the ability of eager systems to cope with contention. The detection of conflicts is accelerated—as our scheme detects conflicts in one hop instead of two—and the number of messages generated for the task is reduced—which is particularly important if the resolution is a simple *stall-and-retry* approach. While detecting sooner does not imply that dependent transactions will serialize quicker, our design dispatches conflicting requests without forwarding additional coherence messages, and thus without blocking the directory. This enables faster reaction to high-contention scenarios in which the same line is accessed by several conflicting transactions, and it has the potential to avoid many aborts and improve performance.

As a second contribution, our proposal stores the transactional bookkeeping information more efficiently, reducing the amount of false positives and consequent performance degradation, when compared to signature-based HTM schemes. We find that having each tile track its transactional addresses is not an efficient global encoding, as transactions often access the same shared data and thus keep redundant metadata on their read and write sets, which in turn may increase false positives when Bloom filters are used for bookkeeping. In contrast, our scheme extends the role of the directory not only to map addresses with cache residence, but also with transactional ownership. In addition, the new functionality is introduced as a separate hardware module that acts as a directory-level conflict controller that works independently from the coherence controller, thus leaving the coherence protocol unmodified.

A first approximation to the advantages offered by directory-based conflict detection in distributed shared-memory multiprocessors was presented in [23]. Here, we extend that work with the following contributions:

- We have modified the design to reduce the overhead of the directory metadata. *Transaction serial numbers* are no longer used to maintain the correspondence between each block and its owner transaction(s).
- We have also introduced a *conflict signature* that acts as a filter for noncontended addresses, in order to decrease the traffic due to metadata propagation.
- The design has been evaluated more comprehensively, with the inclusion of additional benchmarks from STAMP, and the comparison against new configurations with both perfect and real signatures.

2 Motivation

In this section, we discuss a number of reasons that support our claim that the directory is a well-suited location for the detection of transactional conflicts in HTM systems that detect and resolve conflicts eagerly.

2.1 Decoupling Conflict Detection from Coherence

Maintaining coherence means guaranteeing that all processors see the writes to a given location as having happened in the same order. In a bus-based system, all accesses to any location are serialized by the order in which requests appear on the bus. In a distributed system with coherent caching, it is the directory that acts as the serialization point for all the requests to the same memory location, since all relevant operations first come to the home tile. A common solution to ensure serialization to a location while keeping the complexity of the protocol low is to use additional directory states called *busy* or *transient* states [8].

When a line is in a busy coherence state, subsequent requests that target the same block must wait until an *unblock* message is received from the last requestor, indicating the (perhaps unsuccessful) completion of the previous coherence transaction. Only when the line returns to a *base* state is the next queued request considered. The result of this serialization is that coherence requests targeted to the same line can pile up in the input buffers of the directory controller when a cache line experiences high contention. The situation is much worse in the context of transactions, because the directory ignores whether a given request is conflicting, and is unaware of the priority scheme used by transactions. Oblivious to the status of the transactions running on a given moment, the directory attends messages in a first-in-first-out (FIFO) basis. As a consequence, undesired scenarios may arise during high contention: low priority requests are serviced while high priority ones are sitting at the input buffers, when indeed the former do not produce in any useful work since their transactions will probably end up aborting as a result of the conflict. The scenario is depicted in the left part of Fig. 1. The figure shows a transactional interleaving in which four transactions access the same cache

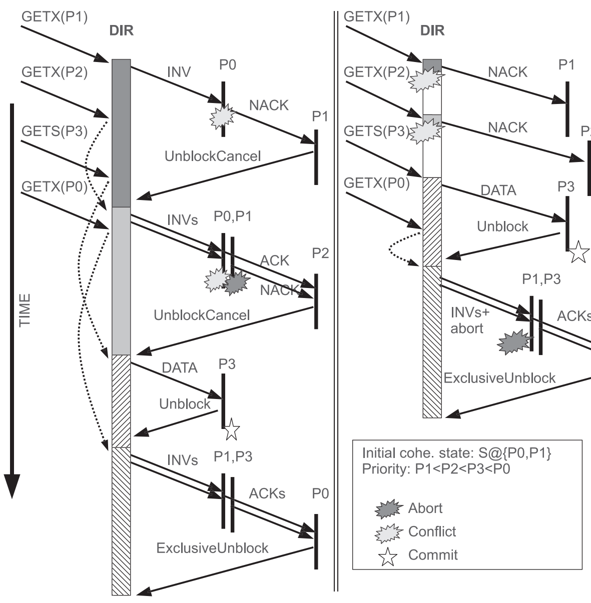

Fig. 1. Cache-based versus directory-based detection.

line, which are initially in the read set of transactions running in processors P0 and P1, whose caches have shared copies of the line. We can see how a nonconflicting shared request from P3 must wait for other conflicting requests that arrived earlier at the directory, delaying unnecessarily the commit of the reader transaction. Furthermore, the directory's obliviousness to time stamps makes the high-priority request from P0 wait for a lower priority request from P2, which does not succeed in acquiring exclusive ownership.

In contrast to the behavior of the cache-based approach to conflict detection shown in left part of Fig. 1, the directory could handle contention more efficiently, provided it had information about the transaction's time stamp and read and write sets. In the right part of Fig. 1, we see how a directory-based scheme of conflict detection serializes more quickly the same transactional interleaving described earlier. In this other case, the directory detects the conflict itself using its own bookkeeping metadata, and thus does not need to forward requests to the private caches that have a copy of the line. As a result, the line does not need to go into a busy state and nonconflicting or high-priority requests can be dispatched without having to wait for the completion of coherence requests known not to make any useful work.

As a quantiative measure of how much this pathology can deteriorate performance, we take the benchmark raytrace from the SPLASH-2 suite [25] and run a transactified version of it (locks replaced with transactions) on the baseline LogTM system described in Section 5. In raytrace, threads synchronize when obtaining a global ray identifier. For the *teapot* input, the highly contended cache line that contains this *rid* variable is responsible for more than 100.000 aborted transactions, versus 30.000 that committed successfully. This pathology results in almost $2\times$ slowdown with respect to the lock-based code.

2.2 Reducing Traffic Generated during Stalls

Eager systems generally attempt to resolve conflicts using a *requester stalls* policy [4], [16], [27]. The simplest way to

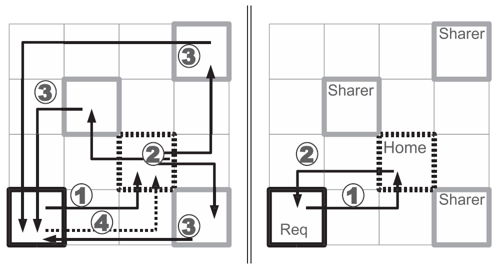

Fig. 2. Messages generated on a write-read conflict in cache-based versus directory-based conflict detection.

implement this policy is simply to retry the conflicting coherence request a few cycles after the nack response has arrived. From the perspective of the processor, the conflicting memory access appears as a long-latency miss that takes hundreds of cycles to be serviced. The conflicting coherence request is retried until it succeeds in bringing the data with the right permissions, or until the deadlock detection mechanism indicates the possibility of a cyclic dependency.

The straightforward *nack-and-retry* scheme of the popular LogTM design [16], [27] has the advantage of its simplicity, but results in a substantial amount of network messages generated when we consider the entire length of the stall, specially if the *nacker* is a long-running transaction. In general, detecting a conflict typically takes $2 + 2n$ messages, where n is the number of bits set in the bit-vector kept at the directory (number of sharers). Fig. 2(left) illustrates all network messages generated on a write-read conflict when a cache-based style of conflict detection is employed. The tiles with a shared copy of the line are shown in gray, a dotted line indicates the home tile and a bold black line the requester. A dotted arrow differentiates the final unblock message from the initial request. On the right side of Fig. 2 we show how by detecting conflicts at the directory, the requester transaction can observe the conflict after only two messages, independently of the number of sharers.

