Characterizing Energy Consumption in Hardware Transactional Memory Systems

Epifanio Gaona-Ramírez, Rubén Titos-Gil, Juan Fernández, Manuel E. Acacio
Computer Engineering Department
University of Murcia
Murcia, Spain
{fanios.gr, rtitos, juanf, meacacio}@ditec.um.es

***Abstract*—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 CMP architectures may need to provide hardware support for transactional memory. On the other hand, power dissipation constitutes a first class consideration in multicore processor design. 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 the LogTM-SE *Eager-Eager* system and a version of the Scalable TCC *Lazy-Lazy* system that enables parallel commits. To the best of our knowledge, this is the first characterization in terms of energy consumption of hardware transactional memory systems. To do that, 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. Results show that the energy consumption of the *Eager-Eager* system is 60% higher on average than in the *Lazy-Lazy* case, whereas performance differences between the two systems are 42% on average. Finally, we found that although on average *Lazy-Lazy* beats *Eager-Eager* there are considerable deviations in performance depending on the particular characteristics of each application.**

***Keywords*-Hardware Transactional Memory (HTM); version management; conflict detection; lazy-lazy; eager-eager.**

I. Introduction and motivation

In recent years we have witnessed the replacement of single-core processors by multi-core ones, which has made parallel computing resources commonplace. Whereas 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 endeavor, even for experienced programmers. On the other hand, power consumption constitutes nowadays a first class consideration in multicore processor designs, and energy-efficient architectures are a must.

Transactional Memory (TM) is currently considered 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 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 in order to satisfy the atomicity property. In this way, the benefits derived from transactional memory are twofold. Transactions are speculatively executed which 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, i.e. 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, i.e. 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.

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 since the original values remain unmodified in cache memory. On the other hand, while eager CD incurs a bigger overhead due to the persistent checking process, lazy CD usually wastes a larger amount of work every time a transaction aborts. The comparison gets 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.

To the best of our knowledge, even though *Lazy-Lazy* systems are considered as the the best choice in the general case [5], 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 an *Eager-Eager* HTM systems for general purpose systems. Ferri *et al.* [6] perform an analysis of both HTM systems but only for embedded architectures. Because of the specific conditions of this architecture and their inherent harder hardware constraints, their proposals strongly focus on the energy efficiency issue at the expense of getting worse performance. In this work, we conduct a fair comparison of two well-known HTM systems. In particular, we compare LogTM-SE [7], as an example of a *Eager-Eager* system, with Scalable TCC [8], a *Lazy-Lazy* HTM system. To do this, 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. Results show that the energy consumption of the *Eager-Eager* system is 60% higher on average than in the *Lazy-Lazy* case, whereas performance differences between the two systems are 42% on average. We found that although on average *Lazy-Lazy* beats *Eager-Eager* 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.

The rest of the paper is organized as follows. Section II fully describes the two HTM systems targeted by our study. In Section III, 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 IV. Finally, conclusions are given in Section V.

II. Characterized HTM systems

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

A. LogTM

LogTM [9] is a widely-known *Eager-Eager* system that makes use of eager version management, storing new values directly in the memory location of the variable (or “in place”), while preserving old values “on the side”. 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, while aborts are slower since the system must trap to a software handler to unroll the log in order to restore pre-transactional 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 acknowledgment (NACK), indicating that the requester transaction must stall its execution until the offended transaction releases isolation over the requested data upon commit/abort. This scheme can result in cycles, so LogTM uses a conservative deadlock avoidance mechanism based on timestamps, always giving priority to the eldest transaction. LogTM-SE [7] 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 book-keeping 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 [10], and the *Eager-Eager* system we characterize in this work.

B. Scalable TCC

Scalable-TCC [8] (STCC) is a popular, scalable, non-blocking implementation of TM that is tuned for continuous use of transactions within parallel programs. STCC provides non-blocking synchronization and an easy-to-understand consistency model. STCC is based on a directory-based implementation of the Transactional Coherence and Consistency (TCC) [11] 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 writebacked into coherent memory only when a transaction commits. STCC uses a two-phase parallel commit algorithm which is supported by an 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. Once in the second commit phase, a transaction 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) [12] 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 write set in increasing order. A transaction cannot send the subsequent *book message* until the previously requested directory bank have 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 gets all permissions, 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 committing transaction at the head of the queue is evicted in each bank previously booked by the same (*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 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. A more advanced version of the algorithm, known as SEQ-PRO [12] 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.

