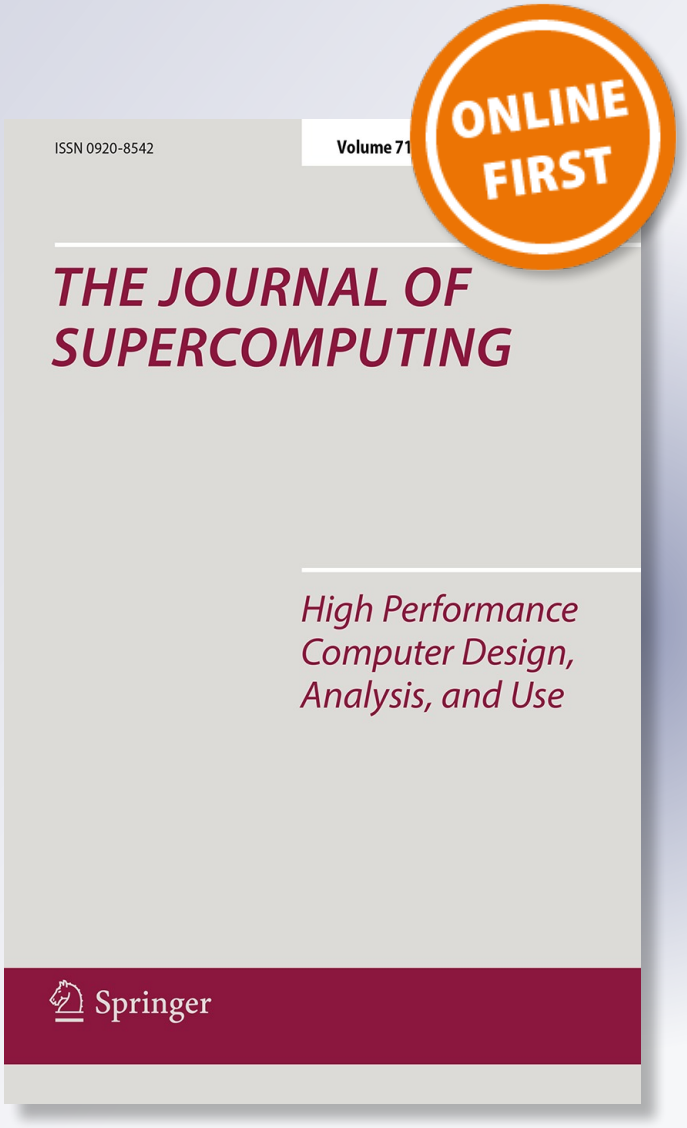
# *Fast and efficient commits for Lazy-Lazy hardware transactional memory*

**Epifanio Gaona, José L. Abellán & Manuel E. Acacio**

**The Journal of Supercomputing**
An International Journal of High-Performance Computer Design, Analysis, and Use

ISSN 0920-8542

J Supercomput
DOI 10.1007/s11227-015-1523-8




Springer

J Supercomput
DOI 10.1007/s11227-015-1523-8

# Fast and efficient commits for Lazy-Lazy hardware transactional memory

**Epifanio Gaona**[1] · **José L. Abellán**[2] · **Manuel E. Acacio**[1]



**Abstract** Transactional memory (TM) is a compelling alternative to simplify multithreaded programming that traditionally relies on error-prone lock-based synchronization for implementing cooperative tasks. *Lazy-Lazy* hardware TM is one of the most efficient schemes in today's hardware TM systems. Nonetheless, the commit protocol in these systems has severe impact on performance and energy. The SEQ in Scalable-TCC implementation (STCC-SEQ) is the most popular and efficient commit protocol to date. In this paper, we propose GCommit, a cost-effective hardware-based STCC-SEQ protocol. GCommit employs a G-Arbiter microarchitecture for achieving minimal-latency and high-efficient commits. We implement G-Arbiter with a standard 45 nm cell library. For a target 16-core CMP, a G-Arbiter just represents 0.07 % of the whole on-chip area, requiring marginal energy consumption. Full-system simulations of the target system with the STAMP benchmarks show that GCommit achieves average reductions of 15.7 and 13.7 % in execution time and energy, respectively, when compared with STCC-SEQ.

**Keywords** Manycore CMPs · Hardware transactional memory (HTM) · *Lazy-Lazy* HTM · Transaction commit

✉ José L. Abellán
jlabellan@ucam.edu

Epifanio Gaona
fanios.gr@ditec.um.es

Manuel E. Acacio
meacacio@ditec.um.es

1 Universidad de Murcia, Murcia, Spain

2 Catholic University of Murcia, Murcia, Spain

Published online: 21 September 2015

## 1 Introduction and motivation

Manycore microprocessors have been designed to allow energy-efficient and scalable computing. Most processor manufacturers have already released products that incorporate several execution cores on a single chip, and the trend is reaching hundreds or even thousands of them in the near future [1]. Nowadays, these manycore chips are available in almost all market segments. For example, Intel i7 and AMD FX series are desktop processors that come with up to 8 dual-threaded processor cores. Intel Knights Landing [2] and Tilera Tile-Gx8072 CPU [3] are examples of servers and embedded systems with 72 cores each.

Transactional memory (TM) has arisen as a promising alternative to the error-prone and complex lock-based synchronization. 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. A TM system can be implemented in either software [4–8], hardware [9–12], or as a combination of both [13]. The common denominator in all implementations is that transactions are speculatively executed which hides the main pathologies associated with locking techniques, such as priority inversion, convoying and deadlocks, from programmers. As a consequence, programmers are armed with an intuitive synchronization abstraction that can greatly help simplify the development of multithreaded programs.

In hardware transactional memory (HTM) systems, the hardware provides the illusion that each transaction is executed atomically and in isolation while threads are executing in parallel. HTM systems usually work at the word or cache line level. Conceptually, each transaction is associated with two initially-empty read and write sets that are populated every time a transactional load or store is issued. To comply with the serializability property, both the old values and the transactional ones must coexist until the transaction is allowed to commit. A transaction can commit only after the HTM system can assure that there are no other running transactions whose write sets collide with its read or write sets. The commit process makes the read and write sets of the winner transaction visible to the whole system. HTM systems are usually classified attending to how they tackle with data version management (VM) and conflict detection (CD). In this work, we focus our attention on the extensively used *Lazy-Lazy* systems [13,14]. For example, the Sun's Rock [15] falls into this category. On lazily versioned systems, updates are aside, i.e., transactional stores are placed in a *write buffer* until the transaction acquires privileges to make its changes visible to the whole system. In this way, transactional data are dumped into the memory hierarchy at commit time. With lazy CD, dependency violations are also checked during the last phase of any transaction, i.e., at commit time. Sequential (SEQ) commit protocol [16] used in Scalable Transactional Coherence and Consistency system [17] (STCC-SEQ) is the most popular HTM *Lazy-Lazy* system to date.

In this work, we present an efficient hardware implementation of the STCC-SEQ system called GCommit that reduces the duration of the *commit* phase (more precisely the *precommit* component) implemented in STCC-SEQ to boost performance and reduce energy consumption. GCommit leverages a dedicated on-chip network

based on GLock architecture [18]. GLock was designed as a hardware alternative to accelerate Lock/Unlock primitives in shared-memory programming, especially when highly contended Lock/Unlock scenarios occur among many competing threads. The reason why we utilize GLock for improving STCC-SEQ system is that STCC-SEQ also requires to implement very fast exclusive access in highly contended scenarios. Particularly, when HTM transactions want to commit and require exclusive access to a rank of addresses at *precommit* time in shared L2 cache banks. As we will demonstrate, implementing this functionality using dedicated hardware brings important benefits in terms of execution time as well as energy consumption with respect to traditional implementations of commit protocols that use the general-purpose interconnection network to coordinate commit ordering.

