CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2013; **25**:862–880
Published online 4 June 2012 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.2866

SPECIAL ISSUE PAPER

On the design of energy-efficient hardware transactional memory systems

E. Gaona*,†, R. Titos, J. Fernández and M. E. Acacio

CAPS Research Group, University of Murcia, Spain

SUMMARY

Transactional memory is currently being advocated as a promising alternative to lock-based synchronization because it simplifies multithreaded programming. In this way, future many-core chip multiprocessor architectures may need to provide hardware support for transactional memory. On the other hand, energy consumption constitutes nowadays a first class consideration in multicore processor designs. In this work, we characterize the performance and energy consumption of two well-known hardware transactional memory systems that employ opposite policies for data versioning and conflict management. More specifically, we compare a LogTM-SE *eager-eager* system and a version of the Scalable Transactional Coherence and Consistency *lazy-lazy* system that enable parallel commits. To do so, we extended the Multifacet GEMS simulator to estimate the energy consumed in the on-chip caches according to CACTI and used the interconnection network energy model given by Orion 2. Results show that the energy consumption of the eager-eager system is 38% higher in average than in the lazy-lazy case, whereas performance differences between the two systems are 26% in average. We found that even though lazy-lazy beats eager-eager on average, there are considerable deviations in performance depending on the particular characteristics of each application and the settings of both systems. Finally, from this characterization, we observe that a significant part of the energy consumed in some applications in eager-eager is spent on the back-off delay phase and explore more energy-efficient hardware back-off mechanisms. For lazy-lazy systems, the way in which memory lines are assigned to the L2 cache banks affects the number of parallel commits in some applications, and we study an alternative fine-grained assignment. Copyright © 2012 John Wiley & Sons, Ltd.

Received 8 June 2011; Accepted 4 April 2012

KEY WORDS: hardware transactional memory (HTM); conflict detection; version management; eager-eager approach; lazy-lazy approach; performance; energy efficiency; aborts; commits

1. INTRODUCTION AND MOTIVATION

Over recent years, we have witnessed the replacement of single-core processors by multicore ones, which has made parallel computing resources commonplace. Although it is expected that the number of cores will grow, reaching dozens or even hundreds of them in the next years [1], multithreaded programming remains a challenging task, even for experienced programmers. On the other hand, energy consumption constitutes nowadays a first class consideration in multicore processor designs, and energy-efficient architectures are a must.

Transactional memory (TM) is currently being fostered as a promising parallel programming paradigm, and processors implementing transactional memory support in hardware have already been announced [2]. TM borrows the concept of transaction from the database world and brings it into the shared-memory programming model [3]. Transactions are no more than blocks of code

*Correspondence to: Epifanio Gaona, Departamento de Ingeniería y Tecnología de Computadores, Facultad de Informática, Campus de Espinardo s/n, 30100 Murcia, Spain.

†E-mail: fanios.gr@ditec.um.es

Copyright © 2012 John Wiley & Sons, Ltd.

whose execution must satisfy the serializability and atomicity properties. Programmers simply declare the transaction boundaries leaving the burden of how to guarantee such properties to the underlying TM system thereafter. Next, the TM system executes the transactions in parallel, as if they were not to perform conflicting memory accesses that could violate the serializability property. If so, this optimistic behavior pays off over the pessimistic lock approach. Otherwise, one of the offending transactions must be aborted. In this case, the TM must guarantee that there are no side effects left behind by the aborted transaction to satisfy the atomicity property. In this way, the benefits derived from transactional memory are twofold. Transactions are speculatively executed that hides to programmers the main pathologies associated with locking techniques, such as priority inversion, convoying, and deadlocks. As a consequence, programmers are armed with an intuitive synchronization abstraction that can greatly help to simplify the development of multithreaded programs.

A TM system can be implemented in either software or hardware or as a combination of both [4]. Hardware transactional memory (HTM) systems usually work at the word or cache line level. Conceptually, each transaction is associated two initially empty read and write sets that are populated every time a transactional load or store is issued. To comply with the serializability property, both the old values and the transactional ones must coexist until the transaction is allowed to commit. A transaction can commit only after the HTM system can assure that there are no other running transactions whose write sets collide with its read or write sets. The commit process makes the read and write sets of the winner transaction visible to the whole system. In this general scheme, there are two opposite ways to tackle data version management (VM) and conflict detection (CD). Eagerly versioned systems perform updates in place, that is, transactional stores overwrite old values residing in cache memory after storing them in an *undo log*. In lazy version management, transactional stores are performed aside; that is, produced values are kept on a private *write buffer* until the transaction is granted permission to commit. In turn, eager conflict detection checks dependency violations on the fly during the transaction lifetime for each transactional load and store, as opposed to lazy conflict detection that leaves this task until the last phase of the transaction execution. In this way, HTM systems could come in four flavors considering these different approaches to VM and CD. Table I classifies some well-known HTM systems appeared in the literature in terms of the policies used to deal with VM and CD.

This classification raises the question of which combination constitutes the best trade-off between cost and performance. The answer has no clear winner because all of them pose some drawbacks. On transaction success, eager VM is faster than lazy VM because transactional values are already in place. On the contrary, if a transaction aborts, lazy VM is a better choice because the original values remain unmodified in cache memory. On the other hand, whereas eager CD incurs a bigger overhead because of the persistent checking process, lazy CD usually wastes a larger amount of work every time a transaction aborts. The comparison gets even more complicated when energy consumption comes into play. Note that the diverse VM and CD management policies have distinct hardware requirements and may lead to different behaviors depending on the transaction interaction pattern. At the end, this translates into quite different energy consumption figures depending on the particular implementation of the HTM system and the characteristics of the workload.

Table I. TM classification.

		VM	
		Eager	Lazy
CD	Eager	LogTM-SE [5] TokenTM [7]	EazyHTM [6] VTM [8], LTM [9]
	Lazy	×	Scalable TCC [10], Bulk [11]

VM, version management; CD, conflict detection; TM, transactional memory; TCC, Transactional Coherence and Consistency; VTM, Virtual Transactional Memory; LTM, LargeTransactional Memory.

Copyright © 2012 John Wiley & Sons, Ltd.

To the best of our knowledge, even though *lazy-lazy* systems are considered as the the best choice in the general case [12], no previous work can be found in the literature that performs a direct comparison of the most popular HTM implementations, namely lazy-lazy HTM systems and *eager-eager* HTM systems for general purpose systems. Ferri *et al.* [13] performed an analysis of both HTM systems but only for embedded architectures. Because of the specific conditions of the embedded domain and their inherent harder hardware constraints, their proposals strongly focus on the energy efficiency issue at the expense of obtaining worse performance. In this work, which is an extension of our previous work appeared in [14], we conduct a fair comparison of two well-known HTM systems. In particular, we compare LogTM-SE [5], as an example of an eager-eager system, with Scalable Transactional Coherence and Consistency (TCC) [15], a lazy-lazy HTM system. To do so, we rely on well-known simulators, tools, and transactional benchmarks widely accepted by the scientific community. In particular, we extended the GEMS simulator to estimate the energy consumed in the on-chip caches according to CACTI and used the interconnection network energy model given by Orion 2.0. Results show that the energy consumption of the eager-eager system is 38% higher on average than in the lazy-lazy case, whereas performance differences between the two systems are 26% on average. We found that even though lazy-lazy beats eager-eager on average, there are considerable deviations in performance depending on the particular characteristics of each application. Our main contribution in this work is a comprehensive analysis of both systems in terms of performance, energy consumption, and network traffic.

On the basis of this analysis, we identify two aspects that could negatively affect energy consumption in future eager-eager and lazy-lazy systems. For eager-eager systems, we found that relying on a software back-off mechanism can drastically increase energy consumption in some cases, and we explore the effects of two hardware back-off mechanisms. For lazy-lazy systems, the way in which memory lines are assigned to the L2 cache banks affects the number of parallel commits in some applications, and we study an alternative fine-grained assignment.

The rest of the article is organized as follows. We start with a profound description of the two HTM systems targeted in this study in Section 2. Later on, in Section 3, we detail the implementation of both systems, the configuration of the simulation environment, and the workload used to generate the results. Performance, energy consumption, and network traffic figures are analyzed in Section 4 for the base systems. In this section, we also analyze the impact that both the election of the back-off mechanism and the distribution of memory lines between L2 cache banks have on the performance and energy consumption of eager-eager and lazy-lazy systems, respectively. Finally, conclusions are given in Section 5.

