# Characterization of Conflicts in Log-Based Transactional Memory (LogTM)

J. Rubén Titos, Manuel E. Acacio, José M. García
Universidad de Murcia
Dpto. Ingeniería y Tecnología de Computadores
Campus de Espinardo, 30100 Murcia, Spain
{rtitos,meacacio,jmgarcia}@ditec.um.es

## Abstract

*The difficulty of multithreaded programming remains a major obstacle for programmers to fully exploit multicore chips. Transactional memory has been proposed as an abstraction capable of ameliorating the challenges of traditional lock-based parallel programming. Hardware transactional memory (HTM) systems implement the necessary mechanisms to provide transactional semantics efficiently. In order to keep hardware simple, current HTM designs apply fixed policies that aim at optimizing the most expected application behaviour, and many of these proposals explicitly assume that commits will be clearly more frequent than aborts in future transactional workloads.*

*This paper shows that some applications developed under the TM programming model are by nature prone to experience many conflicts. As a result, aborted transactions can get to be common and may seriously hurt performance. Our characterization, performed with truly transactional benchmarks on the LogTM system, shows that certain programs composed by large transactions suffer indeed very high abort rates. Thus, if TM is to unburden developers from the programmability-performance trade-off, HTM systems must obtain good performance levels in the presence of frequent aborts, requiring more flexible policies of data versioning as well as more sophisticated recovery schemes.*

## 1 Introduction

Over the past few years, it has become clear that neither instruction level parallelism nor higher clock frequencies are feasible strategies to further enhance single processor performance, and even investing the available transistor budget in larger on-chip caches is reaching the point of diminishing return. These have been the pillars that sustained an extraordinary exponential growth in processor performance, achievement that customers are used to experience one generation of processors after another. For these reasons, we are currently witnessing a fundamental paradigm shift towards multicore architectures that has broad repercussions at both hardware and software levels. Chip multiprocessors (CMPs) are becoming ubiquitous and every manufacturer's road-map shows that the trend in the near future will be towards chips with steadily more and more execution cores.

As CMPs come to be the mainstream, the way software is developed must be reconsidered if computer systems are to maintain such high expectations of application performance in the near future. Programmers will need to write concurrent programs that exploit thread-level parallelism available in applications, in order to use on-chip resources effectively. However, the practical success of CMP-based systems is limited by the difficulty of parallel application development. Writing concurrent programs under conventional programming models is still a fairly complex task that only experienced developers can tackle, because it poses a demanding trade-off between performance and correctness. Extracting fine-grain parallelism is a laborious, time consuming and error prone task, whereas programming conservatively can shrink development time at the cost of sacrificing performance and scalability. Unless other models are developed that both ease concurrent programming and allow the system to extract plenty of parallelism, the performance potential of CMPs systems will be bounded to multiprogrammed workloads and a few server-domain applications.

To this end, Transactional Memory (TM) has aroused as a promising approach to ease parallel programming while still producing efficient multithreaded programs. Under the TM model, the programmer declares *what* regions of the code must appear to execute atomically and in isolation (critical sections), leaving the burden of *how* to provide such properties to the underlying levels. The TM system then executes optimistically transactions, stalling or aborting them whenever real run-time data conflicts occur amongst concurrent transactions. The TM model replaces explicit syn-

0-7695-3089-3/08 $25.00 © 2008 IEEE
DOI 10.1109/PDP.2008.63

chronization mechanisms like locking, thus getting rid of problems such as priority inversion or deadlock, and also simplifying code composability. Summarizing, the TM programming model tends to decouple performance pursuit from programming productivity, and it has the potential to achieve both goals simultaneously.