We evaluate our proposal using a full-system simulated 16-core CMP with 9 STAMP transactional benchmarks. Our experimental results reveal that, compared to STCC-SEQ, GCommit accelerates the *precommit* subphase about 68.5 % on average, which results in average reductions of 15.7 and 13.7 % in terms of execution time and energy consumption, respectively. To prove that GCommit can be fabricated in a real chip for improving STCC-SEQ system, we implement GCommit through a mainstream industrial synthesis toolflow. We corroborate that GCommit can meet the timing closure constraints needed to yield the performance advantages obtained through the full-system simulation platform. Furthermore, we obtain that GCommit just accounts for 0.07 % on-chip area of the whole target system.

A preliminary version of this work was presented in [19], where we basically introduced our proposal and demonstrated through full-system simulation that STCC-SEQ can be improved by a hardware-based solution. In this work, apart from including more detailed explanations to help the reader understand how GCommit actually operates, we highlight the following contributions:

- We introduce *G-Arbiter*, the low-level microarchitecture utilized by GCommit to implement the *precommit* suphase of a STCC-SEQ system. To clearly show how G-Arbiter operates, a walk-through example is illustrated with four different situations of transactions attempting to commit.
- We demonstrate that GCommit can be integrated in a real chip. To achieve this, GCommit is implemented through a mainstream industrial synthesis toolflow with an STMicroelectronics 45 nm standard cell technology library. From this study, we validate the correct operation of GCommit and we accurately account for on-chip hardware costs that GCommit entails.
- We include more details of the transactional benchmarks under study to shed light on the experimental results obtained when evaluating our GCommit proposal.

The rest of the paper is organized as follows. Section 2 presents our proposal, describing in depth the hardware components and operation of GCommit. In Sect. 3, we detail the simulation environment and the workloads used to evaluate GCommit. Results in terms of execution time, energy consumption and network traffic are analyzed in Sect. 4. Section 5 discusses related work. Finally, conclusions are given in Sect. 6.

## 2 GCommit for efficient commits in Lazy-Lazy HTM systems

Our GCommit proposal is an efficient and low-cost hardware implementation of the STCC-SEQ protocol that significantly accelerates the precommit subphase of commits in Lazy-Lazy TM systems. To help understand how our mechanism works, we first describe the baseline STCC-SEQ system. Afterwards, we detail G-Arbiter, the microarchitecture GCommit relies on to ensure exclusive access to a rank of addresses at precommit time (as L2 cache banks do in STCC-SEQ). Once we detail the G-Arbiter microarchitecture, we describe our GCommit protocol and give an example of use. We compare our GCommit architecture against the baseline STCC-SEQ system our proposal is based on. To do so, we show the size of the structures needed by them to demonstrate the superior scalability of our proposal. In the last part, we explain the implementation of our GCommit architecture presented in [19] that leverages a state-of-the-art G-lines technology, utilizing a cost-effective mainstream industrial synthesis toolflow with an STMicroelectronics 45 nm standard cell technology library.

### 2.1 STCC with sequential commit (STCC-SEQ)

In *Lazy-Lazy* systems, transactions are allowed to run as if they were alone in the system. Only when a transaction reaches its end and before it is allowed to propagate their results, conflicts are checked. This stage is known as the commit phase. Sequential commit (SEQ) is nowadays the most popular commit algorithm for *Lazy-Lazy* HTM systems. Instead of employing a centralized arbiter to enforce commit ordering, SEQ makes use of the L2 directory banks to manage an implicit order between transactions with clashing read and write sets. To do so, it tries to book every directory (L2 cache bank) in its read and write sets. A directory belongs to those sets if at least one transactional address belongs to the corresponding L2 cache bank. The process is explained below:

(1) A committing transaction sends a request message to each directory in its read and write sets in ascending order. This prevents deadlock conditions. In the case of two different transactions competing to commit, only the first that achieves to book the first conflicting directory will continue with the process. The other transaction will have to wait till the completion of the first one.

(2) Once a transaction has booked a particular directory, an "Occupied" bit is set and an ACK message is sent back. Other requests will be buffered.

(3) The precommit phase finishes once a transaction has collected all the ACKs from every directory bank in its read and write sets.

(4) The transaction's write set is sent to the reserved directory banks. Involved memory lines will be marked as Owned and invalidations will be sent to the other sharers of these lines to signal a conflict with this transaction. When all the ACKs for these invalidations have been received, the directory bank clears its "Occu-

pied" bit. If another transaction is waiting to book the directory bank, then an ACK message is sent to it.

(5) The transaction finishes with a release message that is sent to the directories in its read set.

The scheme above represents the basic approach where Read-After-Read (RAR) situations are managed as a possible source of conflict. This happens when two transactions share the same directory bank in their read sets. Although in this case there is no conflict between the transactions (the directory is only used for reads), SEQ only allows one of the transactions to book the directory, delaying the commit of the other. An advanced algorithm called SEQ with Parallel Reader Optimization (SEQ-PRO) distinguishes between directories booked by read and write access, allowing several transactions to occupy the same directory bank for read accesses. To do so, several Read "Occupied" bits (and one Write "Occupied" bit) are required in every directory bank.

### 2.2 G-Arbiter microarchitecture to enforce commit ordering in GCommit

The design of a G-Arbiter for a 4-core CMP is shown in Fig. 1a. A G-Arbiter is made up of two kinds of components: controllers (*R*, *Sx*, and *Cx*), and global links that are used by the controllers for data transmission to implement the G-Arbiter communication protocol. The controllers can be classified as follows: *local controllers* (*Cx* in Fig. 1b), a *primary lock manager* (*R* in Fig. 1b) and *secondary lock managers* (*Sx* in Fig. 1b). Each core *x* has a *local controller* (*Cx*) that sends and receives information from its associated *secondary lock manager Sx*. *Secondary lock managers* are located in the first core of each row of the CMP layout. The behavior of the *local controller* of these cores is encapsulated in the *secondary lock manager*. There is a single *primary lock manager* or root located at *C0*. The *secondary lock manager* of this core is encapsulated in the *primary lock manager*. In both cases, the communication is performed locally by means of a flag. Finally, while *secondary lock managers* monitor signals from their

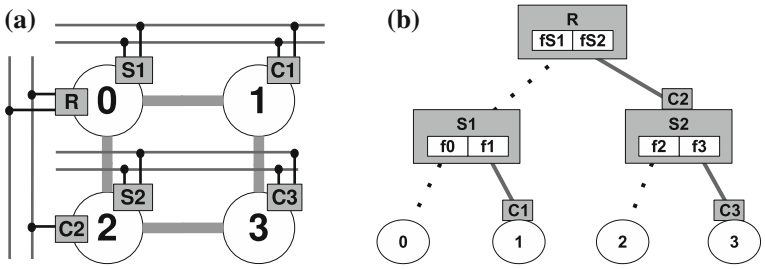



**Fig. 1** Implementation of a G-Arbiter in a 4-core CMP-Controllers: *R* primary lock manager, *S1∼2* secondary lock managers, *C1∼3* local controllers, Internal flags: *fS1∼2 and f0∼3* buffers to hold controllers' signals. *Solid lines* global links for communication between controllers. *Dashed lines* logical links for internal communication between controllers through internal flags. **a** G-Arbiter architecture. **b** Logical view of a G-Arbiter

remote *Cx* controllers or their *local controller* (by means of a flag), the *primary lock manager* monitors signals from the rest of remote *secondary lock managers* or its local *secondary lock manager*. Figure 1b describes the above architecture in a three-level tree structure form. The root is the *primary lock manager*. The *secondary lock managers* are represented by the intermediate nodes. Finally, the leaves are the *local controllers* and cores. All elements are connected using global links (solid lines) or by means of an internal flag (dashed lines). The flags (*fx* and *fSx*) store the signals sent by the controllers to the corresponding *lock managers* and by the *secondary lock managers* to the *primary lock manager*, respectively.