2. CHARACTERIZED HARDWARE TRANSACTIONAL MEMORY SYSTEMS

This section summarizes the main characteristics of the two HTM systems evaluated in this work: LogTM-SE [5] and Scalable TCC [10].

2.1. LogTM

LogTM [16] is a widely known eager-eager system that makes use of an eager version management, storing new values directly in the memory location of the variable (or *in place*), while preserving old values *on the sid*e. Before the completion of a write access, the hardware automatically backs up the old value of the cache block in a per-thread undo log allocated in cacheable virtual memory. This eager versioning policy makes commits fast, whereas aborts are slower because the system must trap to a software handler to unroll the log to restore pretransactional state. Each undo log entry contains the virtual address of the stored block and the block's old value. LogTM performs eager (or pessimistic) conflict detection leveraging the coherence protocol to detect conflicts by observing forwarded requests and invalidations for blocks that belong to a transaction's read and write sets. LogTM augments each L1 cache block with a read (R) and a write (W) bit used to track the blocks that belong to the transaction read and write sets.

When a transaction detects a conflicting remote request, it responds with a negative acknowledgement (NACK), indicating that the requester transaction must stall its execution until the offended

Copyright © 2012 John Wiley & Sons, Ltd.

transaction releases isolation over the requested data upon commit/abort. The default behavior of the requestor after receiving a NACK is to resend the same request until it obtains an (affirmative) acknowledgment (ACK), regardless of needed retries. This scheme can result in cycles, so LogTM uses a conservative deadlock avoidance mechanism on the basis of timestamps, always giving priority to the eldest transaction. More precisely, every time a transaction sends a NACK message to an elder requestor (according to their timestamps) a *possible cycle bit* is activated. The transaction must subsequently abort only if a NACK from an elder transaction is received. Because a transaction keeps on repeating the request when a NACK message is received, the mechanism is guaranteed to be deadlock free. To reduce the number of aborts that may arise during long-running conflicts, LogTM introduces a back-off delay when a transaction is aborted. After this delay, the transaction is retried from the beginning. The back-off mechanism is implemented in software through a library used by all applications. The back-off routine is called by a software handler as part of a work that must be performed during a transactional abort. In detail, the routine computes a cumulative sum over an array in a loop as many times as defined by a specific function. The upper limit of the loop is increased exponentially as the number of retries suffered by the transaction grows.

LogTM-SE [5] is a refinement of LogTM that replaces RW bits with hash signatures (bloom filters) that conservatively summarize a transaction's read and write sets, decoupling transactional bookkeeping from the caches, and enabling virtualization of transactions (as signatures are accessible by software and the operating system). LogTM-SE is the version implemented in GEMS simulator [17] and the eager-eager system we characterize in this work.

2.2. *Scalable TCC*

Scalable TCC [15] (STCC) is a popular, scalable, nonblocking implementation of TM that is tuned for continuous use of transactions within parallel programs. STCC provides nonblocking synchronization and an easy-to-understand consistency model. STCC relies on a directory-based implementation of the TCC[18] model, which defines coherence and consistency in a shared memory system at transaction boundaries. Transactional stores are performed on the side using a write buffer that keeps the speculative new values. The lazy approach to data versioning of STCC requires that transactional data is write-backed into coherent memory only when a transaction commits. STCC uses a two-phase parallel commit algorithm that is supported by a central arbiter. In the validation phase, a transaction checks if it has conflicts with others. The central arbiter gives numbers in upward order (TID) to transactions to prioritize them in case of conflicts. Transactions with higher TID must abort in case of conflicts with a transaction with smaller TID (prioritizing eldest transaction). Once in the second commit phase, a transaction has acquired the privileges to do commit and cannot be violated by other transactions. In this phase, a transaction makes visible its changes to the rest of the system. *Sequential Commit* (SEQ) and its optimized flavor SEQ with *Parallel Reader Optimization* (SEQ-PRO) [10] enhance the STCC system to improve the performance of the commit phase. SEQ allows parallel commits using a distributed mechanism that entails less message overhead than the original STCC commit algorithm. In SEQ (and SEQ-PRO), the physically distributed banks of the L2 cache act as a distributed arbiter. Each bank has a waiting queue. When a transaction reaches the (commit) phase, it sends a *book directory* message—COMMIT_XACT—(in case of a directory-based protocol) to each bank in its read and write sets in increasing order. One particular L2 bank will belong to a transaction's write set if at least one virtual address of this set is mapped to that bank. A transaction cannot send the subsequent *book message* until the previously requested directory bank has acknowledge the booking request, XACT_ACK. This confirmation is sent by the directory when the requester transaction reaches the head of the waiting queue in that bank, acquiring the needed permissions. When a transaction obtains all permissions from all L2 banks, it proceeds with the *commit* itself, making visible its writes, aborting other conflicting transactions (if any), and dumping the values of its write buffer into memory. At the end of the commit, the first place of the queue is released in each bank (*release* subprocess), and new transactions (which have not been aborted by previous commits) can proceed with the directory booking process if they reach the new head of the queue. From here on, we will

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

call the phase of booking directories as *precommit*. Our implementation of SEQ involves a multicast release petition—XACT_RELEASE—and its corresponding confirmation, RELEASE_ACK. When all confirmations have been received, the transaction completes successfully. If a transaction is enforced to abort during its precommit phase, it sends a—EXIT_XACT—message to release each L2 cache bank already booked. The corresponding L2 cache banks will update the waiting queue and send an acknowledgement—EXIT_ACK—message back to the aborting transaction.

As in LogTM, a back-off mechanism is also employed in STCC. However, because aborts in the case of STCC do not involve any software handler but are managed in hardware, the back-off mechanism is also implemented in hardware. More specifically, in the case of abort, all changes stored in the write buffer are discarded, and the number of cycles for the back-off delay is computed taking into account the number of retries suffered by the aborting transaction. Subsequently, STCC will disable the processor where the aborting transaction's thread is running during the calculated number of cycles.

The address mapping policy to the L2 cache banks is critical in STCC. To allow for parallel commits, STCC needs that different transactions are able to book different L2 cache banks. In case of a specific L2 cache bank belongs to two transactions' read or write sets, there will not be any chances for parallel commits between these two transactions because they will try to reserve simultaneously the same L2 cache bank. Obviously, this situation is expected for transactions accessing the same memory addresses. In this case, a conflict between them had been encountered, and eventually, one of the transactions must be aborted. On the contrary, when two different memory lines belonging to distinct read/write sets have been mapped to the same L2 cache bank, their corresponding transactions would compete for it, although no real conflicts occur. In this case, STCC would be causing the well-known *serialized commits* pathology [19] as a side effect of what we call *directory aliasing*.In our implementation of STCC, the address mapping is carried out by assigning chunks of 128 KB of contiguous addresses per L2 cache bank.

Finally, a more advanced version of the algorithm, known as SEQ-PRO [10], differentiates between transactions that want to book a directory for reading from those that intend to write, allowing the promotion to the final stage of commit of all readers as long as there are no writers waiting. We are prioritizing writers over readers. In case some readers are stored in the readers queue (and they have passed to book others banks automatically), a new writer must wait its turn, but no other new reader will continue with the precommit subphase until the writer has finished its commit (or abort) to avoid writer starvation.

3. EVALUATION ENVIRONMENT

In this section, we describe the evaluation environment used in this article. We start by giving the details about how the eager-eager and lazy-lazy HTM systems considered in this work have been implemented in the simulator. Additionally, we delineate the consumption models used to characterize energy consumption. In particular, we focus on the energy consumed in the on-chip memory hierarchy. Finally, we conclude with a description of the benchmarks used to conduct the simulations.

