J Supercomput (2014) 68:914–934
DOI 10.1007/s11227-013-1072-y

Selective dynamic serialization for reducing energy consumption in hardware transactional memory systems

Epifanio Gaona · J. Rubén Titos-Gil · Juan Fernández · Manuel E. Acacio

Published online: 22 December 2013

Abstract In the search for new paradigms to simplify multithreaded programming, Transactional Memory (TM) is currently being advocated as a promising alternative to deadlock-prone lock-based synchronization. In this way, future many-core CMP architectures may need to provide hardware support for TM. On the other hand, power dissipation constitutes a first class consideration in multicore processor designs. In this work, we propose Selective Dynamic Serialization (SDS) as a new technique to improve energy consumption without degrading performance in applications with conflicting transactions by avoiding wasted work due to aborted transactions. Our proposal, which is implemented on top of a hardware transactional memory (HTM) system with an eager conflict management policy, detects and serializes conflicting transactions dynamically (at run-time). In its simplest form, in case of conflict, one transaction is allowed to continue whilst the rest are completely stalled. Once the executing transaction has finished, it wakes up several of the stalling transactions. More elaborated implementations of SDS try to delay this behavior until serialization of transactions is profitable, achieving the best trade-off between performance, energy savings and network traffic. SDS implementations differ from each other in

E. Gaona · M. E. Acacio (✉)
Universidad de Murcia, Murcia, Spain
e-mail: meacacio@ditec.um.es

E. Gaona
e-mail: fanios.gr@ditec.um.es

J. R. Titos-Gil
Chalmers University of Technology, Göteborg, Sweden
e-mail: ruben.titos@chalmers.se

J. Fernández
Intel Barcelona Research Center, Barcelona, Spain
e-mail: juan.fernandez@intel.com

the condition that triggers the serialization mode. We have evaluated several SDS schemes using GEMS, a full-system simulator implementing the LogTM-SE *Eager–Eager* HTM system, and several benchmarks from the STAMP suite. Results for a 16-core CMP show that SDS obtains reductions of 6 % on average in energy consumption (more than 20 % in high contention scenarios) in a wide range of benchmarks without affecting, on average, execution time. At the same time, network traffic level is also reduced by 22 %.

Keywords Many-core CMPs · Hardware transactional memory · Transactions · Run-time serialization · Energy consumption · Execution time

1 Introduction and motivation

During the past 10 years, we have witnessed a transcendental change of paradigm towards parallel architectures. Most processor manufacturers have released products that incorporate several execution cores on a single chip. The Intel Core and AMD APU families are two well-known examples of how chip multiprocessors (CMP) span every market segment, from server solutions to laptop computers. 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 task, even for experienced programmers. The majority of the software has not been able to keep up with this fundamental architectural shift. Software developing techniques have not evolved fast enough to grasp these new computational resources offered by multiple computational cores, mainly because of the difficulty of parallel programming. Exploiting thread-level parallelism constitutes a major challenge for programmers, and without it we can no longer expect that the increasing number of available transistors on chip will yield application performance improvements with the same efficiency as in the past.

Transactional memory (TM) is currently considered as a promising parallel programming paradigm, and processors implementing TM support in hardware have already been announced. Examples go from the AMD's Advanced Synchronization Facility, a set of $\times 86$ extensions that provide a very limited form of hardware TM support [2] to processors supporting TM, such as the Sun's Rock prototype [3], the chips of IBM BlueGene/Q [4], or the Intel's Haswell microarchitecture [5].

TM borrows the concept of transaction from the database world and brings it into the shared-memory programming model [6]. 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. A TM system can be implemented in either software, hardware, or as a combination of both [7]. The common denominator in all implementations is that transactions are speculatively executed, which hides from 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 simplify the development of multithreaded programs.

Hardware transactional memory (HTM) systems are usually classified in terms of how they tackle with data version management (VM) and conflict detection (CD). In this work, we focus our attention on the extensively used *Eager–Eager* systems. On eagerly versioned systems, updates are done in place, i.e., transactional stores overwrite old values residing in cache memory after storing them in an *undo log*. With eager CD, dependency violations are checked on the fly during the transaction lifetime for each transactional load and store.

On the other hand, the emphasis in microprocessor design has shifted from high performance to a combination of both high performance and low power. Power consumption has also become a first class consideration in multicore processor designs, and energy-efficient architectures are a must. This is true not only for embedded systems [8,9] (such as mobile devices), but also for server and even desktop systems [10]. HTM literature has mostly focused on improving performance, simplicity [11] or even flexibility [12]. A recent study [13] has compared the two predominant HTM approaches (*Eager–Eager* and *Lazy–Lazy*) in terms of their performance and energy consumption, concluding that there is significant room for improvement when considering energy consumption in *Eager–Eager* approaches. The main reason for this is that *Eager–Eager* approaches perform poorly in high-contention scenarios [13]. Unfortunately, these scenarios may not be rare in some future transactional applications.

In this work, we present *Selective Dynamic Serialization* (SDS henceforth), a new technique aimed at reducing energy consumption in HTM systems implementing eager conflict management by dynamically (at run-time) serializing transactions that cannot make progress. Instead of continuously retrying a memory access that caused a conflict with another active transaction that previously accessed the same memory position (as done in *Eager–Eager* systems [14]), the offending transaction is completely stalled entering into a low-power mode that saves energy and bandwidth. Once the offended transaction has completed its execution, it wakes the stalled transaction up. The stalled transaction can still abort if another transaction conflicts with it, but a priori wasted work would have already been avoided. In this way, an *Eager*system with SDS would be able to manage conflicts more efficiently in a high-contention scenario, obtaining significant reductions in terms of energy consumption and, in some cases, execution time. The latter is due to the fact that in these situations SDS would also facilitate forward progress. As an example, typical critical sections in transactional applications include modifying an iterator over a list, or inserting an element in some linked structures. The sequence of addresses would be as follows:

`Read A — Read B — Read C — Write C`

In a high-contention scenario (i.e., several transactions trying to execute the sequence at the same time), this sequence implies that when a transaction reaches the last operation (Write C), the rest of executing transactions could have already read address C. This leads to a conflict. Eventually only the highest priority transaction will commit, but experiencing a significant delay (once all conflicts are solved) and aborting other transactions (which results in wasted work). The competition will start again after restarting transactions' execution, thus entering a cycle of conflicts.

In this case, forward progress is compromised. Situations of this kind are found in *intruder*, for example, a benchmark of the STAMP suite [15]. With SDS, after the conflict is detected, transactions would be executed in turn. In particular, just one of the conflicting transactions would be allowed to continue execution. Once this transaction commits, it would signal another transaction to resume execution beyond the conflicting point.

