Int J Parallel Prog (2018) 46:1304–1328
https://doi.org/10.1007/s10766-018-0569-7

# Hardware Transactional Memory Exploration in Coherence-Free Many-Core Architectures

**Dimitra Papagiannopoulou[1] · Andrea Marongiu[2,3] · Tali Moreshet[4] · Luca Benini[2,3] · Maurice Herlihy[1] · R. Iris Bahar[1]**

Received: 5 May 2015 / Accepted: 1 April 2018 / Published online: 4 April 2018


**Abstract** High-end embedded systems, like their general-purpose counterparts, are turning to many-core cluster-based shared-memory architectures that provide a shared memory abstraction subject to non-uniform memory access costs. In order to keep the cores and memory hierarchy simple, many-core embedded systems tend to employ simple, scratchpad-like memories, rather than hardware managed caches that require some form of cache coherence management. These "coherence-free" systems still

This work is supported in part by the United States National Science Foundation under Grants CNS-1319495, CNS-1319095, and CNS-1301924. Additional support was provided by European Union projects FP7 P-SOCRATES (g.a. 611016) and ERC MULTITHERMAN (g.a. 291125). A preliminary version of this work appeared in [20].

✉ R. Iris Bahar
iris_bahar@brown.edu

Dimitra Papagiannopoulou
dimitra_papagiannopoulou@alumni.brown.edu

Andrea Marongiu
a.marongiu@iis.ee.ethz.ch; a.marongiu@unibo.it

Tali Moreshet
talim@bu.edu

Luca Benini
lbenini@iis.ee.ethz.ch; luca.benini@unibo.it

Maurice Herlihy
mph@cs.brown.edu

1 Brown University, Providence, RI 02912, USA

2 ETH Zurich, 8092 Zurich, Switzerland

3 DEI — University of Bologna, 40136 Bologna, Italy

4 Boston University, Boston, MA 02215, USA

require some means to synchronize memory accesses and guarantee memory consistency. Conventional lock-based approaches may be employed to accomplish the synchronization, but may lead to both usability and performance issues. Instead, speculative synchronization, such as hardware transactional memory, may be a more attractive approach. However, hardware speculative techniques traditionally rely on the underlying cache-coherence protocol to synchronize memory accesses among the cores. The lack of a cache-coherence protocol adds new challenges in the design of hardware speculative support. In this article, we present a new scheme for hardware transactional memory (HTM) support within a cluster-based, many-core embedded system that lacks an underlying cache-coherence protocol. We propose two alternative data versioning implementations for the HTM support, *Full-Mirroring* and *Distributed Logging* and we conduct a performance comparison between them. To the best of our knowledge, these are the first designs for speculative synchronization for this type of architecture. Through a set of benchmark experiments using our simulation platform, we show that our designs can achieve significant performance improvements over traditional lock-based schemes.

**Keywords** Transactional memory · Embedded systems · Parallel processing · Coherence-free memory architectures

## 1 Introduction

High-end embedded systems, like their general-purpose counterparts, are turning to many-core cluster-based shared-memory architectures that are subject to non-uniform memory access (NUMA) costs. Memory organization is the single, most far-reaching design decision for such architectures, both in terms of raw performance, and (more importantly) in terms of programmer productivity.

For many-core embedded systems, in order to meet stringent area and power constraints, the cores and memory hierarchy must be kept simple. In particular, scratchpad memories (SPM) are typically preferred to hardware-managed data caches, which are far more area- (40%) and power-hungry (34%) [1]. Several many-core embedded systems have been designed without the use of caches and cache-coherence (the Epiphany IV Processor from Adapteva [1] and ST Microelectronics p2012/STHORM [13] are two examples). These kind of platforms are becoming increasingly common.

Shared-memory multi-core and many-core systems require a way to synchronize memory accesses and guarantee memory consistency. The conventional approach to shared-memory synchronization is to use locks, but locks suffer from well-known performance and usability limitations. Speculative approaches such as Transactional Memory [7] and Speculative Lock Elision [21] are attractive alternatives to locking, improving both performance and energy consumption. We are interested here in speculative synchronization mechanisms supported by hardware. In particular, we focus on Hardware Transactional Memory (HTM).

As embedded systems move to many-core and cluster-based architectures, the design of high-performance, energy-efficient synchronization mechanisms becomes more and more important. Yet, speculative synchronization for such embedded sys-

tems has received little attention. Moreover, implementing speculative synchronization in embedded systems that lack cache-coherence support is particularly challenging, since hardware speculative techniques traditionally rely on the underlying cache-coherence protocol to synchronize memory accesses among the cores. For these cacheless systems, a completely new approach is necessary for handling speculative synchronization. Therefore, the goal of our work is to present the *first ever* implementation of hardware transactional memory (HTM) support within a cluster-based system that lacks an underlying cache-coherence protocol. As we shall describe later, this implementation requires explicit data management and implies a fully-custom design of the transactional memory support.