3.1. System settings

We use a full-system execution-driven simulation based on the Wisconsin GEMS (University of Wisconsin-Madison, Wisconsin, USA) toolset [17], in conjunction with Simics (Wind River, Berkeley, CA, USA) [20]. We rely on the detailed timing model for the memory subsystem provided by GEMS's Ruby module, with the Simics in-order processor model. Simics provides functional correctness for the SPARC ISA and boots an unmodied Solaris 10. We perform our experiments on a tiled chip multiprocessor system, as described in Table II. We assume a 16-core configuration with private L1 I&D caches and a shared, multibanked L2 cache consisting of 16 banks of 512 KB each for the execution of the applications' threads. We have left another core aside to run the operating system in an isolated way from the application threads to avoid intrusions from the same one in benchmarks' execution and obtaining uncorrupted statistics. The operating system still takes the

Copyright © 2012 John Wiley & Sons, Ltd.

Table II. System parameters.

MESI directory-based CMP	
Cores	16, single issue, in order, nonmemory IPC=1
Memory and directory settings	
L1 cache I&D	Private, 32 KB, split 2 way, 1-cycle latency, 64 B lines
L2 cache	Shared, 8 MB, unified 4 way, 12-cycle latency
L2 directory	Bit vector, 6-cycle latency
Memory	4 GB, 300-cycle latency
Network settings	
Topology	2D mesh
Link latency	1 cycle
Link bandwidth	16 bytes/cycle

CMP, chip multiprocessor; 2D, two-dimensional.

Table III. Parameters of Orion 2.0.

Parameter	**Value**
in_port	6
tech_point	45
Vdd	1.0
transistor type	NVT
flit_width	128 (bits)

control of the benchmarks' execution when needed (i.e., during an exception). The L1 caches maintain inclusion with the L2 cache. The private L1 data caches are kept coherent through an on-chip directory (at L2 cache banks), which maintains bit vectors of sharers (which are included in the tags' part of the L2 cache banks) and implements the MESI protocol. The tiles are connected through a two-dimensional mesh network. Each tile contains a router where the private L1, the slice of L2, the memory controller, and the links to the neighboring tiles are connected to. In this 4 x 4 two-dimensional mesh network, each router has between five and seven ports, with six ports per router on average.

To compute energy consumption in the on-chip memory hierarchy, we consider both the caches and the interconnection network. The amount of energy consumed by the interconnection network has been measured on the basis of the Orion 2.0 [21]. In particular, we have extended the network simulator provided by GEMS with the consumption model included in Orion 2.0. Table III shows the values of some parameters assumed for the interconnection network. For those not listed in the table, we use the default values given in Orion 2.0. On the other hand, energy spent in memory structures (L1, L2, Write_Buffer) has been measured on the basis of the consumption model of CACTI 5.3 rev 174 [22]. For the particular case of the L2 cache consumption measurement, we distinguish the accesses that return cache blocks from those that only involve the tags' part of the L2 cache (i.e., those that would be performed by the directory controller to retrieve just the sharing information for a particular cache memory block). Obviously, the latter entails less energy.

The Ruby module contains an implementation of LogTM-SE, an eager-eager system that uses signatures for transactional bookkeeping. Additionally, Ruby provides a naive version of a lazy-lazy system that employs a commit token to serialize transaction commits and whose arbitration takes places through an idealized zero-latency broadcast bus. This sequential commit process with the presence of a centralized referee is similar to that proposed in [15], although it does not use the interconnection network to coordinate the entire process. Thus, the lazy-lazy implementation provided in GEMS is not only nonscalable, because the central referee would become a bottleneck,

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

but also unrealistic because a zero-latency bus is being assumed. For this reason, we have modified Ruby to implement the more efficient and scalable commit algorithms described in Section 2 that uses the two-dimensional mesh network. More specifically, we have evaluated in this work the SEQ-PRO algorithm proposed by Pugsley *et al*. in [10]. SEQ-PRO allows for parallel commits (as the SEQ algorithm) and implements the parallel reader optimization. To implement SEQ-PRO, we had to add three new request messages involved in the commit and abort processes: XACT_COMMIT, XACT_RELEASE, and XACT_EXIT and their confirmations: COMMIT_ACK, RELEASE_ACK, and EXIT_ACK, respectively. In this way, the lazy-lazy system evaluated in this work resembles to that presented in [10].

The undo log of the eager-eager system is a data structure mapped in virtual memory, and thus, its size is not limited by any hardware structure. On the contrary, the write buffer required for the lazy-lazy system has fixed size, which has been limited to 128 entries. Overflows of these write buffers will entail accessing main memory for storing the data. The waiting buffers (queues) of the directories in the lazy-lazy system contain 16 positions (as many as cores in the architecture), which means that there will not be any NACK because of lack of space in the queues during the process of precommit. Finally, the read and write sets of transactions in the lazy-lazy system are handled via memory addresses. For the eager-eager system, we assume perfect signatures for detecting conflicts.

3.2. Benchmarks settings

For the evaluation, we use seven transactional benchmarks extracted from the Stanford Transactional Applications for Multi-Processing (STAMP) benchmark suite 0.9.10 [23]. These applications allow to stress a TM system in several ways. To show a wide range of cases, we evaluate STAMP applications using the most significant input size in each case (in general, what is called medium size). For the sake of focusing on transactional workload, ssca2 has been modified to skip a lot of barrier work at the beginning of the benchmark that provokes a different behavior and hence results change from [14]. Table IV describes benchmarks and values for the input parameters used in this work.

4. EVALUATION

In this section, we present the results obtained for the eager-eager LogTM-SE system and the lazy-lazy Scalable TCC system with the SEQ-PRO as commit algorithm (STCC-SP from now on).

We start with a fair comparison between these two HTM systems in terms of execution time. After that, we study the energy consumption of each system while transactional workloads are running. Next, a comparison of the network traffic generated by both HTM systems is evaluated. Note that some minor deviations of the results from those presented in [14] are unavoidable because the new configuration used in this study prevents detrimental interferences from operating system (see Section 3.1). Finally, we present some considerations for the two systems in an isolated way, specifically, how different back-off implementations and address mapping policies affect the LogTM-SE and the STCC-SP systems, respectively.

Table IV. Workloads and inputs.

Benchmark	**Input**
Bayes	-v32 -r4096 -n2 -p20 -i2 -e2
Intruder	-a10 -l16 -n4096 -s1
Kmeans	-m40 -n40 -t0.05
	-i random-n16384-d24-c16
Labyrinth	-i random-x32-y32-z3-n96
Ssca2	-s13 -i1.0 -u1.0 -l3 -p3
Vacation	-n4 -q60 -u90 -r1048576 -t4096
Yada	-a10 -i ttimeu10000.2

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

4.1. Performance

For the seven transactional benchmarks pointed out in Section 3, Figure 1 shows the execution times that are obtained for both LogTM-SE and STCC-SP. In all cases, execution times have been normalized with respect to the STCC-SP system. Moreover, to have a clear understanding of the results, Figure 2 divides the execution times into the following categories: abort (time spent during aborts), barrier (time spent in barriers), commit (time needed to propagate the write sets), non_xact (time spent in non-transactional execution), precommiting (time taken by the process of booking directories in STCC-SP), stall (time waiting until another transaction ends), xact_useful (useful transactional time), and xact_wasted (transactional time wasted because of aborts). The *back-off* fraction represents the time spent before restarting transactions. The use of back-offs aims to avoid contention situations that arise when several transactions are being aborted repeatedly because they conflict with each other over and over again after being restarted. Its upper bound in cycles raises according to the number of retries of the current aborting transaction. We have observed that without this back-off mechanism, the wasted time (xact_wasted) drastically increases in some cases as the number of aborts grows.

