2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing

Dynamic Serialization: Improving Energy Consumption in Eager-Eager Hardware Transactional Memory Systems

Epifanio Gaona[1], Rubén Titos-Gil[1], Manuel E. Acacio[1] and Juan Fernández[2*]

[1]Universidad de Murcia, Spain. Email: {fanios.gr,rtitos,meacacio}@ditec.um.es
[2]Intel Barcelona Research Center, Spain. E-mail: juan.fernandez@intel.com

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 Dynamic Serialization (DS) as a new technique to improve energy consumption without degrading performance in applications with conflicting transactions. Our proposal, which is implemented on top of a hardware transactional memory system with an eager conflict management policy, detects and serializes conflicting transactions dynamically. Particularly, 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. This brings important benefits in terms of energy consumption due to the reduction in the amount of wasted work that DS implies. Results for a 16-core CMP show that Dynamic Serialization obtains reductions of 10% on average in energy consumption (more than 20% in high contention scenarios) without affecting, on average, execution time.

1 Introduction and motivation

Transactional Memory (TM) is currently considered as a promising parallel programming paradigm. TM borrows the concept of transaction from the database world and brings it into the shared-memory programming model [7]. A TM system can be implemented in either software, hardware, or as a combination of both. The common denominator in all implementations is that transactions are speculatively executed which hides from programmers the main pathologies associated with locks, 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.

Hardware transactional memory systems (HTM) 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 for each transactional load and store.

On the other hand, in the design of new systems, the implications of energy consumption are increasingly important, requiring tradeoffs against performance. This is true not only for embedded systems [4] (such as mobile devices) but also for server and even desktop systems [1]. HTM literature has mostly focused on improving performance, simplicity [2] or even flexibility [17]. A recent study [5] 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 [5]. Unfortunately, these scenarios may not be rare in some future applications.

In this work we present Dynamic Serialization (DS henceforth), a new technique aimed at reducing energy consumption in HTM systems implementing eager conflict management. Instead of continuously re-trying a memory access that caused a conflict with another active transaction (as done in *Eager-Eager* systems [18]), 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 an-

*This work was done while Juan Fernández was a member of the Computer Engineering Department of the University of Murcia.

978-0-7695-4633-9/12 $26.00 © 2012 IEEE
DOI 10.1109/PDP.2012.76

other transaction conflicts with it, but a priori wasted work would have already been avoided. In this way, an *Eager* system with DS 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 DS would also facilitate forward progress. As an example, typical critical sections in transactional applications include modifying an iterator over a list. The sequence of addresses would be as follows:

Read A — Read B — Read C — Write C

This sequence in a high-contention scenario (i.e. several transactions trying to execute the sequence at the same time) 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 and aborting other transactions (which results in wasted work). The competition will start again after restarting transactions' execution. This behavior is produced in a cycle. In this case forward progress is compromised. Situations of this kind are found in *intruder*, for example, a benchmark of the STAMP suite [13]. With DS, 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.

Serialization has already been considered in two previous works ([14], [4]). In particular, Moreshet [14] 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 [4] consists of stopping non-serialized cores until the commit of the serialized one, with the subsequent performance penalization. In this way, and compared with these two proposals, DS 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. Second, DS favors parallel speculation as much as possible since serialization is performed at lower level (cache line).

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

2 Dynamic Serialization (DS)

Dynamic Serialization (DS) refines the conflict management mechanism of *Eager-Eager* HTM systems in order to reduce the wasted energy due to aborted and even stalled transactions. Our implementation is based on LogTM-SE [18] but it is extensible to any other *Eager-Eager* HTM system that employs the cache-coherent protocol to detect conflicts on the fly. DS does not change the default behavior of LogTM-SE in absence of conflicts. However, unlike other serialization mechanisms [4, 14], DS serializes transactions when a conflict arises and not just after there have already been some aborts. DS operates at cache line level so that transactions run smoothly until a conflict in a particular cache block is detected. In such a situation, DS 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 transactions. In this way, DS not only minimizes the energy consumption of the stalled transactions but also lessens the number of aborted transactions thus reducing the wasted energy due to them. To do so, DS requires the hardware support detailed in Section 2.1 to implement the protocol exemplified in Section 2.2.

2.1 Architecture

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 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.