2.3 Reducing False Positives of Signatures

One of the main advantages of eager-eager systems is their ability to accommodate transactions with large footprints. The versioning hardware takes care of logging the old contents of the line before it is speculatively modified "in place." Therefore, evictions of both clean and dirty transactional data from the private cache are tolerated, provided that the cache controller can detect conflicts on spilled lines. The scheme to summarize overflowed addresses can vary from a single bit [16] to hash signatures [6], [18], [27]. Unfortunately, these Bloom filters are susceptible to false conflicts that arise as a consequence of their conservative encoding of addresses.

While signatures can be designed to minimize aliasing [20], [21], [22], [28], an inherent limitation of cache-based conflicting detection is that the bookkeeping metadata is always recorded on a per-core basis. Using a data structure simile and observing the transactional metadata from a global perspective, the typical organization resembles an array of n hash-tables, where n is the number of cores. Since the data accessed inside transactions is often shared, we can expect the elements (addresses) found in different hash-tables to overlap to a certain extent. Given such locality, for coarse grain transactions the number of accessed addresses is often larger than the number of cores in the system, suggesting a more efficient encoding of the metadata that could avoid having the same element repeated across different sets. Instead, one hash table could track the union of all addresses, each one mapped to a bit-vector that indicates which cores are transactional accessors of the address.

The aforementioned idea of tracking metadata on a per-address basis can be naturally applied at the directory level. The directory already acts as a hash table that maps addresses to presence bit-vectors; extending it to track transactional ownership is straightforward. The extra overhead of having a *transactional directory* (TXDIR) is low: A small cache per directory bank is generally sufficient to contain all the lines accessed inside transactions which are mapped to that bank.

2.4 Avoiding Broadcast upon Off-Chip Misses

An unbounded design like LogTM-SE [27] is able to tolerate the loss of the directory information, by further extending the protocol with broadcast *filter check* messages to private (L1) caches on every shared level (L2) miss, and use the responses to conservatively rebuild the directory information. However, the approach adopted by LogTM-SE burdens each and every L2 miss with a broadcast of filter check messages in order to maintain transactional isolation at all times, increasing network traffic in all cases. The reason for such broadcast is that the bookkeeping information required to detect conflicts is solely kept at the private cache level. A directory-based scheme can easily avoid this broadcast by having each tile keep summarized information about its overflowed lines. L2 misses are not a frequent event, and thus L2 evictions of transactional data are an even more uncommon situation—at least in the context of a CMP with a large, shared level L2 cache—which can be handled with simple solutions.

3 Background and Related Work

The early HTM proposal from Herlihy et al. [12] added a separate cache to track the transaction's read and write sets. When TM was revived a decade later by Stanford's TCC system [10], transactional bookkeeping was accomplished by augmenting the existing private caches with transactional status bits associated to each entry. For bus-based systems like TCC, integrating the conflict detection logic into the cache controller is undoubtedly the most natural and straightforward solution because all cache controllers are able to snoop all potentially conflicting memory references issued by remote transactions. In such context, a straightforward solution to conflict detection is to incorporate the transactional status bits (SR and SM bits) along with the coherence state—as part of the cache line metadata—and modify the state machine to interpret those bits as well and take suitable actions when a conflict is detected. This is a rather simple change in the internals of the coherence controller, and is enough to detect conflicts on a best effort HTM design which does not allow evictions.

A simple yet conservative solution upon spills of transactional data are to enforce transaction serialization, letting the overflowed transaction write its results directly to shared memory [10]. Nonetheless, transactions of larger footprints can be accommodated without resorting to global serialization if the system is capable of keeping track of overflowed addresses, even if it does it in a summarized way. Replacements of read-set data can be tolerated regardless of the version management policy used, whereas speculatively written lines can only be spilled to the shared levels of the memory hierarchy if the system logs values before they are speculatively written [16], [27]. To retain isolation on overflowed lines, cache controllers need a way of determining if an address whose tag is not found in cache indeed belongs to the read and write sets of its transaction. The solution can range from a single overflow bit [16], to an overflow signature [18] or a permissions-only cache [2].

A different solution is to remove the transactional status bits from caches and only use signatures for transactional bookkeeping [6], [27]. A key disadvantage of hash-signatures is the possibility of false positives. Addresses that do not belong to the read and write set of the transaction may be considered as such due to aliasing, and false conflicts are then signaled when none exists, causing unnecessary performance degradation. This poses a dangerous situation if the ratio of false positives becomes significant as the transaction footprint grows, as it may discourage programmers from using coarse grain synchronization, somehow jeopardizing one of the main goals of TM.

Other eager conflict detection systems use memory-side conflict detection [1], [3] to allow transactions of an arbitrary size, yet they require extensive modifications to the memory interface or incur in significant space overheads when tracking metadata across all memory.

4 Directory-Based Conflict Detection

In this section, we describe how the directory is augmented with several components in order to support the directory-level conflict detection introduced in the previous sections.

4.1 Transactional Status

Using the directory to check for conflicts over lines that remain cached by transactional owners does not necessarily need any more information about a block than what is already stored in its directory entry. For example, let W be a transactional writer that locally caches a block B with exclusive ownership, and let R be a reader that tries to acquire nonexclusive ownership of B. When R's read request arrives at the directory, the standard protocol dictates that the request must be forwarded to W, which would then detect the conflict. However, if the directory only knew that W is executing a transaction, forwarding the request to W would be unnecessary; the directory itself could conservatively detect a conflict on B and directly send a nack response to R. A simplistic solution is to extend the directory with a *transactional status register* that records which cores are executing a transaction at the moment. This register could be kept updated by sending explicit messages to all directories at transaction begin and commit/abort, and waiting for the corresponding acknowledgement before resuming the execution. Other schemes that avoid stalling the execution of the transaction and reduce the number of contacted directories are discussed later in this section. For now, let us assume the simplistic solution based on transaction begin/end reports.

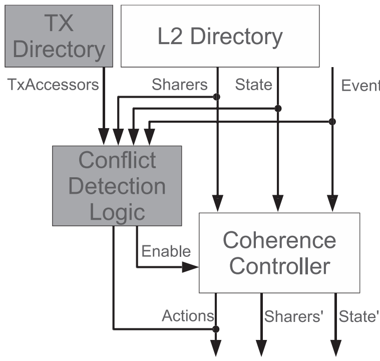

Fig. 3. Block diagram of the new directory organization.

Once the directory knows which cores are executing transactions in a given moment, it can start detecting conflicts on their behalf. Again, the simplest solution would be to perform the logical *AND* of the transaction status register and the presence bit-vector of the requested line, and interpret the result as the current transactional accessors of the line. Obviously, this would conservatively consider as transactional all privately cached lines, and while it would suffice to provide correct transactional semantics, it would result in a tremendous amount of false conflicts. Hence, the directory needs some sort of transactional bookkeeping to distinguish between cache residence and transactional ownership, in order to detect conflicts more accurately.

4.2 Transactional Metadata

Transactional metadata is kept at the directory by means of a small, set-associative cache that we call the *transactional directory*. Just like the regular directory tracks cache residence, the TXDIR tracks transactional ownership. The TXDIR is accessed in parallel with the L2 directory, and the outputs from both structures are provided to the module that contains the conflict detection logic. This organization of the directory, including the newly added components, is shown in Fig. 3.

The internal organization of the TXDIR is detailed in Fig. 4. We can observe how it combines a cache-like structure that maintains precise metadata for a limited number of lines, with a set of small signatures (one per core) which conservatively encode those transactional addresses that do not fit in the aforementioned buffer. Along with the *TxAccessors* bit-vector, each entry includes an additional *TxWriter* bit indicating if the line has been written (not shown in the figure, for clarity); this bit is only meaningful when only one transactional accessor exists. On each incoming request that arrives at the directory, the TXDIR cache is accessed. If a tag hit is found, the transactional accessors and writer status are taken from the entry. Otherwise, the result of the parallel signature check is selected.