As it can be derived from Figure 1, there is no clear winner when LogTM-SE and STCC-SP are compared in terms of performance. In particular, LogTM-SE outperforms STCC-SP for *ssca2* and *vacation*. In turn, STCC-SP beats LogTM-SE for *bayes*, *intruder*, *labyrinth*, and *yada*. For *kmeans*, there is no noticeable difference between the performance of LogTM-SE and STCC-SP because most of the execution time is performing nontransactional work. However, the extents of the differences are quite small when LogTM-SE is the winner (except for ssca2) and very significant when STCC-SP beats LogTM-SE (even reaching a difference of 190% more for intruder). In this way, the

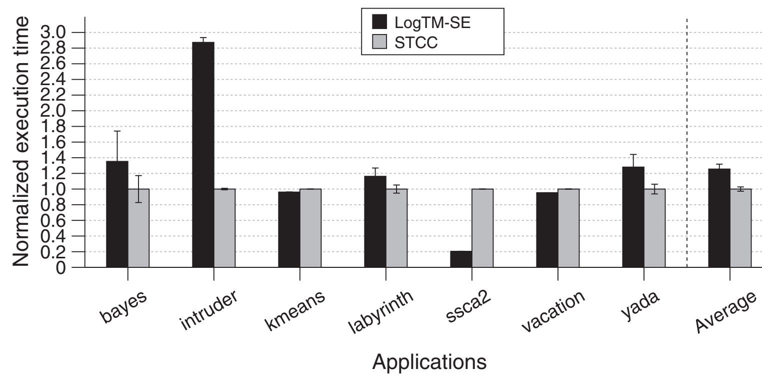

Figure 1. Normalized execution times.

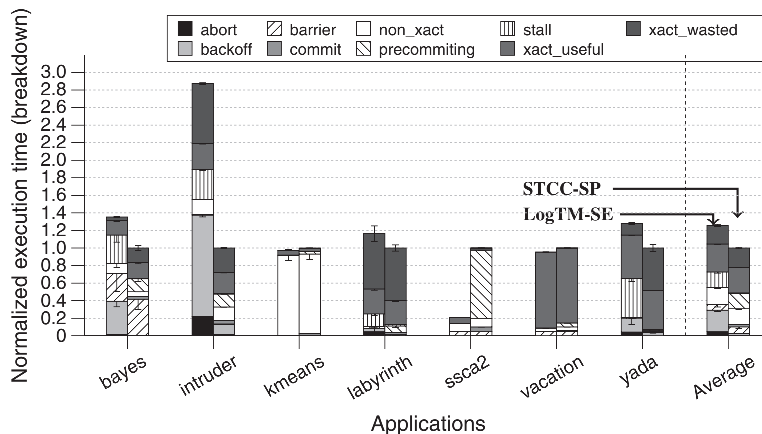

Figure 2. Breakdown of the execution times.

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

lazy-lazy STCC-SP system improves performance (about 26% on average) when compared with the eager-eager LogTM-SE system. In the succeeding paragraphs, we try to explain the differences observed for each benchmark taking into account the breakdown of the execution times presented in Figure 2 and the characteristics of each application along with its data access patterns.

The algorithm implemented in bayes is not deterministic. In particular, its behavior depends on how the branches of the bayes network are carried out, which can change between executions. As a consequence, there is great variance between executions. Its large contention, transactional time, and write sets [23] also lead to a nonnegligible number of conflicts, which is detrimental to LogTM-SE. STCC-SP does not have to deal with the contention until the commit phase, and there will be always one committer (transaction that performs commit) at least.

High contention and short transactions are the main characteristics of intruder. As before, many conflicts take place with LogTM-SE has to deal with lots of conflicts what makes difficult forward progress of its transactions. Besides, this high degree of contention provokes many aborts. This is what causes the bad behavior of LogTM-SE. The back-off time needed grows hugely because of the exponential implementation of the back-off upper bound used in STAMP for LogTM-SE. As a consequence, LogTM-SE degrades the performance by a factor of 3 according to STCC-SP. On the other hand, STCC-SP leverages on its optimistic concurrency control to perform a more fluent behavior, leading to both fast transactional executions and fast abort processes.

For ssca2, LogTM-SE reduces execution time of STCC-SP to a fifth part. The reason can be appreciated in Figure 2. Almost 80% of the time of STCC-SP is spent in the precommit phase (precommiting in figures). As already commented, in this phase, transactions are waiting for each other in the queues of the directories. In a normal execution, if two transactions do not present conflicts between their read and write sets, they will be able to make parallel *fast commits* if their consumed data are mapped in different directories banks. Otherwise, an *induced conflict* for acquiring the directory is produced. Furthermore, there is no significant wasted time due to the absence of real conflicts. We call this behavior as directory aliasing that entails the well-known serialized commits pathology [19] of the lazy-lazy systems. With other physical address mapping, precommiting time could be smaller as it is shown in Section 4.5.

Such as bayes, yada has significant transactional time and write sets and medium contention [23], what entails a significant number of conflicts. The behavior of LogTM-SE with yada is characterized by the importance of the *stall* time together with a mix of back-off and wasted time. Most of the transactions spend 30% of their time in an active waiting (stall) trying to solve conflicts. Although aborts are frequent, the back-off time is much smaller than with bayes because the number of retries per transaction is smaller too. As opposed to LogTM-SE, STCC-SP time is characterized by the fraction of the time wasted by aborting transactions. There is barely precommit time, what means that conflicts are not *induced* by the directory aliasing phenomenon found in ssca2. Conflicts arise in this benchmark because most transactions want to have access to the same addresses. This behavior leads to an important number of aborts, which in turn results into a significant fraction of wasted time in lazy-lazy systems (47% approximately).

Time patterns in labyrinth are quite similar to those found in yada. The differences are in wasted time allocation with LogTM-SE. The abort takes place before, resulting in shorter stalls but increasing the wasted time because of aborting transactions. Labyrinth's characteristics are the same ones as bayes.

The rest of the benchmarks show similar results with both systems. On the one hand, taking into account the average breakdown execution time, the bottleneck found for STCC-SP is its precommit phase, especially when directory aliasing takes place. On the other hand, LogTM-SE and its pessimistic concurrency control involve a worse general behavior with larger stall and back-off times than in STCC-SP. Allowing LogTM-SE and STCC-SP to use a lineal back-off function and a more refined precommit stage, respectively, could improve their performance.

4.2. Energy

Figure 3 shows the dynamic energy consumption of LogTM-SE and STCC-SP. As before, results have been normalized with respect to STCC-SP. Additionally, in Figure 4, we split the energy

Copyright © 2012 John Wiley & Sons, Ltd.

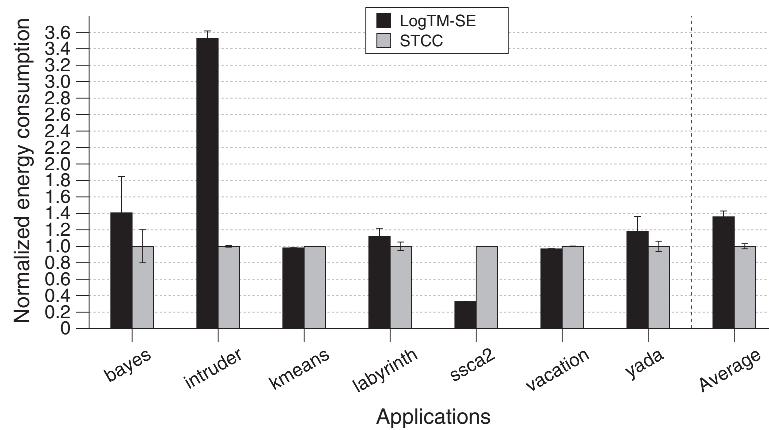

Figure 3. Normalized energy consumption.

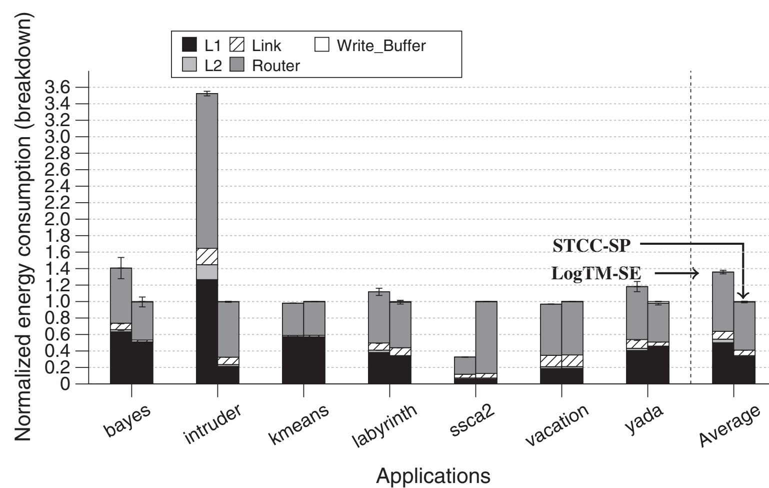

Figure 4. Breakdown of energy consumption. Hardware structures.

consumed in each case for LogTM-SE and STCC-SP into the following categories: energy spent accessing the L1 and L2 caches (*L1* and *L2*, respectively), the write buffer in STCC-SP (*Write_Buffer*), and the network routers and links (*Router* and *Link*, respectively). Again, for most applications, STCC-SP beats LogTM-SE when energy consumption is considered. Only for ssca2, LogTM-SE shows better results. For kmeans and vacation, there are no noticeable differences between both systems. Note that although the average difference in performance among the two systems was 26% in favor of STCC-SP, the latter outperforms about 38% LogTM-SE when energy is considered.