Despite being one of TM's fundamental principle, programming ease has been implicitly left outside the scene of hardware transactional memory (HTM) research. All current HTM designs use simple recovery mechanisms to resolve conflicts, and some of them are designed to accelerate commits at the cost of slowing down rollbacks [1, 11, 2]. These policies behave well for applications that spend a small portion of their run-time in short transactions that usually experience few aborts. However, when conflicts become more numerous as transaction size grows, these optimistic policies may cause aborts to have broad repercussions on performance. On one hand, it is reasonable to expect that programmers will make frequent use of coarse-grain transactions. On the other, compilation techniques find extremely difficult to extract fine-grain parallelism from applications, and so they may also generate code often composed by coarse-grain tasks wrapped inside transactions. Both for programmers and compilers, producing code abundant in long transactions results not only in a larger amount of wasted work per abort, but also in a higher probability of abort. These side-effects of coarse-grain transactional code may cause serious performance degradation problems for applications with frequent inter-thread (inter-transaction) communication. In our opinion, current HTM research has failed so far to produce designs that open the TM model to the common developer, because these designs still depend on skilled programmers or advanced parallelizing compilers to extract fine-grain transactions, in order to achieve good levels of performance.

In this paper, we make two main contributions. First, by focusing on the importance of dealing efficiently with aborted transactions, we take a novel perspective on the design dimensions of a hardware transactional memory system (HTM). Unlike most of the related literature, we find at least dubitable the general assumption that commits will be much more frequent than aborts in future transactional workloads. If the TM model is to simplify parallel programming, HTMs should not add more pressure on programmers (or compilers) than the own task of parallelization strictly requires. From our point of view, HTMs should be designed to achieve good performance even in presence of frequent conflicts and aborts, since these systems will face applications with a great variability in transaction sizes, thread communication patterns, etc. In this context, we re-analyze the influence on abort penalty of data version management, conflict detection and resolution policies for hardware transactional memory systems.

As a second contribution, this paper tries to quantitatively show that aborts will be fairly frequent in some applications developed with the TM paradigm. We present the first characterization of conflicts in truly transactional benchmarks composed by large, coarse-grain transactions. We perform our evaluation on a popular log-based transactional system (LogTM) [11], using the Stanford Transactional Applications for Multi-Processing (STAMP) [4].

The rest of the paper is organized as follows: Section 2 covers the related work, briefly describing the most relevant contributions to hardware transactional memory. In Section 3 we discuss the influence on abort penalty of the main design dimensions –namely, data version management, conflict detection and conflict resolution– in a transactional memory system. In Section 4 we perform a characterization of conflicts in LogTM using a suite of transactional workloads. We end with Section 5, that summarizes the main conclusions of this study.

## 2 Related Work

In the early nineties, Herlihy and Moss introduced *Transactional Memory* (TM) [8] as a hardware alternative to lock-based synchronization. Their main idea was to generalize the LL/SC primitives in order to perform atomic accesses not to one but to several independent memory locations, thus eliminating the need for protecting critical sections with lock variables. Almost a decade later, architects began to recover their interest in transactions at a hardware level. Rajwar and Goodman's *Transactional Lock Removal* (TLR) [14] was the first to apply the concept of transaction to the execution of lock-protected critical sections, merging the idea of *Speculative Lock Elision* (SLE) [13] with a timestamp-based conflict resolution scheme. Hammond *et al.* present *Transactional Coherence and Consistency* [7], a novel coherence and consistency model that uses continuous transactional execution. The novelty of TCC stems from its *all transactions, all the time* philosophy, where transactions are the basic unit of parallel work, synchronization, memory coherence and consistency. Later on, several proposals such as UTM [1] or VTM [15] focus on hardware schemes that provide virtualization of transactions, i.e., support for transactions of unlimited duration, size and nesting depth. However, both of them achieve this goal by introducing large amounts of complexity in the processor and the memory subsystem. Moore *et al.* take a more evolutionary approach to transactional memory in LogTM [11]. Unlike TCC, LogTM combines transactional support with a conventional shared memory model, enabling a more gradual change towards transactional systems. The authors present a log-based implementation of transactional memory that makes commits fast by storing old values to a per-thread log in cacheable virtual memory, and enables conflict de-

tection of evicted blocks through an elegant extension to a MOESI directory protocol. LogTM has been subsequently refined to better support nested transactions [12] and to decouple transactional support from caches [17]. This latest improvement, called LogTM-SE (*Signature Edition*), borrows the idea of using hash signatures to detect conflicting threads, introduced by Ceze *et al.* in [5]. LogTM's approximation of making commits fast has also inspired OneTM [2], a recent contribution by Blundell *et al.* that uses a cache to reduce the frequency with which transactions overflow on chip resources, and simplifies the way the system handles overflowed transactions.