III. Evaluation Environment

In this section, we describe the evaluation environment used in this paper. 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 list the consumption models used to characterize energy consumption. In particular, we focus on the energy consumed in the on-chip memory hierarchy. Finally, we end with a description of the benchmarks used to conduct the simulations.

A. System Settings

We use a full-system execution-driven simulation based on the Wisconsin GEMS toolset [10], in conjunction with Virtutech Simics [13]. 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 CMP 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 512KB each. The L1 caches maintain inclusion with the L2. 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 2D-mesh network. Each tile contains a router where the private L1, the slice of L2 and the memory controller are connected to, plus the links to the neighboring tiles. In this 4x4 2D-network, each router has between 5 and 7 ports, with an average of 6 ports per router.

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 based on Orion 2.0 [14]. In particular, we have extended the network simulator provided by GEMS with the consumption model included in Orion. Table I shows the values of some of the parameters assumed for the interconnection network. For those not listed in the table, we use the default values given in Orion. On the other hand, the energy spent in the memory structures (L1, L2, *Write Buffer*) were measured based on the consumption model of CACTI 5.3 rev 174 [15]. In the case of the L2 cache, 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 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 book-keeping. Additionally, Ruby provides a naive version of a *Lazy-Lazy* system that employs a commit token to serialize transactions 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 [8], although it does not use the interconnection network to coordinate the entire process. Thus, the *Lazy-Lazy* implementation provided in GEMS is not only non-scalable, since the central referee would become a bottleneck, but also unrealistic since 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 II that uses the 2D-mesh network. More specifically, we have evaluated in this work the SEQ-PRO algorithm proposed by Pugsley *et al*. in [12]. SEQ-PRO allows for parallel commits (as the SEQ algorithm) and implements the parallel reader optimization. In order to implement SEQ-PRO, we had to add three new request messages involved in the *commit* process: 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 [12].

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 due to 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.

B. Benchmarks Settings

For the evaluation, we use eight transactional benchmarks extracted from the STAMP suite [16] on its version 0.9.10. These applications allow to stress a TM system in several ways. To show a wide range of cases, we evaluate all STAMP applications using the most significant input size in each case (in general, what is called medium size). Table III describes the benchmarks and the values of the input parameters used in this work.

IV. 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 commit algorithm (STCC-SP from now on). We start with a comparison between these two HTM systems in terms of execution time. Next, we will study the energy consumption of each system when executing the transactional workloads. Finally, we also compare the traffic that both HTM systems generate.

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

Table I
Parameters of Orion 2.0.

MESI Directory-based CMP	
Cores	16, simple issue, in order, non-memory IPC=1
Memory and Directory settings	
L1 Cache I&D	Private, 32 KB, split 2 way, 1-cycle latency
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

Table II
System Parameters.

Benchmark	**Input**
Bayes	-v32 -r4096 -n2 -p20 -i2 -e2
Genome	-g512 -s32 -n32768
Intruder	-a10 -l16 -n4096 -s1
Kmeans	-m40 -n40 -t0.05 -i random-n16384-d24-c16
Ssca2	-s13 -i1.0 -u1.0 -l3 -p3
Labyrinth	-i random-x32-y32-z3-n96
Vacation	-n4 -q60 -u90 -r1048576 -t4096
Yada	-a10 -i ttimeu10000.2

Table III
Workloads and inputs.

A. Performance

For the eight transactional benchmarks pointed out in Section III, 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 clear understanding of the results Figure 2 divides the execution times into the following categories: *Abort* (time spent during aborts), *Back-off*, 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), *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. Its upper bound 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*)

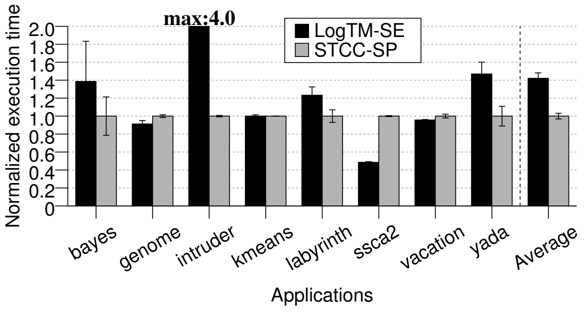

Figure 1. Normalized execution times.

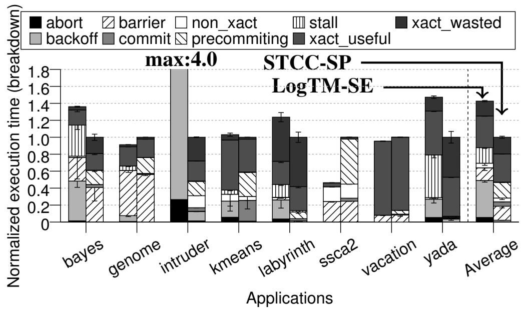

Figure 2. Breakdown of the execution times.