The differences in terms of energy consumption found in bayes, kmeans, labyrinth, and vacation for LogTM-SE and STCC-SP are almost identical to the ones previously reported in terms of execution time. For kmeans and vacation, the compared systems neither show any noticeable difference in execution time nor in energy consumption. For bayes and labyrinth, the improvement of 38 and 18%, respectively, found for STCC-SP in execution time directly translates into a reduction of almost the same extent in energy consumption.

For intruder, we can see that LogTM-SE consumes much more energy than STCC-SP. In this case, the differences are not justified by just taking into account the execution times. Labyrinth algorithm tries to write a road in a 32×32×3 matrix. Each thread first locks the entire matrix to read data; next, it operates with the data locally until a road is found; after this, it tries to write the road, which potentially brings modifications into the cells that form the road that other transaction is processing; finally, if any of the cells have been previously modified by other transaction, the transaction must

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

abort and restart from the beginning. With 16 cores, the probability of collision between roads is high, entailing a significant number of conflicts being detected and therefore aborts. An eager-eager system would have more difficulties in these situations to commit transactions because it is possible that one road that was colliding with another has also collisions with a third one and so on, leading to long chains of dependencies between transactions. Transactions with STCC-SP will commit more easily because when they acquire the commit permissions (reserve all directories in our case) no other transactions can abort them. Additionally, the fact that there are 16 elements of one row per cache block in labyrinth creates a high degree of false sharing, which leads to a noticeable number of *false* conflicts between transactions. All these conflicts (*true* and false conflicts) provoke in LogTM-SE a big number of messages on the interconnection network (to check conflicts) and cache accesses, which drastically increases energy consumption. Something similar happens with intruder: significant contention and a large fraction of aborts lead to much more energy consumption and execution time in eager-eager systems than in lazy-lazy systems.

Although STCC-SP is more energy-efficient than LogTM-SE in yada, the difference is about 20%, whereas the performance gap is close to 30%. This is because STCC-SP experiences a great amount of aborts, increasing significantly the energy consumed in the L1 caches as well as in the network. On the contrary, LogTM-SE spends much of its time in the stall phase, what is translated into request messages and their corresponding NACK responses, which increases the amount of energy consumed in the interconnection network.

Finally, ssca2 is the only application that exhibits noticeable differences in terms of energy consumption in favor of LogTM-SE (energy reductions about 65% are obtained). In this case, the difference is concentrated on the energy consumed in the interconnection network (routers in particular). As already discussed, STCC-SP spends almost 80% of its time in ssca2 in the precommit phase. During this phase, messages for booking directories are being exchanged between processors and L2 cache banks. There is another consideration we have to take into account. Orion 2.0 uses clock frequency to calculate energy spent by its component (routers and links) and because of the intrinsic relationship between frequency (cycles/second) and the execution time (cycles), this last parameter can affect the network consumption.

The breakdown of energy consumption presented by Figure 4 shows that the interconnection network takes the most important part of the consumed dynamic energy, with 65% in STCC-SP and 59% in LogTM-SE. It is worth noting that the energy consumed in the L2 cache, the links of the interconnection network, and the write buffer (only for STCC-SP) is almost negligible (8% of total energy). Consumption in L1 caches, however, can be between 20 and 50% of the total energetic expense (36% on average). The fraction of the energy consumed in the network is more evident for applications with high contention such as intruder. Some issues that motivate the increased energy consumption found in LogTM-SE are the following:

- Cache coherence protocol: assume the case of a request for a data block in an exclusive mode, that is, in SS state in the L2 (0 o more sharers). In this case, the directory must first send invalidations to the sharers (if any) and then provide the data block to the requester. Acknowledgements for the invalidations are collected by the requester. If any of the the shares answers with a NACK message, the received data must be discarded by the requester. In these cases, the L1 cache, the L2 cache bank, and the interconnection network have been used, and because the data has to be discarded, all this energy is wasted. Something similar happens when the data is in Mt state (data in L2 and so it is sent to the requester, but this operation must be checked with the current owner, which can invalidate the action).
- Use of retries: the eager-eager system retries continuous memory requests (see stall state in Figure 2) in case of conflicts until the corresponding transaction aborts or achieves its goal. This supposes a considerable energetic expense.

With regard to the energy consumed in the interconnection network, we have found that the most important fraction is due to the routers (60% in average), whereas the links only consume an 8%. Obviously, the amount and the distribution of energy consumed in the interconnection network will depend on its particular characteristics. In this work, we are assuming a relatively small flit

Copyright © 2012 John Wiley & Sons, Ltd.

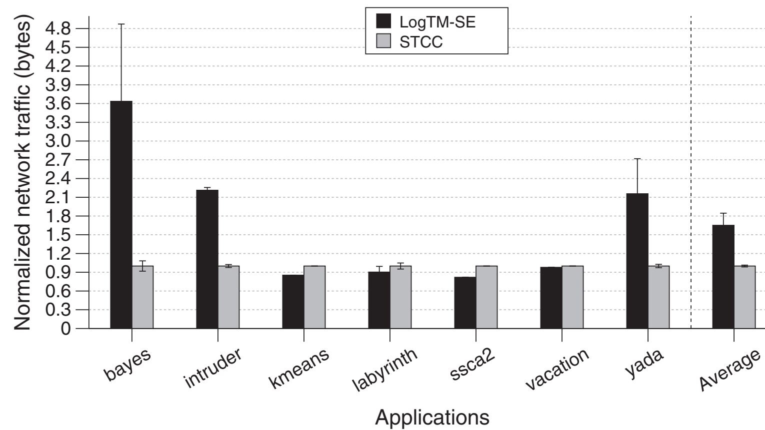

Figure 5. Normalized network traffic.

size (16 bytes) and six-input port routers, which makes the routers the most important source of energy consumption.

4.3. Network traffic

Figure 5 shows the levels of traffic in the interconnection network (measured as flits per cycle) for both LogTM-SE and STCC-SP. As before, results have been normalized with respect to the last one. In general, LogTM-SE entails higher traffic levels than STCC-SP (approximately 70%, on average). The reason can be found in the high number of retries needed in LogTM-SE to achieve data in case of conflicts. This fact is highlighted because sometimes the received data are not valid and must be discarded (see Section 4.2). The difference in network traffic is considerable in the case of bayes, intruder, and yada. These benchmarks are characterized by exhibiting high contention and/or by the large size of their transactions [23]. During the stall phase of LogTM-SE's execution, intensive usage of interconnection network is made because a transaction retries continuously the access to the corresponding memory address until the owner stops sending the NACK response or the transaction aborts. In STCC-SE, the precommit phase does not make such an intensive use of the interconnection network because the number of messages required to book directories is limited [10].

4.4. Influence of the back-off mechanism in eager-eager systems

In this section, we analyze the influence that the election of the back-off mechanism used in eager-eager systems has on performance and energy consumption. As already discussed in Section 2, the back-off mechanism originally implemented in LogTM-SE, which is called exponential software back-off (ESB), consists of a software routine invoked by the software handler that manages the aborts before the aborting transaction can be retried. Figure 6 presents the pseudocode of the routine. As it can be observed, it simply computes a cumulative sum over an array in a loop as many times as determined by a specific function. Particularly, the upper limit of the loop grows with the number of retries suffered by the aborting transaction as shown in Equation (1) (N stands for

```
i = 0;
max_i = random(0 to upperbound);
while (i < max_i) {
    index = i mod #elements in A;
    B = B + A[index];
    i = i + 1;
}
```

Figure 6. Software back-off routine implemented in LogTM-SE.