The synchronization protocol inside of a G-Arbiter is based on the exchange of signals between the *local controllers*, the *secondary lock manager* and the *primary lock manager* through global links. In particular, when a transaction wants to access a G-Arbiter, its *local controller* notifies its *secondary lock manager* of the request by means of a REQ signal. Next, the *secondary lock manager* asks for the G-Arbiter by sending the REQ signal to the *primary lock manager*. Similarly, when the G-Arbiter is acquired, the ACK signals flow in the opposite direction towards the corresponding *local controller*. To release the G-Arbiter, REL signals perform the same travel than REQ ones but no other confirmation is expected from the G-Arbiter or its components. *Primary* and *secondary lock managers* guarantee fairness using a Round-Robin policy, that is, if all transactions wanted to get the same G-Arbiter concurrently, they would be granted access in a cyclic fashion. As we will detail in Sect. 2.3, in absence of contention, a transaction will receive the ACK signal in four cycles if the REQ signal must reach the *primary lock manager* and in two cycles if the intermediate *secondary lock manager* does not send back the REL signal to the primary one. Release process spends only one cycle.

Note that, as compared to a GLock, G-Arbiter extends its functionality to manage release petitions (unlock) even if the ACK signal that grants access to the lock has not been sent yet. In particular, this behavior is found when a transaction is aborted just after requesting access to a GLock, and the corresponding ACK has not been received. In this case, and as part of the abort, the transaction desists from its request to commit by explicitly sending the release signal to the lock manager.

### 2.3 GCommit protocol

GCommit makes use of one or more G-Arbiters to enforce commit ordering between transactions with clashing read and write sets during the *precommit* subphase. The use of a few global links (only 6 single-bit links in Fig. 1 for a 4-core CMP system) to design such a signalization protocol is feasible because there is no need to transport large amounts of information. In addition, this approach yields a twofold benefit: improved performance and reduced energy consumption. The former is mainly due to the low latency of G-Arbiter communication protocol. The latter stems from the elimination of the traffic induced by the *precommit* subphase that makes use of the general-purpose interconnection network.

Each G-Arbiter is conceptually a lock associated with a range of memory addresses that prevents simultaneous accesses over them at commit time. The policy to assign memory block addresses to G-Arbiters and the granularity of the assignment are arbi-

trary because GCommit does not impose any particular restriction on them. In our study, we either have a single G-Arbiter or 16 G-Arbiters (one per L2 cache bank). In the former case, the assignment is direct. In the latter case, each L2 cache bank is assigned to a different G-Arbiter to maximize the probability of having parallel commits. Note that in our results, consecutive chunks of 128 KB are mapped across all L2 cache banks with a Round-Robin policy. In this sense, when a transaction reaches the *precommit* subphase, it has to acquire all the G-Arbiters in charge of the addresses of its read and write sets. To keep track of the required G-Arbiters, each transaction makes use of a hardware bit-vector structure called *G-Arbiter vector*. In this way, each core would need a *G-Arbiter vector* per thread in a SMT configuration. A transaction sets position *i* of its *G-Arbiter vector* if it accesses any data mapped to that G-Arbiter. When a transaction reaches the *precommit* subphase, it must ask for permission to commit to all the G-Arbiters in its *G-Arbiter vector*. This process is performed sequentially in ascending order to avoid deadlock situations. To do so, the transaction sends a request (REQ) signal to acquire the first G-Arbiter. After receiving an acknowledgement (ACK) signal, the transaction proceeds with the next one. Once a transaction has acquired all the G-Arbiters, it cannot be aborted and dumps the contents of the *write buffer* into the memory hierarchy (*commit* subphase). Next, the committing transaction sends a release (REL) signal to all the previously acquired G-Arbiters. The *G-Arbiter vector* is reset at the end of a commit (or an abort). It is worth noting that during the *commit* phase there is a likelihood of aborting other transactions. This conflict may happen when a committing transaction updates the memory hierarchy. If any updated memory block is in the read or write set of any other ongoing transaction, that transaction must abort (like in any other *Lazy-Lazy* system). Besides, if that transaction had already acquired one or more G-Arbiters, it would have to send a release (REL) signal to each of those G-Arbiters.

To have a clear understanding of how our GCommit protocol works, Fig. 2 shows an example where four transactions try to commit at the same time. We assume one thread per core. Figure 2a shows the initial state of the four transactions ($T0$–$T3$) with their *G-Arbiter vectors* above them—*G-a x* stands for *G-Arbiter vector* of core *x*. On the right, there are two G-Arbiters (G0 and G1). Finally, the table on the top of the figure represents the mapping table that specifies the assignment of memory block addresses (letters from A to F) to L2 cache banks and G-Arbiters. Note that, in the figure, RX means read address X, and WY means write address Y. Each G-Arbiter layout corresponds to the design of Fig. 1b. While the root node represents the registers *fSx* of the *primary lock manager*, the leaves are the *fx* registers of the *secondary lock managers*. The left leaf node stores REQ signals from core 0 and core 1 ($T0$–$T1$), whilst the right leaf node does the same with core 2 and core 3 ($T2$–$T3$).

Figure 2b shows two successful attempts to commit. Lower case letters on the left of transactions execution bar represent the cycle in which a communication with a G-Arbiter has started. When $T3$ finishes its normal execution (white bar), it enters the *precommitting* phase (gray bar). Before committing $T3$ must acquire access to those G-Arbiters enabled in its *G-Arbiter vector* (only G0). In cycle the *p*, the *local controller* of core 3 (C3) enables REQ signal to the immediate *secondary lock manager* of G0 ($S2_0$) and writes its corresponding flag. Since $S2_0$ has not enough privileges to ensure access to $T3$, the REQ signal travels to the *primary lock manager* ($R_0$). Two cycles

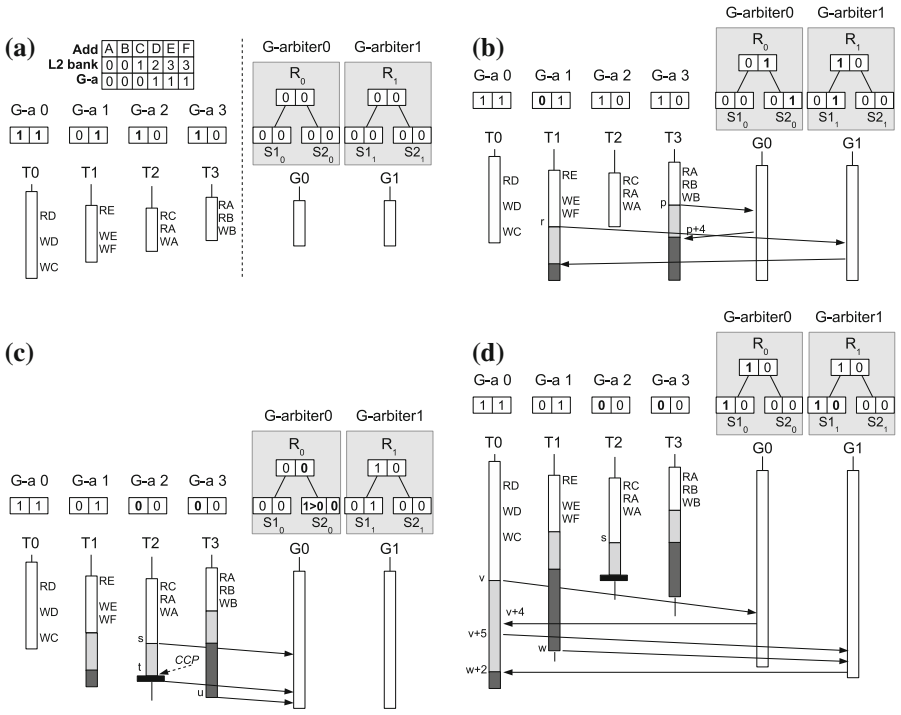



**Fig. 2** Example with four attempts to commit. **a** Step 1. **b** Step 2. **c** Step 3. **d** Step 4

later, $R_0$ has managed that petition and responds with an ACK to $S2_0$ which forwards it to C3. As a result, $T3$ is allowed to commit (dark bar) with a total delay of 4 cycles. In the cycle *r*, T1 launches the same process and gets the same result with G1. At the end of this figure, both transactions are committing in parallel.