In this work, we study several implementations of SDS, which differ in the condition that triggers the serialization mode (SM). In the simplest one, which we call DS (from Dynamic Serialization), one transaction is allowed to continue in case of conflict whilst the rest are completely stalled. Once the executing transaction has finished, it wakes up several of the stalling transactions. This "always serialize from the first conflict" case was originally proposed in [16]. Now, we extend such work by presenting in this manuscript more elaborated implementations of SDS that try to delay this behavior until serialization of transactions is profitable, achieving the best trade-off between performance, energy savings and network traffic.

Serialization has already been considered in two previous works ([8,17]). In particular, Moreshet [17] proposed a naive static serialization mechanism in which two conflicting transactions are re-issued in serialized mode, preventing parallel speculation in other aborting transactions. On the other hand, serialization in [8] consists of stopping non-serialized cores until the commit of the serialized one with the subsequent performance penalty. In this way, and compared with these two proposals, the dynamic behavior of SDS brings the following two advantages. First, transactions can still make progress from the beginning of their execution until the presence of a conflict. A transaction will not be serialized if it is not necessary, unlike with static implementations [8,17]. Second, SDS favors parallel speculation as much as possible since serialization is performed at lower level (cache line). SDS also presents a delay in the activation of the serialization mode, achieving a better trade-off between situations where normal *Eager–Eager* systems scale well and the energy efficiency of serialization when forward progress is compromised. The condition of when this serialization mode is activated is the key to distinguish the different SDS flavors proposed in this work.

The rest of the paper is organized as follows. Sect. 2 contains an in-depth description of the proposed SDS technique. In Sect. 3, we detail the configuration of the simulation environment and the workloads used to evaluate SDS. Performance, energy consumption and network traffic figures of several SDS implementations are analyzed in Sect. 4. In Sect. 5, we discuss some related researches. Finally, conclusions are given in Sect. 6.

2 Selective dynamic serialization of transactions in HTM

In this section, we present several implementations of the SDS idea proposed in this work. All implementations are carried out on top of an *Eager–Eager* HTM system (in particular, LogTM-SE [14]), and they only differ in the condition that triggers the serialization of transactions. Through run-time serialization of some transactions, our proposal tries to reduce the energy wasted due to aborted and even-stalled transactions.

Finally, SDS can be applied to any other *Eager–Eager* HTM system that employs the cache coherence protocol to detect conflicts on the fly.

SDS does not change the default behavior of LogTM-SE in absence of conflicts. However, unlike other serialization mechanisms [8,17], SDS serializes transactions in a more flexible way depending on some parameters. From the simplest "always serialize from the first conflict" case originally presented in [16] (base DS), to serializing only when several conflicts or aborts have taken place. This flexibility allows us to perform an in-depth analysis of the benefits of the dynamic serialization of transactions without incurring in some of its drawbacks with some access patterns. Results show that SDS can save energy with no performance penalty (the latter is even improved in some situations). To do that, SDS incorporates a decision logic that enables the serialization mode when high levels of contention have been detected. SDS operates at cache line level so that transactions run smoothly until a conflict in a particular cache block is detected and the system enables the serialization mode. In such a situation, SDS serializes the conflicting transactions by guaranteeing forward progress of one transaction and stalling the others in a low-power state. Once the winner transaction has finished, it wakes up the highest-priority transaction among all the stalled ones. In this way, SDS not only minimizes the energy consumed by the stalled transactions, but also the number of aborted transactions, thus, reducing wasted energy. To do so, SDS requires the hardware support detailed in Sect. 2.1. This hardware allows to implement the different flavors of dynamic serialization (DS) presented in Sects. 2.2 and 2.3. The protocol exemplified in Sect. 2.4 refers to the base DS protocol where serialization mode is enabled from the beginning.

2.1 Architectural aspects of SDS

In LogTM-SE, all transactions make progress by storing new values directly in the memory location of the variable (or "in place"), while preserving old values "on the side" during its execution and making the changes visible to the rest of the system during commit. When a transaction detects a conflicting remote request thanks to the cache coherence protocol, 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. Thereafter, the offending transaction keeps retrying until the commit/abort of the offended transaction, thus wasting a variable amount of energy that depends on the level of contention. Differences between several approaches to SDS rely on choosing the event that triggers the activation of the *Serialization Mode*, while the rest of the system remains equal. There are two possible events: number of subsequent conflicts received for the same address (NACK_SDS) and number of subsequent aborts experimented (ABORT_SDS).

Instead of continuously trying to get access to an address, SDS avoids this persistent retrying process by stalling the offending transaction after entering into the serialization mode. In this low-power state, the offending transaction will not try to get access to the conflicting cache block again without prior notification from another transaction. During that period, the offending transaction will not generate any cache coherence message, but it will have to process incoming cache coherence requests

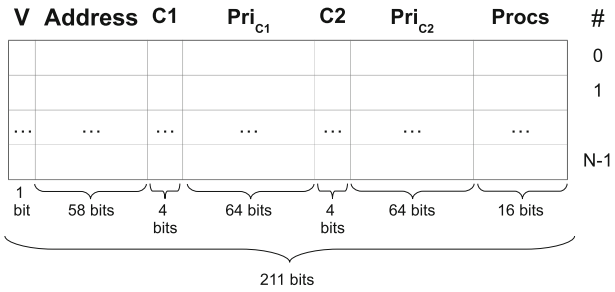

Fig. 1 Structure of the serialization table (ST)

from other transactions. Moreover, it must keep track of all NACKed transactions to wake up the highest-priority one at commit/abort time. To accomplish this task, every transaction has a hardware structure called *Serialization Table* (ST) with one valid entry per each different cache block address that was NACKed by the transaction. Experiments conducted in Sect. 4 have shown that just six entries are enough to prevent overflow situations. This way, the total size of each ST is just 127 bytes. Figure 1 shows the structure of the ST. As it can be seen, each ST has the following fields:

- *V*: valid bit that indicates whether the entry is currently being used (has valid information).
- *Address*: cache block address that has been NACKed.
- *C1 (Core 1)*: core which runs the NACKed transaction that requests *Address* with the *a priori* highest priority (single threaded core).
- Pri_{C1}: priority level of C1 (timestamp of C1).[1]
- *C2 (Core 2)*: core which runs the NACKed transaction that also requested *Address* with the second highest priority.
- Pri_{C2}: priority level of C2 (timestamp of C2).
- *Procs*: bit vector of the NACKed cores for *Address*.

When a transaction receives a request that conflicts with any of the addresses in its read or write sets, there are four possible courses of action:

1. The cache block address of the request is present in the ST and the offending transaction has higher priority than C1. The transaction copies C1/Pri_{C1} into C2/Pri_{C2}, sets the new values for C1/Pri_{C1} and updates the Procs field.
2. The cache block address of the request is present in the ST and the offending transaction has higher priority than C2. The transaction sets the new values for C2/Pri_{C2} and updates the Procs field.
3. The cache block address of the request is present in the ST and the offending transaction has lower priority than C2. The transaction just updates the Procs field.

[1] Our implementation uses the timestamps employed by LogTM-SE as priority mechanism, but any other similar method could be used.