Copyright © 2012 John Wiley & Sons, Ltd.

the number of retries). In this way, the larger the number of aborts experienced by a transaction, the longer the amount of time before the transactions can be re-executed. The upper limit grows exponentially until the number of retries reaches 16. After this, the upper limit barely increases. Additionally, to ensure that transactions aborting at the same time find a different duration for the back-off phase, the number of iterations over the loop is a random between zero and the calculated upper limit.

$$\text{upper bound} = \begin{cases} 2^N & \text{if } N \leq 16; \\ 2^{16} + (N - 16) & \text{if } N > 16. \end{cases} \tag{1}$$

Obviously, the use of the back-off mechanism is aimed to reduce the degradation arising in high-contention situations (several transactions fighting for the same data at the same time), and therefore, its election conditions the performance and energy consumption of transactional applications experiencing this kind of situations. The back-off mechanism implemented in LogTM-SE is an example of an energy inefficient design decision. During the back-off phase, the processor is active, executing a loop that does nothing. This translates into energy wasted in the processor (not accounted in our study) and the memory hierarchy (extra accesses to the L1 cache).

To analyze the importance that the back-off mechanism has in some applications, we study the case of implementing it in hardware. To do so, after the abort process has been completed, the number of cycles that the processor must wait until it can re-execute the transaction is calculated, and the thread is disabled during this period of time (because we are simulating one thread per core; this would be equivalent to suspending the processor during the computed amount of cycles). In particular, we have implemented two flavors of hardware back-off. The first, which is called exponential hardware back-off (EHB), is inspired by the software mechanism originally included in LogTM-SE and tries to show how implementing the back-off in software is a bad design decision from the energy point of view. The second, referred to as linear hardware back-off (LHB), uses initially more aggressive increases in waiting time to cut off congestion, and therefore, improve also performance in applications that suffer a high level of contention.

We implement the EHB scheme by modifying the software routine that is called as part of the abort process in LogTM-SE. More specifically, we estimate the number of cycles that it would take to execute one iteration of the loop shown in Figure 6 and calculate the waiting time by multiplying this by a random number between zero and the upper limit. As in the ESB scheme, the upper limit is calculated following Equation (1). Figure 7 summarizes how the hardware back-off has been implemented. As it can be observed, we have estimated that each iteration of the loop shown in Figure 6 takes approximately 14 cycles because it would involve the execution of 14 machine instructions, and we are assuming single issue, in order cores (Instructions per cycle (IPC) value of 1).

Finally, the LHB scheme differs from EHB in the way in which the upper limit is calculated. As already commented, the EHB and ESB mechanisms increase the upper limit exponentially with the number of retries, as illustrated by Equation (1). This results in a slight increase in the upper limit when suffering the first aborts. We have observed that for applications with high contention, initial small waiting times cause a significant number of aborts, and consequently, wasted work. Alternatively, the LHB scheme explores the effects of a more aggressive way of computing the upper limit. In particular, we consider the linear approach presented in Equation (2). Compared with Equation (1), this approach differs in how the upper limit is increased for the first 256 retries of a transaction, which is performed linearly in steps of 256. After 256 retries, both the EHB and ESB mechanisms would update the upper limit in the same way. These values have been chosen through experimentation.

$$\text{upper bound} = \begin{cases} 2^8 * N & \text{if } N \leq 256; \\ 2^{16} + (N - 256) & \text{if } N > 256. \end{cases} \tag{2}$$

```
waiting_time = random(0 to upperbound) * 14;
suspend processor for waiting_time cycles;
```

Figure 7. Implementation of the hardware back-off.

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

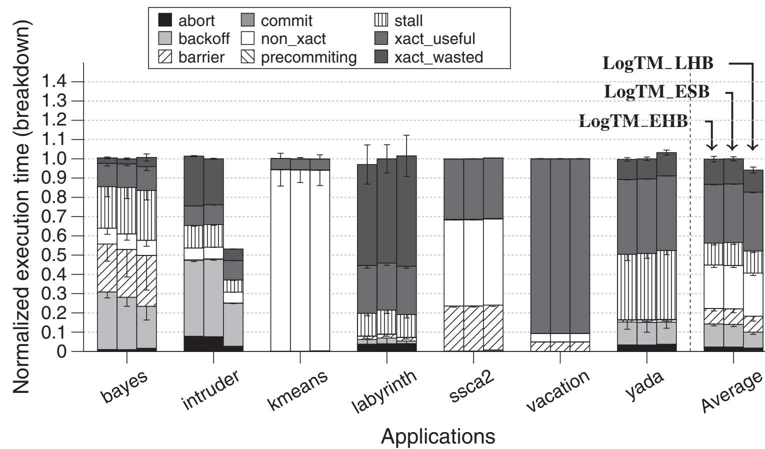

Figure 8. Breakdown of the execution times of eager-eager for different back-off mechanisms.

Figure 8 plots the execution times that are obtained for the three back-off schemes previously discussed. All execution times have been normalized with respect to the values obtained for the base case (LogTM-SE with the ESB mechanism). Again, all bars have been split into the categories presented in Section 4.1. From this graph, we can extract two main observations. The first is that, as expected, LogTM_ESB and LogTM_EHB obtain virtually the same results in terms of execution time. As discussed, the EHB mechanism approaches the behavior of the ESB but implements the waiting by stalling the processor instead of executing the loop of Figure 6. The second observation is that for applications with high contention and short transactions, such as intruder, the fact that the LHB scheme increases waiting times more rapidly than both the EHB and ESB ones, results in a very significant reduction in the number of aborts (see the abort category). In particular, the number of aborts in intruder is about 150,000 for both the EHB and ESB schemes, hogging one of the three transactions found in this application almost two-thirds of the total number of aborts. On the contrary, the number of aborts suffered when the LHB scheme is used is reduced to 43,000, being approximately 22,000 caused by the specially conflicting transaction. This reduction in the number of aborts translates also into noticeable reductions in the fraction of the execution time wasted because of aborts (Xact_wasted) and the back-off time. At the end, these improvements translate into a significant reduction in the execution time (47%). For the rest of the applications, the use of the LHB scheme barely affects execution time. The exception is yada, which is penalized by the longer waiting times entailed by LHB.

Also, very interesting are the results obtained in terms of energy reductions, which are presented in Figure 9. Again, all results are normalized with respect to the base configuration (LogTM_ESB). Contrary to the case of execution times, the energy figures exhibited by the ESB and EHB schemes differ significantly for applications having a noticeable fraction of the execution time because of the back-off, namely bayes, intruder, and yada. As it can be observed, these differences come fundamentally from the amount of energy consumed in the L1 caches. The fact that the ESB scheme keeps the processor, executing instructions during a back-off results into an increased number of accesses to the L1 caches when compared with the EHB scheme. Finally, the LHB scheme also brings a very significant reduction in terms of energy consumption for intruder application. This reduction comes from the ability of the LHB scheme to drastically reduce the number of aborts in this application.

4.5. Influence of directory aliasing in lazy-lazy systems

One of the performance pathologies in HTM systems identified in [19] is serialized commit. HTM systems that use lazy CD must serialize transactions during the commit phase to guarantee the serialization property, that is, to ensure a global serial order in transaction completion. In this way, this pathology exclusively affects lazy-lazy systems such as STCC-SP. Committing transactions may

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

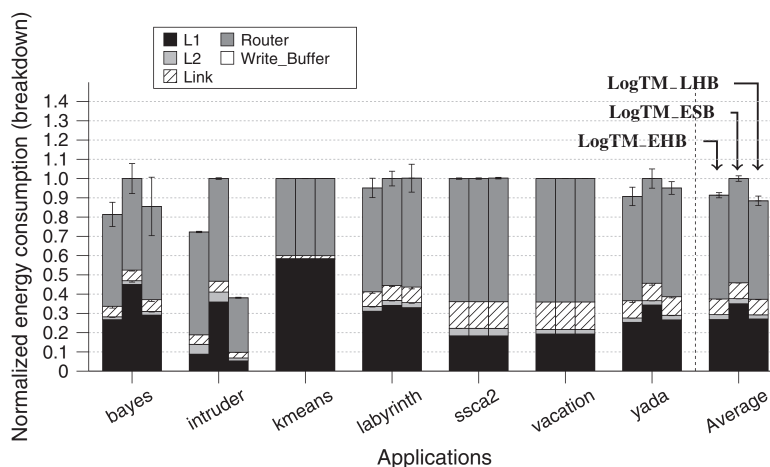

Figure 9. Breakdown of the energy consumed by eager-eager for different back-off mechanisms.

stall while they are waiting for other transactions to commit. In this way, if there is contention during the commit phase, the stall time might considerably increase the overall transaction execution time. Moreover, the performance impact of serialized commits may be quite significant for applications with small transactions.