Speculation for embedded devices raises two challenges:

- How to keep the underlying hardware and software interfaces simple enough to use.
- How to scale to cluster-based architectures that do not provide cache coherence, but do provide memory hierarchies encompassing scratchpad memories (SPM), which can be thought of as software-maintained caches.

The lack of cache-coherence brings major challenges in the design of HTM support, which needs to be designed from scratch. At the same time though, the lack of cache coherence provides a significant benefit: A more lightweight and simple environment to build upon, that could be more appropriate for the embedded systems domain. Building on such an environment, we create a novel hardware transactional memory design that is self-contained, and does not rely on an underlying cache coherence protocol to provide synchronization and safety guarantees.

To sum up, the main contributions of this work are the following:

1. We design from scratch a Hardware Transactional Memory (HTM) scheme within a cluster-based, many-core embedded architecture, that does not rely on an underlying cache coherence protocol to manage read/write memory conflicts. We propose novel hardware support to keep track of read/write memory accesses and guarantee execution correctness. To the best of our knowledge, this is the first design for speculative synchronization in this type of architecture.
2. We provide full speculative synchronization support for multiple transactions accessing data within a single cluster. While our current implementation is limited to single-cluster accesses, our proposed scheme is designed such that it is scalable and can be extended to multiple clusters. We introduce the idea of distributing the synchronization management, which makes it inherently scalable.
3. We provide two alternative implementations of the Transactional Memory scheme, proposing a *Full-Mirroring* and a *Distributed Logging* scheme for data versioning management. We perform an overhead analysis of the proposed designs and a performance comparison based on simulations.
4. We investigate how our proposed HTM support scheme can improve the performance of multi-threaded applications on a simulation platform for a coherence-free, many-core embedded system. Through a set of simulations on data structure applications, benchmarks from the STAMP benchmark suite [15] and on the Eigenbench application [8], we show that our scheme can achieve significant performance improvements over traditional lock-based schemes.

The rest of this article is organized as follows. In Sect. 2 we provide background on shared memory synchronization techniques, including speculative techniques designed on top of cluster-based architectures. In Sect. 3, we describe our underlying simulation platform used to model a cluster-based embedded architecture with no cache coherence. Next, we describe in Sect. 4 our proposed hardware implementation that supports hardware transactional memory within our simulation platform. Simulation results showing the viability and potential benefits of our proposed scheme are reported in Sect. 5, followed by conclusions in Sect. 6.

## 2 Background

As the demand for more compute-intensive capabilities for embedded systems increases, multi-core embedded systems are evolving into many-core systems in order to achieve improved performance and energy efficiency, similar to what has happened in the high-performance computing (HPC) domain. The memory is then shared across these multiple cores; however, the specific memory organization of these many-core systems has a significant impact on their potential performance. For small-scale systems, shared-bus architectures are attractive for their simplicity, but they do not scale. For large-scale systems, architects have embraced hierarchical structures, where processing elements are grouped into clusters, interconnected by a scalable medium such as a network-on-chip (NoC). In such systems [11,17], cores within a cluster communicate efficiently through common on-chip memory structures, while cores in different clusters communicate less efficiently through bandwidth-limited higher-latency links. Architectures that provide the programmer with a shared-memory abstraction, but where memory references across clusters are significantly slower than references within the clusters, are commonly referred to as *non-uniform memory access* (NUMA) architectures.

With a shared memory structure, multiple cores may try to modify the same block of memory. Therefore, it is important to employ some means to guarantee memory consistency. Locks are typically used to guarantee memory consistency in shared memory systems. Locks, however, can slow performance and consume excessive energy, as they typically require energy-expensive read-modify-write operations that traverse the memory hierarchy. In addition, locks must be deployed conservatively whenever conflicts are possible, even if they are very unlikely. By contrast, *speculative* synchronization mechanisms detect conflicts dynamically, rolling back and retrying computations when conflicts actually occur, instead of delaying computations that might encounter them. Speculative synchronization mechanisms have been the subject of much prior work, but such work has tended to focus on throughput and ease of use. While we share these goals, we feel that embedded systems demand an equal focus on architectural simplicity and scalability.

Transactional memory [7,16], Lock Elision [21], and Transactional Lock Removal (TLR) [22] are hardware speculation techniques that allow critical sections without run-time conflicts to execute in parallel. If a data conflict does take place, it is detected, and one or more of the conflicting threads is rolled back and restarted.

When designing speculative synchronization mechanisms for embedded devices, it is essential to keep both the underlying hardware and the software interface simple and scalable. Prior transactional memory proposals restrict speculative computations to the L1 (or sometimes L2) caches, relying on native cache-coherence protocols to detect conflicts. Some prior proposals for speculative synchronization in embedded devices considered only shared-bus single-cluster architectures [5,18]. While popular for their simplicity, such bus-based architectures are inherently not scalable, because the bus becomes overloaded when shared by more than a handful of processors. As a result, it is necessary to rethink the design of speculative mechanisms for scalable, cluster-based embedded systems where inter-cluster communication is restricted.