Next, Fig. 2c shows an abort because of a previous commit. $T2$ tries to acquire G-Arbiter 0 in the cycle *s*. As G0 has previously granted access to $T3$, $T2$ must wait for an ACK signal. While $T2$ is being serialized in its *precommitting* phase, it observes a change in cache that crashes with its sets (address A). The cache coherence protocol (CCP) sends an invalidation for that block to $T2$, which must abort (cycle *t* in Fig. 2c). Finally, $T3$ sends a REL signal to G0 at the end of the commit (cycle *u*).

To conclude, Fig. 2d shows a successful serialized commit. At the end of the normal execution, $T0$ tries to get access to G0 and G1. This process must be performed sequentially in ascending order. $T0$ (its *local controller*) sends first a REQ signal to G0 (cycle *v*). G0 is able to respond with an ACK in 4 cycles. Next, T1 enables another REQ signal to G1 (cycle *v+5*). G1 has previously been acquired by T1, so the ACK signal is sent only when G1 receives the REL signal from T1 in cycle *w*. Finally, $T0$ is allowed to commit too.

### 2.4 GCommit vs STCC-SEQ

Table 1 shows the size of the structures needed to implement STCC-SEQ, their parallel reader optimization (STCC-SEQPRO [16]) and our proposal, GCommit. To avoid

**Table 1** Size of the structures needed in STCC-SEQ, STCC-SEQPRO and GCommit

| System | # Buffer sets | # Entries per buffer set | Bits of each entry |
|---|---|---|---|
| STCC-SEQ | $\#L2DB$ | $T$ | $\text{Log}_2 T$ |
| STCC-SEQPRO | $\#L2DB \times 2$ | $T$ | $\text{Log}_2 T$ |
| GCommit | $\#GA$ | $T + \log_2 T$ | 1 |

overflows in the buffers of the L2 directory banks, STCC-SEQ needs at least one entry per thread (T is the number of threads). STCC-SEQPRO doubles that resources since it needs two buffers per directory bank (L2DB)—one for readers and one for writers. Moreover, to keep track of the arrival order, each buffer must store a unique ID per thread, whose size is equal to $\log_2 T$ bits. On the other hand, the equivalent mechanisms in GCommit are the flags implemented inside of the *lock managers*. Instead of using buffers, GCommit employs sets of flags. Each flag is one bit and each G-Arbiter needs $T + \log_2 T$ flags ($T$ flags for the *secondary lock managers* and $\log_2 T$ flags for the primary one). Moreover, GCommit does not need as much G-Arbiters (GA) as L2DB (*# buffers* in Table 1) to perform better than STCC-SEQ (see Sect. 4). The total size (in bits) of the structures required by each one of the configurations considered in this work can be calculated by multiplying the values in the corresponding row of Table 1.

In the previous paragraph, we demonstrate that GCommit achieves better scalability than both STCC-SEQ and STCC-SEQPRO systems in terms of the structures required to keep track of the transactions willing to commit. However, another important consideration is whether the hardware hierarchical infrastructure that implements GCommit (i.e., the G-Arbiter architecture illustrated in Fig. 1a) can also be scaled well with the number of processor cores. As explained in Sect. 2.2, our G-Arbiter is a slight modification of the GLock mechanism proposed in [18]. It is based on the same scalable hierarchical infrastructure as follows. First, using the same 2-level hierarchy presented in Fig. 1a for larger manycore systems requires complexer and/or slower lock managers due to having higher radixes to interconnect more local controllers. This scheme can scale well to up to 32 processor cores [18]. Second, to further improve scalability of the G-Arbiter for higher core counts, as explained in [18], another strategy to avoid augmenting the controllers radixes so that they can always run at the same frequency is to increase the number of levels in the hierarchy. This strategy would affect the latency of the G-Arbiter operation, which would be increased. This extra latency, however, will not be excessive (one extra clock cycle when going up/down through every extra level) and will not hurt performance in a significant way.

### 2.5 G-Arbiter manufactured using an industrial synthesis toolflow

In [19], we introduce our GCommit protocol utilizing a state-of-the-art full-custom technology, namely G-lines. We demonstrated that our proposal reported minimal area overhead, energy consumption and very fast operation. Nevertheless, this implementation is not cost-effective as it is not within reach of a standard cell design methodology. For that reason, in this section, we study the hardware costs of our proposal in the

context of a mainstream industrial synthesis toolflow with an STMicroelectronics 45 nm standard cell technology library. This technology will be referred to as *Standard* technology.

When using Standard technology, since traditional resistance–capacitance (RC) wires are very critical to performance degradation, we have implemented each G-Arbiter's controller by separating the delay that signals take along the wires, from the effective computation that the controllers require to generate their output signals. Notice that, for small manycore CMPs, the critical path that limits the maximum operating speed in our G-Arbiter infrastructure is defined by the most complex controller (i.e., the lock manager that samples signals from the local controllers), but as the wire length increases for larger CMPs, the wires could represent such critical path. Consequently, separating wire delays from controllers delays become essential to achieve maximum clock speeds. In this way, using this technology, we cannot directly assume the synchronization latencies achieved using G-lines, and a higher number of cycles will be required for the gather and release phases. In addition, to minimize the length of wires, we have situated the lock managers (both primary and secondary ones) in the central column/row of the 2D-mesh topology, rather than the first column and first row as depicted in Fig. 1a. Note that, in case of G-lines technology, this optimization would not be necessary since every G-line is specially designed to implement one-cycle latency, one-bit transmissions across one dimension of the chip.

For a realistic characterization of this implementation, placement-aware logic synthesis is performed through Synopsys Physical Compiler. Moreover, the final place-and-route step is performed with Cadence SoC Encounter [20] which also involves clock tree synthesis. In addition, we assume a single clock domain with a unique clock tree for the whole CMP layout. It is worth noting that our mechanism has been synthesized by defining non-routable obstructions. In this work, we assume that this area is equal to $550 \times 550\,\mu m^2$ (each core). Additionally, fences are defined to limit the area where the cells of each G-Arbiter's controller can be placed. Such obstructions and fences also ensure minimum-length routing for the *links* to reduce their impact on performance and area overhead as the wire length increases. The resulting floorplan is shown in Fig. 3 that shows the controllers of a G-Arbiter in the context of a 4-core

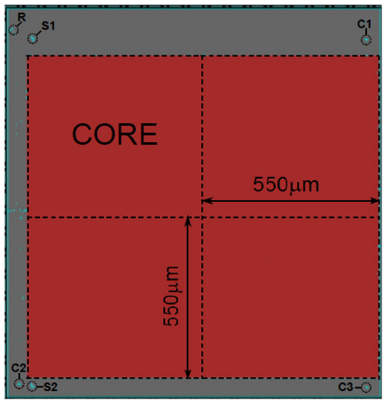



**Fig. 3** Floorplan of the G-Arbiter's controllers in a 4-core CMP designed with Cadence SoC Encounter tool

**Table 2** Raw statistics using *G-lines* technology and a *Standard* technology for a single G-Arbiter in a 16-core CMP layout

| Technology | Freq (MHz) | Latency (cycles) | Area ($\mu m^2$) | Power (mW) |
|---|---|---|---|---|
| Standard | 1000 | 9 (worst), 5 (best) | 3618 | Negligible |
| G-lines | 2500 | 4 (worst), 2 (best) | Negligible | 14 |

CMP system (see Fig. 1a for a logical view of this scheme). As we can see in the figure, it is worth nothing the very small area required by the controllers as compared to the area assumed per core.