The TXDIR cache has a special feature: It is augmented with circuitry for flush-clearing the *TxAccessors* at the granularity of bits. Individual bits can be flush-cleared in a single cycle by enabling the corresponding clear signals, which are controlled by the conflict detection logic. The TXDIR is implemented in a similar way to the L1-cache tag array of traditional HTM systems which use transactional status bits (SR and SM) for bookkeeping. The extra circuitry

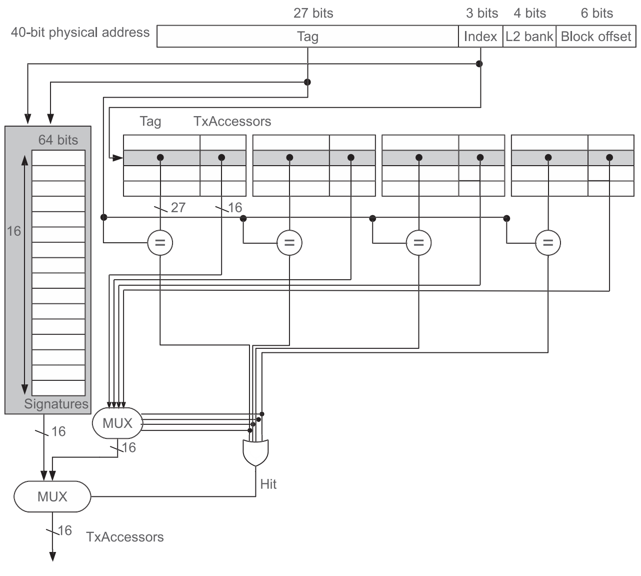

Fig. 4. Transactional directory internals.

required for flush-clearing each bit is very modest: for each bit, an extra transistor connected to the corresponding clear signal is added. In comparison to the augmented L1 cache tag entries commonly found in HTM systems, the TXDIR simply extends the number of bits from 2 to n, and the number of clear signals from 1 to n (for an n-way CMP).

A small number of entries per TXDIR suffices to precisely track the accessors for a large number of transactional lines, as the combined entries of all TXDIRs are available to any transaction. For a 16-core tiled CMP design, we find that an 8-way set associative TXDIR with eight sets per L2 bank suffices to keep accurate transactional metadata in the common case. Under this configuration, the aggregation of all 64-entry TXDIRs is capable of bookkeeping up to 1,024 different transactional addresses before resorting to a conservative scheme based on signatures. If a fine-grain L2 bank mapping policy is used—i.e., consecutive line addresses are mapped to subsequent L2 banks—the read and write sets of the transaction are evenly distributed across all directories, thus minimizing the frequency of metadata overflows at the TXDIR. Though not shown in Fig. 4, a victim buffer can be optionally incorporated to the TXDIR cache to reduce the overflows due to its limited associativity.

An inherent advantage of the directory-based bookkeeping is that, even when the limited capacity or associativity of the TXDIR requires the use of signatures to track transactional accessors, some false positives can be identified and properly ignored: The use of sticky states at the directory ensures that every transactional accessor is marked as a holder of the line in the presence bit-vector. Thus, the opposite scenario—transactional accessor not in *sharers*—is clearly the result of address aliasing and can be ignored. For this reason, a set of small, per-core signatures (64 bits each, 1 Kbit overall) suffices to support conflict detection over an unbounded number of lines without introducing frequent false positives.

4.3 Conflict Detection Logic

As shown in Fig. 3, the module that performs the conflict detection takes as inputs two bit-vectors—transactional accessors and sharers—the coherence state, the event—type of request- and the identity of the requestor. Its main output is a *no-conflict* signal, which is in turn connected to the *enable* input of the coherence controller module. This signal is usually asserted, allowing the coherence controller to operate as usual. When a conflict is detected, the coherence controller is disabled so that the line's state and sharers remain unchanged, while the actions taken by the directory are directly controlled by the conflict detection module. Its simple logic is specified in Algorithm 1.

Algorithm 1. Conflict Detection Logic

```
$conflict \leftarrow false$
if $TxAccessors \neq \{Requestor\}$ AND $TxAccessors \neq \{\}$ then
  if $Event = Write$ then
    if $State = M$ then
      $conflict \leftarrow true$
    else
      if $Priority(Requestor) < HighPrio(TxAccessors)$
      then
        $conflict \leftarrow true$
      end if
    end if
  else
    if $Event = Read$ then
      if $State(Address) = M$ AND $TxWriter$ then
        $conflict \leftarrow true$
      end if
    end if
  end if
end if
```

The conflict controller only attempts to detect a conflict if the line has at least one transactional accessor different from the requestor. In that case, a conflict is signaled if a write request finds a line that is exclusively owned by another transaction (write-write or write-read conflict), as dictated by the M coherence state. In any other state, a conflict is immediately detected if the requestor does not have higher priority than the current transactional accessors. If it does, the directory omits the detection of the conflict and forwards invalidations to the sharers of the line, in order to support a hybrid resolution policy [4], similarly to the baseline eager-eager system evaluated throughout this thesis. Finally, shared requests cause a conflict if they find the line held in exclusive ownership whose writer status indicates that the only transactional accessor is indeed a write transaction (read-write conflict). It should be noted that this detection logic is able to process conflicting requests to lines in busy states, effectively decoupling conflict detection from coherence maintenance.

4.4 Propagation of Transactional Metadata

The fundamental drawback of detecting conflicts at the directory is that not all memory references within a transaction must go through directory, but only those that result in cache misses. Because the notification of a transactional load or store hit to the directory cannot happen instantly, conflicts still need to be temporarily detected at the cache level until the directory has knowledge of the transactional access and can take over the task. Thus, transactional loads and stores that hit in the private cache must notify the directory in order to update the TXDIR metadata. This communication takes place asynchronously—off the critical path of the memory reference—by means of a special write-back message we call *txaccess*. Note that this new type of message is not part of the protocol and thus does not participate in the coherence mechanisms. *Txaccess* messages simply update the metadata kept at the directory, adding the sender as a transactional accessor and appropriately setting the writer status bit (a write flag distinguishes the type of access). For memory references that do go through directory, the TXDIR must be updated after the miss has been successfully solved. In this case, the *txaccess* is piggybacked as a couple of flags in the final *unblock*.

Since propagation of transactional metadata happens asynchronously, caches must be able to detect conflicts in those cases when a request reaches the directory before it has been informed about a transactional hit at the L1 cache level. Once the directory receives the corresponding *txaccess* report it resumes the task of conflict detection, and the offended caches no longer observe conflicting traffic. We discuss several alternatives for dealing with such racing requests in a later section.

4.5 Awareness to Priority and Deadlock Detection

Eager HTM systems that rely on a pure *requester stalls* policy are susceptible to a pathology known as *starving writer* [4]. A priority scheme is required to support the aforementioned hybrid resolution policy, which resolves write-read conflicts in favor of the requester when the writer has higher priority than all readers. Time stamps transported in all coherence requests for deadlock avoidance [16] are now also leveraged to support such hybrid policy at the directory, which must keep a *time stamp table* and constantly update it by snooping incoming requests.

Thanks to this time stamp table, the conservative deadlock avoidance mechanism commonly employed by eager systems [16], [27] can be kept unmodified. Negative acknowledgement messages are sent from the directory on behalf of the eldest accessor of the line—including its time stamp in the response—so that caches are oblivious to the fact that most of the *nack* responses they receive are in fact sent by the directory, and not by other caches. This allows the resolution scheme at the caches to remain unchanged: A transaction aborts if it is *nacked* by an older transaction and it has its *possible-cycle* bit asserted. This bit gets set when a transaction *nacks* a request from an older transaction. Because caches no longer *nack* forwarded requests, the *possible-cycle* bit is set upon reception of a new *txnacked* message: To emulate the original behavior, the directory sends a *txnacked* message to the eldest transactional accessor when an even older requestor gets *nacked* on its behalf.

4.6 Clearing Transactional Metadata

Transaction begin is implicitly communicated to each directory bank on the first access to a line mapped to the bank, via *txaccess* (L1 hit) or a coherence request (L1 miss). Each directory bank snoops these messages and updates its transactional status register and time stamp table when it detects that a core has entered transactional mode. Coherence request messages are guaranteed to arrive in order, as guaranteed by an in-order processor model. *Txaccess* messages that arrive out of order (e.g., that belong to the previous transaction) are detected by observing their time stamp and properly discarded.

