Directory-Based Conflict Detection in Hardware Transactional Memory

Rubén Titos, Manuel E. Acacio, and José M. García

Departamento de Ingeniería y Tecnología de Computadores
Universidad de Murcia, 30100 Murcia, Spain
{rtitos,meacacio,jmgarcia}@ditec.um.es

Abstract. One of the key design points of any hardware transactional memory (HTM) system is the conflict detection mechanism, and its efficient implementation becomes critical when conflicts are not a rare event. While many contemporary proposals rely on the coherence protocol to carry out conflict detection at the private cache levels, this approach is not optimal for systems that use a directory to maintain coherence over an unordered, scalable network, such as tiled CMPs. In this paper, we present a new scheme of conflict detection for HTM systems, which moves this key mechanism from the private caches to the directory level. We propose a novel transactional book-keeping method and describe how this detection can be carried out more efficiently at the directory. Simulation results show that our approach obtains reductions in execution time between 25 and 55% for transactional benchmarks with a high number of conflicts, with an average improvement over LogTM-SE of 15%.

1 Introduction

Transactional Memory (TM) has arisen as a promising programming model targeted to ease parallel programming while still producing efficient multithreaded programs that exploit the computational resources available in present and future multicore chips. Using the TM model, the programmer declares *what* regions of the code must appear to execute in mutual exclusion, leaving the burden of *how* to provide atomicity and isolation to the underlying levels. The system then optimistically executes transactions, stalling or aborting them whenever real runtime data conflicts appear. Although a TM system can be entirely implemented in software, moving some basic transactional functionality to the hardware level is essential to minimize its performance overhead. This paper focuses on hardware TM systems (HTMs) whose aim is to bring the TM model to the high-performance computing arena.

One of the key mechanisms of any transactional system is conflict detection. A conflict occurs when two or more concurrent transactions access the same data block and at least one of the accesses is a write. In order to detect such violations of isolation, a TM system must keep track of its transactions' read and write sets. Some proposed HTMs perform this book-keeping by extending each cache

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 541–554, 2008.
© Springer-Verlag Berlin Heidelberg 2008

entry with R/W bits [4][8]. Other designs opt for per-thread hash signatures to encode address sets using Bloom filters [12].

Regardless of how the TM system records R&W sets, another major design dimension of conflict detection is *when* to use this information to check for violations of isolation. This can be done either immediately after every memory request – eager policy – or it can be delayed until the end of the transaction – lazy detection –. Most HTM systems proposed to date implement eager conflict detection by modifying standard ownership-based cache-coherence protocols [5] [1][9][8][12]. These systems monitor the coherence traffic for transactional blocks to determine if another processor is performing a conflicting access.

Semantically, a transaction must retain exclusive ownership over its written blocks, and non-exclusive ownership over its read blocks, until it reaches commit. Because ownership is usually associated with cache residence, any coherence protocol capable of detecting ownership conflicts can also detect transaction conflicts at no extra cost. However, limited cache capacity and associativity lead to replacements of active transactional blocks, thus breaking the ownership-cache residence connection that basic conflict detection relies upon. In order to support transactions of an arbitrary size, HTMs should ensure isolation in the presence of overflowed transactional blocks (evicted from the private cache level). Thus, a transactional node needs to see the coherence traffic for blocks that are no longer locally cached. While this happens naturally in systems with snoopy-based cache coherence, like the original TM proposal by Herlihy and Moss [5], it constitutes an abnormal behaviour for a directory-based protocol that maintains coherency over an unordered, point-to-point network, as that of a tiled CMP.

The introduction of the so-called *sticky states* in directory-based protocols [8][12] basically consists of using the directory entry to track the current transactional owner of an evicted block, and forward requests for that block to the transactional owner so that it can detect conflicting accesses that try to revoke its transactional ownership. Somehow, this can be regarded as a timid first step towards the fusion of cache coherence and conflict detection. Such combination of two seemingly independent mechanisms is not new, but it was already a fundamental part of TCC [4], an HTM in which lazy conflict detection and snoopy coherence were merged to provide a consistency model based on transactions.