Table 2 shows the main raw performance statistics obtained from this *Standard*-based G-Arbiter implementation and with original *G-lines* characteristics. We illustrate the maximum operating speed, the latencies of the synchronizations and also the area overhead that our proposal entails. It is worth noting that the area devoted to a G-Arbiter is equal to $3618\,\mu m^2$ that corresponds to a negligible 0.07 % of the total area employed for the simulated 16-core CMP layout (remember that we assumed that each core is $550\times550\,\mu m^2$ in size). This marginal overhead also will lead to a negligible impact on power consumption. Moreover, worse latencies and frequency are shown in case of the Standard technology in comparison to the G-lines technology. While these new performance parameters were incorporated into the simulation environment presented in Sect. 3, we observed that the performance results when using Standard technology are virtually the same than those reported by the implementation using the G-line technology. This means that our GCommit protocol does not depend on the full-custom G-lines technology to achieve superior efficiency against the baseline STCC-SEQ system.

## 3 Evaluation environment

### 3.1 System settings

We use a full-system execution-driven simulation based on the Wisconsin GEMS toolset [21], in conjunction with Wind River Simics [22]. 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 simulate a tiled CMP system configured as described in Table 3. More precisely, we assume a 16-core configuration with private L1 I&D caches and a shared, multibanked L2 cache consisting of 16 banks of 512 KB each. The L1 caches maintain inclusion with the shared L2 cache. The private L1 data caches are kept coherent through an on-chip directory (at L2 cache banks), which maintains bit vectors of sharers (which are included in the tags' part of the L2 cache banks) and implements the MESI protocol. All tiles are connected through a router-based 2D-mesh network. In this $4\times4$ 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

**Table 3** System parameters

| | |
|---|---|
| MESI Directory-based CMP | |
| Cores | 16, single 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 cycles |
| Link bandwidth | 16 Bytes/cycle |

**Table 4** Parameters of Orion 2.0

| Parameter | Value |
|---|---|
| In_port | 6 |
| Tech_point | 45 |
| Vdd | 1.0 |
| Transistor type | NVT |
| Flit_width | 128 (bits) |

the interconnection network has been measured based on Orion 2.0 [23]. In particular, we have extended the network simulator provided by GEMS with the consumption model included in Orion. Table 4 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 amount of energy spent in the memory structures (L1, L2) has been measured based on the consumption model of CACTI 5.3 rev 174 [24]. 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. Furthermore, it provides support for a naive implementation of a *Lazy-Lazy* system. We have extended the latter one to achieve an implementation of a *Lazy-Lazy* system that mimics the behavior of Scalable-TCC with the sequential commit algorithm (STCC-SEQ) described in [16]. We have implemented GCommit on top of STCC-SEQ. For that, all we had to change was the commit protocol. In GCommit, all commits are performed in hardware using the mechanism described in Sect. 2. The remaining aspects are identical in both GCommit and STCC-SEQ. Finally, we have also evaluated the parallel reader optimization for STCC-SEQ (STCC-SEQPRO [16]). This optimization allows multiple transactions to simultaneously occupy a directory as long as none of these transactions write to this directory. The counterpart is that it increases complexity and area requirements.

**Table 5** Workloads and inputs

| Benchmark | Input |
|---|---|
| Genome | -g512 -s32 -n32768 |
| Intruder | -a10 -l16 -n4096 -s1 |
| Kmeans-high | -m40 -n40 -t0.05 -i random-n16384-d24-c16 |
| Kmeans-low | -m40 -n40 -t0.05 -i random-n16384-d24-c16 |
| Labyrinth | -i random-x32-y32-z3-n96 |
| Ssca2 | -s13 -i1.0 -u1.0 -l3 -p3 |
| Vacation-high | -n4 -q60 -u90 -r1048576 -t4096 |
| Vacation-low | -n2 -q90 -u98 -r1048576 -t4096 |
| Yada | -a10 -i ttimeu10000.2 |

### 3.2 Workloads

For the evaluation, we use nine transactional benchmarks extracted from the STAMP suite [25]. These applications allow to stress a TM system in several ways. To show a wide range of cases, we evaluate the most relevant STAMP applications using the recommended input size in each case (in general, what is called the medium size). The application *Bayes* was excluded since it exhibits unpredictable behavior and high variability in its execution times [26]. For *Kmeans* and *Vacation*, both high and low contention configurations were used. Results presented have been averaged over twenty runs for each application, each with very minor randomization of some system parameters just sufficient to excite different interleavings. Table 5 describes the benchmarks and the values of the input parameters used in this work.

## 4 Experimental results

### 4.1 Performance comparison analysis

For the nine transactional applications pointed out in Sect. 3, Fig. 4 shows the relative breakdown of the execution times that are obtained for STCC-SEQ, GCommit and STCC-SEQPRO. In all cases, execution times have been normalized with respect to those obtained with the STCC-SEQ. Moreover, to have clear understanding of the results, Fig. 4 splits each of the bars into the following categories: *Abort* (time spent during aborts), *Back-off* (delay time between an abort and the next re-execution), *Barrier* (time spent in barriers), *Commit* (time needed to propagate the write sets to the memory hierarchy), *Non_xact* (time spent in non-transactional execution), *Precommitting* (time taken to acquire privileges to commit–book the corresponding directory modules/gain access to the*G-Arbiter*–), *Stall* (time waiting until another transaction finishes its *commit* phase), *Xact_useful* (useful transactional time), *Xact_wasted* (transactional time wasted because of aborts).

As it can be derived from Fig. 4, GCommit shows noticeable improvements in overall performance with respect to both STCC-SEQ (average reduction of 15.7 % in execution time) and STCC-SEQPRO (average reduction of 11.3 % in execution time). It is important to note that these improvements come as a result of a significant reduc-

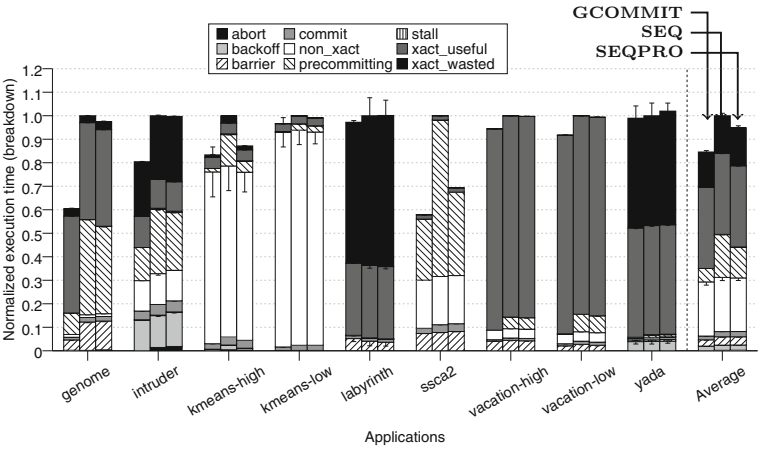



**Fig. 4** Breakdown of the execution times

tion in the amount of time needed to complete the *precommitting* phase (*Precommitting* category in the bars). Observe also that GCommit consistently reduces the duration of the *precommitting* phase in all the applications. The fact that in GCommit all the booking process of the directory modules during the *precommitting* phase is done through a fast protocol operating on dedicated wires brings down the duration of this phase (68.5 % on average compared with STCC-SEQ). On the contrary, in STCC-SEQ and STCC-SEQPRO, the *precommitting* phase is based on the interchange of regular messages travelling on the general-purpose interconnection network. Observe also that the distinction between reads and writes carried out in STCC-SEQPRO results in small improvements over STCC-SEQ (average reduction of only 5.1 % in execution time), which are mainly a consequence of the reductions in the duration of the *precommitting* phase observed in just two of the nine applications (*Kmeans-high* and *Ssca2*).