4. The cache block address of the request is not present in the ST. The transaction allocates a new entry in the ST with the cache block address of the request (Address), the identity of the requesting core (C1) and the priority of the offending transaction (Pri_{C1}). Finally, the transaction sets the corresponding bit in the Procs field.

2.2 NACK_SDS

The two implementations of SDS presented in this work (called NACK_SDS and ABORT_SDS) have the same hardware requirements: a serialization mode bit and a counter of one or more bits (saturating counter) per thread. When a conflict is detected in NACK_SDS, the saturating counter (SC) (initially set to zero) is incremented. When the SC saturates (current transaction has experienced certain number of conflicts to the same address), NACK_SDS sets the serialization mode bit enabling the serialization mode for that thread. From that time all conflicting transactions with the one running within this thread will be serialized (also entering in the serialization mode) until commit.

To accelerate the process of serializing transactions, the saturating counter can be initially set to a value greater than zero. In our particular case, we initialize every saturating counter with half of its capacity in case the conflicting transaction is in the serialization table. On the contrary, it is set to zero.

When a conflict is subsequently resolved before entering the serialization mode (the transaction "pass" the conflicting situation but serialization is not needed), the SC is reset to give the transaction the same probability of success to normal execution (not serialized one) in the presence of new conflicts.

Section 4 shows results for NACK_SDS with a one-, two- or three-bit SC. We have decided to evaluate a special case with a zero-bit SC to represent the most aggressive and simplest scheme where serialization mode is always enabled. This design is equivalent to DS, originally presented in [16].

2.3 ABORT_SDS

The condition used in ABORT_SDS to increase the SC counter is the presence of aborts. That is to say, instead of counting the number of conflicts to a particular address, ABORT_SDS increases the SC counter every time an abort is experienced (because of repeated conflicts to the same address or aborts to several addresses). In this case, the saturating counter can only be reset at commit (unlike in NACK_SDS where it is reset every time a conflict is resolved). Again, every SC in ABORT_SDS is initialized to half of its capacity during a commit in case the transaction has been recorded in the serialization table.

2.4 Protocol

Our implementations of SDS are based on the MESI cache coherence protocol, although SDS could be built atop any other cache coherence protocol with a similar behavior such as MOESI. SDS does not modify the cache coherence protocol at

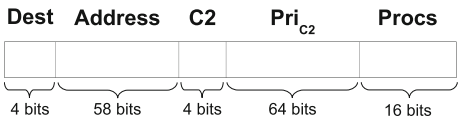

Fig. 2 Anatomy of the UNSTALL message added in our proposal.

all, it only adds a single short control message called UNSTALL with the following fields (Fig. 2):

- *Dest*: destination core.
- *Address*: cache block address that was NACKed by this transaction. Address field in the ST.
- *C2*: C2 field in the ST.
- Pri_{C2}: Pri_{C2} field in the ST.
- *Procs*: bit vector of the NACKed cores for address. Procs field in the ST.

At commit/abort time, a transaction scans its ST and sends an UNSTALL message per each valid entry, that is, per each conflicting cache block NACKed during its lifetime. The destination of the message is C1 and the message includes the Address, C2, Pri_{C2} and Procs fields of the ST entry (the bit corresponding to C1 is reset). Upon reception of an UNSTALL message, a stalled transaction updates its ST using the information carried by the UNSTALL message and resumes execution. If there is an ST entry for the Address of the UNSTALL message, the Procs field of the ST entry is ORed with the Procs field of the UNSTALL message, and the C1/Pri_{C1} and/or C2/Pri_{C2} fields of the ST entry are modified following steps 1 through 3 of the procedure explained in Sect. 2.1. Otherwise, a new entry is added to the ST where Address, C2, Pri_{C2} and Procs fields of the UNSTALL message are copied into the Address, C1, Pri_{C1} and Procs fields of the new ST entry. In this way, the protocol enables transactions to build a list of stalled transactions so that the highest priority ones are intended to occupy the first positions. It is worth noting, though, that transactions deal with imprecise information because transactions only know the first two positions of the list in the best case (the Procs field is unordered). Nevertheless, our experimental results revealed that two ordered elements at the head of the list approach the ideal case with exact information. Finally, since highest-priority transactions are supposed to populate the heads of the stalled transactions lists, they will not abort in case of conflicts with other transactions, thus guaranteeing forward progress. Figure 3 shows and example in which DS avoids aborts and network traffic.

Figure 3 shows the initial state of five transactions (T0–T4) with the STs of transactions T1 and T2 at the top. Transactions T0 through T4 start their execution at time t0 through t4, respectively—RX means that the transaction reads cache block address X, while WY means that the transaction writes cache block address Y. In Fig. 3b, T1 sends a NACK message from T0 since address D was previously written by T1. Then, T1 adds a new entry to its ST with Address D, and 0 and t0 as C1 and Pri_{C1}, respectively, and sets the bit of the Procs field corresponding to T0. In the meantime, T0 stalls so that no further retries are sent to T1 until the UNSTALL message is received. A similar scenario due to conflicting access to address A takes place between T2 and T3. Next, in Fig. 3c, T1 serializes T2, adds a new entry to its ST with Address C, and 2 and t1 as C1 and Pri_{C1}, respectively, and sets the bit of the Procs field corresponding