In this paper, we propose a novel approach to eager conflict detection that further extends a directory protocol in order to provide a fast detection scheme in tiled CMP architectures. By comparing our proposal with an HTM system such as LogTM-SE [12], we observe several advantages of implementing conflict detection at the directory level instead of at the cache level. First and foremost, detection itself is accelerated, as conflicts are always detected in one hop instead of two. Considering that one of TM's fundamental principles is to achieve programming ease by allowing coarse-grained transactions, it is of great importance that conflicts are handled as efficiently as possible, as they are likely to occur often when the programmer relies on large transactions. Indeed, most of the transactional workloads from the Stanford Transactional suite (STAMP) [3] already pose this high-conflict behaviour, as shown in [10]. Second, by detecting

conflicts faster, the proposed TM system reacts more rapidly to high-contention scenarios and has the potential to avoid many aborted transactions, improving performance. Simulation results using GEMS (*General Execution-driven Multi-processor Simulator*) show that our conflict detection approach obtains reductions in execution time of 15% on average for the selected benchmarks, with better performace gains – up to 55% – for those workloads that suffer frequent transaction conflicts.

The rest of the paper is organized as follows: Section 2 briefly describes the different approaches to conflict detection adopted by some of the most relevant contributions to hardware transactional memory, and motivates our work. In Section 3 we describe our directory-based conflict detection scheme. Section 4 evaluates the performance of our proposal, comparing it to an ideal LogTM-SE system. We end with Section 5, which summarizes the main conclusions of this study and presents our future work.

2 Motivation and Related Work

In the early nineties, Herlihy and Moss introduce *Transactional Memory* (TM) [5] as a hardware alternative to lock-based synchronization. Their proposal relies on a snoopy coherence protocol to detect conflicting accesses, providing atomic accesses to several independent memory locations. More than a decade later, Hammond *et al.* present TCC, *Transactional Coherence and Consistency* [4], a novel coherence and consistency model based on transactions. The TCC system is also built upon a broadcast network that allows transactions to snoop commit traffic to maintain coherence and detect possible dependence violations (conflicts). Later on, several proposals such as UTM [1] or VTM [9] focus on hardware schemes that provide virtualization of transactions, i.e., support for transactions of unlimited duration, size and nesting depth. Both UTM and VTM monitor the coherence traffic for the transaction's cache lines to determine if another processor is performing a conflicting operation. In LogTM [8], Moore *et al.* combine transactional support with a conventional shared memory model, also taking the coherence protocol as a means to perform conflict detection. LogTM-SE [12] is a subsequent refinement that decouples transactional support from caches using hash signatures to detect conflicting threads.

Some of these HTM proposals perform transactional book-keeping by extending each cache entry with R/W bits [4][8]. Despite losing the information needed to perform conflict detection when when transactional blocks are evicted from the cache, these systems manage to guarantee isolation in this circumstances at a performance cost. On one hand, TCC [4] enforces transaction serialization by letting a transaction write its results directly to shared memory. On the other hand, LogTM [8] lets blocks leave the cache, and modifies a directory coherence protocol with *sticky states* so that the overflowed cache keeps receiving forwarded requests and performing conflict detection on the evicted blocks. LogTM's approach of lazily cleaning up sticky states suffers from frequent false positives when overflows become more frequent, due to stale directory information. Other

HTM designs opt for per-thread hash signatures to encode address sets using Bloom filters [12][3]. Under this alternative, transactional blocks that overflow cache are no longer a problem, as the information needed to detect conflicts is decoupled from the data block and stored at the core level. However, due to their conservative encoding, hash-signatures may signal a conflict when none exists (a false positive), causing unnecessary rollbacks that degrade performance. The ratio of false positives becomes significant when the transaction footprint grows, disencouraging the programmer from using coarse grain synchronization and somehow jeopardizing one of the main goals of TM.

2.1 Why Detect at the Directory Level

Up until now, conflict detection has always been performed at the private cache levels of the memory hierarchy. This makes the most sense when private caches are able to snoop on every memory transaction that takes place across the system, by being connected to a shared, ordered network like a bus [5][4]. However, in more scalable networks where directory-based protocols are more appropriate to maintain coherence, a cache only observes the requests for those blocks that are locally cached. Despite this substantially different scenario, eager conflict detection schemes that rely on directory protocols have so far implicitly inherited the same style of *private-level* conflict detection [8][12].

In this context, a reason why the directory is best suited for conflict detection is its location. From the perspective of a memory transaction, L1 caches are *end-points* – a request's origin or destination – whereas the L2 directory acts as a *middle-point* that orchestrates the traffic – routing requests so that they arrive at their destination –. As end-points entities, L1 caches are not a straightforward location to perform conflict detection: For them to detect conflicts on their evicted transactional blocks, the directory needs to behave abnormally and forward requests for blocks that are no longer cached at the private level. The directory, however, is not only a middle point that naturally observes all the traffic for its mapped blocks, but also the *first stop* of any request message, thus becoming the perfect location to provide a fast (one-hop) detection scheme.