Recently, Intel [10] and IBM [2] announced new processors with direct hardware support for speculative transactions, and it seems likely that others will follow suit. The IBM transactional memory mechanism, like ours, is intended for a clustered architecture.

Prior designs for hardware transactional memory based on network-on-chip (NoC) communication include Kunz et. al [12], who describe a LogTM [16] implementation on a NoC architecture, and Meunier and Petrot [14], who describe a novel embedded HTM implementation based on a write-through cache coherence policy. Like us, these researchers propose speculation mechanisms for many-core embedded NoC systems, but unlike us, they build on top of an underlying cache coherence protocol. We eschew such an approach from the belief that cache coherence will become more and more unwieldy as cluster sizes grow, so we think it is time to consider speculative synchronization mechanisms that do not rest on inherently unscalable foundations.

## 3 Target Architecture

Our work is based on a virtual platform environment called VSoC, which simulates a cluster-based many-core architecture at a cycle-accurate level [3]. Like recent many-core chips such as Kalray MPPA256 [11], ST Microelectronics p2012/STHORM [13], and even GPGPUs such as NVIDIA Fermi [17], the VSoC platform encompasses multiple computing clusters, and is highly modular. These systems achieve scalability through a hierarchical design. Simple processing elements (PE) are grouped into small-to-medium sized subsystems (*clusters*) sharing a high-performance local interconnect and memory. In turn, clusters are replicated and interconnected with a scalable network-on-chip (NoC) medium, as depicted in Fig. 1.

Figure 2 shows the basic cluster architecture. Each cluster consists of a configurable number, $N$, of 32-bit ARMv6 RISC processors,[1] one L1 private instruction cache for each of the $N$ processors, and a shared multi-ported and multi-banked tightly coupled data memory (TCDM). Note that the TCDM is not a cache, but a first level Scratchpad Memory (SPM) structure. As such, it is not managed by hardware but with software and it lacks cache coherence support. The TCDM is partitioned into $M$ banks, where all banks have the same memory capacity. For the ARMv6 processor models we use

[1] The original simulator was designed with ARMv6 processors. Using a later version would not change our expected observations.

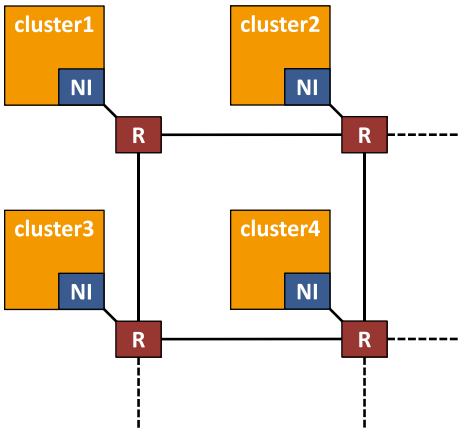



**Fig. 1** Hierarchical design of our cluster-based embedded system

the Instruction Set Simulator in [6] wrapped in a SystemC module. A logarithmic interconnect supports communication between the processors and the TCDM banks. The TCDM can handle simultaneous requests from each processor in the cluster. If all requests are to different banks, those requests are serviced simultaneously. Otherwise requests to the same bank are serialized. An off-chip Main Memory is also available. Note that this is not part of the cluster, but each cluster's cores can access it through an off-cluster Main Memory bus (see Fig. 2).

The logarithmic interconnect is a mesh-of-trees (Fig. 3) and provides fine-grained address interleaving on the memory banks to reduce banking conflicts. The latency of traversing the interconnect is normally one clock cycle. If multiple processor cores are requesting data that reside within different TCDM banks, then the data routing is done in parallel. This allows the cluster to maintain full bandwidth for the communication between the processors and the memories. In addition, when multiple processors are reading the same data address simultaneously, the network can broadcast the requested data to all readers within a single cycle.

However, if multiple processor cores are requesting different data that reside within the same TCDM bank, conflicting requests will occur, which will trigger a round-robin scheduler to arbitrate access for fairness. In this case, additional cycles will be needed to service all data requests. Specifically, the conflicting requests will be serialized, but with no additional latency between consecutive requests. In order to reduce banking conflicts, the number of banks $M$ should be an integral multiple of the number of cores $N$. In our case, we chose $M$ to be equal to the maximum number of cores that the cluster can accommodate, that is 16.

Regardless of whether the processors within the cluster requested conflicting requests or not, when a memory access request arrives at a bank interface, the data is available on the negative edge of the same clock cycle. Hence the latency for a TCDM access that has not experienced conflicts is two clock cycles [3]: one cycle for traversing the interconnect in each direction.