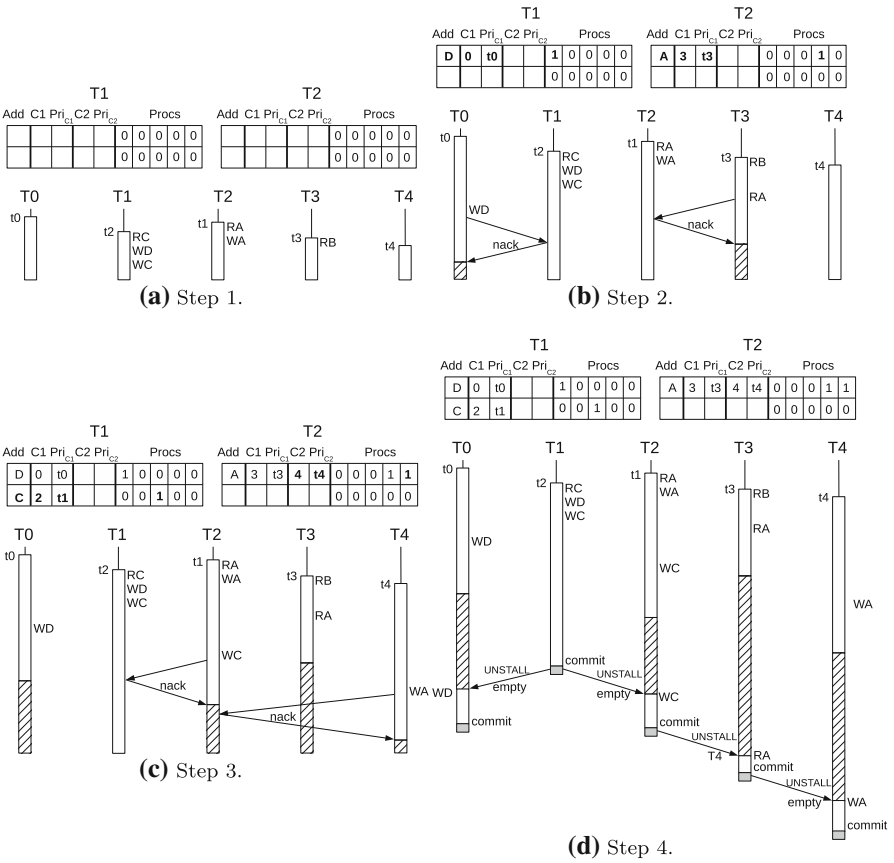

Fig. 3 Dynamic serialization example

to T2. While stalled, T2 serializes T4 due to a conflicting access to address A. T2 updates C2 and Pri_{C2} with 4 and t4, respectively, because T4 has lower priority than T3. At this point, all transactions other than T1 are stalled. In Fig. 3d, T1 commits and scans its ST. Therefore, it sends an UNSTALL message to T0 and T2. The Procs fields of both messages are empty, since T1 did not serialize any other transactions on addresses D and C. When T2 commits, it sends an UNSTALL message to T3 whose Procs fields identifies T4 as an stalled transaction. Finally, T3 commits and unstalls T4 which also commits. Note that T3 did not send any NACK message to T4, but T3 inherited T4 from T2.

Finally, SDS also copes with deadlock detection. LogTM-SE implements a conservative deadlock detection mechanism based on timestamps and a "possible cycle bit" that it sets whenever a NACK message is sent to an older transaction. In this way, if a transaction receives a NACK message from an older transaction and the "possible cycle bit" is set, the transaction is enforced to abort. With SDS, a deadlock could go unnoticed because NACKed transactions get stalled. To prevent this situation from happening, SDS adds a second *"possible cycle bit"* that is set whenever a NACK

message from an older transaction is received. In this case, if the transaction is about to send a NACK message to an older transaction and the second *"possible cycle bit"* is set, the transaction must abort as well.

3 Evaluation environment

In this section, we describe the evaluation environment used in this work. We start by giving the details about how the *Eager–Eager* HTM systems considered in this work have been implemented. 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 experiments.

3.1 System settings

We use a full-system execution-driven simulator based on the Wisconsin GEMS toolset [18], in conjunction with Wind River Simics. 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 unmodified Solaris 10. We perform our experiments assuming a tiled CMP system, as described in Table 1. Particularly, we simulate a 16-core configuration with private L1 I&D caches and a shared, multibanked L2 cache consisting of 16 banks of 512 kB each. We have left another core aside to run the operating system (OS) in a isolated way from the application threads to avoid intrusions from the same one in benchmarks' execution and getting uncorrupted statistics. The OS still takes the control of the benchmarks execution when needed (i.e., during an exception). 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 4×4 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 [19]. In particular, we have extended the network simulator provided by GEMS with the consumption model included in Orion. Table 2 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) were measured based on the consumption model of CACTI 5.3 rev 174 [20]. 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.

Table 1 System parameters

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	2 cycle
Link bandwidth	16 Bytes/cycle

Table 2 Parameters of Orion 2.0

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

The Ruby module contains an implementation of LogTM-SE, an *Eager–Eager* system that uses signatures for transactional book-keeping. We have extended the MESI cache coherence protocol originally used by LogTM-SE to support the SDS process described in Sect. 2. Our serialization table uses six entries, more than enough to avoid any overflow situation with the STAMP benchmarks. Finally, the *undo log* used in LogTM-SE is a data structure mapped in virtual memory and thus, its size is not limited by any hardware structure. We assume perfect signatures to check for conflicts.

3.2 Benchmarks settings

For the evaluation, we use seven out of eight transactional benchmarks extracted from the STAMP suite version 0.9.10 [15]. These applications allow to stress a TM system in several ways. To show a wide range of cases, we evaluate all kinds of benchmarks that present low/moderate/high contention and/or large read and write set sizes and. The application *Bayes* was excluded, since it exhibits unpredictable behavior and high variability in its execution times [21]. Table 3 describes the benchmarks and the values of the input parameters used in this work.

4 Evaluation

In this section, we present the results obtained for an *Eager–Eager* system (particularly, LogTM-SE) extended with the different flavors of the SDS idea proposed in this work.

Table 3 Workloads and inputs

Benchmark	Input
Genome	-g512 -s32 -n32768
Intruder	-a10 -l16 -n4096 -s1
Kmeans-high	-m15 -n15 -t0.05 -i random-n2048-d16-c16
Labyrinth	-i random-x32-y32-z3-n96
Ssca2	-s14 -i1.0 -u1.0 -l9 -p9
Vacation-high	-n4 -q60 -u90 -r1048576 -t4096
Yada	-a10 -i ttimeu10000.2

In particular, we consider the simplest implementation of SDS, in which transactions are serialized every time a conflict is observed (LogTM-SE_DS), and the two less aggressive implementations of SDS described in Sects. 2.2 and 2.3 (ABORT_SDS and NACK_SDS, respectively). In LogTM_DS, the serialized mode is enabled from the beginning of the execution. Remember that it does not affect the normal execution until a conflict takes place. In this case, one of the transactions will be serialized[2]. For ABORT_SDS and NACK_SDS, we have evaluated three counter sizes in each case. ABORT_SDS_*x* and NACK_SDS_*x* refer to both flavors of SDS, respectively, where *x* represents the size of the serialized counter in bits. For example, results for ABORT_SDS with a serialized counter of three bits are labeled as ABORT_SDS_3 in the following graphs. In this way, ABORT_SDS_3 will allow up to seven retries of the same transaction before entering into serialized mode. The same nomenclature is used for NACK_SDS. Note that LogTM_DS is equivalent to NACK_SDS_0.

We perform a comparison between these HTM systems in terms of execution time. Next, we study the energy consumption of each system when executing the transactional workloads. Finally, we also compare the amount of network traffic that has been generated. Note that some graphs include error bars. They represent several executions (20 times) of the same benchmark, but with some random variations in the memory latency (within 2 %) that generate different data access patterns.

4.1 Execution time results

For the seven transactional benchmarks pointed out in Sect. 3, Fig. 4 shows the execution times that are obtained for LogTM-SE (from left to right, last bar) and the three implementations of SDS considered in this work: LogTM-SE_DS (first bar), ABORT_SDS (next three bars) and NACK_SDS (next three bars). In all cases, execution times have been normalized with respect to those obtained with the LogTM-SE system. Moreover, to have a clear understanding of the results, Fig. 4 divides the execution times into the following categories: *Abort* (time spent during aborts), *Backoff* (explained next), Barrier (time spent in barriers), *Commit* (1 cycle), *Non_xact* (time spent in non-transactional execution), *stall_active* (active time waiting until another transaction ends: in this case the access to the address is continuously retried),

[2] Note that LogTM_DS would be equivalent to NACK_SDS_0.