Besides its privileged location, the directory's role makes conflict detection a simple addition to its responsibilities. Considering that i) the directory is in charge of tracking each cached block's ownership[1], and that ii) transactional ownership is connected to cache residence in the common case (except for evicted transactional blocks), the directory has most of the information required to detect memory accesses that attempt to revoke a node's transactional ownership over its read and written blocks. Therefore, such an extension in its functionality becomes a natural evolution of its role within a TM system.

Furthermore, an HTM with directory-based conflict detection also mitigates the performance implications of signature false positives. In systems where not every memory block has its corresponding directory entry, the directory controller still needs to keep detecting conflicts for those transactional blocks that

[1] We use the term ownership throughout this paper to stand for cache residence, independent of the state of the block in the cache(s).

are spilled from the coherence level. Per-bank signatures in the directory are a good solution to this problem because the number of transactional active blocks that overflow this level (i.e. the L2 cache in a CMP) is insignificant in comparison to total number of blocks accessed by a transaction, and so is the probability of false positives, compared to using signatures to encode the entire access set.

3 Directory-Based Conflict Detection

Using the directory to check for conflicts over blocks that remain cached by transactional owners does not 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 non-exclusive ownership of B. When R's read request arrives to 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 immediately detect a conflict on B and take the appropriate actions to resolve it. To do this, the directory only needs to keep a record of which cores are executing a transaction at any moment. To this end, our base conflict detection scheme explicitly notifies the directory about transaction begin and transaction commit. A simplistic solution could consist of sending dedicated begin/commit messages and waiting for acknowledgment before resuming the execution.

Once the directory knows that a core P is executing a transaction, it could immediately start to detect conflicts for all accesses to blocks locally cached by P. However, doing so would lead to many unnecessary conflicts since not all cached blocks may have been accessed by the transaction – in other words, cache residence does not necessarily imply transactional ownership –. In order to avoid them, the naive approach of our base scheme is flush-clearing the local data cache at transaction begin, writing back all modified/exclusive blocks. Following this simple approach, the first reference to each data block from inside a transaction misses in the local cache, so that the directory observes all transactional addresses and perform the book-keeping required for conflict detection. While flush-clearing the data cache is clearly not desirable, those workloads composed of large transactions should not be too affected, as flushes happen infrequently. The main drawback of flushing appears in applications with short, frequent transactions, in which not only the transaction but also the following code, find *an almost* empty data cache. Most misses suffered by the post-transactional code (which presumably operates on local data) are directly caused by the recent cache-flush. For this reason, more sophisticated schemes are necessary, which would allow for conflict detection at the directory level without requiring a cache-flush on every transaction begin.

So far, we have assumed in our elaboration that transactional ownership implies cache residence, but that is not always the case because transactional blocks can exceed the capacity or associativity of the local cache. Since no explicit

information about a transaction's R&W sets is stored at the core level (no signatures), the directory needs to track transactional blocks that are evicted from the private level while the transaction runs, in order to keep detecting conflicting accesses on those blocks. To this end, we introduce the concept of *Transaction Serial Number* (XSN), a small, reusable, *per-core* identifier that is used by the directory to tag transactional blocks and maintain a correspondence between a block and its owner transaction(s). While we have not determined the ideal size of the transaction serial number, performing lazy clean-up of non-matching XSNs greatly reduces the overhead of these identifiers. A few bits per XSN should suffice to avoid virtually all false conflicts due to XSN reuse. Nonetheless, these false positives caused by stale XSNs that become *fresh* only affect the performance but not the correctness of the transactional execution.

Hardware Requirements. On the core side, each core has a counter (*XSN register*) that contains the XSN assigned to its last/current transaction. The *XSN register* is incremented every time the instruction begin_transaction is executed – hence also after an abort –. Its content is copied to all the outgoing transactional messages (set to zero for all non-transactional requests), allowing the directory to differentiate between transactional and non-transactional requests. On the directory side, each directory bank keeps a vector of XSN's (*global XSNs*), one XSN per core. The corresponding XSN of the vector is updated on every begin_transaction with the XSN indicated in the message, while it is set to zero ("not in transaction") upon arrival of a commit_transaction message. As for the directory entry, each one is augmented with a new field, *xact owners XSN*, whose function is to keep a correspondence between the block and its current transactional owner(s). For simplicity, we can think of this field as a vector with as many XSN as cores. In practice, each entry does not need to store one XSN per core; instead, the hardware overhead of this mechanism can be minimized by having a separated XSN buffer that the directory controller uses "on demand". Finally, the directory uses a set of per-core signatures to track those transactional active blocks that are evicted from the directory level. Before the replacement, the address is added to the signature of its transactional readers/writer. These signatures are only checked in case a request misses at the coherence level, and cleared on transaction commit/abort.