A stage of demultiplexers between the logarithmic interconnect and the cores selects whether the memory access requests coming from the processors are for the main memory or for the TCDM. Accesses to memory external to a cluster go through a peripheral

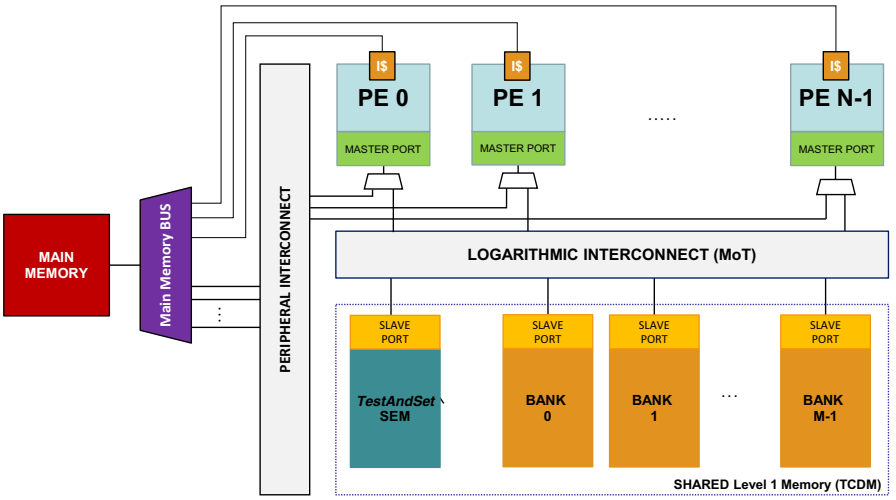



**Fig. 2** Single cluster architecture of target platform

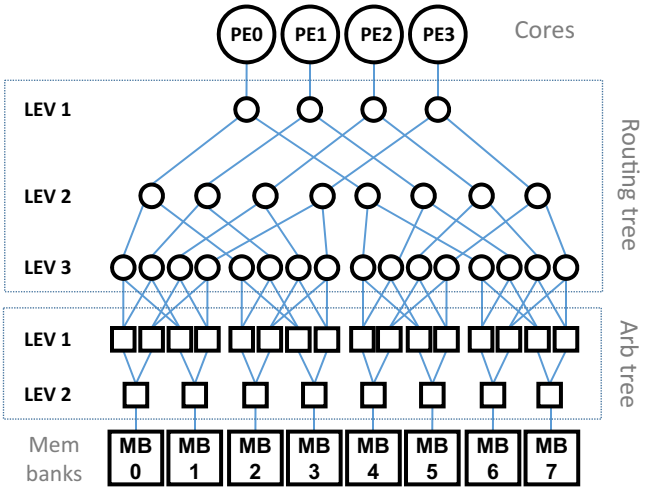



**Fig. 3** A 4 × 8 Mesh of Trees. Circles represent routing and arbitration switches. Taken from [3]

interconnection. The off-cluster (main memory) bus coordinates the accesses and services requests in round-robin fashion.

The basic mechanism for processors to synchronize is standard read/write operations to a dedicated memory space, which provides *test-and-set* semantics (a single atomic operation returns the content of the target memory location and updates it). We use this memory for locking, the baseline against which we compare our transactional memory design.

Note that cores within the cluster do not have private data caches or memories (just private per-core instruction caches). Instead, all data accesses go through the TCDM. The absence of coherent data caches implies a completely custom design of the

transactional memory support. Indeed, speculative synchronization through hardware transactional memory generally relies on the underlying cache coherency protocol to manage conflict detection. Instead, we will have to employ a different mechanism to achieve this. This mechanism will be explained in more detail in Sect. 4.

## 4 Implementation

Designing a hardware transactional memory system for the cluster-based VSoC system presents novel challenges. Prior HTM proposals relied on the cache hierarchy to buffer intermediate results, and on a native, bus-based cache-coherence protocol for conflict detection. Neither of these options are available on VSoC, so we must rethink the HTM design from scratch.

Instead of buffering tentative updates in an L1 cache like prior designs, we chose to integrate the HTM mechanism with the TCDM memory, implying that the TCDM memory holds both speculative and non-speculative data. In prior designs for small-scale embedded devices [4], a centralized unit would snoop on bus transactions, detect data conflicts, and resolve them (i.e., decide which of the conflicting transactions should be aborted). Monitoring all ongoing traffic in a shared bus environment is fairly easy, since only one transaction can traverse the shared bus medium at each cycle. However, in the VSoC system, the logical interconnect permits multiple transactions to traverse the interconnect in the same cycle. Since interconnect access is concurrent, not serial, snooping on the cluster interconnect is not feasible. Serializing and routing transactional memory traffic through a centralized module would create a substantial sequential bottleneck and drastically limit scalability.

For these reasons, we concluded that transactional management must be distributed to be scalable. Thus, we divide conflict detection and resolution responsibilities across the TCDM memory banks, into multiple *Transaction Support Modules (TSM)*. By placing a transactional support module at each bank of the TCDM, we allow conflict detection and resolution mechanisms to be decentralized. In this way, transactional management bandwidth should scale naturally with the number of banks. Our design consists of three parts, *Transactional Bookkeeping*, *Data Versioning* and *Control Flow*. We will describe each of these in more detail next.