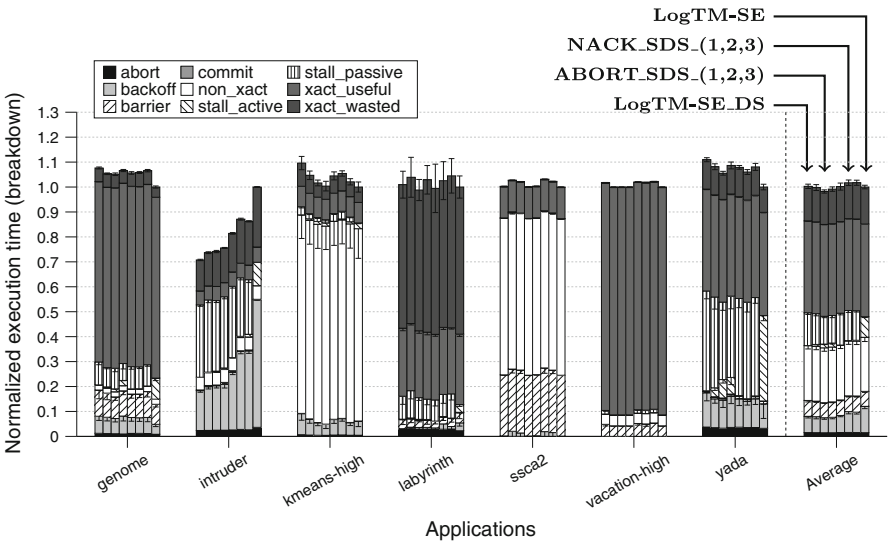

Fig. 4 Breakdown of the execution times

stall_passive (time waiting until another transaction ends: in this case the core is in a low power mode), *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 amount of wasted time (*Xact_wasted*) drastically increases in some cases as the number of aborts grows. All systems employ a hardware exponential back-off mechanism that emulates the original software method, to reduce energy consumption due to interferences from that kind of implementation.

As it can be derived from Fig. 4, none of the implementations of SDS outperforms the others for all the applications. As expected, serialization of transactions conveys a small overhead in execution time for the majority of the benchmarks. This overhead goes from 0 % (in the case of *ssca2*) to 11 % (for *yada*). Remember that the main goal of SDS is to reduce energy consumption by serializing conflicting transactions without increasing significantly the execution time. Nevertheless, for applications exhibiting significant contention among the transactions (such as *intruder*), very important reductions in execution time (about 30 %) can be obtained when contention is reduced by means of serializing the execution of transactions.

Regarding the two most elaborated SDS implementations proposed in this work (ABORT_SDS and NACK_SDS), it is worth noting that both of them are able to reduce the performance degradation introduced by the simplest and aggressive SDS implementation (LogTM-SE_DS), which turns into serialized mode whenever the first conflict is detected. On the contrary, several aborts and conflicts (for ABORT_SDS and NACK_SDS, respectively) must be experienced before transactions are started

to be serialized. Obviously, these less aggressive implementations take more time to react in high contention scenarios (such as *intruder*), where LogTM-SE_DS is the winner. Among all the configurations plotted in Fig. 4, ABORT_SDS_2 achieves the best results in average execution time, even improving LogTM-SE by 2 %.

Another important observation is that in LogTM-SE_DS, ABORT_SDS and NACK_SDS the predominant stall phase is the *stall_passive*. Whereas, all stalls fall into the *stall_active* category in LogTM-SE. Remember that *stall_passive* means that the transactions enters into a low-power mode, where no new messages are generated and dumped into the network and of course there are no new cache accesses from these transactions allowing to save energy and network traffic.

The discussion below highlights important observations and presents insights gained from a detailed analysis of the interaction between the three SDS implementations and the behavior of individual workloads.

Genome This workload exhibits three execution phases where the second one dominates the overall performance. These accesses are predominantly non-contended and a *Eager*scheme such as LogTM-SE performs really well. With this premises, the serialization is not justified because the amount of saved work that it would entail is quite small compared with the time to wake up in chain the serialized transactions. Nevertheless, the overhead that the simplest SDS implementation (LogTM-SE_DS) introduces is noticeable (7.5 %). ABORT_SDS and NACK_SDS reduce the probability of serialization, resulting in lower overhead compared with LogTM-SE (just 5 % for ABORT_SDS_3, ABORT_SDS_2, NACK_SDS_1 and NACK_SDS_2).

Intruder This workload shows high contention and transactions acquire exclusive ownership to data before they are guaranteed to commit. Furthermore, it presents a high probability of conflicts and leads to large stalls and chains of dependencies and aborts. This is a pathological behavior exhibited by *Eager–Eager* systems. This kind of applications typically presents a poor performance when compared with lazy approaches [13]. Obviously in this scenario, reducing the likelihood of conflict favors forward progress. LogTM-SE_DS achieves the most significant reduction in execution time (29 %), closely followed by ABORT_SDS_1 and ABORT_SDS_2 (26 %). We have observed that the number of aborts is reduced from 153,000 to 95,000 in LogTM-SE_DS. In addition, we have found that in these applications the number of conflicts before a *possible cycle* is detected, and consequently, an abort is raised, is very small. This reduces the chance of entering into the serialized mode in NACK_SDS and it obtains the worst results, especially for the larger size counters (remember that the counter is reset each time a transaction aborts).

Kmeans-high This workload is mainly non-transactional (white bars in Fig. 4), but it exhibits moderate levels of contention in its transactional part. Despite the fact that its transactional part is about 20 %, LogTM-SE_DS increases the execution time by 10 %, which represents 50 % of its transactional execution time. An *Eager–Eager* approach performs well with this level of contention, and serialization is counterproductive from the execution time point of view. On the contrary, the other more elaborated SDS schemes are able to completely hide this overhead (especially, ABORT_SDS_3) by precluding the serialization of transactions in most cases. In addition, they translate the small stall_active phase exhibited by LogTM-SE into the energy-efficient stall_pasive phase, increasing energy efficiency and reducing network traffic levels.