Operation. By jointly considering both a block's XSNs and the global XSN vector, the directory can unequivocally determine if a certain block is owned by some currently running transaction(s) or if, on the contrary, some transaction that made use of it has committed/aborted. The basic idea behind this mechanism is that a block is considered part of a transaction's R/W set running in P when the P-th XSN of its *xact owners* matches the P-th XSN of the global XSN vector. Comparing a block's XSN against the global XSN vector, the directory tracks transactional ownership even when the block is not privately cached, enabling conflict detection regardless of the actual location of the block. Figure 1 illustrates the proposed conflict detection mechanism, showing how the forementioned hardware elements work together to provide fast conflict detection at the

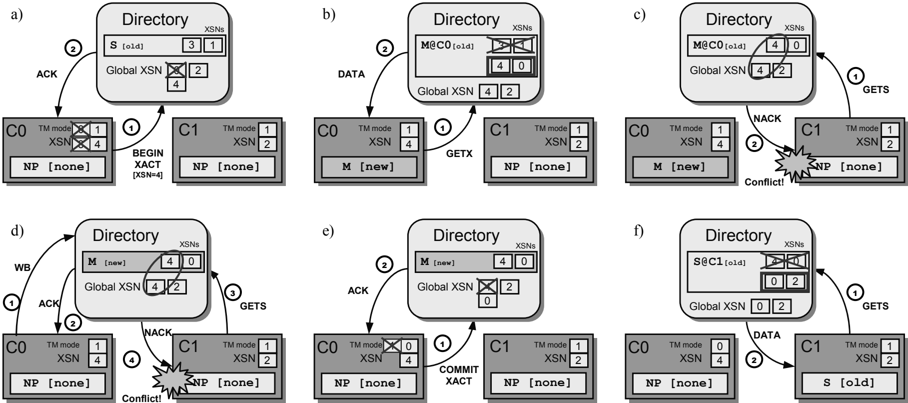

Fig. 1. Examples of Directory-Based Conflict Detection

directory level. The figure also shows the coherence state for one block at the directory and in two core's private caches, as well as the block's XSNs. The directory's global XSN vector and each core's XSN register are also shown.

Core 0 (C_0) begins its transaction by incrementing its XSN register and sending it to the directory through an explicit *begin_transaction* message (Figure 1 a). The directory uses this message to update its global XSN register and responds with an acknowledgment, allowing the core to begin its transaction. In Figure 1 b, C_0 attempts to write a block, but misses in its private cache and sends an exclusive request to the directory. The directory checks the block's *xact owners XSN*, observes that the reader transaction in C_1 is no longer running and sends exclusive data to C_0, setting both the state and *xact owners* accordingly – lazily clearing C_1's stale XSN –. In Figure 1 c, the transaction in C_1 tries to read the same block, missing in its L1. Comparing *xact owners* and the global XSN, the directory finds out that C_0 is a transactional owner, and then it uses the coherence state to find out whether C_0 is a reader or a writer. In this case, the block is not in shared state, which means C_0 is a writer and hence the directory detects the conflict. Figure 1 d is an example of the out-of-cache conflict detection: C_0 writebacks the block and when C_1 retries its read request, the directory detects the conflict once again, since the writeback did not change the xact owners. At last, C_0 commits its transaction, notifies the directory (Figure 1 e), allowing C_1 to finally obtain a shared copy of the block (Figure 1 f).

3.1 Enhancements to the Base Detection Scheme

Augmenting the Private Cache to Avoid Flushing. Instead of flush-clearing the L1 cache on every transaction begin, a more elaborated solution could serve those accesses that hit on privately cached data immediately, and allow the core to continue its execution without any extra delay, while sending a notification down to the directory (off the critical path). This *report* messages contain the new transaction serial number of the just-started transaction and are used to

update the block's *xact owners XSN* vector at directory. A *Transactional* bit must be added to each L1 cache line, to deal with forwarded conflicting request as well as to reduce the number of reports sent down to the directory. This bit is set each time a block is accessed and flush-cleared on transaction commit. Report messages are only sent out if the bit is not set. If a race occurs between a remote request and a report message, so that the remote message arrives before at the directory, the core receives the forwarded request and it signals a conflict if it finds the *Transactional* bit set for the block. Eventually, the directory information for that block will be updated with the new XSN and subsequent conflicting requests will be handled entirely at the directory level.