On the contrary, DS avoids this persistent retrying process by stalling the offending transaction after receiving a NACK in a low-power state. In this 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 from other transactions. Moreover, it must keep track of all NACKed transactions in order 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 ad-

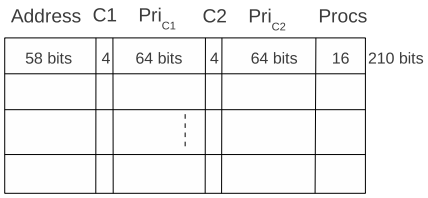


Figure 1. Serialization Table.

dress that was NACKed by the transaction. Experiments conducted in Section 4 have shown that only 6 entries are enough to prevent overflow situations. Figure 1 shows the ST structure whose fields are the following:

– *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.

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 Protocol

Our implementation of the DS protocol is based on the MESI cache coherence protocol, although DS could be built atop any other cache coherence protocol flavor with a similar behavior such as MOESI. DS does not modify the cache coherence protocol, it only adds a single control message called UNSTALL with the following fields (Figure 2):

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

Figure 2. UNSTALL message format.

– *Address*: cache block address that was NACKed by this transaction. Address field in the ST.

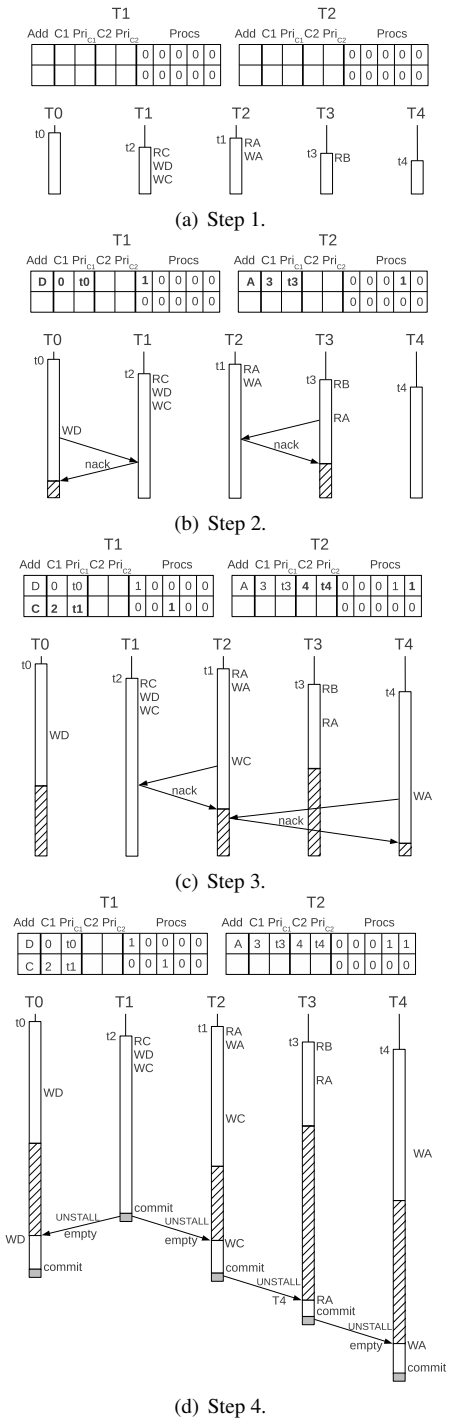
– *C2*: C2 field in the ST.

– *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 from 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 Section 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(a) 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 Figure 3(b), 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



(a) Step 1.

(b) Step 2.

(c) Step 3.

(d) Step 4.

Figure 3. Dynamic Serialization example.

T2 and T3. Next, in Figure 3(c), 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 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 Figure 3(d), 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.

To conclude, the DS protocol also copes with deadlock detection. LogTM-SE implements a conservative deadlock detection mechanism based on timestamps and a "possible cycle bit" that it set 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 DS, a deadlock could go unnoticed because NACKed transactions get stalled. To prevent this situation from happening, DS 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 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.

3.1 System Settings

We use a full-system execution-driven simulation based on the Wisconsin GEMS toolset [12], in conjunction with Virtutech 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 2. Particularly, we simulate a 16-core configuration with private L1 I&D caches and a shared, multibanked L2 cache consisting of 16 banks of 512KB each. We have left another core aside to run the operating system (OS) in a isolated way from the application threads in order to avoid intrusions from the same one in benchmarks' execution and getting uncorrupted statistics. The OS still takes the control of the benchmarks execution

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

Table 1. Parameters of Orion 2.0.