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

**Genome** This workload exhibits moderate to high degree of contention from the very beginning of its execution. Readers–writers conflicts dominate at the first phase of the application. The second phase is characterized by transactions with moderate write sets that access to predominantly non-contended data. This phase dominates overall performance. We have found that the average number of directory banks that must be booked during the *precommitting* phase is considerably high (between 10 and 16). Due to directory, booking in STCC-SEQ is performed in ascending order and a transaction is not allowed to access the next bank until having received confirmation from the previous one, the amount of time needed to complete the *precommitting* phase is significant. Additionally, the number of parallel commits keeps low and the distinction between read and write accesses enabled by STCC-SEQPRO barely improves execution time. On the contrary, in GCommit only one *G-Arbiter* must be acquired during the *precommitting* phase. Moreover, in absence of contention, few cycles (2 or

4) are enough to gain access to the *G-Arbiter*. The result is that execution time in this application is dramatically reduced (40 %) when GCommit is considered.

**Intruder** This workload shows high contention with three transactions. Nevertheless, only transaction TID0 accumulates the major part of the aborts. We have found that this transaction needs to book just two directory banks, one for reading and the other one for writing, so there is no chance of parallel commits between transactions TID0 running on different cores. Nevertheless, parallel commits are still possible for the other two transactions (which have different TIDs). We have observed that STCC-SEQPRO can take advantage of this situation and can achieve up to 4 commits at the same time. Moreover, when a commit is issued, half of the times there is already another commit being performed on another core. STCC-SEQ does not reach that level of parallel commits. Unfortunately, transaction TID0 dominates execution time and no meaningful benefits are obtained for STCC-SEQPRO. On the other hand, GCommit accelerates commits in TID0. Moreover, reducing *precommitting* time for this transaction leads to much less contention, reducing the number of aborts and improving overall performance (reduction of 20 %) despite not allowing parallel commits.

**Kmeans (high/low)** Despite the fact that this benchmark is mainly non-transactional, there are some differences in *Kmeans-high*. In this case, there are three transactions but only TID0 and TID1 represent important fractions of transactional execution time. TID0 has to access four banks (three for reading, one for writing), whilst TID1 only needs one bank (for writing). Both transactions can commit in parallel in case of STCC-SEQ and STCC-SEQPRO. Nevertheless, there is no possibility of parallel commits between transactions with the same TID running on different cores. In case of TID0, the bank booked for writing is always the same, the last one (number 15). This benefits STCC-SEQPRO since transactions TID0 running on different cores only fight for the last bank (transactions in STCC-SEQ need to fight for at least three additional banks before being allowed to commit). Nevertheless, *Kmeans* is a highly concurrent application, so the efficiency of GCommit exceeds its lack of parallel commits when a single *G-Arbiter* is employed.

**Labyrinth** Results for this workload depend significantly on the interleaving of executing threads. Its most important transaction (TID1) presents large write sets (more than 200 addresses) and a long execution time. An abort is extremely costly. Due to the long execution time of TID1, there is not much contention and a transaction that reaches the *precommitting* phase usually does not have to compete against others. Nevertheless, when a transaction commits, it is frequent that other transactions must be aborted. Transactional execution phases (*xact_useful* and *xact_wasted* bars) dominate overall execution time, and the election of the commit algorithm is not as important.

**Ssca2** It has a large number of tiny transactions and transaction TID2 is which dominates execution time. This transaction has three read addresses and two write ones in its sets. The read set is always mapped to the same directory bank, so there is no chance of parallel commits with STCC-SEQ. Nevertheless, *Ssca2* spreads its write set between ten directory banks. Hence, STCC-SEQPRO is able to achieve up to four parallel commits at the same time and almost two-thirds of the commits are in parallel. Even the fact that GCommit does not allow parallel commits for one *G-Arbiter*,

it outperforms STCC-SEQ and STCC-SEQPRO. In this application, commits are frequent and the lower latency of the mechanism implemented in GCommit drastically improves commit bandwidth.

**Vacation (high/low)** This benchmark does not exhibit real conflicts. Its main transaction has a moderate write set (6.7 addresses) and a large read set (96.5 addresses). The number of directory banks booked for writing and the low level of contention eliminates any possibility of parallel commits. Only the fast commits enabled by GCommit can make *precommitting* time disappear.

**Yada** It has a large working set and exhibits high contention. The dominant transaction (TID2) spreads its large write set (69.3 addresses) among all L2 banks, hindering parallel commits. Many transactions in the *precommitting* phase are aborted and hence, *precommitting* time in this application is much less significant than the *xact_wasted* time.

#### *4.1.1 Performance with 16 G-Arbiters*

Until now we have assumed just one *G-Arbiter*, neglecting the positive effects that parallel commits could have on performance for GCommit. Next, we perform a brief analysis of GCommit with 16 *G-Arbiters*, one per directory bank.[1] This configuration mimics the ability of SCTCC-SEQ to book a subset of the directory banks, and thus, enables parallel commits. The main downside is that more than one *G-Arbiter* must be acquired, which increases the duration of the *precommitting* phase. Figure 5 shows a comparison between the execution times that are obtained with one *G-Arbiter* (left bar for each application) and those that are reached when 16 *G-Arbiters* are considered (right bar). Results are normalized to the first case. As it can be seen, there are no noticeable performance differences between these two configurations. When a single *G-Arbiter* is considered, GCommit can complete the *precommitting* phase with extremely low latency, and thus, the number of transactions blocked at *precommitting* is very small. This provokes that the possibility of parallel commits is lower for GCommit than for STCC-SEQ. On the other hand, because of the particularities described above, GCommit with 16 *G-Arbiters* only achieves parallel commits in *Genome*, *Kmeans* and *Yada*. In these cases, we have observed that up to 2 parallel commits can be done at the same time. In *Genome* and *Yada*, just 0.5 and 1.5 % of the transactions coincide with another one while committing. This percentage grows for *Kmeans-high* and *Kmeans-low* (24 and 11 %, respectively). Unfortunately, *precommitting* time is insignificant in both cases. Therefore, it is more important to ensure short *precommitting* time (as done with one *G-arbiter*) than enabling parallel commits at the cost of increasing the time needed to acquire every *G-Arbiter*.