Reporting Begin/Commit to the Directory without Extra Delay. Instead of sending one begin and one commit message to each directory bank for each transaction, a more scalable solution could use *on-demand piggybacking* for these reports. This can be done by recording which L2 banks the core has accessed during the transaction, using a simple bit-vector that is updated by the address-to-bank mapping logic on each L1 miss and cleared after commit. In this way, the begin transaction report is inserted as a field (XSN) in the first request message sent to a directory bank, without delaying the execution of the transaction. At transaction commit, only the appropriate directory banks need to be notified, according to the forementioned bit-vector.

4 Evaluation

In this section, we evaluate the performance of the proposed conflict detection scheme (DirCD). We use the LogTM-SE hardware transactional memory system as the basis of our simulations, and we modify it to introduce two versions of our proposal: a naive implementation that empties the L1 cache on every begin_transaction (*DirCD+L1Flush*), and an enhanced version that avoids cache flushing (*DirCD+NoL1Flush*). To provide a better perspective over the results, we also consider an identical configuration to the baseline LogTM-SE system that flush-clears the L1 cache on transaction begin (*CacheFlush*). We compare these two DirCD flavours against an ideal configuration of LogTM-SE in which perfect signatures are used to track R/W sets and detect conflicts (*Base*).

For simplicity, our version of *DirCD+NoL1Flush* does not use *hit-report* messages nor L2-overflow signatures; instead, we approximate a *flush-free* configuration by relying on the original address signatures of LogTM-SE, which we have made directly accessible to the directory conflict detection logic. The resulting implementation emulates a more sophisticated DirCD-based system, which incorporates the enhancements described in 3.1. Regarding conflict resolution (CR), it is now performed at the directory level, although the CR policy remains fixed – requester stalls, with conservative deadlock avoidance –. Our DirCD implementation also reuses the functionality that the simulator provides for LogTM-SE, so that the directory does not track timestamps, possible cycles nor does it issue abort messages when a possible deadlock is detected. Lastly, our

conflict detection scheme is evaluated without restricting the size of transaction serial numbers, and using a full XSN-vector in each directory entry.

4.1 Summary of LogTM-SE

LogTM is a hardware transactional memory system proposed by the Multifacet group at the University of Wisconsin-Madison. LogTM implements eager version management and eager conflict detection. It uses a per-thread log in cacheable virtual memory that contains address and old values of memory locations modified by the current transaction. It extends a directory protocol in order to perform conflict detection of evicted blocks by using *sticky states*. LogTM-SE (*Signature Edition*) is a refined version of LogTM in which R/W sets are tracked using hash signatures. We use LogTM's basic algorithm to detect potential deadlocks using timestamps: A processor sets a bit if it nacks an older transaction; if in turn it receives a nack from an older transaction, this represents a potential cycle and the transaction aborts. The abort traps to a software handler, which walks the transaction log and restores the old values into memory. The system uses randomized linear backoff to reduce contention after an abort.

4.2 Simulation Methodology and Environment

We use a full-system execution-driven simulation based on the Wisconsin GEMS toolset [7], in conjunction with Virtutech Simics [6]. We use an implementation of the LogTM-SE protocol and the detailed timing model for the memory subsystem of GEMS v2.1, with the Simics in-order processor model. Simics provides functional correctness for the SPARC ISA and boots an unmodified Solaris 10.

We perform our characterization on a tiled CMP system, as described in Table 1. We use a 16-core configuration with private L1 I&D caches and a shared, multibanked L2 cache consisting of 16 banks of 512KB each. The L1 caches maintain inclusion with the L2. The cores and L2 cache banks are connected through a 2D mesh network. The private L1 data caches are kept coherent through an on-chip directory (at L2 cache banks), which maintains a bit vector of sharers and implements the MESI protocol. We compare our proposal against an ideal

Table 1. System parameters

MESI Directory-based CMP	
Core Settings	
Cores	16, single issue, in-order, non-memory IPC=1
Memory and Directory Settings	
L1 I&D caches	Private, 32KB, split, 2-way, 1-cycle latency
L2 cache	Shared, 8MB, unified, 4-way, 12 cycle-latency
L2 Directory	Full bit vector, 6-cycle latency
Memory	4GB, 300-cycle latency
Network Settings	
Topology	2D Mesh (4x4)
Link latency	1 cycle
Link bandwidth	40 bytes/cycle