# 3 Influence of HTM design dimensions on abort penalty

## 3.1 Data version management

A TM system must satisfy the property of atomicity at any time: transactions must execute completely, or else not execute at all. To maintain atomicity, a transactional system must make the updates performed by a transaction visible to the rest of the system at once, and therefore it must retain both new and old values of all modified memory locations during the entire execution of the transaction. The data versioning policy dictates how the system handles the simultaneous storage of both versions, and it constitutes a major design point of the system.

TM systems that implement eager version management copy the old value to a separate data structure (i.e. a transaction log) upon each write, and then update the memory location with the new value. This policy makes commits fast and it is desirable in those scenarios where conflicts are rare. However, aborts are slowed down since the structure that keeps old values must be traversed in order to restore each memory location with its original content. Examples of proposed HTMs that use eager data versioning are UTM [1], LogTM [11] and others based on the latter [12, 17].

TM systems that make use of lazy version management keep old values in their memory locations until the commit phase, and then overwrite them with the new values, which are stored somewhere else in the meantime (i.e. in a speculative buffer). Since the old values stay in place, a system with lazy versioning can get rid of an aborted transaction quickly, simply by discarding the new values (flushing the speculative buffer/cache lines). At the cost of slowing down commits, this *fast-rollback* policy alleviates the performance degradation experienced by applications with frequent cyclic conflicts between transactions that lead to high abort rates. Among the systems that rely on lazy version management are TCC [6], LTM [1], VTM [15] and Bulk [5].

Some authors claim that an ideal transactional memory system should use eager version management [11], an statement based on the assumption that commits are much more frequent than aborts. This hypothesis is confirmed by the results of the same work, whose evaluation was performed using parallel applications from the SPLASH-2 suite. These benchmarks have been carefully optimized over the years to avoid synchronization overhead and effectively exploit fine-grain parallelism available in the code [16]. In such parallel codes, replacing lock/unlock by begin_transaction/end_transaction calls leads to programs that spend a small amount of their run-time in brief transactions, experience few conflicts and thus infrequent aborts. In our opinion, this behaviour may not be representative of future transactional memory workloads at all, because it goes against one of the main goals of TM –ease parallel programming by using coarse-grain transactions– or would not include the transactional code generated by parallelizing compilers. For this reason, realistic transactional benchmarks should reflect expected programmer practises such as the use of conservative synchronization, which leads to applications that spend almost all their run-time in large transactions. In these circumstances, the frequency of conflicts and aborts will depend on factors such as programmer's expertise and knowledge of the problem, inter-thread communication patterns, inherent parallelism available in the application, etc.

## 3.2 Granularity of conflict detection

A conflict between two concurrent transactions happens when a transaction's write set overlaps other transactions' read or write set. In other words, a conflict happens when two simultaneous transactions access the same memory location, and at least one of the accesses is a write. To detect such violations of isolation, an HTM must track the data both read and written by each transaction. This can be done at different levels of granularity, commonly either blocks or words. In the first case, only two bits per block are needed, but false conflicts may arise when concurrent transactions access different words of the same block. False conflicts may have a significant impact in application performance if they happen frequently enough. Tracking read and write sets at a word granularity avoids this undesirable effect at the cost of additional overhead ($2w$ bits per block, assuming $w$ words per block). Although the compiler can prevent potential false conflicts in some cases, detecting conflicts at block granularity creates yet another responsibility for the programmer. The avoidance of these spurious violations is an additional burden that clearly goes against the principle of parallel programming ease pursued by TM.