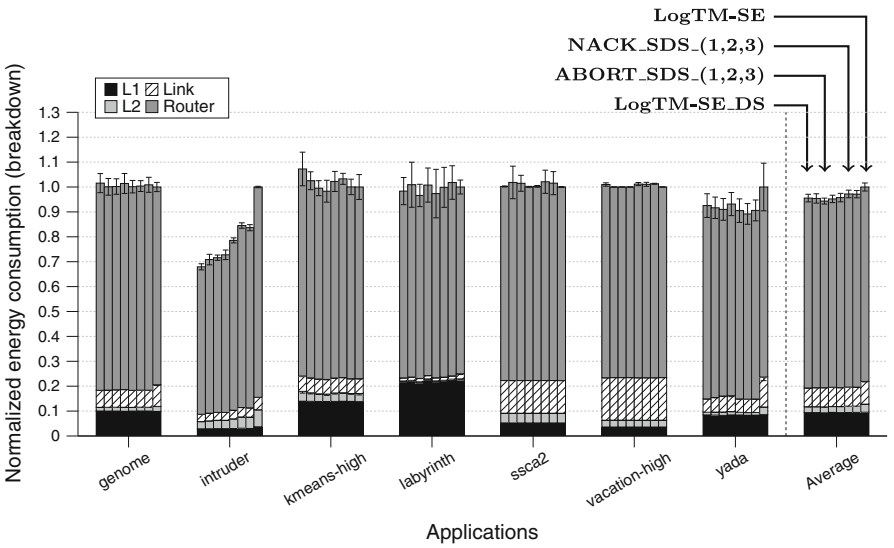

Fig. 5 Breakdown of energy consumption

Labyrinth It is characterized by long transactional time, large write sets and medium contention. Furthermore, its behavior is not always the same. *Labyrinth* tries to find a path in a maze (three-dimensional matrix of $32 \times 32 \times 3$) following a variant of Lee's algorithm. The calculation of the path and its addition to the global maze is performed in a single transaction. First the global matrix is read locally by the transaction. Next, a path is worked out with the local copy, and finally the path is updated to the global maze if no conflicts happen. In order to scale performance with the number of the on-chip cores, *labyrinth* makes use of early release, that is, the isolation over the set of read addresses is released after doing the copy with the aim of reducing conflicts (hardware support is needed). Transactional times are so long that if a conflict occurs at the beginning of a transaction, it will take a lot of time to be resolved. Results generated by this workload present a high variability, because it depends significantly on the interleaving of threads. Its most important transaction presents large write sets (more than 200 addresses) and a long execution time. An abort is extremely costly and there is no clear winner in this case.

Ssca2/Vacation-high These benchmarks do not have real conflicts, and thus *Eager–Eager* approaches perform and scale really well. There are no significant differences between normal execution and those with SDS.

Yada It has a large working set and exhibits high contention. Aborts are common in these workloads and penalize the overall execution and the *xact_wasted* proportion represents a significant fraction of the execution time. The stall_active fraction of the execution is also very significant. Serialization in this scenario does not favor execution time because of the extra overhead in waking up the serialized transaction, but it manages to translate the stall_active fraction of the execution into the energy-efficient stall_pasive. As we will see in Sects. 4.2 and 4.3, this results into important savings in terms of energy and network traffic.

4.2 Energy consumption results

Figure 5 plots dynamic energy consumption for LogTM-SE, LogTM-SE_DS, NACK_SDS and ABORT_SDS. Again, several sizes for the saturating counter are considered in both NACK_SDS and ABORT_SDS. As before, results have been normalized with respect to LogTM-SE. Additionally, we split the energy consumed in each case into the following categories: energy spent accessing the L1 and L2 caches (*L1* and *L2*, respectively), and energy spent in the network routers and links (*Router* and *Link*, respectively). The amount of energy spent in the caches is related to the number of accesses to each one of them, and thus with the number of aborts. More aborts means retrying more accesses to the caches. Link energy is due to the average link utilization or the number of flits that cross every link. The energy model for the router in Orion 2 exhibits a sublinear growth with respect to the network average load. Furthermore, energy consumed in routers is related with the execution time too.

For most applications, the three SDS implementations improve or at least keep the same results than the base case when energy consumption is considered. Only for *kmeans-high* LogTM-SE_DS increases energy consumption by 7.5 % when compared to LogTM-SE as a consequence of the degradation in execution time it introduces. Fortunately, the most sophisticated SDS implementations do not degrade energy consumption. Among all SDS schemes considered in this work, ABORT_SDS_2 represents the most balanced alternative in terms of both execution time and energy consumption. On the one hand, it does not degrade energy consumption in *genome*, *kmeans-high*, *ssca2* and *vacation-high*. On the other hand, it reduces energy consumption by 4, 9 and 29.5 % in *labyrinth*, *yada* and *intruder* respectively, resulting in average reductions of 6 %.

The advantages in terms of energy consumption that SDS brings comes from two sources: the reduction in the number of aborts and the reduction in the number of network messages during conflicts. The latter mainly affects the energy consumed in the links of the interconnect, whilst the former also has impact on the energy consumed in the routers and L2 caches. By reducing the number of aborts, SDS implementations are able to cut down the amount of wasted work, which results in energy savings in the L2 caches and network routers and links. On the other hand, by stopping transactions when conflicts arise and, when solved, letting them go, SDS schemes do not generate any network traffic while a transaction is stopped (*stall_passive*), which translates into important energy savings in the the links of the interconnection network. Finally, less network utilization and lower execution times mean that less energy is spent in the routers.

In general, results in terms of energy consumption follow closely those reported in terms of execution time. The only exception is *yada*, for which the SDS schemes introduced some degradation in terms of execution time. However, when energy consumption is considered, we observe that all SDS implementations bring significant savings, as a consequence of important reductions in the energy consumed in the links of the interconnect, which also translates into less network traffic (see Sect. 4.3).