### 4.1 Transactional Bookkeeping

Each bank's TSM intercepts all memory traffic to that bank and is aware of which cores are executing transactions. When a TSM detects a conflict, it decides which transaction to abort, and notifies the appropriate processor. This *transactional bookkeeping* keeps track of transactional readers and writers. For each data line, there can be multiple transactions reading that line, or a single transaction writing it, which we call the line's *owner*. Each bank keeps track of which processors have transactionally accessed each data line through a per-bank array of $k$ $r$-bit vectors, where $k$ is the number of data lines at that bank, and $r = 1 + N + log_2 N$, where $N$ is the number of cores in the cluster. The first bit indicates whether the line has been written transactionally, and if so, the next $log_2 N$ bits identify the owner. The next $N$ bits indicate which processors


**Fig. 4** Bookkeeping example. At time t1 address location A had not been read or written. By time t2, cores 1, 7, 8, and 13 have read the address. At time t3 core 13 writes the address and generates a conflict. So, core 13 will be aborted and its read flag will be cleared. Since core 13 was also the writer of address location A, the Writer ID bits and the Owner bit will be cleared as well

are transactionally reading that line. For example, for $N = 16$ cores, a 21-bit vector is needed, as shown in Fig. 4. The transactional support mechanism is integrated within each bank of the TCDM memory.

When a transaction accesses a bank, the bank's TSM checks the corresponding vector to determine whether there is a conflict. A transactional write to a memory location that is currently being read by another core will trigger a conflict (and vice versa). A transactional read to a memory location concurrently being read by other cores does not trigger a conflict.

### 4.2 Data Versioning

Any transactional memory design must manage *Data Versioning*, keeping track of speculative and non-speculative versions of data. Previous designs relied on data caches to buffer speculative data changes. Each core would keep track of its modifications using either a dedicated transactional cache or its private L1 cache. However, our target architecture does not have private L1 data caches, but a single shared TCDM memory structure that is distributed in multiple banks. Hence, we must rethink how speculative data should be saved. Using the main memory to buffer transactional data would not be a good choice, since that would require multiple off-cluster main memory accesses and could cause substantial delays. Thus, transactional data must be saved in on-cluster memory. Another option would be to add dedicated per-core memory structures on the cluster to store speculative data. However, since cores can access data located in different TCDM banks, redirecting and buffering these data to dedicated per-core structures could create a substantial bottleneck. Instead, we opt to store speculative data directly in the TCDM. Since the non-speculative data is distributed between the banks, we similarly distribute the speculative data.

Like conflict detection and resolution, data versioning can be eager or lazy. Lazy versioning leaves the old data in place and buffers speculative updates in different locations, while eager versioning makes speculative updates in place and stores back-

up copies of the old data elsewhere. Keeping the original data in place makes the abort scenario very fast, but it delays commits, since extra time is necessary to write the speculative data back to memory during commits. Eager data versioning on the other hand, makes commits faster but increases the abort recovery time, since the original data need to be fetched back to memory. For applications with low contention, hence low abort rates, eager data versioning is more attractive. However, care must be taken to make sure that the abort recovery time does not become a major bottleneck, even with lower abort rates.

We chose to use eager versioning for our design. Existing eager versioning designs such as LogTM [16], store original values in a software log that has a stack structure. In LogTM, a per-thread transaction log is created in cacheable virtual memory, which holds original data values and their virtual addresses, for all data modified by a transaction. In our platform, keeping per-thread transaction logs would have some significant drawbacks. First, the transaction log could reside anywhere in the memory, across multiple banks. This would imply that during every transactional memory access, the log would first need to be located and then traversed in order to find out whether the memory location has been accessed by that transaction. Moreover, the abort process would require traversing the log once more and restoring data back to their original locations, which would create data exchanges across different banks through the interconnect and hence significant delays. For these reasons, we propose two alternative data versioning designs that avoid this costly cross-bank data exchange: a *Full-Mirroring* and a *Distributed Logging* design. Both designs utilize the local per-bank TSMs for performing the transactional data saving and restoration processes. Next, we describe each of these designs in detail.

#### *4.2.1 Full-Mirroring*

This design is based on the idea that, for every address in the memory space, we create a mirror address in the same TCDM bank that holds the original data, to be recovered in case of abort. In this way, the restore process does not involve any exchange of data between different banks. Instead, the data saving and restoring process is triggered internally by the TSM of each bank, and it is completed by simply performing an internal bank check in the mirroring address without requiring interaction with other banks' TSMs. Although this solution consumes more space than keeping a dedicated per-transaction log, it yields a very simple and fast design.