On the other hand, when a transaction ends—whether it is or not successfully—the transactional metadata kept for it at the directory must be cleared in order to release isolation over the lines in the read and write set. To this end, we introduce a second protocol-independent message called *txend*, sent by cores both at transaction commit or after the rollback has completed. When a *txend* message from core *k* reaches the directory, flush-clears the *k*th bit *TxAccessors* of each entry in the TXDIR cache. The writer status flag of the entry is also conditionally cleared (if the *k*th bit is set). The transactional status bit and overflow signature associated with the core are also cleared. Not all directory banks need to be informed about the end of the transaction, but only those whose mapped addresses were part of the finished transaction. Thus, for every memory reference performed during the transaction, each core records the directory bank that the address is mapped to. This bit-vector of *txaccessed directories* is then used at commit/abort time to selectively issue *txend* messages. The core can continue its execution without delay, as *txend* messages do not need acknowledgement because they carry the transaction's time stamp, which is used in conjunction with the internal time stamp table kept at the directory to handle cases in which *txend* or *txaccess* messages arrive out of order.

4.7 Dealing with Races

While a *txaccess* message traverses the network toward the directory after a cache hit, the core must be able to detect conflicts on forwarded requests that reached the home tile before the *txaccess*. Several solutions are possible to handle conflict detection in these races. The most straightforward approach is to maintain the typical transactional status bits in cache, which are used to detect conflicts on such races, as well as to identify those lines whose transactional status has been already reported to the directory—avoiding repeated *txaccess* messages on subsequent hits. For simplicity, this is the approach we use in our design, but other solutions are possible too, since the directory already keeps accurate information about the read and write sets, and therefore it is not necessary to have such precise—redundant—metadata at the cache level at all times. In this case, replacement of transactional lines in S state cannot be silent, as conflicts could go undetected if the L1 cache loses the transactional metadata before the corresponding *txaccess* reaches the directory. Hence, transactional S replacements are treated much like M replacements, by sending a write-back message that updates the TXDIR metadata, and waiting for the *ack* to arrive back to the L1 cache before finally deallocating the line.

One way to detect conflicts at the cache level without having to add bits to the private caches, is to use a small *summary signature* that encodes both read and write sets, and then modify the conflict logic so that the signature is only checked if there is a potential race, i.e., a *txaccess* message on-the-fly whose destination is the same directory bank that forwarded this request. Thus, the *txaccessed directories* bit-vector acts as a first filter to distinguish nonconflicting from potentially conflicting requests. More accurate ways of filtering the check of the signature require some kind of *txaccess* acknowledgement scheme from the directory, which could be easily accomplished by means of serial numbers that are piggybacked in existing messages.

Another method consists in maintaining a separate *txaddress buffer* with the most recently accessed addresses of the transaction. This scheme decouples transactional bookkeeping from caches at the cost of increasing the amount of network messages, as now *txaccess* messages need to be acknowledged before an address can be deallocated from the buffer. Addresses can be buffered indefinitely in order to avoid redundant communications with the directory. However, the buffer should be drained after a certain occupancy threshold in order to leave room for new addresses. Performance can suffer if at some point the buffer fills up completely, since the processor will stall on the next memory reference that results in a cache hit until space becomes available, or else violations of isolation would be risked.

4.8 Reducing Metadata Propagation

Obviously, reporting every cache hit to the directory is an expensive solution in terms of its traffic demands. However, the performance benefits of directory-based conflict detection only apply to contended lines. Hence, it makes more sense to propagate only those accesses to lines that have seen conflicts in the past. We optionally introduce a *conflict signature* which is updated every time a cache sends or receives a *nack* message for an address, and checked to determine if a *txaccess* message needs to be sent. A *txaccess* is sent only for L1 hits to lines whose transactional status is not yet set, whose address also belongs to the conflict signature. Because the fraction of lines that experience contention is usually small in comparison with the size of the transactional set, a small signature should suffice to filter out most of the traffic that propagates metadata from the caches to the directory. This optimization does not affect correctness, and the filter is periodically cleared (i.e., in barriers). This trades off some performance for reduced traffic, when conflicts are forgotten and accesses to contended data are not immediately reported to the directory.

It is important to note that neither the conflict signature nor the queuing of *txaccess* messages lay on the critical path of a cache hit. The L1 cache does not need to wait for the result of the signature check to service the data, and so the memory access can indeed complete before the message is enqueued, if the signature check signals a positive. The introduction of the conflict signature is thus an optimization which does not impact L1 hit latency. Similarly, *txaccess* messages are not strictly necessary to maintain correctness and can be safely dropped if need be.

4.9 Design Scalability

As the number of tiles in the CMP increases, the size of the proposed per-bank TXDIR could be proportionally reduced while maintaining accurate bookkeeping metadata over a constant number of lines which suffices in the common case, even for most coarse-grain transactions. As for the overflow signatures, several tiles could be grouped to share one signature in order to keep the area of this array constant when scaling up the system. Since L2 transactional spills

TABLE 1
HTM Configurations Evaluated

Configuration	Description
EE_base	The LogTM-SE design[27] with *perfect signatures*
EE_pred	Idem. as EE_base, with write-set predictor [4]
BitSel_2048, H3_2048,H3_4096	Idem. as EE_base, with parallel Bloom signatures [22] Type: bit-selection/hash function; Size: 2/4 Kbits
DirCD_Magic	Idem. as EE_base, with magic conflict detection @ dir
DirCD_TxDir64	EE system w/ conflict detection @ dir (64-entry TXDIR)

would be an even rarer event in larger chips, the number of false positives would still remain low. The extra latency due to extra hops does not substantially affect messages for metadata propagation since they are asynchronous. The number of *txend* messages does increase proportionally to the number of tiles, despite being sent only to those directory banks whose data were accessed. Their overhead may become significant in large chips for workloads with large transactions and high contention, though this problem could be alleviated using multicast messages [13].

5 Methodology and Evaluation

In this section, we evaluate the proposed scheme of directory-based conflict detection, comparing it against several pertinent HTM design points.

5.1 Experimental Setup

We use a full-system execution-driven simulator based on the Wisconsin GEMS tool-set (v2.1) [15], in conjunction with Wind River Simics [14]. We use the detailed timing model for the memory subsystem provided by GEMS, with the Simics in-order processor model. Simics provides functional simulation of the SPARC-V9 ISA and boots an unmodified Solaris 10 operating system. Table 1 lists all HTM configurations evaluated in this paper. LogTM-SE [27] acts as the eager-eager (EE) HTM system of reference, and all other HTM configurations are derived from it. The *EE_base* configuration employs perfect signatures at the private cache level, and uses a hybrid resolution policy to avert the *starving writer* pathology [4]. Experiments were performed on a 16-core tiled CMP system, described in Table 2, with private L1 caches and a shared, multibanked L2 cache. For each workload-configuration pair we gathered average statistics over 10 randomized runs.

The *EE_pred* augments this baseline with a 256-bit, 256-entry write-set predictor, in order to also target the *dueling upgrades* pathology [4], by selectively requesting exclusive permission for predicted loads. Our configuration is similar to Bobba's EE_{HP} system, except for one detail: In our case, the block is not added to the transaction's write set until it is indeed written, as we observed that doing so consistently results in worse relative performance than if only added to the read set, for the benchmarks here considered.

We also evaluate the EE_base system using parallel Bloom signatures. All three signature configurations considered use four hashes. We consider both bit-selection and high-quality H3 functions, with sizes of 2-4 Kbits.