### 4.2 Energy consumption analysis

In this section, we compare STCC-SEQ, STCC-SEQPRO and GCommit in terms of the amount of dynamic energy consumed in each case. Figure 6 shows the dynamic

[1] The latency for acquiring each arbiter is the same than in the case of a single *G-Arbiter*

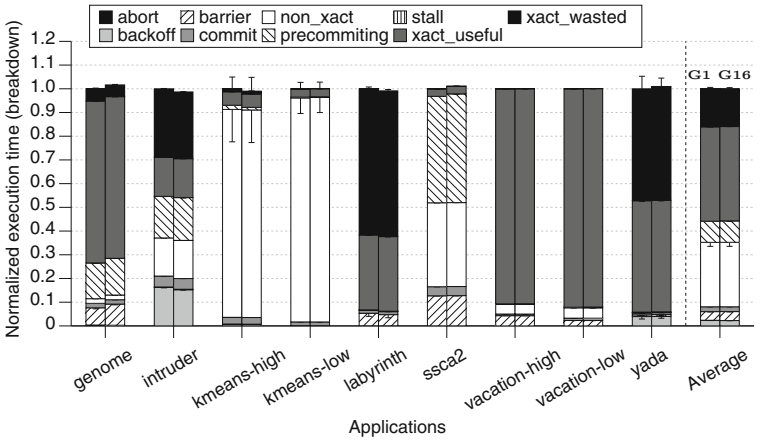



**Fig. 5** Breakdown of the execution times: 1 *G-Arbiter* (*left bars*) vs. 16 *G-Arbiters* (*right bars*)

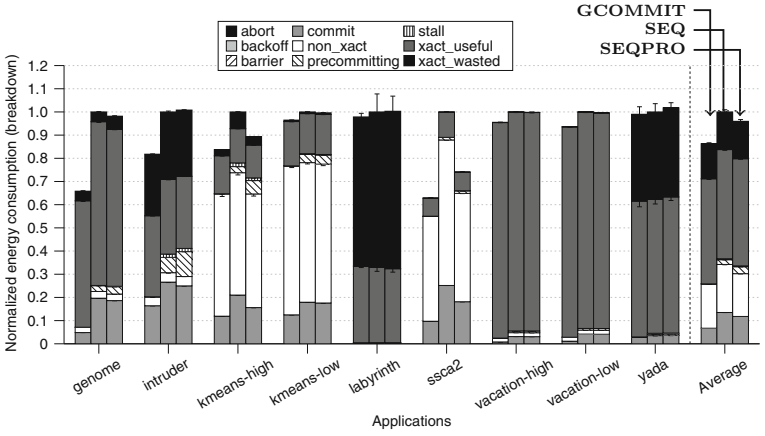



**Fig. 6** Breakdown of the energy consumed

energy consumption for the three systems considered in this work. As before, results have been normalized with respect to STCC-SEQ. Additionally, we split the energy consumed in each case assuming the same categories than in Sect. 4.1. To do so, we track the amount of energy consumed during each one of the categories. For messages sent through the general-purpose interconnection, we track data accesses in any cache structure as well as other possible messages generated because of the first one, and accumulate all this energy consumption into the corresponding execution phase of the transaction that issued the original message.

As with execution time, our proposal significantly improves overall energy consumption compared to STCC-SEQ and, as shown in Fig. 6, average reductions of 13.7 % are obtained. The energy due to the duration of the *precommitting* phase is virtually eliminated with GCommit. Our proposal does not issue any messages on the general-purpose interconnect, and the energy consumed during the *precommitting* phase comes just from the much more efficient dedicated links and controllers that our

mechanism involves. Additionally, in both STCC-SEQ and SETCC-SEQPRO, several directory banks must be reserved and therefore much more work has to be done. As in Fig. 6, results have been normalized with respect to those obtained with STCC-SEQ. As it can be seen, GCommit entails smaller traffic levels (contention) than the other approaches (reductions of 10.6 % on average).

On the other hand, the amount of energy spent during *precommitting* is small in the case of STCC-SEQ and STCC-SEQPRO. The number of control messages across the network during this arbitration period is moderate in both protocols, so part of the improvements in energy consumption comes from the reduction in the number of aborts that GCommit entails. This is why the amount of energy due to backoff is reduced in GCommit.

### 4.3 Network traffic analysis

Figure 7 shows the levels of traffic in the interconnection network (measured as flit per cycle) for the three commits algorithms: GCOMMIT, SEQ and SEQPRO. In general, as explained, GCommit does not generate any messages on the general-purpose interconnection network during precommitting. This keeps interconnection network less over-saturated during the smaller execution time of the applications. Benchmarks with less traffic levels are those whose precommitting phase acquires higher relevance: genome, intruder, kmeans and ssca2. These traffic levels are supported in these benchmarks by the number of aborts too, which are smaller with GCOMMIT.

## 5 Related work

Research in HTM has been very active since the introduction of multicores in mainstream computing. The initial proposal by Herlihy et al. [27] was revived in the previous decade with more sophisticated designs such as UTM [28], TCC [29], Bulk [11] or LogTM-SE [12]. HTM systems have been traditionally classified into two categories according to the approaches to version management (VM) and conflict detection (CD) that they implement: *Eager-Eager* (eager VM and eager CD) and *Lazy-Lazy* (lazy

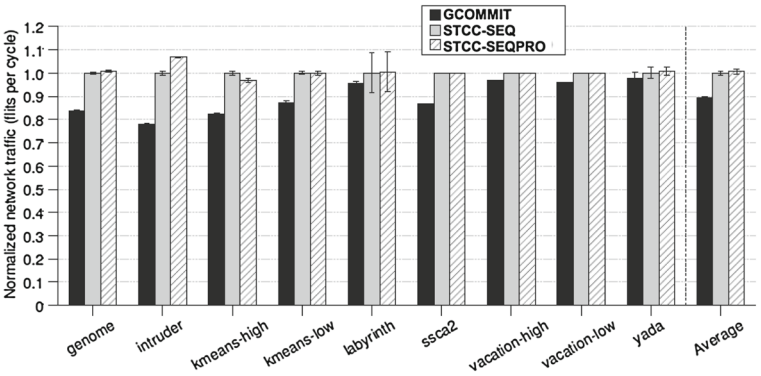



**Fig. 7** Normalized network traffic

VM and lazy CD). Although *Lazy-Lazy* HTMs have been identified as being more efficient than eager designs [30], the necessity of en-masse publication of updates at a transaction commit raises issues of scalability and several hybrid approaches such as EazyHTM [9], ZEBRA [31] or FlexTM [32] have been recently proposed. Differently from these previous works, our proposal builds directly on the simpler *Lazy-Lazy* HTM model and deals with the heart of the problem by deploying fast and efficient dedicated hardware to reduce commit time.

TCC [29] has probably been the most seminal *Lazy-Lazy* system. The original design which was based on a bus was later adapted to a more scalable architecture that uses directory-based coherence, giving rise to Scalable-TCC [17]. Subsequently, Pugsley et al. proposed STCC-SEQ and STCC-SEQPRO to significantly reduce commit time in Scalable-TCC [16]. Both STCC-SEQ and STCC-SEQPRO are free of deadlocks/livelocks, do not employ a centralized agent and allow for parallel commits. In this work, we study a hardware implementation of an algorithm similar to STCC-SEQ and show that to improve commit bandwidth it is more important to reduce *commit transfer time* (time that goes since a transaction completes commit until another one gets permission to commit) than enabling parallel commits.

Regarding energy consumption in the context of TM, Klein et al. [33] 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. [34] 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. [35] 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. [36] present a simple and energy-efficient TM design for embedded architectures, at the cost of performance. In the context of Eager-Eager HTMs, Gaona et al. [37] try to save energy by serializing transactions' execution in scenarios of high contention without hurting performance. In [38], the two well-known HTM systems (namely, *Eager-Eager* LogTM-SE system [12] and *Lazy-Lazy* Scalable-TCC system [16,17]) are compared in terms of energy consumption. On the other hand, in [39], Cristal et al. show a case use of how efficient 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.

Finally, GCommit is based on the use of Global lines (*G-lines*). A *G-line* is basically a shared wire that broadcasts 1-bit messages (signals) across one dimension of the chip in a single clock cycle. Practical uses of *G-lines* are presented by Khrisna et al. [40] to enhance a flow control mechanism (EVC) in terms of latency and power consumption in the context of networks-on-chip (NoC), and by Abellán et al. to accelerate lock and barrier synchronization and reduce power consumption in manycore CMPs [18,41].