When a transaction first writes an address, the TSM sends a request to the log space to record that address's original data. As shown in Fig. 5, the log space is designed so that each address's mirrored address is in the same bank. When a log needs to be saved, the TSM will trigger a write to the corresponding mirroring address in the bank. Note that only the first time that an address is written within a specific transaction, we need to log its original value. Hence, we pay the cost of writing to the log space only once for each address that is written during a transaction. If a transaction aborts, the data it overwrote is restored from the log.

Because each address and its mirror reside at the same bank, the latency overhead of recording the address's original value and restoring it on a transaction abort is quite modest (two extra cycles). No additional cost will be paid to search for the location

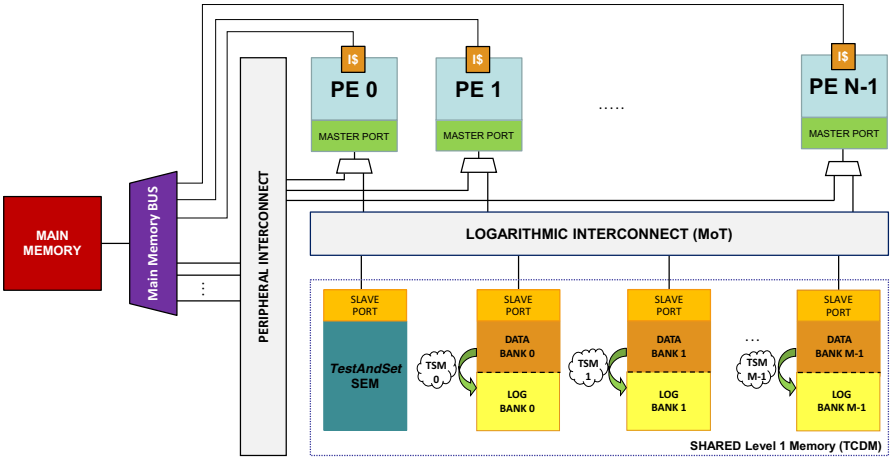



**Fig. 5** Modified single cluster architecture

where the address is logged, since each address's mirror lies at a certain location that can be found by a simple calculation and does not require traversing a log. Moreover, the use of an eager versioning scheme makes commits fast, since no data need to be moved. Thus, while full-mirroring has a significant area overhead (i.e., we need to utilize half of the TCDM memory to accommodate the log space), it has an advantage when it comes to simplicity. However, the fact that it uses memory less efficiently may require extra delay overhead to move data to/from the TCDM. In Sect. 5.1 we perform a detailed overhead comparison of full-mirroring with the alternative distributed logging scheme in terms of space and time.

### *4.2.2 Distributed Logging*

As just noted in the previous section, half of the available on-chip TCDM must be reserved for the mirror in our full-mirroring scheme, yet the amount of this memory space that will actually be used to save log data depends on the write footprint of the transactions and these typically cover only a small subset of the available memory space. Our second proposed data versioning design, *Distributed Logging*, offers a solution to the space inefficiency of the *Full-Mirroring* design, by using *distributed per-address logs* instead of mirrors. A preliminary version of this design was presented in [19], but no details were given on how it can be used for the purposes of traditional Transactional Memory and no simulation analysis was done on how it actually compares against the *Full-Mirroring* design.

Figure 6 depicts how the *Distributed Logging* design works. In this design, distributed per-address logs are used to save backups of the original values of data that are written during transactions, so that they can be recovered in case of aborts. Again, as in *Full-Mirroring* the transactional handling and log managing responsibilities are divided across the TSMs of the banks. Each bank's TSM monitors transactional accesses to the bank and manages the cores' logs that reside in that bank. It is also responsible for restoring the log data of the cores that abort their transactions and

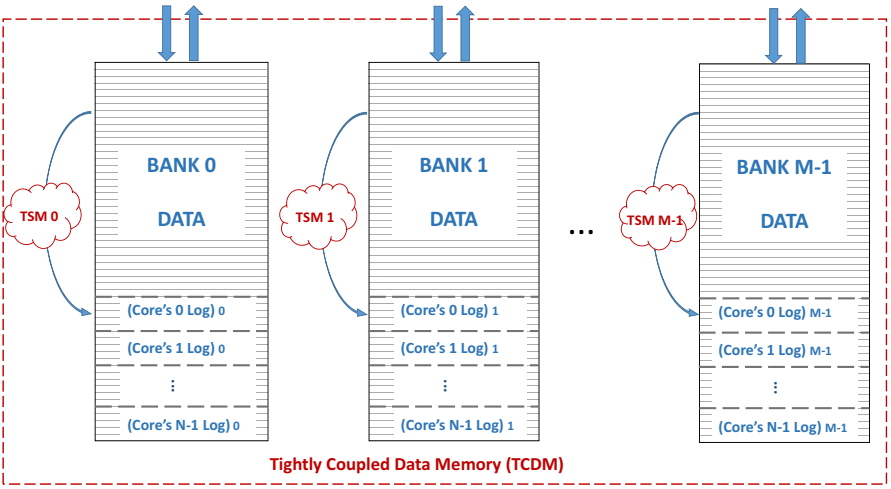



**Fig. 6** Distributed per-address log scheme for M banks and N cores

cleaning the logs of the cores that commit their transactions. Again, all banks' TSMs work in parallel and independently of one another.