As for our design, we consider two systems. On the one hand, *DirCD_Magic* acts as an upper bound of the performance achievable by a directory-based conflict detection scheme. It is a modified version of the ideal EE_base

TABLE 2
System Parameters

MESI Directory-based CMP	
Core Settings	
Cores	16, single issue in-order, non-memory IPC=1
Conflict-Retry interval	50 cycles
Conflict signature	256 bits
Memory Settings	
L1 I & D caches	Private, 32KB, split 4-way, 1-cycle latency 1 Tx bit per line
Log Buffer	8 cache lines
L2 cache	Shared, 512KB per tile, unified 8-way, 12 cycle-latency
Memory	4GB, 300-cycle latency
Directory Settings	
L2 Directory	Bit vector, 6-cycle latency
TXDIR	64-entry, 8-way associative
TXDIR victim cache	8-entry, fully associative
TXDIR overflow signature	16 x 64-bit H3 parallel
L2 cache TX overflow	16 x 1 bit
Network Settings	
Topology	2D Mesh
Link latency	1 cycle
Link bandwidth	40 bytes/cycle
Flit size	16 bytes

(perfect signatures) in which the directory has instant access to any read or write signature across the chip—without involving any message—and directly *nacks* conflicting requests, while magically informing the nackers about the conflict—to maintain deadlock detection.

On the other hand, *DirCD_TxDir64* is a detailed implementation of the scheme described in previous section, which uses a TXDIR of finite size to perform transactional bookkeeping at the directory, and extends the protocol with *txaccess*, *txend*, and *txnacked* messages to propagate metadata, communicate transaction commit/abort and detect deadlocks, respectively. The specific parameters of this configuration are shown in Table 2. In order to provide a fair comparison in terms of network traffic, in the DirCD_TxDir64 system we adapted the amount of cycles that a processor stalls after detecting a conflict—before retrying the request—so that both EE_base and our design have similar intervals of retry, i.e., result in a similar number of reissued requests on a stall of the same length. While EE_base retries after only three cycles, DirCD waits longer (50 cycles) before reissuing the conflicting memory request, because it inherently detects conflicts quicker. This implementation of DirCD extends each L1 cache entry with a single transactional bit, used to detect conflicts on racing requests, as well as to decide if a *txaccess* message needs to be sent to directory. When this *tx* bit is asserted, the coherence state conservatively determines if the line belongs to the read set (S or E states) or write set (M state). As for the ability to tolerate L2 evictions of transactional data, we model a single overflow bit-vector (one bit per core) on each tile. Nonetheless, we have not experienced any such events in our experiments, mainly due to the L2 cache's fairly large size and associativity of the CMP modeled throughout the thesis (512 MB per tile, 8-way associative).

Workloads. STAMP workloads were chosen as they are among the most representative TM benchmarks available so far, and exhibit a fair diversity in behavior. The parameters for applications have been taken from [5] and are shown in Table 3. We changed one of genome's compile-time parameters in order to use a larger chunk size in the first

TABLE 3
Benchmarks and Inputs Used in the Simulations

Benchmark	Input
genome+	-g512 -s32 -n32768
genome	-g256 -s16 -n16384
intruder+	-a10 -l16 -n4096 -s1
intruder	-a10 -l4 -n2048 -s1
kmeans-low	-m40 -n40 -t0.05 -i random-n2048-d16-c16
kmeans-high	-m15 -n15 -t0.05 -i random-n2048-d16-c16
ssca2+	-s14 -i1.0 -u1.0 -l9 -p9
ssca2	-s13 -i1.0 -u1.0 -l3 -p3
vacation-low	-n2 -q90 -u98 -r16384 -t4096
vacation-high	-n4 -q60 -u90 -r16384 -t4096
vacation-vhigh	-n2 -q1 -u1 -r128 -t4096
yada+	-a10 -i ttimeu10000.2
yada	-a20 -i 633.2

step of the algorithm. We have increased *chunk_step1* from its default value of 12 in the release of STAMP, to 36, so as to stress the bookkeeping mechanism of the evaluated HTM systems with transactions of a larger footprint. Yen adopts similar strategies when evaluating signatures [26]. Furthermore, we excluded labyrinth because it requires support for the *early release* construct in order to benefit from parallel execution: Removing addresses from read sets is not possible in systems that use real signatures, including our DirCD scheme, and thus the utility of labyrinth in this evaluation is limited. To make up for the loss benchmark, we modified the input parameters of vacation, as shown in Table 3, to create a new configuration, *vacation-vhigh*, which exhibits very high levels of contention on transactions of a fairly large footprint. Very high contention levels in vacation are achieved by reducing the initial size of the database as well as the percentages of queried relations and *user* (read-only) transactions.

5.2 Performance Analysis

5.2.1 Idealized Systems

Fig. 5 shows the potential performance gains of a directory-based scheme to conflict detection (DirCD), relative to the

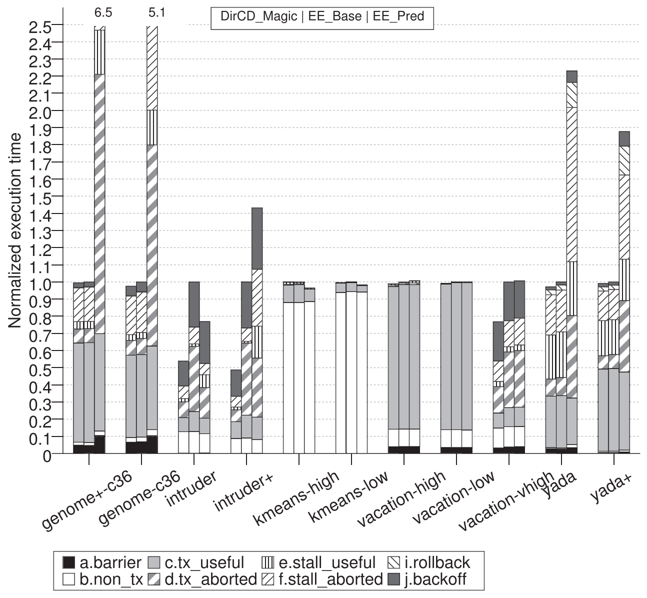

Fig. 5. Relative performance of "magic" directory-based versus ideal cache-based conflict detection.

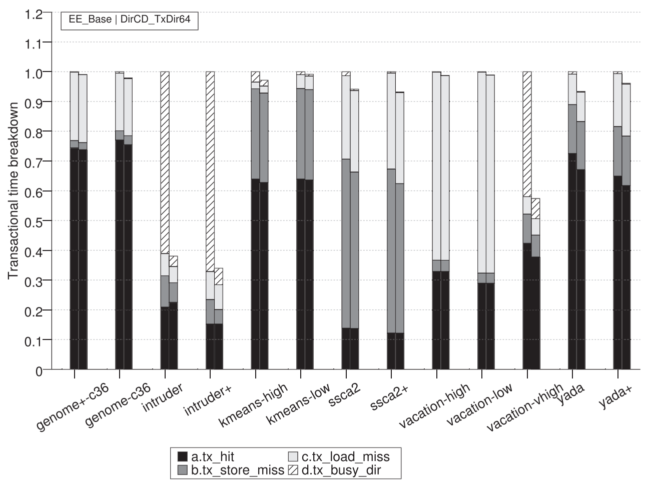

Fig. 6. Transactional cycle breakdown in baseline versus directory-based schemes.

cache-based approach traditionally used by EE systems. In comparison to EE_base, it shows how the "magic" DirCD system greatly reduces the execution time of highly contended applications such as intruder and vacation-vhigh, in percentages that vary from 22 percent in vacation-vhigh to roughly 50 percent in intruder+. This confirms that the directory indeed creates a bottleneck in the conflict management mechanism in situations of high contention affecting a few cache lines. DirCD_Magic anticipates the performance gains that can be expected if detection is decoupled from coherence, i.e., if it can be carried out without the need of request forwarding—thus without transitioning to the line to a busy state.