As explained in Section 2, STCC-SP enhances the original lazy-lazy HTM system proposed in [15] to improve the performance of the commit phase with the SEQ-PRO algorithm that allows for parallel commits in some cases [10]. To do so, the physically distributed banks of the L2 cache act as a distributed arbiter. When a transaction reaches the precommit phase, it must book all L2 cache banks belonging to its read and write sets in increasing order. One particular L2 cache bank will belong to a transaction's write set if at least one address of this set is mapped to that bank. Two or more transactions can go through the precommit phase simultaneously if they book disjoint sets of L2 cache banks. Otherwise, there will be a single committing transaction while the others will have to wait.

Conflicting L2 cache banks may be due to transactions accessing the same memory address. If that is the case, one of the transactions must be aborted. However, it may also happen that two or more transactions access different memory addresses that map to the same L2 cache bank. Therefore, even though no real conflict occurs, a side effect of the serialized commit pathology that we call directory aliasing takes place. The directory aliasing phenomenon directly depends on the address mapping policy of the underlying chip multiprocessor architecture, the data access pattern, and the size of the read and write sets of the transactions. It is worth noting that the combination of both determines the set of L2 cache banks that need to be reserved during the precommit phase. The smaller the number of L2 cache banks to be booked, the faster the precommit phase will come to an end. Nevertheless, this does not necessarily mean a better behavior because of directory aliasing.

So far, all the experiments have assumed the default address mapping policy used by GEMS. Such an address mapping policy is depicted in Figure 10 and proceeds as follows. Cache block size is 64 bytes; hence, the block offset comprehends the six lowest order address bits. Each L2 cache bank is composed of 2048 sets of four cache blocks each so that the set index requires the following eleven lowest order bits. Then, the next four bits are used to index the particular L2 cache bank. In this scheme, each L2 cache set is mapped to the same L2 cache bank, that is, blocks of 128 KB are assigned to the same L2 cache bank. As opposed to the block address mapping policy, the cyclic address mapping policy, also shown in Figure 10, uses the four lowest order bits following the block offset to select the L2 cache bank. In this way, consecutive L2 cache banks are cyclically assigned cache blocks in ascending order. The rest of this section analyzes the impact of the address mapping policy in terms of performance and energy efficiency on the very same set of benchmarks used in the previous sections.

Copyright © 2012 John Wiley & Sons, Ltd.

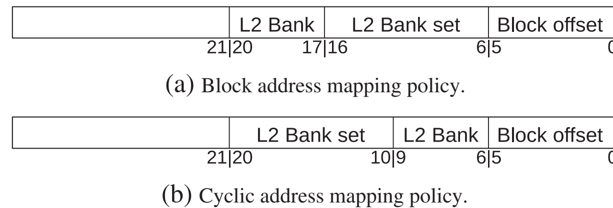

Figure 10. Address mapping policies of cache blocks to L2 banks. (a) Block and (b) cyclic.

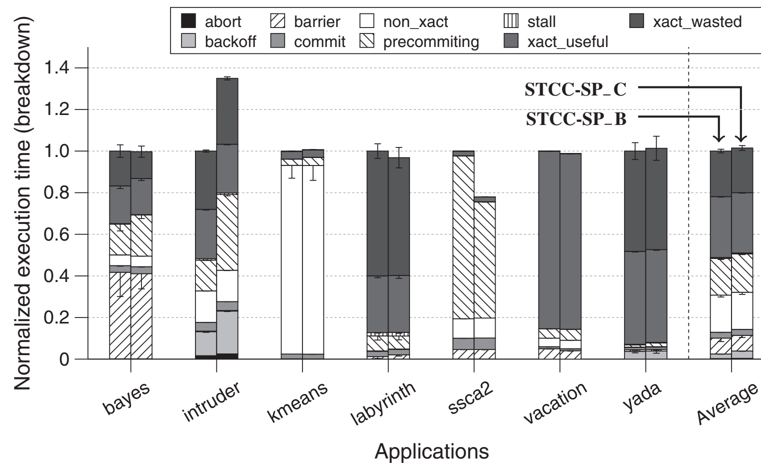

Figure 11. Breakdown of the execution times of lazy-lazy for different address mapping polices.

In Figure 11, we compare the execution time of the benchmarks when using a block address mapping policy as opposed to a cyclic address mapping policy. Results are normalized to the block address mapping policy used by GEMS. As we can see, the impact of this policy is not relevant for most benchmarks. In particular, there are minor differences for bayes, kmeans, labyrinth, vacation, and yada. On the contrary, intruder and ssca2 suffer considerable deviations from the base case. In particular, intruder and ssca2 experience an increase and a reduction of 35 and 22% in the execution time, respectively. With ssca2, even though the cyclic address mapping policy may involve booking a larger number of L2 cache banks, it is scattering memory accesses to the main data structure of the application on different L2 cache banks. This directly translates into a much shorter precommit phase because more transactions can go through the precommit phase in parallel (54% of parallel commits with two or more transactions). Because transactions in ssca2 are quite small, the precommit phase accounts for a significant percentage of the total execution time. Consequently, reductions in the precommit phase time directly decrease the overall execution time of *ssca2*. Unlike ssca2, the cyclic address mapping policy negatively affects intruder. In this case, the cyclic address mapping policy unnecessarily increases the number of L2 cache banks involved in the precommit phase. One of the transactions manipulates a queue of elements, that is, all threads running such a transaction access the very same queue and dequeue pointers. The problem arises because the cyclic address policy makes it necessary to book two L2 cache banks rather than one. This fact triggers a chain reaction of increased precommit phase time, number of aborts (32% increment), back-off time, and transactional wasted time.

Finally, Figure 12 shows the energy consumption breakdown by hardware structures when comparing the block address mapping policy with the cyclic address mapping policy. With ssca2, the router energy expenses decrease because of the shortened precommit phase. Note that the pronounced reduction is due to the Orion 2.0 router energy model that exhibits a sublinear power growth with respect to the network average load. For the same reason, intruder energy requirements substantially increase mostly due to the router energy consumption.

Copyright © 2012 John Wiley & Sons, Ltd.

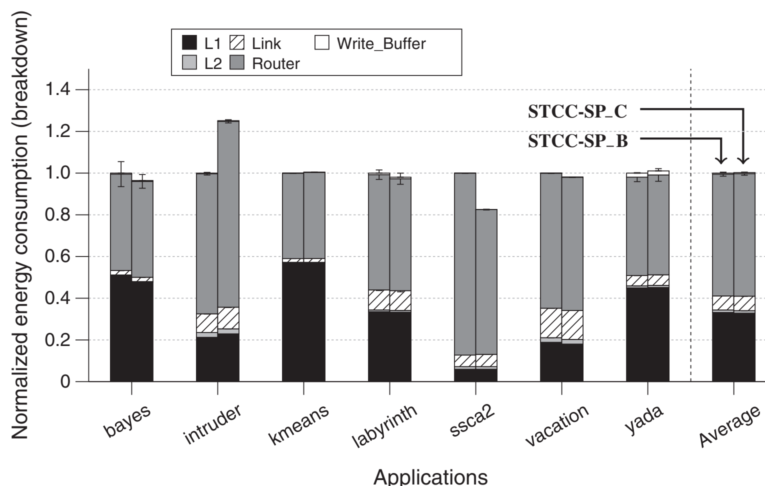

Figure 12. Breakdown of the energy consumed by lazy-lazy for different address mapping polices.