At every bank of the TCDM, we keep a fixed-size log space for each core in the system. Each core's log holds the addresses that belong to that bank and are written transactionally by that core. In this way, we keep a log space only for the addresses of the bank that are actually written transactionally and not for all of them as in *Full-Mirroring*. At the same time, with this distributed logging design we still avoid cross-bank data exchange when saving and restoring the log, since each addresses' log falls within the same bank. Thus the log saving and restoration process is triggered internally by the TSM of each bank and it does not require interaction with the TSMs of other banks. This would not be feasible if we used per-thread transaction logs as it was proposed in [16].

When a core writes transactionally to an address of a bank, its log is traversed to check whether it already holds an entry for that address. If not, a new log entry is created to store the original data of the address. Note that the data only need to be logged the first time the address is written within a specific transaction. Therefore, the log size depends on the write footprint of each transaction. Since the log of each core is distributed among all the TCDM banks, we expect that the log writes will also be divided among the banks. The size of each core's log space per bank is a parameter in our design, so it can be easily adjusted to the needs of different applications domains. In case of an overflow, our technique resorts to software-managed logging into the main L2 memory. The capability of tuning the log size is intuitively key to reducing the number of overflows. If a conflict is detected and a transaction must abort, each bank's log is traversed to restore the original data back to its proper address. If a transaction commits, the logs associated with that transaction are all discarded and the speculative data now becomes non-speculative. In Sect. 5.1, we further detail the overhead analysis of each of our proposed data versioning schemes in terms of space and time.

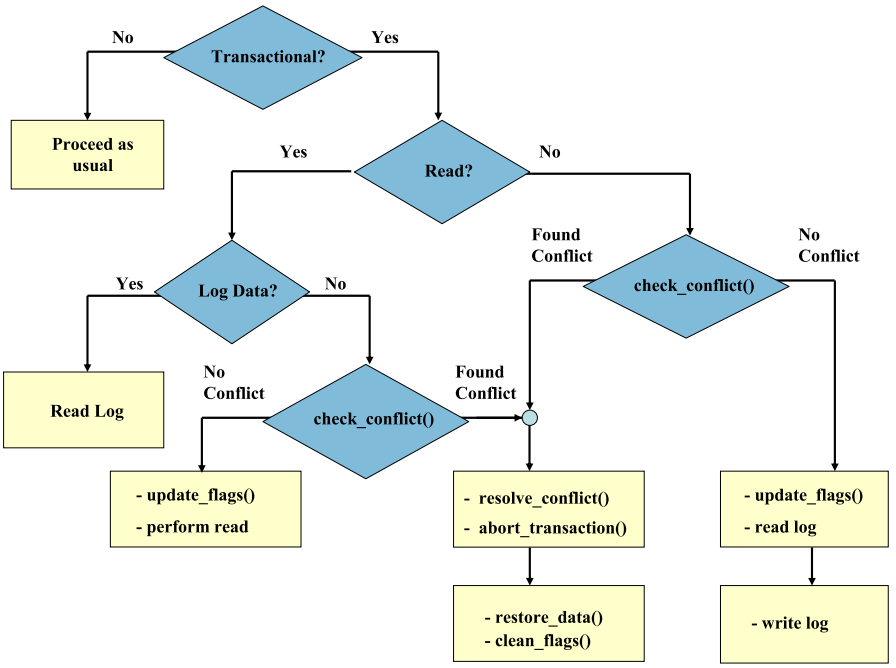



**Fig. 7** Transactional control flow

### 4.3 Transaction Control Flow

Next, we describe the transaction control flow. Before a transaction starts, it reads a special memory-mapped *transactional base register*. When this request reaches the memory, the bit corresponding to the core that made the request is set internally, to indicate that this processor is executing a transaction. When the transaction starts, its core saves its internal state (program counter, stack pointer and other internal registers), to be able to roll back if the transaction aborts. All transactionally executed memory accesses are marked with a special transactional bit set when the memory accesses are issued to the system. When a transaction ends, it triggers another access to a memory-mapped *transactional commit register*, which activates a special process at the memory bank level that cleans all the transactional flags and saved logs associated with that core's transactional accesses. Note that the access to these special transactional registers is a read access, hence it is non-blocking, meaning that multiple cores may access those registers simultaneously without serialization. These special registers do not impose the serialization and contention costs associated with traditional semaphores.

Figure 7 depicts the transactional memory control flow. Each TSM has a process that, in each cycle, checks whether the request received by the corresponding bank is transactional. If so, and it is a request to read a saved log value, then the process returns the data from the log.