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 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 [10]. In particular, we have extended the network simulator provided by GEMS with the consumption model included in Orion. Table 1 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 [9]. 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. We have extended the MESI cache coherence protocol originally used by LogTM-SE in order to support the Dynamic Serialization (DS) process described in section 2. Our Serialization Table uses 6 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 five transactional benchmarks extracted from the STAMP suite version 0.9.10 [13]. These applications allow to stress a TM system in several ways. To show a wide range of cases, we evaluate the benchmarks

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 2. System Parameters.

Benchmark	Input
Bayes	-v32 -r4096 -n2 -p20 -i2 -e2
Intruder	-a10 -l16 -n4096 -s1 -i random-n16384-d24-c16
Labyrinth	-i random-x32-y32-z3-n96
Vacation	-n4 -q60 -u90 -r1048576 -t4096
Yada	-a10 -i ttimeu10000.2

Table 3. Workloads and inputs.

that present moderate/high contention and/or large read and write set sizes and, at the same time, their transactional execution time represents more than 70% of the total execution time. 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) with the Dynamic Serialization technique proposed in this work (LogTM-SE_DS from here on) and the original LogTM-SE system. We will perform a comparison in terms of execution time, energy consumption and network traffic.

4.1 Execution time results

For the five transactional benchmarks pointed out in Section 3, Figure 4 shows the execution times that are obtained for both LogTM-SE and LogTM-SE_DS. In all cases, execution times have been normalized with respect to those obtained with the LogTM-SE system. Moreover, to have clear understanding of the results Figure 4 divides the execution times into the following categories: *Abort* (time spent during aborts), *Back-off* (explained next), Barrier (time spent in barriers), *Commit* (1 cycle), *Non_xact* (time spent in non-transactional execution), *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

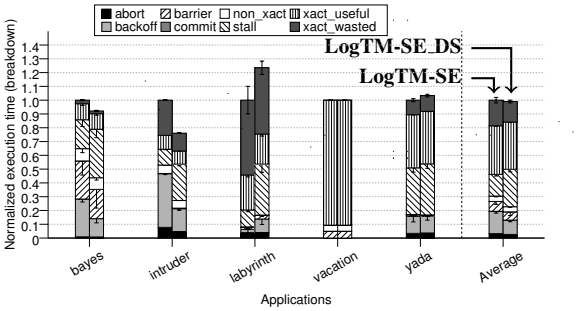


Figure 4. Breakdown of the execution times.

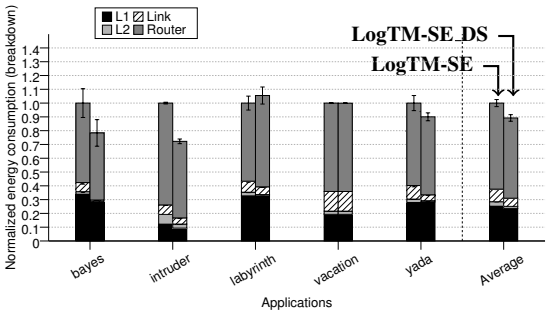


Figure 5. Breakdown of energy consumption.

transaction. Both systems employ a hardware exponential back-off mechanism that emulates the original software method, in order to reduce energy consumption due to interferences from that kind of implementation.

As it can be derived from Figure 4, none of the systems outperforms the other for all the applications. Dynamic Serialization (DS) improves performance in *bayes* (8%) and *intruder* (24%) but looses in a similar proportion with *labyrinth* (23%). For *vacation* there is no noticeable difference between the performance of LogTM-SE and LogTM-SE_DS. Below, we try to explain the differences observed for each benchmark taking into account the breakdown of the execution times, and the characteristic data access patterns of each application.

Bayes implements a non-deterministic algorithm what leads to high variability results. Its main transactional characteristics are moderate-high contention, long transactional time and considerably large write sets [13] and as a consequence a significant number of conflicts that are very expensive in case of abort. DS favors this benchmarks from the performance point of view because of the cost of the aborts and the level of contention exhibited by this application.