Table 2. Benchmarks and inputs

Benchmark	Input	Benchmark	Input
DELAUNAY	Mesh gen3.2, min. angle 30	BARNES	4096 bodies
GENOME	8K segments, gene length 256, segment length 16	CHOLESKY	tk14
BAYES	32 variables, 1K records, 2 parents, 20%chance	RAYTRACE	teapot
KMEANS	16/16 clusters, thres. 0.05, 2048 16-dim points		
VACATION	64K entries, 4K tasks, 8 queries, 10 rel, 80 users		

implementation of LogTM-SE in which conflict detection uses *perfect* signatures – mere lists of addresses read/written by the transaction – instead of actual hash signatures that lead to unnecessary conflicts as a result of false positives.

For the evaluation, we use five transactional benchmarks extracted from the STAMP suite [3]. These benchmarks use coarse-grain transactions to execute concurrent tasks on irregular data structures such as graphs or trees. We have also selected a few *non-transactional* workloads from the SPLASH-2 suite [11], in order to evaluate our proposal with substantially different applications. Note that the latter may not be representative of future transactional applications, and are just included for comparison purposes.

4.3 Results

Figures 2 and 3 summarize the performance evaluation of the proposed directory-based conflict detection (DirCD) mechanism. We can observe how the optimized version of our proposal (DirCD+NoL1Flush) outperforms LogTM-SE in every STAMP transactional benchmark as well as in raytrace, and obtains similar results in non-transactional applications from SPLASH such as barnes or cholesky.

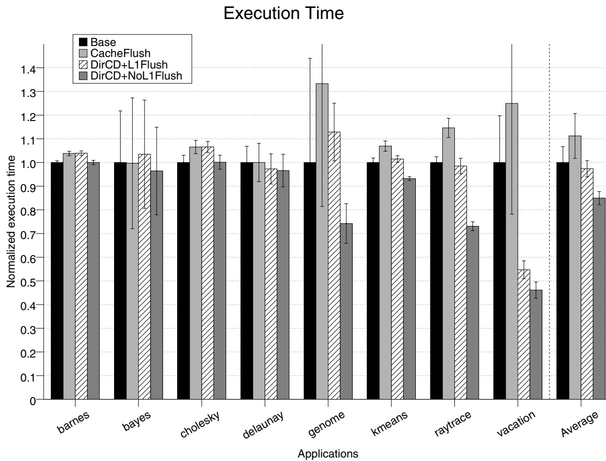

Fig. 2. Normalized execution time

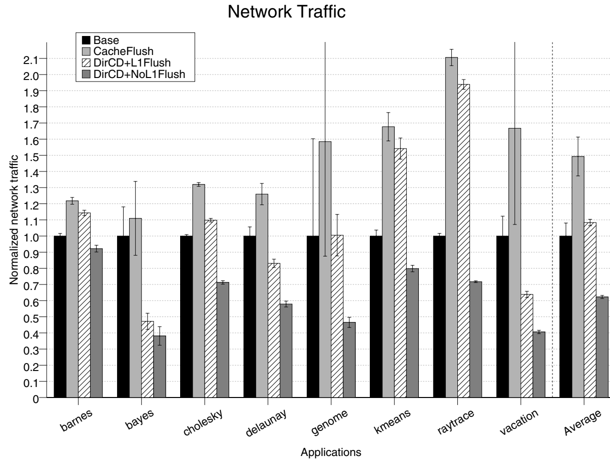

Fig. 3. Normalized network traffic

Table 3. L1 Miss Rate. Committed vs. Aborted Transactions

	L1MissRate				Commits	Aborts			
	Base	Flush	DirCD Flush	DirCD NoFlush		Base	Flush	DirCD Flush	DirCD NoFlush
barnes	1,44%	1,71%	1,74%	1,46%	17399	316	333	306	296
bayes	1,93%	2,40%	3,67%	3,73%	526	1315	1368	1363	1285
cholesky	0,87%	1,25%	1,31%	0,87%	6567	73	41	39	75
delaunay	4,26%	6,13%	9,32%	7,66%	6312	16462	15491	15137	15408
genome	3,15%	4,49%	7,75%	2,88%	5234	3178	3908	2941	1120
kmeans	0,54%	1,08%	1,14%	0,53%	8238	6883	8172	6059	2693
raytrace	1,88%	5,14%	6,13%	2,70%	47766	203968	174393	154819	171681
vacation	4,14%	7,42%	9,32%	6,61%	4096	10573	10596	3233	2953

The performance gain of DirCD+NoL1Flush is considerable for genome (25%), raytrace (27%) and vacation (55%), three benchmarks that suffer many conflicts, as shown in the last four columns of Table 3. Other transactional workloads such as bayes, delaunay or kmeans present more modest improvements in their execution time of 3 to 7%.