### 3.3 Conflict Detection and Resolution

Conflict detection strategies vary depending on when a processor examines the information of its R/W sets. Most HTM proposed to date implement conflict detection by extensions to ownership-based cache-coherence protocols [8, 1, 15, 11, 2]. These systems monitor the cache coherence traffic for the transactional blocks to determine if another processor is performing a conflicting access, according to the transaction's R&W sets. With this eager policy –sometimes also referred as pessimistic–, conflicts are detected as soon as they happen. As no other transaction can observe uncommitted state, early detection may improve performance by resolving some conflicts using stalls rather than drastic aborts. Eager conflict detection may reduce the amount of wasted work to be discarded if the transaction must be finally aborted to avoid deadlock. However, this policy can experience a series of execution patterns that harm performance –friendly fire, dueling upgrades, futile stall and starving writer–, as described by Bobba *et al.* in a recent paper [3].

Other HTM approaches [7, 5] face this design dimension with a different policy, called lazy or optimistic conflict detection. In these proposals, the check for conflicting accesses is delayed until transaction commit, and the resolution is always based on a committer-wins scheme. The committer transaction broadcasts its write set to the rest of the system, so that every other transaction can check against its R&W sets, and proceed to abort when necessary. A transaction T that performed a conflicting access early in its execution does not detect such violation until it receives the committer transaction's write set, which may as well happen when T is close to its end. All the computation carried out between the conflict and its detection must be redone, therefore consuming power and network bandwidth. A favourable side-effect of this policy is that a restarted transaction may find a significant amount of its data already in cache, since the previous (unsuccessful) execution acted as a prefetch mechanism. Anyhow, this policy leads to potentially larger amounts of wasted work than eager conflict detection, since all conflicts (even non cyclic ones) lead to aborts. This approach can suffer other undesirable pathologies such as serialized commit, restart convoy or starving elder, as shown in [3].

Regardless of the detection policy, HTM proposals rely on *discard-everything and restart* recovery mechanisms against violations of isolation. This approach keeps hardware simple and has little effect on performance for applications composed by short transactions, since aborts do not happen very often, and when they do, restarting the entire transaction requires only a small number of instructions to be re-executed. An ideal recovery mechanism would not just discard all work done, but instead would maintain as much of the partial results as possible and only redo those computations that must use new data versions obtained from the transaction that won the conflict. This is particularly beneficial when conflicts that lead to abort are detected at the end of long transactions, which are likely to be the common case in future parallel workloads developed with TM techniques [4]. In such scenario, current conflict resolution schemes may severely hurt performance, as significant amounts of *valid* work is wasted on each abort. Under the current discard-everything approach, the performance degradation suffered by transactional applications that experience high abort rates on large transactions can make programmers look for finer-grain transactions. The result is a similar programming effort to fine-grain locking, opposite to the very purpose of programming ease sought by TM.

## 4 Characterization

In this section, we perform a characterization of conflicts under a popular hardware transactional memory system such as LogTM [11]. We show that for some TM applications developed under the paradigm of large transactions, the frequency of aborts is high, thus invalidating the assumption that commits are clearly the common case in TM systems. This is the first characterization of conflicts on a hardware transactional memory system that uses representative benchmarks developed from scratch under the TM paradigm.

### 4.1 Summary of LogTM

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. Each cache block is augmented with two bits (R&W) so that conflicts are detected at a granularity of blocks. It extends a directory protocol in order to perform fast conflict detection of evicted blocks, by using *sticky states*. Transaction nesting is supported by flattening inner nested transactions into the top-level one. LogTM detects 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.

**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 |

## 4.2 Simulation Methodology and Environment

We use a full-system execution-driven simulation based on the Wisconsin GEMS toolset [10], in conjunction with Virtutech Simics [9]. We use an implementation of the LogTM protocol and the detailed timing model for the memory subsystem included in GEMS v2.0, with the Simics in-order processor model. Simics provides functional correctness for the SPARC ISA and boots an unmodified Solaris 10. Two of the main metrics of our characterization are conflict rate and abort rate, defined as[1]:

$$conflict\_rate = \frac{xacts\_aborted + xacts\_conflicted\&committed}{xacts\_aborted\ + xacts\_committed}$$

$$abort\_rate = \frac{xacts\_aborted}{xacts\_aborted + xacts\_committed}$$

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. Instead of augmenting each cache block with R&W bits and using an *overflow bit*, conflict detection is performed through *perfect* signatures –mere lists of addresses read/written by the transaction–, similar to LogTM-SE [17]. In this way, our characterization is isolated from all kinds of false conflicts, either due to signature's false positives or to stale sticky states resulting from overflowed transactional blocks.