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 *genome*, *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. 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 300% more for *intruder*). In this way, on average, the *Lazy-Lazy* STCC-SP system improves performance (about 42%) when compared with the *Eager-Eager* LogTM-SE system. Below, 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 [16] also lead to a non negligible 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, 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 LogTM-SE. As a consequence, LogTM-SE degrades the performance by a factor of 4 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 halves the execution time of STCC-SP. The reason can be appreciated in Figure 2. Almost half of the time of STCC-SP is spent in the precommit phase. 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 absent of real conflicts. We will call this behavior as "directory aliasing" that entails the well-known "Serialized Commits" pathology [17] of the *Lazy-Lazy* systems.

As *bayes*, *yada* has significant transactional time and write sets and medium contention [16], 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 its time in an active waiting (*stall*) trying to solve conflicts. Though 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 since 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 due to aborting transactions. *Labyrinth*'s characteristics are the same as *bayes*.

The rest of the benchmarks show similar results with

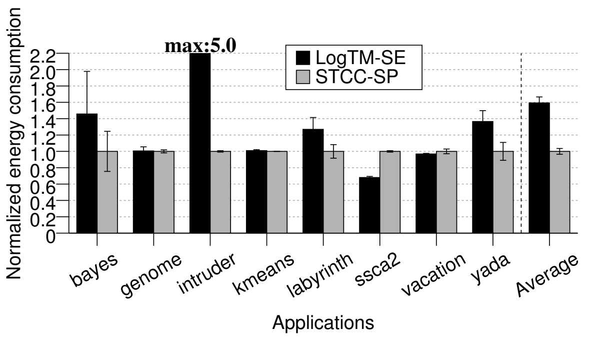

Figure 3. Normalized energy consumption (Power Delay Product).

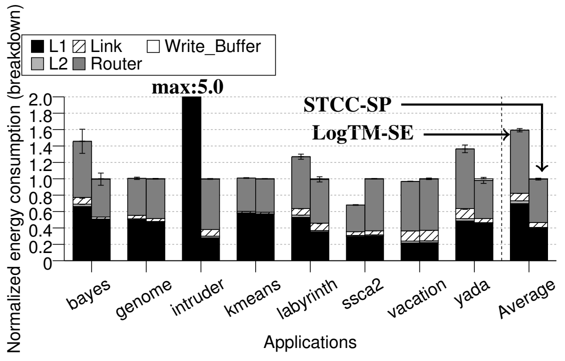

Figure 4. Breakdown of energy consumption.

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.

B. *Energy*

Figure 3 shows the dynamic energy consumption or more commonly known as dynamic Power Delay Product (PDP) 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 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 *genome*, *kmeans* and *vacation* there are no noticeable differences between both systems. Note that while the average difference in performance among the two systems was 42% in favor of STCC-SP, the latter outperforms about 60% LogTM-SE when energy is considered.

The differences in terms of energy consumption found in *bayes*, *genome*, *kmeans*, *labyrinth* and *vacation* for LogTM-SE and STCC-SP are almost identical to the ones previously reported in terms of execution time. For *genome*, *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 40% and 24% respectively found for STCC-SP in execution time directly translates into a reduction of almost the same extent in the Power Delay Product.

For *intruder*, we can see that LogTM-SE consumes much more energy than STCC-SP. In this case, the differences can not be justified by just taking into account the execution times. Labyrinth algorithm tries to write a road in a 32x32x3 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 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. Transaction with STCC-SP will commit more easily since when they acquire the commit permission (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 leads to much more energy consumption and execution time in *Eager-Eager* systems than in *Lazy-Lazy* systems.

Although STCC-SE is more energy-efficient than LogTM-SE in *yada*, the difference is about 38% whereas the performance gap reaches 48%. This is because STCC-SP experiences a great amount of aborts, increasing significantly the energy consumed in the L1 caches. 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 (improvements of 30% are obtained). In this case, the difference is concentrated on the energy consumed in the interconnection network (in particular in the routers). As already discussed, STCC-SP spends half 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.

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 60% in STCC-SE and 55% in LogTM-SE. It is worth noting that the energy consumed in the L2 cache, the links of the network and the write buffer (only for STCC-SP) is almost negligible (8% of total energy). Consumption in L1 caches, however, can be between 22% and 60% of the total energetic expense. The fraction of the energy consumed in the network is more evident for applications with high contention as *intruder*. Some issues that motivate the increased energy consumption found in LogTM-SE are:

- 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 or 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. Acknowledgments 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 continuously memory requests (see *stall* state in the figure 2) in case of conflicts until the corresponding transaction aborts or achieves its goal. This supposes a considerable energetic expense.