First, we start by analyzing the performance degradation caused by flushing the L1 cache on every transaction, shown by the CacheFlush bar in Figure 2. This will help us understand the obtained results for our flush-based detection scheme (DirCD+Flush). As expected, flushing causes an increase in the L1 miss rate (shown in Table 3) that has a direct effect over the execution time of all benchmarks, particularly for raytrace, vacation and genome (up to 15-33%). The case of raytrace is clear: it executes a very high number of transactions (see "Commit" column of Table 3), most of which have a very small size (basically read&increment a ray id), and thus flushing the L1 on each transaction continuously leaves the post-transactional code with an almost empty data cache. The increase in on-chip network traffic (see Figure 3, is more dramatic for raytrace

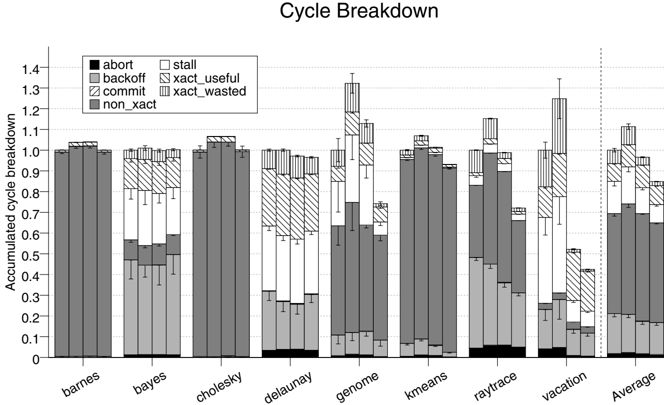

Fig. 4. Normalized cycle breakdown

than for any other benchmark, as a result of its fine-grain, abundant synchronization. However, the effects on execution time and network traffic are different for genome, a benchmark that spends the majority of its runtime in transactional code (see Figure 4). In this case, the degradation does not come from non-transactional cache misses, but from an increased number of aborted transactions – see Table 3 – that arises as a result of having transactions that span a longer period of time (probability of conflict is directly proportional to the duration).

By avoiding the flush, the DirCD+NoL1Flush configuration offers a clearer look upon the benefits of fast conflict detection than the flush-based DirCD version, as the latter introduces overheads that shadow the potential gains of our proposal over the base LogTM-SE system. The remarkable speedup achieved by genome, raytrace and vacation is not directly caused by faster conflict detection, but it happens as a result of it. As shown in Table 3 (columns "Abort"), our faster detection scheme manages to reduce the total number of aborted transactions for many benchmarks, and the gains are higher for those applications in which conflicts are not a rare event. In vacation and genome, respectively, 75% and 65% of the aborted transactions are avoided by our DirCD+NoL1Flush scheme, in comparison to the base LogTM-SE configuration. This is due to the early detection and resolution of contended situations achieved by our approach.

In the LogTM system, when a conflict is detected, the requesting processor stalls. If the conflict is detected sooner, as in the proposed scheme, the stall will likely last longer. Since execution time is determined by how quickly the system serializes conflicting transactions, detecting conflicts quicker does not speed up the execution when stalling the conflicting transaction(s) is enough to solve the conflict. However, our DirCD configuration does achieve a faster serialization of multiple conflicting transactions, when multiple conflicts cannot be resolved by stalling, but they require some transaction(s) to be aborted. In this case, the sooner the system detects the conflict, the faster it can take action and abort the

appropriate transactions. The directory not only is able to detect the conflict in one hop, but it can also take action without having to wait until the conflicting block is in a base state (unlike the base approach that relies on forwarded coherence traffic), contributing to even faster detection/action. Aborting conflicting transactions earlier reduces the effect of pathological execution patterns such as *futile stall* and other conflicting interactions that affect eager CD systems like LogTM-SE [2]. The remarkable reductions in the execution time of vacation and genome are due to this quicker and more effective response. Compared to the base case, our DirCD scheme causes more aborts at first, but later on this allows more transactions to execute concurrently without interference and commit, reducing the overall number of aborts as well as the total stalled time and wasted work (see Figure 4). Quantitatively speaking, we observed this behaviour by taking a look at the first two million cycles of execution of vacation, given the two configurations (Base and DirCD+NoL1Flush) and the same random seed: we found that the former manages to commit 108 transactions by aborting 748 in that period of time, while the latter commits more than twice as many (236) at the cost of aborting around 50% more (1135). The same kind of pattern is found in other simulations with different seeds for the same benchmark.