Fig. 6 presents a breakdown of transactional cycles for the EE_base system, compared to the realistic implementation of DirCD analyzed in the next section. The total transactional time corresponds the sum of tx-useful and tx-aborted components of Fig. 5. The figure divides transactional cycles into *tx-hit* (nonmemory or L1 hits) and cycles waiting for a memory request to complete—excluding retried requests, which are accounted as stall time in Fig. 5. Memory access time is in turn further broken into the time the request was queued at the directory due to busy states (*tx-busy-dir*), and actual miss time (*tx-load-miss* or *tx-store-miss*), which reflects the compulsory time taken by messages to travel across the interconnect, L2 cache access time, etc. The figure demonstrates how the reductions achieved by DirCD in the tx-useful and tx-aborted components of intruder and vacation-vhigh are due to the removal of the bottleneck formed at the directory in the baseline system, which limits its ability of resolving conflicts during contention.

Intruder. Both DirCD and EE_Base systems suffer the aforementioned pathology of *dueling upgrades* that leads to many aborts, shown in Table 4. The improvement of DirCD with respect to the EE_Base is not so much due to the reduction in the number of aborts—aborts of transaction with TID0 go down by 40 to 60 percent, as we can see in Table 4—as it is due to the higher throughput of detected conflicts achieved by DirCD. TID0 corresponds to a *queue pop* operation that first reads and then writes a highly contended line. The detection at the directory allows the

TABLE 4
Number of Aborts for the Systems in Fig. 5

	Total Aborts			TID0 Aborts		
	DirCD	EE_Base	EE_Pred	DirCD	EE_Base	EE_Pred
genome+	1880	1841	31059	1146	1186	30535
genome	1267	1265	14714	759	784	14383
intruder+	103431	133610	121074	71520	109076	150
intruder	22076	29668	20805	6057	14859	249
kmeans-high	679	598	8	473	426	2
kmeans-low	239	215	5	86	79	0
ssca2+	258	190	64	14	13	13
ssca2	330	280	112	47	51	51
vacation-high	159	122	204	150	116	180
vacation-low	22	23	35	22	23	35
vacation-vhigh	8207	8132	7850	193	198	189
yada+	6643	6520	16526	2066	2016	398
yada	3261	3312	8362	893	890	222

highest priority writer to proceed quickly—since its write request is processed immediately when it arrives at the directory, as it demonstrated Fig. 6—causing the abort of the lower priority readers (upgraders). The fact that lower priority conflicting transactions are aborted much faster in DirCD is reflected in the radical decrease of the *tx-aborted* component seen in Fig. 5 when compared to EE_Base; the total number of aborts also decreases and accounts for the shrinkage of the *backoff* component too. In comparison to EE_Pred, the write-set predictor successfully averts the dueling upgrades and almost completely removes all TID0 aborts by directly requesting the line for exclusive access, as shown in Table 4. TID0 transactions serialize one after another without incurring in the huge number of aborts seen in the other two systems. However, the predictor does not represent a generalizable solution. For the small input of this benchmark, we see how the total number of aborts is decreased by around 1K, in spite of a reduction in TID0 aborts of almost 6K; for the medium input, despite removing 110K TID0 aborts, the total number of aborts goes up by 18K. This indicates that the write set predictor penalizes transactional execution of other transactions of the program, mainly due to the combination with a hybrid resolution policy: more write requests means more (often unnecessary) aborts of concurrent readers.

Vacation-vhigh. This benchmark corroborates our observation that the performance improvements seen in DirCD do not stem exclusively from a reduction in the number of aborts, but rather from a reduction in the amount of cycles that transactions waste while waiting for a conflicting request to be processed at the directory, as shown in Fig. 6. In both traditional EE systems, the directory spends most of its time in a busy state while messages are forwarded to the possible transactional owners of the line, which are responsible for the conflict check. In DirCD, the directory spends much less time in busy states and thus can attend and respond to messages immediately, which in turn is translated in faster resolution of the conflict.

Other benchmarks. DirCD and EE_Base perform comparably for applications whose transactions are either long running or not heavily contended. In these cases, there is no performance advantage in detecting the conflict sooner, since the resolution consists in stalling the requester the majority of the times. In regards to EE_Pred, we observe that the effect of mispredicted upgrades becomes very acute for

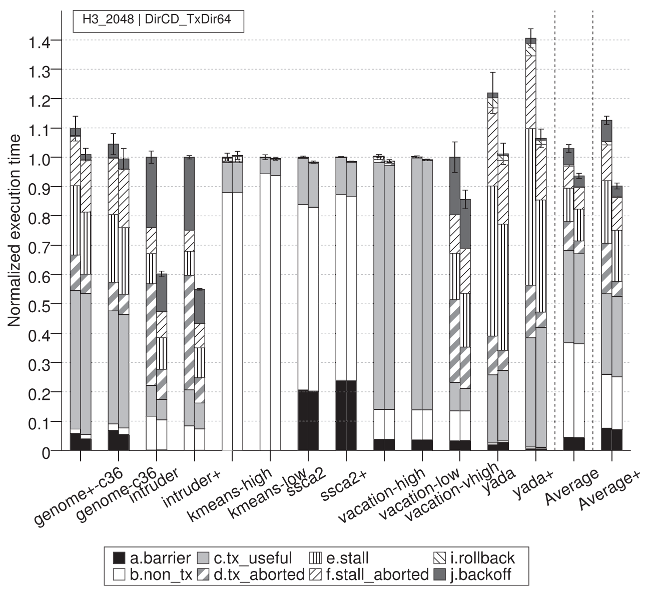

Fig. 7. Relative performance of realistic directory-based versus cache-based conflict detection.

benchmarks with large transaction footprints like yada and genome, which experience severe performance drops that vary from around $2\times$ slowdowns in the case of yada, up to 5-6 times slower in the case of genome. We can see in Table 4 how TID0 is the transaction responsible for the pathological behavior of the write-set predictor in genome, which is precisely the transaction with the largest read and write sets.

5.2.2 Realistic Systems

Fig. 7 shows the relative performance of our detailed implementation of DirCD that uses a transactional directory, compared to an EE system that uses Bloom signatures whose total size is comparable to the overhead of the structures introduced by our scheme. Fig. 8 presents performance numbers for four EE systems that use real Bloom filters to track transactional read and write sets. The results shown in

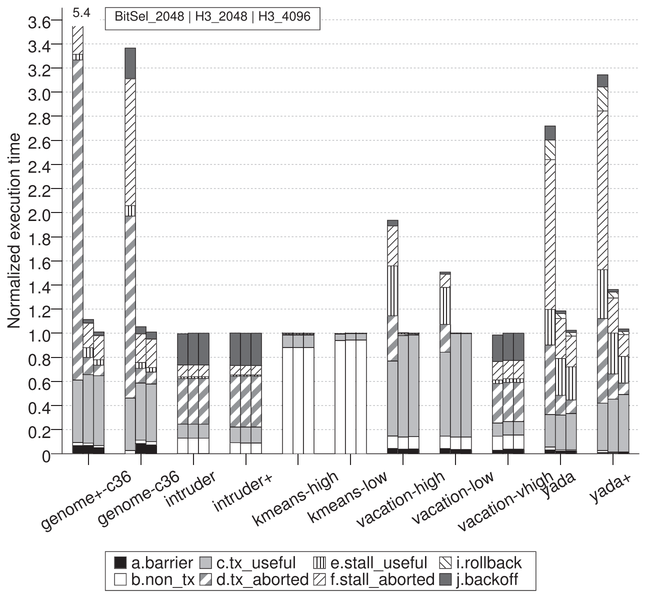

Fig. 8. Performance effects of false positives.

both figures are normalized with respect to the EE_Base configuration. We can see in Fig. 7 how DirCD excels for applications with high contention like intruder and vacation-vhigh, for the reasons discussed earlier. Furthermore, DirCD closely tracks the performance of the EE system with perfect signatures for workloads with large transactions like genome and yada. This reveals that the number of false conflicts that arise in DirCD is very low, in spite of its finite capacity to precisely track transactional metadata. This is due to its ability to identify and ignore false positives signaled by the TXDIR overflow signatures, by performing a logical *and* between the result of the signature check and the corresponding bit in *sharers*.