High contention and short transactions are the main characteristics of *intruder*. This kind of applications presents a poor performance when compared with lazy approaches [5]. In this case the cost of aborts is less punitive but the contention levels are much higher. The latter makes difficult forward progress. *Backoff*, *abort* and *xact_wasted* phases represent 73% of the total execution time for the base case of LogTM-SE (see Figure 4). In the meantime, LogTM-SE_DS reduces the number of aborts to 95000 approximately, while the base case incurs in 153000 aborts. This leads to significant reductions in the amount of time wasted due to aborts (*xact_wasted*) and in the duration of the backoff phase (*backoff*). Furthermore, DS increases the number of cycles that transactions are stalled. Differently from LogTM-SE, stalls with DS do not involve any activity.

Labyrinth 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 (tridimensional matrix of 32 x 32 x 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. 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. Aborts are also very expensive for the same reason. DS increments execution time 23% because of the random task of the transactions. After an abort, a transaction probably will find another path in the maze what means that the previous conflict will not happen again (the other conflicting transaction will likely still be in execution). The probability of conflicts between paths is not so considerable to justify the use of DS. Therefore, LogTM-SE_DS will serialize unnecessarily during long times transactions that would do forward progress after a conflict.

The difference in execution time between the LogTM-SE and LogTM-SE_DS is barely 3% in *yada*. For this application we have found that although DS is able to avoid some aborts, it creates others due to the cycles between transactions that appear when serialization is applied. Finally, both systems obtain the same result in *vacation* because the amount of conflicts is virtually zero.

4.2 Energy consumption results

Figure 5 shows the dynamic energy consumption of LogTM-SE and LogTM-SE_DS. As before, results have been normalized with respect to LogTM-SE. Additionally, we split the energy consumed 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 per cycle 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, LogTM-SE_DS beats the base case when energy consumption is considered. Only for *labyrinth* LogTM-SE shows better results. For *vacation* there are no noticeable differences between both systems since they show identical behavior. Note that while the average difference in performance among the two systems was almost zero, LogTM-SE_DS outperforms about 10% LogTM-SE when energy consumption is considered.

The differences in terms of energy consumption found in *bayes* (22%) and *intruder* (28%) are located at all levels. The reduction of the number of aborts entails energy savings in the cache structures. In addition to this, conflicts in DS are solved by stopping transactions and, when solved, letting them go. Differently than LogTM-SE, DS does not generate any network traffic while a transaction is stopped. This is the reason why the amount of energy spent in the links of the interconnection network in DS is smaller for all applications. Finally, lower network utilization and execution times supposes less energy spent in the routers.

For *labyrinth*, both the energy spent in the L1 caches and the number of aborts are similar for both systems. LogTM-SE consumes more energy in the L2 cache because during the stall phase of a transaction it is continuously retrying the access to an specific address, which is continuously rejected. That not only entails much more bandwidth but also accesses to the tags part of the L2 caches to process the request. This is true for all benchmarks too. Differences in the energy spent in the interconnection links are expected. On the other hand, the amount of energy consumed in the routers is higher because of its dependency with execution time. Only for this reason, LogTM-SE_DS is 5.5% more energy-inefficient than the standard *Eager-Eager* system.

Finally, although the execution time of *yada* is slightly increased when DS is used, the significant reduction in network traffic (explained in Section 4.3) that DS implies, translates into important energy savings in the interconnection network (routers and link). At the end, LogTM-SE_DS achieves an average reduction of 10% in energy consumption.

4.3 Network traffic results

Figure 6 shows the levels of traffic in the interconnection network (measured as flit per cycle) for both LogTM-SE and LogTM-SE_DS. As before, results have been normalized with respect to the obtained for LogTM-SE. In general, LogTM-SE entails higher traffic levels than LogTM-SE_DS (approximately 41% on average). The difference in network traffic is considerable in most cases except *vacation*.

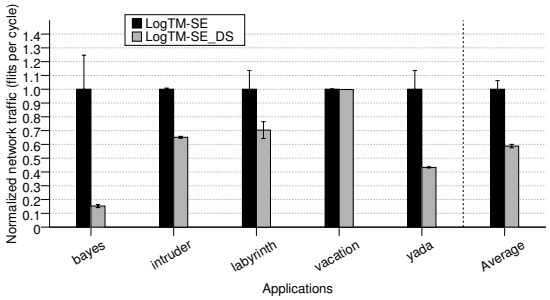


Figure 6. Normalized network traffic.