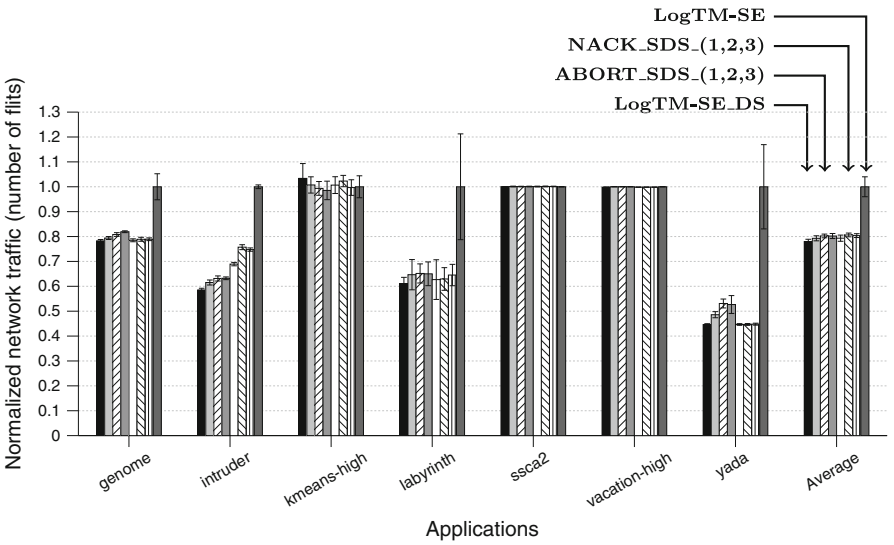

Fig. 6 Normalized network traffic

4.3 Network traffic results

Network traffic measured as number of flits for LogTM-SE, LogTM-SE_DS, NACK_SDS and ABORT_SDS is shown in Fig. 6. As before, results have been normalized with respect to LogTM-SE. In general, LogTM-SE_DS generates less network traffic than LogTM-SE (approximately 22 % less, on average). The reason can be found in the high number of retries that are needed in LogTM-SE in case of conflicts. The difference in network traffic is considerable in most cases except for *kmeans*, *vacation* and *ssca2*. As already explained, transactions in these benchmarks either barely conflict or are very short. The rest of the benchmarks are characterized by high contention and/or by the large size of their transactions [15]. During the *stall* phase in LogTM-SE (stall_active), intensive usage of the 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 has to abort. LogTM-SE_DS not only saves that wasted work, but also avoids conflicts that can lead to an abort by stalling conflicting transactions. The other two SDS schemes try to imitate the behavior of the simplest approach and they achieve 20 % of total reduction in network traffic. In *genome* our proposals reduce network traffic up to 20 %, while the energy consumption is approximately the same than that of the base case. This is because the execution time is higher when transactions are serialized. This trade-off between network traffic and execution time is found in all the applications when energy consumption is considered. Special attention should be paid to *labyrinth* and *yada*. The former shows average reductions of up to 35 % with the SDS schemes while barely affecting to the results in terms of energy consumption. This is because what really dominates both execution time and energy consumption in this case is the cost of the aborts so that the reductions in traffic do not have a big

impact. On the other hand, in *yada*, serialization brings reductions in network traffic ranging between 47 and 55 % when compared with LogTM-SE. In this case, these reductions in network traffic brings important savings in energy consumption even although SDS hurts execution time. Also note that different to *intruder*, NACK_SDS beats ABORT_SDS and reaches the traffic levels of the aggressive LogTM-SE_DS implementation. For *yada*, we have seen that the average number of conflicts before an abort is detected is higher, which makes NACK_SDS start serializing transactions earlier.

5 Related work

Transactional memory has become a promising parallel paradigm alternative to lock synchronization [22]. While locks suffer from deadlocks, priority inversions and convoying, TM trusts in executing transactions in parallel. If any conflict happens, the transaction changes are become visible to the whole system. On the contrary, changes are discarded. TM can be implemented in either software [23–27], hardware [11,14,28,29], or as a combination of both [7,12,30]. Our focus is on HTM.

Nowadays, the implications of energy consumption are a first-class consideration, requiring trade-offs against performance. This is true not only for embedded systems [8,9] (such as mobile devices), but also for server and even desktop systems [10,31]. TM literature has traditionally focused on improving performance, simplicity or even flexibility. For example, [32–34] are several recent proposals aimed at increasing concurrency between transactions to reduce execution times. In [35], Titos-Gil et al. show how effective store management can improve the performance of eager HTM systems. Ceze et al. try to simplify the conceptual and implementation complexity of HTM in [11]. FlexTM [36] is a high-performance TM framework that allows software to determine when (eagerly, lazily, or in a mixed fashion) and how to manage conflicts, while employing hardware to manage transactional state and to track conflicts.

Regarding energy consumption in the context of TM, Klein et al. [37] performed a study comparing STM and conventional lock-based systems, and also proposed new mechanisms to improve energy efficiency of STM. For HTM, Moreshet et al. [17] performed an early comparison in terms of energy consumption and performance between the lock approach and TM considering only the energy spent in the memory structures. In this previous work, Moreshet proposed a naive static serialization mechanism to improve energy efficiency in which two conflicting transactions are re-issued in serialized mode, preventing parallel speculation in other transactions. Subsequently, Sanyal et al. [38] applied the well-known clock gating technique in the context of HTM to save energy. In particular, they propose a novel protocol, which gates processors dynamically on each abort and un-gates them depending on the number of aborts suffered and the state of the conflicting transactions. Ferri et al. [8] present a simple and energy-efficient TM design for embedded architectures, at the cost of performance. One of their proposals is to perform a static serialization of transactions. If one transaction reaches this mode, the rest of the cores must stop their execution until the transaction commits. This reduction in speculation and performance suits well with embedded systems, but not with general purpose ones. In [13], two well-known HTM systems

(namely, *Eager–Eager* LogTM-SE system [14] and *Lazy–Lazy* Scalable TCC system [39,40]) are compared in terms of energy consumption. The results of this study show that LogTM-SE and other eager approaches present a significant potential for improvement in energy consumption. On the other hand, in [41], Cristal et al. show a case use of how efficient the HTM support can help techniques aimed at increasing energy efficiency in current multicores. In particular, it is proposed to use HTM support for rolling back the effects of wrong executions caused by the reduction of the supply voltage of cores. Reducing the supply voltage improves energy efficiency, but at the same time increases the likelihood for wrong executions of programs.

In this work, we have presented and evaluated SDS, a technique that can be implemented on top of any *Eager–Eager* HTM system. SDS is aimed at improving energy efficiency in *Eager–Eager* HTM systems and reducing the pressure on the interconnection network, but without hurting execution time. Our implementations of SDS have been based on LogTM-SE [14]. Different to [8,17], serialization in SDS is only performed when conflicts happen and no extra cost in terms of increased number of aborts is incurred to raise this mode. As another difference, SDS works at cache line level, therefore it is a fine-grained serialization detection mechanism that favors speculation as much as possible.

6 Conclusions

In this work, we present SDS, a new technique that improves energy consumption in HTM systems that implement eager conflict management, such as LogTM-SE. SDS is aimed at dynamically serializing transactions in high-contention scenarios (i.e., several transactions fighting for the same data at the same time). In these cases, previous works [13] have shown that the energy efficiency of *Eager–Eager* systems collapses. This is because conflicts are managed either by re-trying the memory access that caused the conflict until it disappears or by aborting one or more transactions (depending on the interactions among the write sets of the transactions involved in the conflict), which results in a significant amount of energy being wasted. On the contrary, SDS tries to detect this kind of situation and dynamically serializes only the involved transactions.

Selective Dynamic Serialization is a refinement of DS originally presented in [16] that aims to achieve the benefits of DS (i.e., important savings in terms of energy consumption and network traffic), but without incurring the overheads in execution time that DS has for some applications. To do so, SDS makes use of saturating counters that are incremented when conflicts are detected (NACK_SDS) or when aborts raise (ABORT_SDS). Depending on the values of these counters, transactions enter the serialized mode or not. It is important to note that DS is a particular case of SDS, concretely NACK_SDS_0.

We have implemented all these flavors of SDS on top of the GEMS full-system simulator, and we have compared them against the original LogTM-SE *Eager–Eager* HTM system. Results in terms of execution time, energy consumption and network traffic have been presented. In general, SDS obtains average reductions of 6 % in energy consumption (up to 42 % in high-contention scenarios) at no performance cost. Furthermosre, SDS is able to save about 22 % of the network traffic levels generated

by LogTM-SE. This is due to SDS precludes transactions from continuously retrying conflicting memory accesses. In addition, SDS entails minimal cost in terms of extra hardware. For all these reasons, we can conclude that SDS constitutes an interesting optimization for future HTM systems.