DirCD consistently performs equally or better than the H3_2048 configuration, which uses a total of 4 Kbits to hash-encode both transactional sets (2,048 bits each signature). As it can be derived from Table 2, our detailed DirCD implementation spends a bit less than 5 Kbits: 3 Kbits (including tags and "data") in the TXDIR and associated victim cache, an extra 1 Kbit for the 16 overflow signatures of 64 bits, the 256-bit conflict signature, and an additional 512 bits for the transactional flag kept per L1 cache entry (although these bits are only maintained for dealing with races as well as reducing redundant metadata propagation). We choose to compare DirCD against the H3 scheme of encoding, as it is more efficient than bit-selection for same-sized signatures [22], as demonstrated by Fig. 8. For a similar metadata storage capacity of less than 5 Kbits per tile, our scheme keeps the number of false positives to a minimum, demonstrating that storing transactional accessors on a per-address basis at the shared cache level is a more efficient encoding of transactional sets than tracking addresses on a per-core basis using signatures. Using parallel H3 filters, it is necessary to increase their joint size to 8 Kbits in order to avoid most false positives that arise in benchmarks with large-sized transactions like genome or yada, though it would still be of no help for highly contended benchmarks like intruder or vacation-vhigh. Considering that TM does not discourage programmers from using coarse grain transactions, the efficient encoding of transactional addresses is an important point in HTM design that is met by our proposed bookkeeping scheme.

5.3 Traffic Considerations

Fig. 9 plots the network traffic results for the baseline EE system and our detailed DirCD model. It shows both network message counts generated by each system—broken down according to message type—as well as the flit count. Both measures are normalized to the data obtained for EE_base. As we can observe in the average plots, DirCD generates between 25 and 35 percent less traffic (flits) than the baseline system. The first relevant difference shown by the breakdown is how DirCD completely eliminates *filter check* messages broadcast on every L2 miss, as discussed in the motivation section. On average, filter checks approximately account for 15 percent of all network messages generated, although they reach over 30 percent for workloads with large working sets like both original configurations of vacation. The number of *acks* messages is also severely reduced in DirCD, since each filter check is responded with an *ack*.

Another difference is the removal of virtually all *unblock-cancel* messages, as the vast majority of the conflicts are

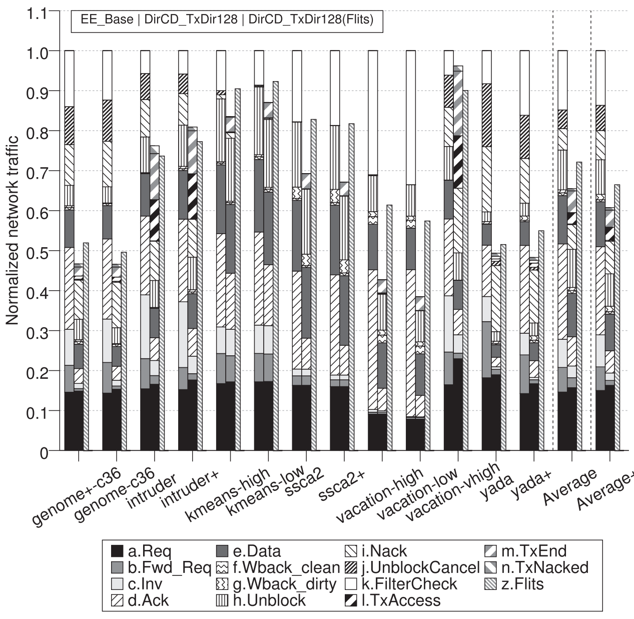

Fig. 9. Network message breakdown and network flits.

detected at the directory, which does not forward requests that are known to be conflicting and thus does not enter a busy state. Formidable reductions in the amount of invalidation messages achieved by DirCD indicate that write-read conflicts are solved with a single *nack* message, avoiding both the *invs* and the corresponding *ack*/*nack* responses, as described in Fig. 2. For contended workloads with long-running transactions like genome and yada, both *invs* as well as requests forwarded to exclusive owners are significantly reduced, which gives an idea of how DirCD allows the simple *stall-and-retry* resolution policy of the baseline system, at a much lower cost in terms of the network traffic generated by retries. The number of requests, data, unblock and *nack* messages stays more or less constant across all benchmarks, for both systems.

DirCD achieves the above reductions in the network traffic associated with conflict detection at the cost of introducing new messages that do not exist in the baseline system. Their main purpose is to propagate (via *txaccess* messages) or clear (via *txend* messages) the metadata kept at the directory. The number of *txend* and *txaccess* messages depends on the total number of transactions attempted as well as the transaction footprint (number of L2 banks accessed). Therefore, they grow proportionally to the level of contention, and much faster if contention affects large transactions. This explains why it is responsible for around 20 percent of all messages in intruder (small transactions), while it can reach almost 30 percent for vacation-vhigh (larger data set). For workloads with few large transactions with moderate contention like yada, the number of *txend* and *txaccess* messages is almost negligible. In all other benchmarks, they account for less than 5 percent of all messages, due to the filtering effect of the conflict signature used at the private cache level to avoid sending *txaccess* messages for lines that have not seen conflicts recently. The role of conflict signature explains the low number of *txaccess* messages in low contended benchmarks like vacation-low and vacation-high, in spite of their large transaction size. For the same reason, both ssca2 inputs should show much larger counts of *txend* as a result of the huge number of noncontended transactions executed, but the conflict signature avoids them by filtering out most or all *txaccess* messages that would have been sent to the directory otherwise.

In summary, despite the newly added messages, the significant reductions in other types of messages associated with conflict detection still tips the scale in favor of DirCD across all evaluated workloads, confirming that the extra cost of detecting conflicts at the directory does not only pay off in terms of performance, but it is also more efficient on its use of the interconnect. While we do not investigate it in this paper, a hybrid approach could further improve the efficiency of this design by adapting to the level of contention seen in the application, in order to switch off the metadata propagation and thus remove *txend* and *txaccess* messages altogether when there is no benefit in detecting conflicts at the directory level.

6 Concluding Remarks

In this paper, we have presented a new approach to conflict detection targeted to eager TM systems which make use of a distributed directory to maintain coherence over a point-to-point network, as it is the case of a tiled CMP architecture. We have demonstrated that in traditional approaches, the directory becomes a bottleneck in situations of high contention by limiting the throughput of the conflict management mechanism. To this end, we have proposed a design that decouples conflict detection from cache coherence in order to overcome pathological situations that degrade the performance of an eager HTM system, enabling quicker reaction to high-contention scenarios. Our experimental evaluation has shown that our technique deals with contention more efficiently, leading not only to fewer aborted transactions, but most importantly to a lower overall latency of contended memory accesses within transactions. Our experiments have shown average reductions in execution time of 6 to 10 percent with respect to a LogTM-SE system with ideal signatures, while simultaneously decreasing its use of the network by 30 percent on average. In particular, we have observed performance gains of up to 45 percent for those workloads that suffer very high contention over a small number of lines. We have also compared our work to systems that use signatures of equivalent hardware cost at the cache level, and found that our scheme reduces the performance degradation caused by false positives. Our bookkeeping scheme leverages the inherent characteristics of the directory to globally encode all transactional sets by associating addresses to transactional accessors, instead of redundantly tracking addresses in each core, and enables the elimination of some false transactional accessors using the directory information itself. In summary, we claim that this extension in the role of the directory is a natural step in its responsibilities within a cache coherent HTM system.

References

[1] C.S. Ananian, K. Asanovic, B.C. Kuszmaul, C.E. Leiserson, and S. Lie, "Unbounded Transactional Memory," *Proc. 11th Symp. High-Performance Computer Architecture,* pp. 316-327, 2005.

[2] C. Blundell, J. Devietti, E. Christopher Lewis, and M. Martin, "Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory," *Proc. 34th Int'l Symp. Computer Architecture,* pp. 24-34, 2007.

[3] J. Bobba, N. Goyal, M.D. Hill, M.M. Swift, and D.A. Wood, "Token TM: Efficient Execution of Large Transactions with Hardware Transactional Memory," *Proc. 35th Int'l Symp. Computer Architecture,* pp. 81-91, 2008.