As already explained, transactions in this benchmark barely conflict. The rest of the benchmarks are characterized by exhibiting high contention and/or by the large size of their transactions [13]. During the *stall* phase in LogTM-SE, 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. LogTM-SE_DS not only saves that wasted work but also avoids conflicts that can lead to an abort by stalling conflicting transactions.

5 Related Work

Transactional Memory (TM) has become a promising parallel paradigm alternative to lock synchronization [6]. While locks suffer from deadlocks, priority inversions and convoying, TM trusts in executing transactions in parallel. TM can be implemented in either software [8] [16], hardware [2] [18], or as a combination of both [17]. Our focus is on hardware transactional memory (HTM).

Nowadays, the implications of energy consumption is a first-class consideration, requiring tradeoffs against performance. This is true not only for embedded systems [4] (such as mobile devices) but also for server and even desktop systems [1]. TM literature has traditionally focused on improving performance, simplicity [2] or even flexibility [17]. In the STM world, Klein *et al.* [11] have performed a study about energy consumption compared with lock techniques and at the same time propose new mechanisms to improve this key factor. In HTM, Moreshet *et al.* [14] 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 in which two conflicting transaction are reissued in serialized mode, preventing parallel speculation in others transactions. Ferri *et al.* [4] present a simple and energy-efficient TM 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 its execution until the transaction commits. This reduction in speculation and per-

formance suits well with embedded systems, but not with general purpose ones. In [5] two well-known HTM systems (*Eager-Eager* LogTM-SE system [18] and *Lazy-Lazy* Scalable TCC system [3] [15]) are compared in terms of execution time, energy consumption and traffic network. Despite the fact that the *Lazy-Lazy* system outperforms the *Eager-Eager* one for the general case, LogTM-SE and other eager approaches present a significant potential for improvement in energy consumption.

6 Conclusions

In this work, we present Dynamic Serialization (DS), a new technique that improves energy consumption in Hardware Transactional Memory (HTM) systems that implement eager conflict management, such as LogTM-SE. DS is aimed at dynamically serializing transactions in high-contention scenarios. In these cases, previous works [5] 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, DS detects this kind of situations and dynamically serializes only involved transactions.

We have implemented DS on top of the GEMS full-system simulator and we have compared it 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, DS obtains an average reduction of 10% in energy consumption (up to 20% in high-contention scenarios) at no performance cost. Furthermore, DS is able to save about 41% of the network traffic levels generated by LogTM-SE. This is because DS precludes transactions from continuously retrying a conflicting memory access.

Acknowledgment

This work was supported by the Spanish MICINN, Plan E funds, under Grant TIN2009-14475-C04-02. 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] L. A. Barroso and U. Hölzle. The Case for Energy-Proportional Computing. *Computer*, 40(12):33–37, December 2007.

[2] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA-33*, pages 227–238, 2006.

[3] 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*, pages 97–108, February 2007.

[4] 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)*, 70(10):1042–1052, October 2010.

[5] E. Gaona, R. Titos, J. Fernandez, and M. E. Acacio. Characterizing energy consumption in hardware transactional memory systems. SBAC-PAD-22, pages 9–16, 2010.

[6] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Morgan & Claypool, 2010.

[7] M. Herlihy, J. Eliot, and B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA-20*, pages 289–300, 1993.

[8] M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III. Software transactional memory for dynamic-sized data structures. In *PODC-22*, pages 92–101, 2003.

[9] HP Labs. http://quid.hpl.hp.com:9081/cacti.

[10] 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-13*, pages 423–428, 2009.

[11] F. Klein, A. Baldassin, G. Araujo, P. Centoducatte, and R. Azevedo. On the energy-efficiency of software transactional memory. In *SBCCI-22*, 2009.

[12] 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*, 33(4):92–99, 2005.

[13] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC-4*, pages 35–46, 2008.

[14] T. Moreshet, R. I. Bahar, and M. Herlihy. Energy-Aware Microprocessor Synchronization: Transactional Memory vs. Locks. In *Workshop on Memory Performance Issues*, 2006.

[15] S. H. Pugsley, M. Awasthi, N. Madan, N. Muralimanohar, and R. Balasubramonian. Scalable and reliable communication for hardware transactional memory. In *PACT-17*, pages 144–154, 2008.

[16] B. Saha, A. Adl-tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *PPoPP-11*, pages 187–197, 2006.

[17] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible decoupled transactional memory support. In *ISCA-35*, pages 139–150, 2008.

[18] 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*, pages 261–272, 2007.