## 4.3 Transactional Workloads

Up to now, the majority of studies on transactional memory systems have used in their evaluation parallel applications from the SPLASH-2 suite [16]. The lack of real transactional workloads parallelized under this model makes difficult to find a set of benchmarks that acts as a reference system for the TM research community. So far, the STAMP suite (*Stanford Transactional Applications for Multi-Processing*) [4] is the only available collection of transactional applications that use coarse-grain transactions to execute concurrent tasks on irregular data structures such us graphs or trees. STAMP comprises four benchmarks: *genome*, that reconstructs a gene sequence from segments of a larger gene; *kmeans*, that clusters objects in k partitions according to certain attributes; and *vacation*, that implements a travel agency system; and *delaunay*, that implements the Delaunay algorithm for mesh generation. To produce the results of this characterization, we used the same set of inputs than Cao Minh *et al.* [4], as shown in Table 2. We also use a few benchmarks extracted from the SPLASH-2 suite, in order to compare the behaviour of the LogTM transactional memory system under substantially different workloads.

[1] xacts: transactions

**Table 2. Benchmarks and inputs.**

| Benchmark | Input |
|---|---|
| DELAUNAY | Mesh gen3.2, min. angle 30 |
| GENOME | 16K segments, gene length 256, segment length 16 |
| KMEANS | 20/20 clusters, thres. 0.05, 1000 12-dim points |
| VACATION | 64K entries, 4K tasks, 8 queries, 10 rel, 80 users |
| BARNES | 4096 bodies |
| CHOLESKY | tk14 |
| RAYTRACE | teapot |

## 4.4 Results

In this section, we present in a quantitative manner how certain applications developed under the TM programming model are prone to experience many conflicts and aborts, using a log-based HTM system as our evaluation environment. We start with a brief characterization of the benchmarks and the transactions they comprise, that will help us to better interpret the results of our experiments later on. This is necessary since each workload has its particular features that distinguish it from the rest in how it is affected by conflicts, despite being developed under the same concept of coarse-grain transactions. We find that the frequency of conflicts is greatly dependant on the granularity of the transactions. We show the performance implications of such overhead –stall, backoff and abort– and notice how its effects are more pronounced in workloads where the transactional code has a significant weight in global execution time.

The amount of work carried out by a transaction –quantitatively measured by the size of its read and write sets– directly affects conflict and abort rates, since the odds of a conflict with concurrent transactions increase with the amount of data accessed by the transaction. Figure 1 shows the average transaction read and write set size of each benchmark. A more detailed analysis of each benchmark's transactions is shown in Table 3 (numbers in bold highlight