Regarding the energy consumed in the interconnection network, we have found that the most important fraction is due to the routers (52% on average), while 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 size (16 bytes) and six-input ports routers, which makes the routers be the most important source of energy consumption.

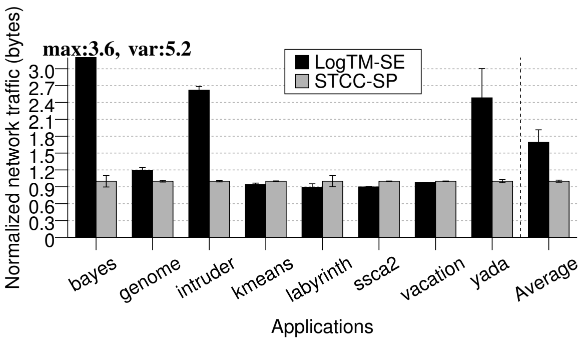

Figure 5. Normalized network traffic.

C. Network traffic

Finally, Figure 5 shows the amount of traffic in the interconnection network for both LogTM-SE and STCC-SP. As before, results have been normalized with respect to the last one. In general, LogTM-SE entails approximately 70% more network traffic than STCC-SE. The reason can be found in the high number of retries that are 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 IV-B). The difference in network traffic is considerable in the case of *bayes*, *intruder*, *labyrinth* and *yada*. These benchmarks are characterized by exhibiting high contention and/or by the large size of their transactions [16]. 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 since the number of messages required to book directories is limited [12].

V. Conclusion

This paper 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, the 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 due to the persistent *stall* process. In the 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 mapping of cache lines to cache banks may cause the appearance of the "directory aliasing" problem in some applications which artificially induces the "Serialized Commits" pathology.

Future work includes a smarter mapping of cache lines to cache banks and a possibly larger number of bookable directories to avoid the "directory aliasing" problem in STCC-SP, and a lineal back-off implementation along with a more energy-efficient protocol to get better results in LogTM-SE.

Acknowledgment

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

References

[1] S. Borkar, "Thousand core chips: A technology perspective," in *DAC-44*, June 2007, pp. 746–749.

[2] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early experience with a commercial hardware transactional memory implementation," in *ASPLOS-14*, November 2009, pp. 157–68.

[3] M. Herlihy, J. Eliot, and B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *ISCA-20*, May 1993, pp. 289–300.

[4] T. Harris, A. Cristal, O. S. Unsal, E. Ayguad, F. Gagliardi, B. Smith, and M. Valero, "Transactional memory: An overview," *IEEE Micro*, vol. 27, no. 3, pp. 8–29, May-June 2007.

[5] A. Shriraman and S. Dwarkadas, "Refereeing conflicts in hardware transactional memory," in *ICS-23*, April 2009, pp. 136–146.

[6] C. Ferri, S. Wood, T. Moreshet, R. I. Bahar, and M. Herlihy, "Embedded-TM: Energy and complexity-effective hardware transactional memory for embedded multicore systems," *Journal of Parallel and Distributed Computing (JPDC)*, February 2010. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2010.02.003

[7] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "LogTM-SE: Decoupling hardware transactional memory from caches," in *HPCA-13*, February 2007, pp. 261–272.

[8] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, "A scalable, non-blocking approach to transactional memory," in *HPCA-13*, February 2007, pp. 97–108.

[9] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: log-based transactional memory," in *HPCA-12*, February 2006, pp. 254–265.

[10] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH CAN*, vol. 33, no. 4, pp. 92–99, March 2005.

[11] L. Hammond, V. Wong, M. K. 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 *ISCA-31*, June 2004, pp. 102–113.

[12] S. H. Pugsley, M. Awasthi, N. Madan, N. Muralimanohar, and R. Balasubramonian, "Scalable and reliable communication for hardware transactional memory," in *PACT*, October 2008, pp. 144–154.

[13] 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*, vol. 35, pp. 50–58, February 2002.

[14] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi, "ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration," in *DATE*, September 2009, pp. 423–428.

[15] HP Labs, "http://quid.hpl.hp.com:9081/cacti."

[16] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *IISWC*, September 2008, pp. 35–46.

[17] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance pathologies in hardware transactional memory," in *ISCA-34*, June 2007, pp. 81–91.