[4] J. Bobba, K.E. Moore, L. Yen, H. Volos, M.D. Hill, M.M. Swift, and D.A. Wood, "Performance Pathologies in Hardware Transactional Memory," *Proc. 34th Int'l Symp. Computer Architecture,* pp. 81-91, 2007.

[5] C.C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," *Proc. IEEE Int'l Symp. Workload Characterization,* pp. 35-46, 2008.

[6] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas, "Bulk Disambiguation of Speculative Threads in Multiprocessors," *Proc. 33rd Int'l Symp. Computer Architecture,* pp. 227-238, 2006.

[7] H. Chafi, J. Casper, B.D. Carlstrom, A. McDonald, C.C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, "A Scalable, Non-Blocking Approach to Transactional Memory," *Proc. 13th Symp. High-Performance Computer Architecture,* pp. 97-108, 2007.

[8] D.E. Culler, J.P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, 1999.

[9] L. Hammond, B.D. Carlstrom, V. Wong, M. Chen, C. Kozyrakis, and K. Olukotun, "Transactional Coherence and Consistency: Simplifying Parallel Hardware and Software," *IEEE Micro,* vol. 24, no. 6, pp. 92-103, Nov./Dec. 2004.

[10] L. Hammond, V. Wong, M. Chen, B.D. Carlstrom, J.D. Davis, B. Hertzberg, M.K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional Memory Coherence and Consistency," *Proc. 31st Int'l Symp. Computer Architecture,* pp. 102-113, 2004.

[11] T. Harris, J.R. Larus, and R. Rajwar, *Transactional Memory,* second ed. Morgan & Claypool, 2010.

[12] M. Herlihy, J. Eliot, and B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," *Proc. 20th Int'l Symp. Computer Architecture,* pp. 289-300, 1993.

[13] S. Ma, N.E. Jerger, and Z. Wang, "Supporting Efficient Collective Communication in NoCs," *Proc. 18th Symp. High-Performance Computer Architecture,* 2012.

[14] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *Computer,* vol. 35, no. 2, pp. 50-58, Feb. 2002.

[15] M.M.K. Martin, D.J. Sorin, B.M. Beckmann, M.R. Marty, M. Xu, A.R. Alameldeen, K.E. Moore, M.D. Hill, and D.A. Wood, "Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset," *Computer Architecture News,* vol. 33, pp. 92-99, 2005.

[16] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood, "LogTM: Log-Based Transactional Memory," *Proc. 12th Symp. High-Performance Computer Architecture,* pp. 254-265, 2006.

[17] A. Negi, R. Titos-Gil, M.E. Acacio, J.M. Garcia, and P. Stenstrom, "π-TM: Pessimistic Invalidation for Scalable Lazy Hardware Transactional Memory," *Proc. 18th Symp. High-Performance Computer Architecture,* 2012.

[18] A. Negi, M.M. Waliullah, and P. Stenstrom, "LV*: A Low Complexity Lazy Versioning HTM Infrastructure," *Proc. Int'l Conf. Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS '10),* pp. 231-240, 2010.

[19] S.H. Pugsley, M. Awasthi, N. Madan, N. Muralimanohar, and R. Balasubramonian, "Scalable and Reliable Communication for Hardware Transactional Memory," *Proc. 17th Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 144-154, 2008.

[20] R. Quislant, E. Gutierrez, and O. Plata, "Improving Signatures by Locality Exploitation for Transactional Memory," *Proc. 18th Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 303-312, 2009.

[21] R. Quislant, E. Gutierrez, and O. Plata, "Multiset Signatures for Transactional Memory," *Proc. 25th Int'l Conf. Supercomputing,* pp. 43-52, 2011.

[22] D. Sanchez, L. Yen, M.D. Hill, and K. Sankaralingam, "Implementing Signatures for Transactional Memory," *Proc. 40th Int'l Symp. Microarchitecture,* pp. 123-133, 2007.

[23] R. Titos-Gil, M.E. Acacio, and J.M. García, "Directory-Based Conflict Detection in Hardware Transactional Memory," *Proc. 15th Int'l Conf High-Performance Computing,* pp. 541-554, 2008.

[24] S. Tomic, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero, "EazyHTM: Eager-Lazy Hardware Transactional Memory," *Proc. 42nd Int'l Symp. Microarchitecture,* pp. 145-155, 2009.

[25] S.C. Woo, M. Ohara, E. Torrie, J. Pal Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Int'l Symp. Computer Architecture,* pp. 24-36, 1995.
[26] L. Yen, "Signatures in Transactional Memory Systems," PhD thesis, CS Dept., Univ. of Wisconsin-Madison, 2009.
[27] L. Yen, J. Bobba, M.R. Marty, K.E. Moore, H. Volos, M.D. Hill, M.M. Swift, and D.A. Wood, "LogTM-SE: Decoupling Hardware Transactional Memory from Caches," *Proc. 13th Symp. High-Performance Computer Architecture,* pp. 261-272, 2007.
[28] L. Yen, S.C. Draper, and M.D. Hill, "Notary: Hardware Techniques to Enhance Signatures," *Proc. 41st Int'l Symp. Microarchitecture,* pp. 234-245, 2008.

Rubén Titos-Gil received the MS and PhD degrees in computer science from the Universidad de Murcia, Spain, in 2006 and 2011, respectively. He is currently a postdoctoral research associate at the Chalmers University of Technology, Sweden. His research interests lay on the fields of parallel computer architecture and programming models, including synchronization, coherence protocols, and memory systems. He was a Spanish MEC-FPU Fellowship recipient from 2007 to 2011. He is a member of the IEEE.

Manuel E. Acacio received the MS degree in computer science. He joined the Computer Engineering Department (DiTEC) in 1998 after receiving the MS degree. He successfully defended the PhD degree in March 2003. He started as a teaching assistant, at the time he began his work on his PhD thesis. He is an associate professor of computer architecture and technology at the University of Murcia, Spain. Before, in the summer of 2002, he worked as a summer intern at IBM TJ Watson, Yorktown Heights, NY. After that, he became an assistant professor in 2004, and subsequently, an associate professor in 2008. Currently, he leads the Computer Architecture & Parallel Systems (CAPS) research group at the University of Murcia, which is a part of the ACCA group. He has published several papers in top conferences such as HPCA, IPDPS, ICS, DSN, PACT or SC, and renown journals such as *IEEE Transactions on Parallel and Distributed Systems (TPDS)* and *IEEE Transactions on Computers (TC)*. Recently, he has got the best paper award in the architectures track at the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011). As well, he has served as a committee member of important conferences, ICPP and IPDPS among others, and is currently an associate editor of *IEEE Transactions on Parallel and Distributed Systems (TPDS)*. His research interests are focused on the architecture of multiprocessor systems. More specifically, he is actively working on prediction and speculation in multiprocessor memory systems, synchronization in CMPs, power-aware cache-coherence protocols for CMPs, fault tolerance, and hardware transactional memory systems. He is a member of the IEEE.

José M. García received the MS degree in electrical engineering and the PhD degree in computer engineering both from the Technical University of Valencia in 1987 and 1991, respectively. He is a professor of computer architecture at the Department of Computer Engineering, and also the head of the Parallel Computer Architecture Research Group. He is currently serving as a dean of the School of Computer Science at the University of Murcia, Spain. He has developed several courses on Computer Structure, Peripheral Devices, Computer Architecture, Parallel Computer Architecture, and Multicomputer Design. He specializes in computer architecture, parallel processing and interconnection networks. His current research interests lie in high-performance power-efficiency coherence protocols for Chip Multiprocessors (CMPs) and shared-memory multiprocessor systems, high-speed interconnection networks, and the use of GPUs for general-purpose applications such as bioinformatics and biomedical apps. He has published more than 150 refereed papers in different journals and conferences in these fields. He is a member of HiPEAC, the European Network of Excellence on High Performance and Embedded Architecture and Compilation. He is also member of several international associations such as the IEEE and the ACM.