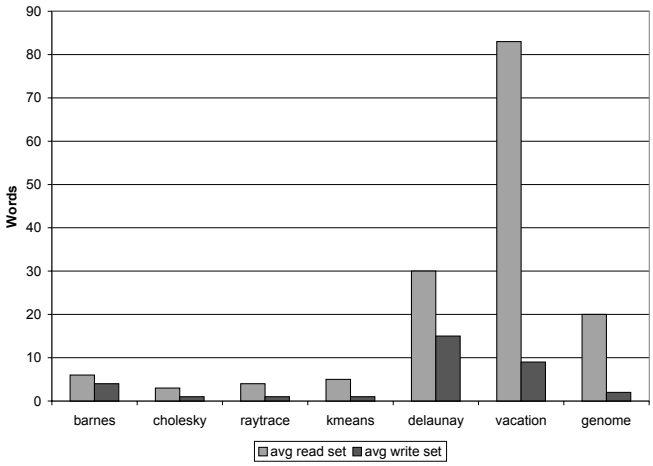


**Figure 1. Average read and write set sizes in SPLASH vs. STAMP benchmarks.**

**Table 3. Characterization of transactions in STAMP benchmarks: Count, Read and Write set sizes**

| Benchmark | XID0 | | | XID1 | | | XID2 | | | XID3 | | | XID4 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | # | R | W | # | R | W | # | R | W | # | R | W | # | R | W |
| kmeans | **1000** | **6** | **1** | 333 | 1 | 1 | 8 | 2 | 1 | - | - | - | - | - | - |
| delaunay | 1898 | 1 | 0 | **3101** | **55** | **28** | 1211 | 8 | 3 | 8 | 1 | 1 | - | - | - |
| vacation | **3232** | **92** | **10** | 433 | 24 | 1 | 431 | 72 | 6 | - | - | - | - | - | - |
| genome | **1024** | **52** | **1** | 241 | 2 | 1 | 3615 | 14 | 3 | 241 | 3 | 11 | 449 | 5 | 2 |

each benchmark's main transaction). In Table 3 we can see that both delaunay and vacation have a main transaction performing the bulk of the computation. In delaunay, XID1 not only has the larger read and write sets but is also the transaction executed most times, and the same can be said about vacation. For both benchmarks, their main transaction has huge R&W sets compared to kmeans and genome. Only the first step (transaction) of the algorithm performed by genome requires a larger read set; its remaining four transactions access a few shared data. As for kmeans, its three transactions have very small data sets –including the main one, XID0– compared to vacation and delaunay. These results anticipate higher conflict and abort rates for delaunay and vacation than for genome and kmeans, although final outcome depends on the actual degree of data communication among threads.

Table 4 presents the first part of the results of this characterization. It shows the total number of conflicts experienced by each benchmark and splits them according to the way they are resolved, either by aborting or stalling the transaction. The column *useful stalls* shows how many stalls successfully solved the conflicts (the transaction was able to reach commit later on). Although eager conflict detection allows LogTM to deal with some conflicts by

**Table 4. Characterization of conflicts according to their resolution.**

| Benchmark | total conflicts | aborted | stalled | useful stalls | % useful stalls | % aborted |
|---|---|---|---|---|---|---|
| barnes | 601 | 386 | 215 | 213 | 99% | 64% |
| cholesky | 123 | 56 | 67 | 67 | 100% | 46% |
| raytrace | 95229 | 89577 | 5652 | 5466 | 97% | 94% |
| kmeans | 218 | 75 | 143 | 137 | 96% | 34% |
| delaunay | 19713 | 15003 | 4710 | 3670 | 78% | 76% |
| vacation | 34355 | 16886 | 17469 | 4104 | 23% | 49% |
| genome | 8459 | 3403 | 5056 | 2904 | 57% | 40% |

**Table 5. Transactions committed, stalled before commit and aborted. Retries per transaction. Conflict and abort rates.**

| Benchmark | committed | stalled bfr commit | aborted | retries per xact | stalled bfr commit | conflict rate | abort rate |
|---|---|---|---|---|---|---|---|
| barnes | 17431 | 542 | 386 | 0,02 | 3,1% | 0,05 | 0,02 |
| cholesky | 6573 | 146 | 56 | 0,01 | 2,2% | 0,03 | 0,01 |
| raytrace | 47766 | 14775 | 89577 | 1,88 | 30,9% | 0,76 | 0,65 |
| kmeans | 1349 | 187 | 75 | 0,06 | 13,9% | 0,18 | 0,05 |
| delaunay | 6284 | 2976 | 15003 | 2,39 | 47,4% | 0,84 | 0,70 |
| vacation | 4096 | 2798 | 16886 | 4,12 | 68,3% | 0,94 | 0,80 |
| genome | 5570 | 1298 | 3403 | 0,61 | 23,3% | 0,52 | 0,38 |