Acknowledgments This work was supported by the Spanish MINECO, as well as European Commission FEDER funds, under grant "TIN2012-38341-C04-03". 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. Borkar S (2007) Thousand core chips: a technology perspective. In: DAC-44
2. Diestelhorst S, Pohlack M, Hohmuth M, Christie D, Chung J-W, Yen L (2010) Implementing AMD's advanced synchronization facility in an out-of-order x86 core. In: Transact-05
3. Dice D, Lev Y, Moir M, Nussbaum D (2009) Early experience with a commercial hardware transactional memory implementation. In: ASPLOS-14
4. The IBM Blue Gene Team (2011) The Blue Gene/Q compute chip. In: Hot Chips 23
5. Kanter D (2012) Analysis of Haswell's transactional memory. In: Real World Technologies (02–15-2012)
6. Herlihy M, Eliot J, Moss B (1993) Transactional memory: architectural support for lock-free data structures. In: ISCA-20
7. Harris T, Cristal A, Unsal OS, Ayguad E, Gagliardi F, Smith B, Valero M (2007) Transactional memory: an overview. IEEE Micro 27(3):8–29
8. Ferri C, Wood S, Moreshet T, Bahar RI, Herlihy M (2010) Embedded-TM: energy and complexity-effective hardware transactional memory for embedded multicore systems. J Parallel Distrib Comput (JPDC) 70(10):1042–1052
9. Ferri C, Wood S, Moreshet T, Bahar RI, Herlihy M (2010) Energy and throughput efficient transactional memory for embedded multicore systems. In: HiPEAC, pp 50–65
10. Barroso LA, Hölzle U (2007) The case for energy-proportional computing. Computer 40(12):33–37
11. Ceze L, Tuck J, Torrellas J, Cascaval C (2006) Bulk disambiguation of speculative threads in multiprocessors. In: ISCA-33
12. Shriraman A, Dwarkadas S, Scott ML (2008) Flexible decoupled transactional memory support. In: ISCA-35
13. Gaona-Ramírez E, Titos-Gil JR, Fernández J, Acacio ME (2013) On the design of energy-efficient hardware transactional memory systems. Concurr Comput Pract Exp 25(6):862–880
14. Yen L, Bobba J, Marty MR, Moore KE, Volos H, Hill MD, Swift MM, Wood DA (2007) LogTM-SE: decoupling hardware transactional memory from caches. In: HPCA-13
15. Minh CC, Chung J, Kozyrakis C, Olukotun K (2008) STAMP: stanford transactional applications for multi-processing. In: IISWC-4
16. Gaona-Ramírez E, Titos-Gil JR, Acacio ME, Fernández J (2012) Dynamic serialization: Improving energy consumption in eager–eager hardware transactional memory systems. In: PDP-20, pp 221–228
17. Moreshet T, Bahar RI, Herlihy M (2006) Energy-aware microprocessor synchronization: transactional memory vs. locks. In: Workshop on memory performance, Issues
18. Martin MMK, Sorin DJ, Beckmann BM, Marty MR, Xu M, Alameldeen AR, Moore KE, Hill MD, Wood DA (2005) Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. SIGARCH CAN 33(4):92–99
19. Kahng AB, Li B, Peh L-S, Samadi K (2009) ORION 2.0: a fast and accurate NoC power and area model for early-stage design space exploration. In: DATE-13
20. Thoziyoor S, Muralimanohar N, Ahn JH, Jouppi NP (2008) Cacti 5.1. Technical Report HPL-2008–20. HP Laboratories, Palo Alto, CA
21. Dragojevic A, Guerraoui R (2010) Predicting the scalability of an STM. In: Transact-05
22. Harris T, Larus J, Rajwar R (2010) Transactional memory, 2nd edn. Morgan & Claypool, San Rafael
23. Dice D, Shalev O, Shavit N (2006) Transactional locking II. In: DISC-20
24. Fraser K, Harris TL (2007) Concurrent programming without locks. ACM TOCS 25(2):1–61

25. Marathe VJ, Scherer-III WN, Scott ML (2005) Adaptive software transactional memory. In: DISC-19
26. Herlihy M, Luchangco V, Moir M, Scherer-III WN (2003) Software transactional memory for dynamic-sized data structures. In: PODC-22
27. Saha B, Adl-tabatabai A, Hudson RL, Minh CC, Hertzberg B (2006) McRT-STM: a high performance software transactional memory system for a multi-core runtime. In: PPoPP-11
28. Tomic S, Perfumo C, Kulkarni CE, Armejach A, Cristal A, Unsal OS, Harris T, Valero M (2009) EazyHTM: eager-lazy hardware transactional memory. In: MICRO-42
29. Rajwar R, Herlihy M, Lai KK (2005) Virtualizing transactional memory. In: ISCA-32
30. Damron P, Fedorova A, Lev Y, Luchangco V, Moir M, Nussbaum D (2006) Hybrid transactional memory. In: ASPLOS-XII, pp 336–346
31. Flores A, Aragón JL, Acacio ME (2008) An energy consumption characterization of on-chip interconnection networks for tiled cmp architectures. J Supercomput 45(3):341–364
32. Lupon M, Magklis G, González A (2010) A dynamically adaptable hardware transactional memory. In: MICRO-43, pp 27–38
33. Negi A, Titos-Gil JR, Acacio ME, García JM, Stenström P (2011) ZEBRA: a data-centric, hybrid-policy hardware transactional memory design. In: ICS-25
34. Negi A, Titos-Gil JR, Acacio ME, García JM, Stenström P (2012) PI-TM: pessimistic invalidation for scalable lazy hardware transactional memory. In: HPCA-18, pp 141–152
35. Titos-Gil JR, Negi A, Acacio ME, García JM, Stenström P (2013) Eager beats lazy: improving store management in eager hardware transactional memory. IEEE Trans Parallel Distrib Syst 24(11):2192–2201
36. Shriraman A, Dwarkadas S, Scott ML (2010) Implementation tradeoffs in the design of flexible transactional memory support. J Parallel Distrib Comput 70(10):1068–1084
37. Klein F, Baldassin A, Araujo G, Centoducatte P, Azevedo R (2009) On the energy-efficiency of software transactional memory. In: SBCCI-22
38. Sanyal S, Roy S, Cristal A, Unsal O, Valero M (2009) Clock gate on abort: towards energy-efficient hardware transactional memory. In: HPPAC-2009
39. Chafi H, Casper J, Carlstrom BD, McDonald A, Minh CC, Baek W, Kozyrakis C, Olukotun K (2007) A scalable, non-blocking approach to transactional memory. In: HPCA-13
40. Pugsley SH, Awasthi M, Madan N, Muralimanohar N, Balasubramonian R (2008) Scalable and reliable communication for hardware transactional memory. In: PACT-17
41. Cristal A, Unsal O, Yalcin G, Fetzer C, Wamhoff J-T, Felber P, Harmanci D (2013) A. Sobe, Leveraging transactional memory for energy-efficient computing below safe operation margin. In: TRANSACT-2013