5 Conclusions and Future Work

In this paper, we present a new approach to conflict detection targeted to TM systems built over a tiled CMP architecture. For these systems, we believe the directory constitutes a natural location for this basic transactional mechanism, and claim that extending its role to include such functionality is a natural evolution of its responsibilities within a cache coherent TM system. We propose a novel book-keeping scheme that augments each directory entry with transaction serial numbers, and describe how the detection is carried out with little assistance from the cores. The results show how the fast conflict detection achieved by our design reduces the number of aborted transactions in workloads that suffer frequent conflicts, resulting in average reductions of 15% in execution time for the selected benchmarks.

Bringing together two independent mechanisms like cache coherence and conflict detection creates a synergistic relationship that opens up a wide spectrum of new opportunities within the TM system. When combined onto the same hardware logic, both entities can cooperate *symbiotically* and accomplish new functionalities that cannot be achieved otherwise. For example, by giving the directory control over the outcome of a transaction, speculation can be applied in a variety of ways, for example, to continue the transactional execution past the occurrence of conflicting accesses. Our conflict detection mechanism already provides a commit_request/commit_ack message exchange, and could naturally support a commit_deny message that forces a transaction to abort if the speculation failed.

Acknowledgements. This work has been jointly supported by the Spanish MEC and European Commission FEDER funds under grants "Consolider Ingenio-2010 CSD2006-00046" and "TIN2006-15516-C04-03', as well as by the EU FP6 NoE HiPEAC IST-004408. Rubén Titos is supported by a research grant from the Spanish MEC under the FPU National Plan (AP2006-04152). The authors would like to thank the anonymous reviewers for their helpful insights.

References

1. Ananian, C.S., et al.: Unbounded transactional memory. In: Proc. of the 11th Int'l. Symposium on High-Performance Computer Architecture, pp. 316–327 (February 2005)
2. Bobba, J., Moore, K.E., Yen, L., Volos, H., Hill, M.D., Swift, M.M., Wood, D.A.: Performance pathologies in hardware transactional memory. In: Proc. of the 34rd Annual Int'l. Symposium on Computer Architecture (June 2007)
3. Cao Minh, C., et al.: An effective hybrid transactional memory system with strong isolation guarantees. In: Proc. of the 34th Annual Int'l. Symposium on Computer Architecture (June 2007)
4. Hammond, L., et al.: Transactional memory coherence and consistency. In: Proc. of the 31st Annual Int'l. Symposium on Computer Architecture, pp. 102–113 (June 2004)
5. M. Herlihy and E. B. Moss. Transactional Memory: Architectural support for lock-free data structures. In *Proc. of the 20th Annual Int'l. Symposium on Computer Architecture*, pages 289–301, May 1993.
6. Magnusson, P.S., et al.: Simics: A full system simulation platform. IEEE Computer 35(2), 50–58 (2002)
7. Martin, M.M.K., et al.: Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) toolset. Computer Architecture News, 92–99 (September 2005)
8. Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A.: LogTM: Log-based transactional memory. In: Proc. of the 12th Int'l. Symposium on High-Performance Computer Architecture, pp. 254–265 (February 2006)
9. Rajwar, R., et al.: Virtualizing transactional memory. In: Proc. of the 32nd Annual Int'l. Symposium on Computer Architecture, pp. 494–505 (June 2005)
10. Titos, R., Acacio, M.E., García, J.M.: Characterization of conflicts in log-based transactional memory (LogTM). In: Proc. of the 16th Euromicro Int'l. Conference on Parallel, Distributed and Network-Based Processing, pp. 30–37 (February 2008)
11. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: Characterization and methodological considerations. In: Proc. of the 22nd Annual Int'l. Symposium on Computer Architecture, pp. 24–36 (June 1995)
12. Yen, L., et al.: LogTM-SE: Decoupling hardware transactional memory from caches. In: Proc. of the 13th Int'l. Symposium on High-Performance Computer Architecture, pp. 261–272 (February 2007)