stalling a transaction rather than aborting it, those benchmarks composed by large transactions cannot benefit as much from this conflict resolution scheme. In vacation –the benchmark with the largest main transaction of all– only 23% of the stalls are useful, in the meaning that they enable the transaction to reach commit. This percentage is somehow higher for genome and delaunay –benchmarks with relatively large transactions– but is still far from those obtained for other workloads formed by small transactions –over 96% in all cases–. In regards to conflicts solved by aborting, they represent between 40 and 50% of the cases in most benchmarks, reaching higher percentages for raytrace and delaunay –94 and 76%, respectively–. In raytrace, two similar critical regions (transactions) that read and increment a global variable together account for more than 90% of the total number of aborts. In delaunay, almost 50% of the aborts correspond to XID0 –also a short transaction that pops an element from the front of the worklist– and only 16% to XID1 –its main transaction that performs the cavity refinement–. In vacation and genome, 98 and 76% of the aborts are by XID0 –their main transaction–, respectively. Therefore, we can expect vacation to be the benchmark that wastes more time doing computations that are discarded afterwards, due to the size of its main transaction and the frequent aborts it experiences.

In Table 5 we summarize the main results of our characterization. First, we can observe how barnes and raytrace execute many more critical regions (transactions) than the

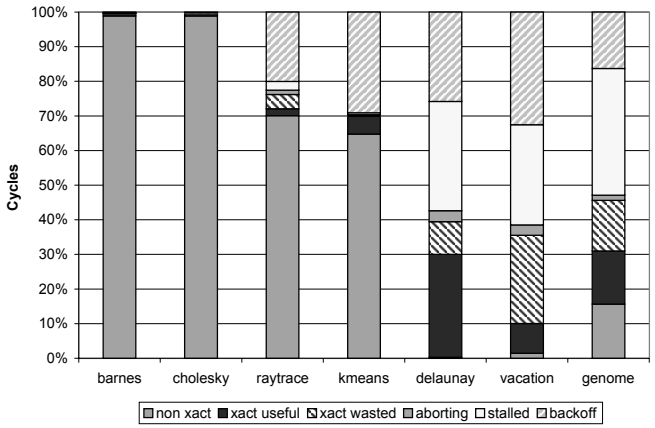


**Figure 2. Execution time breakdown.**

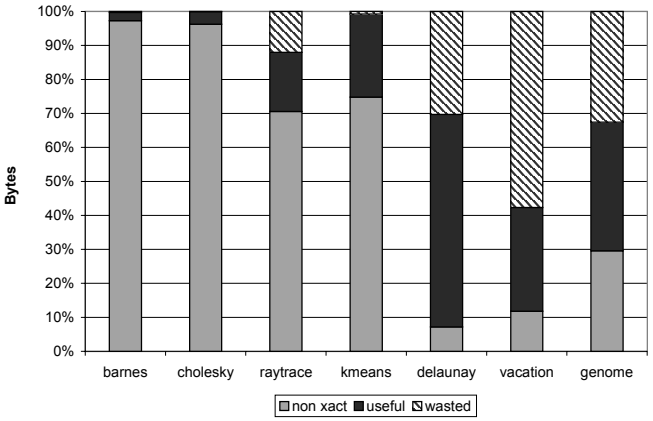


**Figure 3. On-chip network traffic breakdown.**

STAMP benchmarks. As we mentioned earlier in this paper, this is due to the fine-grain critical regions in the SPLASH-2 benchmarks, opposed to the programming style promoted by TM and used to develop STAMP. Second, some benchmarks have an important percentage of committed transactions that are stalled before reaching their end, like vacation (68%), delaunay (47%) or raytrace (31%). This also exhibits the clear relationship between transaction size and the odds of suffering *non-letal* conflicts. Conflict rate summarizes the appearance of both recoverable –stalled– and unrecoverable –aborted– conflicts, indicating the fraction of transactions that conflicted, out of all started transactions. Column 7 in Table 5 shows that conflicts arise very frequently in applications such as vacation, delaunay, raytrace or genome, with moderate to very high conflict rates –ranging from 0,52 in genome to 0,94 vacation–. In other words, these rates reveal that the majority of the transactions in these three benchmarks suffer some kind of conflict during their execution. Lastly, the abort rates obtained for these benchmarks are also remarkably high, particularily in delaunay (0,70) and vacation (0,80), which are precisely the benchmarks with larger transactions. Therefore, this charaterization not only contradicts the assumption that commits will be much more frequent than aborts, but also points out the need for better recovery schemes.