5. CONCLUSIONS

This article presents a comprehensive analysis of two well-known HTM systems, namely LogTM-SE and Scalable TCC, that represent the eager-eager and lazy-lazy approaches for VM and CD, respectively. Our experiments, conducted on a widely accepted simulation platform, compare both HTM systems in terms of execution time, energy consumption, and network traffic. Results show that even though the lazy-lazy system outperforms the eager-eager system on average, there are considerable deviations depending on the particular characteristics of each application. In addition, we also found that reductions in the execution time are not directly proportional to equivalent reductions in either energy consumption or network traffic mainly due to their particular implementations or the pathologies they suffer.

In general, contention with LogTM-SE leads to a large number of either stalled or aborted transactions depending on their write sets interactions. This behavior generates a lot of network traffic because of the persistent stall process, which translates into a significant amount of energy spent in the interconnection network. Additionally, we have found that when the number of aborts is high, a significant fraction of the energy consumed in the memory hierarchy is attributable to the back-off mechanism. In particular, the software version of the back-off mechanism originally included in LogTM-SE (ESB) is energy-inefficient because waitings are implemented by executing several instructions in a loop. This increases the amount of energy consumed in the L1 caches when compared with a hardware implementation of the same algorithm (EHB). In general, the EHB implementation brings reductions in terms of energy consumption ranging from 10 to 30% for applications that spend a nonnegligible fraction of their time in the back-off phase (viz., bayes, intruder, and yada). Additionally, we have found that the LHB implementation can significantly reduce the execution time of intruder application by cutting off the number of aborts that this application would suffer in an eager-eager system, and consequently, by easing forward progress. Although a reduction of 47% in execution time is obtained for intruder when the LHB scheme is employed, STCC-SP still is the clear winner.

Meanwhile, the optimistic concurrency control of Scalable TCC guarantees that at least one transaction will be able to commit in the presence of contention. Nevertheless, the address mapping policy to cache banks may cause the appearance of directory aliasing in some applications that artificially induces the serialized commits pathology. A further analysis reveals that a cyclic address mapping policy of cache blocks to L2 cache banks may thwart the effects of directory aliasing in some applications, such as ssca2, by scattering memory accesses to different L2 cache banks. However, it also may be detrimental to other applications, such as intruder, where contention is due to

Copyright © 2012 John Wiley & Sons, Ltd.

real conflicts. In this case, the block address mapping policy enlarges the precommit phase given the fact that the committing transactions need to reserve a larger number of L2 cache banks.

ACKNOWLEDGEMENTS

This work has been jointly supported by the Spanish MEC and MICINN under grant TIN2009-141475-C04-02 and European Comission FEDER funds under grant Consolider Ingenio-2010 CSD2006-00046. Epifanio Gaona is supported by a fellowship 09503/FPI/08 from Fundación Séneca, Agencia Regional de Ciencia y Tecnología de la Región de Murcia (II PCTRM).

REFERENCES

1. Borkar S. Thousand core chips: a technology perspective. *Design Automation Conference (DAC-44)*, San Diego, California, 2007; 746–749.
2. Dice D, Lev Y, Moir M, Nussbaum D. Early experience with a commercial hardware transactional memory implementation. *14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-14)*, Washington, DC, 2009; 157–168.
3. Herlihy M, Eliot J, Moss B. Transactional memory: architectural support for lock-free data structures. *International Symposium on Computer Architecture (ISCA-20)*, San Diego, California, 1993; 289–300.
4. Harris T, Cristal A, Unsal OS, Ayguad E, Gagliardi F, Smith B, Valero M. Transactional memory: an overview. *IEEE Micro* May-June 2007; **27**(3):8–29.
5. Yen L, Bobba J, Marty MR, Moore KE, Volos H, Hill MD, Swift MM, Wood DA. LogTM-SE: decoupling hardware transactional memory from caches. *13th International symposium on high performance computer architecture (HPCA-13)*, Phoenix, Arizona, 2007; 261–272.
6. Tomic S, Perfumo C, Kulkarni CE, Armejach A, Cristal A, Unsal OS, Harris T, Valero M. EazyHTM: eager-lazy hardware transactional memory. *42nd International Symposium on Microarchitecture (MICRO-42)*, New York, NY, USA, 2009; 145–155.
7. Bobba J, Goyal N, Hill MD, Swift MM, Wood DA. TokenTM: efficient execution of large transactions with hardware transactional memory. *35th International Symposium on Computer Architecture (ISCA-35)*, Beijing, China, 2008; 127–138.
8. Rajwar R, Herlihy M, Lai KK. Virtualizing transactional memory. *32nd International Symposium on Computer Architecture (ISCA-32)*, Madison, Wisconsin, USA, 2005; 494–505.
9. Ananian CS, Asanovic K, Kuszmaul BC, Leiserson CE, Lie S. Unbounded transactional memory. *11th International Conference on High-Performance Computer Architecture (HPCA-11)*, San Francisco, CA, USA, 2005; 316–327.
10. Pugsley SH, Awasthi M, Madan N, Muralimanohar N, Balasubramonian R. Scalable and reliable communication for hardware transactional memory. *17th International Conference on Parallel Architecture and Compilation Techniques (PACT-17)*, Toronto, Ontario, Canada, 2008; 144–154.
11. Ceze L, Tuck J, Torrellas J, Cascaval C. Bulk disambiguation of speculative threads in multiprocessors. *33rd International Symposium on Computer Architecture (ISCA-33)*, Boston, MA, USA, October 6, 2006; 227–238.
12. Shriraman A, Dwarkadas S. Refereeing conflicts in hardware transactional memory. *23rd International Conference on Supercomputing (ICS-23)*, Yorktown Heights, NY, USA, 2009; 136–146.
13. Ferri C, Wood S, Moreshet T, Bahar RI, Herlihy M. Embedded-TM: energy and complexity-effective hardware transactional memory for embedded multicore systems. *Journal of Parallel and Distributed Computing (JPDC)* October 2010; **70**(10):1042–1052.
14. Gaona-Ramirez E, Titos-Gil R, Fernandez J, Acacio ME. Characterizing energy consumption in hardware transactional memory systems. *22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD-22)*, Petropolis, Brazil, 2010; 9–16.
15. Chafi H, Casper J, Carlstrom BD, McDonald A, Minh CC, Baek W, Kozyrakis C, Olukotun K. A scalable, non-blocking approach to transactional memory. *13th International Conference on High-Performance Computer Architecture (HPCA-13)*, Phoenix, Arizona, USA, 2007; 97–108.
16. Moore KE, Bobba J, Moravan MJ, Hill MD, Wood DA. LogTM: log-based transactional memory. *12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, Austin, Texas, 2006; 254–265.
17. Martin MMK, Sorin DJ, Beckmann BM, Marty MR, Xu M, Alameldeen AR, Moore KE, Hill MD, Wood DA. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News* 2005; **33**(4):92–99.
18. Hammond L, Wong V, Chen MK, Carlstrom BD, Davis JD, Hertzberg B, Prabhu MK, Wijaya H, Kozyrakis C, Olukotun K. Transactional memory coherence and consistency. *31st International Symposium on Computer Architecture (ISCA-31)*, Munich, Germany, 2004; 102–113.
19. Bobba J, Moore KE, Volos H, Yen L, Hill MD, Swift MM, Wood DA. Performance pathologies in hardware transactional memory. *34th International Symposium on Computer Architecture (ISCA-34)*, San Diego, CA, USA, 2007; 81–91.

Copyright © 2012 John Wiley & Sons, Ltd.

20. Magnusson PS, Christensson M, Eskilson J, Forsgren D, Hallberg G, Hogberg J, Larsson F, Moestedt A, Werner B. Simics: a full system simulation platform. *IEEE Computer* February 2002; **35**:50–58.
21. Kahng AB, Li B, Peh LS, Samadi K. ORION 2.0: a fast and accurate NoC power and area model for early-stage design space exploration. *Design, Automation & Test in Europe (DATE-1)*, Nice, France, 2009; 423–428.
22. HP Labs. (Available at: http://quid.hpl.hp.com:9081/cacti) [June 2011].
23. Minh CC, Chung J, Kozyrakis C, Olukotun K. STAMP: Stanford transactional applications for multi-processing. *4th IEEE International Symposium on Workload Characterization (IISWC-4)*, Seattle, WA, USA, 2008; 35–46.

Copyright © 2012 John Wiley & Sons, Ltd.
DOI: 10.1002/cpe