## 6 Conclusions and future ways

In this work, we have presented GCommit, a new hardware approach to commit algorithms in *Lazy-Lazy* HTM systems. Instead of trying to achieve more parallelism at commit time, our proposal focuses on reducing the duration of the commit events. More specifically, our proposal dramatically accelerates the *precommit* subphase of a commit, which ensures compliance with the serializability and atomicity properties by enforcing commit ordering between transactions with clashing read and write sets. GCommit is implemented using dedicated global links and a set of controllers. The links are used by committing transactions to both ask for and receive authorization to commit. For that, a dedicated lightweight on-chip global-link-based network is able to manage commit requests and acknowledgements. The controllers are in charge of selecting just one of the transactions aimed to commit. Compared to previously proposed commit protocols that use the general-purpose interconnection network to coordinate commit ordering, our proposal reduces the number of cycles of the *precommitting* phase to a minimum (less than 10 cycles). This results into increased commit bandwidth.

We use full-system simulations of a 16-core CMP running several STAMP applications to evaluate GCommit, and compare it with STCC-SEQ and STCC-SEQPRO systems (the highest performance commit protocols proposed in the literature). We find that GCommit accelerates the *precommit* subphase about 68.5 % on average, which results in average reductions of 15.7 and 13.7 % in terms of execution time and energy consumption, respectively, compared with STCC-SEQ. To quantify hardware costs of our proposal, we implement GCommit using a cost-effective mainstream industrial synthesis toolflow. Our study reveals that negligible requirements in terms of area are needed (0.07 % of the whole chip for the target 16-core CMP system) that leads to negligible power requirements. We have also explored the simultaneous usage of different GCommit architectures in the same CMP system, so that we can increase parallelism when committing different transactions at a particular time. We have found that, due to the particularities of transactional applications, it is more important to ensure short *precommitting* time than enabling parallel commits at the cost of increasing the duration of the process (e.g., using slower *G-Arbiter* acquisition).

As part of our future work, we plan to study the possibility of extending our proposal with the parallel reader optimization without increasing commit latency and while keeping the design simple. Furthermore, this work has demonstrated the important benefits that using a specialized NoC (in conjunction with the main NoC) can bring in the context of Lazy-Lazy HTMs. We are considering also the possibility of applying the same approach for accelerating critical aspects of Eager-Eager HTMs. Finally, we are also considering larger multicore architectures (manycores).

**Acknowledgments** This work has been supported by the Spanish MINECO under grant "TIN2012-38341-C04-03" and by Fundación Séneca, Agencia Regional de Ciencia y Tecnología de la Región de Murcia under grant "19295/PI/14". 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. Anthony S (2013) Intel unveils 72-core x86 knights landing cpu for exascale supercomputing. Available http://www.extremetech.com/extreme/171678-intel-unveils-72-core-x86-knights-landing
3. Corporation T (2014) Tile-gx8072 processor
4. Dice D, Shalev O, Shavit N (2006) Transactional locking II. In: DISC-20
5. Fraser K, Harris T (2007) Concurrent programming without locks. ACM Trans Comput Syst 25(2):5. doi:10.1145/1233307.1233309
6. Marathe VJ, Scherer-III WN, Scott ML (2005) Adaptive software transactional memory. In: DISC-19
7. Herlihy M, Luchangco V, Moir M, Scherer-III WN (2003) Software transactional memory for dynamic-sized data structures. In: PODC-22
8. 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
9. 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
10. Rajwar R, Herlihy M, Lai KK (2005) Virtualizing transactional memory. In: ISCA-32
11. Ceze L, Tuck J, Torrellas J, Cascaval C (2006) Bulk disambiguation of speculative threads in multiprocessors. In: ISCA-33
12. 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
13. 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
14. Sanyal S, Roy S, Cristal A, Unsal OS, Valero M (2009) Dynamically filtering thread-local variables in lazy-lazy hardware transactional memory. In: 11th IEEE international conference on high performance computing and communications. HPCC'09. IEEE, New York, pp 171–179
15. Dice D, Lev Y, Moir M, Nussbaum D (2009) Early experience with a commercial hardware transactional memory implementation. In: ASPLOS-14
16. Pugsley SH, Awasthi M, Madan N, Muralimanohar N, Balasubramonian R (2008) Scalable and reliable communication for hardware transactional memory. In: PACT-17
17. 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
18. Abellán JL, Fernández J, Acacio ME (2013) Design of an efficient communication infrastructure for highly contended locks in many-core CMPS. J Parallel Distrib Comput 73(7):972–985
19. Gaona E, Abellán JL, Acacio ME, Fernández J (2013) Deploying hardware locks to improve performance and energy efficiency of hardware transactional memory. In: Architecture of computing systems-ARCS. Springer, Berlin, pp 220–231
20. Cadence, SoC Encounter. http://www.cadence.com/
21. 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
22. Magnusson PS, Christensson M, Eskilson J, Forsgren D, Hallberg G, Hogberg J, Larsson F, Moestedt A, Werner B (2002) Simics: a full system simulation platform. IEEE Comput 35:50–58
23. Kahng AB, Li B, Peh LS, Samadi K (2009) ORION 2.0: a fast and accurate NoC power and area model for early-stage design space exploration. In: DATE-13
24. HP Labs. http://quid.hpl.hp.com:9081/cacti
25. Minh CC, Chung J, Kozyrakis C, Olukotun K (2008) STAMP: stanford transactional applications for multi-processing. In: IISWC-4
26. Dragojevic A, Guerraoui R (2010) Predicting the scalability of an STM. In: Transact-05
27. Herlihy M, Moss JEB (1993) Transactional memory: architectural support for lock-free data structures. SIGARCH CAN 21(2):289–300
28. Ananian CS, Asanovic K, Kuszmaul BC, Leiserson CE, Lie S (2005) Unbounded transactional memory. In: HPCA-11
29. Hammond L, Wong V, Chen MK, Carlstrom BD, Davis JD, Hertzberg B, Prabhu MK, Wijaya H, Kozyrakis C, Olukotun K (2004) Transactional memory coherence and consistency. In: ISCA-31
30. Shriraman A, Dwarkadas S, Scott ML (2008) Flexible decoupled transactional memory support. In: ISCA-35

31. Titos JR, Negi A, Acacio ME, García JM, Stenström P (2011) ZEBRA: a data-centric, hybrid-policy hardware transactional memory design. In: ICS-25
32. 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
33. Klein F, Baldassin A, Araujo G, Centoducatte P, Azevedo R (2009) On the energy-efficiency of software transactional memory. In: SBCCI-22
34. Moreshet T, Bahar RI, Herlihy M (2006) Energy-aware microprocessor synchronization: transactional memory vs. locks. In: Workshop on memory performance issues
35. Sanyal S, Roy S, Cristal A, Unsal O, Valero M (2009) Clock gate on abort: towards energy-efficient hardware transactional memory. In: HPPAC-2009
36. 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
37. Gaona-Ramírez E, Titos-Gil JR, Fernández J, Acacio ME (2014) Selective dynamic serialization for reducing energy consumption in hardware transactional memory systems. J. Supercomput. 68(2):914–934
38. Gaona-Ramírez E, Titos-Gil JR, Fernández J, Acacio ME (2013) On the design of energy-efficient hardware transactional memory systems. Concurr Comput Pract Exper 25(6):862–880
39. Cristal A, Unsal O, Yalcin G, Fetzer C, Wamhoff JT, Felber P, Harmanci D, Sobe A (2013) Leveraging transactional memory for energy-efficient computing below safe operation margin. In: TRANSACT-2013
40. Krishna T, Kumar A, Peh L-S, Postman J, Chiang P, Erez M (2009) Express virtual channels with capacitively driven global links. IEEE Micro 29(4):48–61
41. Abellán JL, Fernández J, Acacio ME (2012) Efficient hardware barrier synchronization in many-core cmps. IEEE Trans Parallel Distrib Syst 23(8):1453–1466