The weight of transactions in execution time determines the influence of aborts on application performance. Despite the programmer's ability to extract finer grain transactions, the nature of the computation ultimately determines how much synchronization is needed and therefore how much time must be spent in transactional code. Figure 2 shows the normalized execution time breakdown for each benchmark, divided in six components: non transactional, transactional useful (committed work), transactional wasted (discarded work), aborting (rolling back transactional state during an abort), stalled (stalling to resolve a transaction conflict) and backoff (stalling after an abort to reduce contention). We can see that both barnes and cholesky spend more than 97% of their cycles in non transactional code, having virtually no transactional overhead. For raytrace and kmeans more than 65% is spent in non transactional code, and almost all the overhead is due to the exponential backoff (20 to 30%). Note that the numerous aborts of raytrace do not lead to a lot of wasted cycles, as the transactions are very small. The remaining three benchmarks exhibit similar behaviour. Since they spend most or all of their time inside large, coarse-grain transactions, the impact of the transaction overhead on application performance is much higher than in the previous benchmarks. The high conflict and abort rates of vacation presented in Table 5 are responsible for a surprisingly low fraction of useful cycles (10%, including non transactional execution). Roughly 25% of its cycles are unstalled transactional computation that gets discarded afterwards, upon abort. In delaunay, however, 30% of its cycles are useful work and only 10% are wasted work, while genome stays in-between, with around 15% of each. Stall cycles due to conflicts account for 30-35% in these three benchmarks, and backoff cycles are in the 15-35% range. The amount of wasted work can be represented as well from the perspective of network traffic. Figure 3 breaks it into three components: non transactional traffic, wasted transactional traffic and useful transactional traffic. As we can see, those benchmarks with high abort rates not only make a poor use of their cores but also introduce a lot of useless traffic into the on-chip interconnection. This may slow down other threads or applications that find a congested network, and of course it has dramatic consecuences on power consumption.

## 5 Conclusions

In this paper, we show that aborted transactions can be quite frequent for some kinds of future TM applications. We reanalyzed the HTM design space from a novel perspective, assuming that aborts could be as common as commits.

We presented the first characterization of conflicts in a log-based hardware transactional system that uses truly transactional benchmarks composed by large, coarse-grain transactions. Our results show that those applications that spend most of their time in a few large transactions experience significant performance degradations due to the high abort rates. Thus, we argue against commits being always the common case in future transactional workloads, and point out that more flexible schemes of data versioning are needed if HTM systems are to success in environments with a huge variability of transaction granularity. High abort rates experienced by some of the evaluated workloads also indicate that HTM research needs to focus on enhanced recovery mechanisms that are capable of obtaining good performance levels even in the presence of frequent aborts. Otherwise, these systems will burden programmers with the same programmability-performance trade-off that the TM model tries to free them from.

## 6 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'. Rubén Titos is supported by a research grant from the Spanish MEC under the FPU national plan (AP2006-04152).

## References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Feb 2005.

[2] C. Blundell, J. Devietti, E. C. Lewis, and M. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proceedings of the 34st Annual International Symposium on Computer Architecture*, Jun 2007.

[3] 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. In *Proceedings of the 34rd Annual International Symposium on Computer Architecture*, Jun 2007.

[4] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, Jun 2007.

[5] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 227–238, June 2006.

[6] 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*, 24(6), Nov-Dec 2004.

[7] 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. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 102–113, Jun 2004.

[8] M. Herlihy and E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–301, May 1993.

[9] 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. *IEEE Computer*, 35(2):50–58, Feb 2002.

[10] M. M. 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*, pages 92–99, September 2005.

[11] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265, Feb 2006.

[12] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logtm. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 359–370, Oct 2006.

[13] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *34th International Symposium on Microarchitecture*, pages 294–305, December 2001.

[14] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *10th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, October 2002.

[15] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505, Jun 2005.

[16] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. *In Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[17] L. Yen, J. Bobba, M. M. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, pages 261–272, Feb 2007.