If the request is a transactional data read but not for the log space, then the function *check_conflict*() checks the address's flag vector to determine whether this request triggers a conflict. If all concurrent transactions are also reading, then there is no conflict, and the *update_flags*() function adjusts the location's read flags before performing the read.

On the other hand, if some core has written that address, a conflict is triggered. A *resolve_conflict*() function decides which of the transactions currently accessing that address will have to abort. This decision depends on the current conflict resolution policy. As a starting point, we chose to abort the requester (i.e., the core which issued the access that triggered the conflict). When the *resolve_conflict*() function returns, it calls the *abort_transaction*() function, which notifies the cores that need to abort. These cores then restore their internal saved state and respond with an acknowledgment. Control is passed back to the *abort_transaction*() process that now has to call the *restore_data*() and *clean_flags*() functions. The first function is responsible for restoring the original saved data from the logs back to their original address locations and the second function is for cleaning the read/write flags of the aborted core. It is important to mention that, when an abort occurs in the system, all banks' TSMs call *restore_data*() and *clean_flags*() simultaneously and the banks stall normal operation until these functions have been completed, in order to avoid intermediate reads of invalid data. Once the data restoration process has been completed by all banks' TSMs, the aborted core's internal state is restored. Thus, the aborted core is ready to retry the transaction. This will not happen right away, but after waiting for a random backoff period, in order to avoid consecutive conflicts with other cores that might be also retrying their aborted transactions simultaneously. More details on how this backoff period is implemented follow in Sect. 5.

The control flow for a write is similar, but there is a difference in the conflict detection criteria: If the memory location is currently being either read or written by other transactions, then a conflict will be triggered. If no transaction is reading the location, then the *update_flags*() function sets the owner flag and the new owner's ID for that address. The first time the owner writes, the address's original data must be saved to the log.

## 5 Experimental Results

In this section we evaluate our proposed transactional memory design and compare it with the use of a conventional locking scheme. Moreover, we evaluate the overhead of the data versioning schemes we have proposed. To test our design, we chose benchmarks that required some data synchronization and are representative of real embedded systems applications. Also, since our simulation platform does not include operating system support, we eliminated all OS calls within the benchmarks. We start with the following data structure benchmarks, as well as benchmarks from the STAMP benchmark suite. Later in Sect. 5.3 we will report results using the EigenBench exploration tool.

– *Redblack*, *Skiplist* These are applications operating on special data structures. The workload is composed of a certain number of atomic operations (i.e., inserts,

deletes and lookups) to be performed on these two data structures. Redblack trees and skip-lists constitute the fundamental blocks of many memory management schemes found in embedded applications.
– *Genome* This is a gene sequencing program from the STAMP benchmark suite [15]. A gene is reconstructed by matching DNA segments of a larger gene. The application has been parallelized through barriers and large critical sections.
– *Vacation* This application also comes from the STAMP benchmark suite [15] and implements a non-distributed travel reservation system. Each thread interacts with the database via the systems transaction manager. Vacation features large critical sections.
– *Kmeans* This is a program from the STAMP benchmark suite [15] that groups objects into K clusters. It uses barrier-based synchronization and features small critical sections.

### 5.1 Overhead Characterization

In this section, we further detail the overhead analysis of our proposed data versioning scheme in terms of space and time. As we described in Sect. 4.2, the full-mirroring design requires half of the TCDM memory space to be reserved for the mirroring addresses, even though not all of them will be actually used. The distributed logging design on the other hand, employs distributed per-address logs instead of mirrors. We can fine tune the size of those logs so that it is based on the actual write footprint of the transactions. As a result, the distributed logging scheme provides a better utilization of the available memory, since it reserves for the logs only the space necessary by the transactions, leaving the rest to the application, while full-mirroring reserves half of the available memory for mirrors that will not be entirely used. For each application that we ran, we measured the maximum per-core transactional write footprint, ie. the maximum size of data that is written within a single transaction by a core, when running our applications with the maximum number of cores that the cluster can accommodate (16 cores). We report the results in Table 1. In the second column, we see how many bytes are actually written within a single transaction of a core. In the third column, we see how many bytes need to be reserved in total for all cores' transactions, that is actually the amount of space we need to keep for the logs. We observe that in the worst case we need 5 KB for all the logs in the system. For a TCDM memory size of 256 KB, that is roughly 2% of the total TCDM memory size, which means that we can use the remaining 98% of the memory for the actual application data. If we use full-mirroring, we are able to utilize only 50% of the TCDM memory space for the application, which is a considerably less space-efficient solution.

The Distributed Logging scheme has its own cost as well. Since we use per-address logs and not mirrors, the position of each address in the log is not straightforward as in full-mirroring. As a result, every time that an address is saved in the log, we have to traverse the log in order to find whether the address already exists in the log and if it is not there, then add a new entry for that address. In full-mirroring, the location of each address's mirror can be computed very simply, by adding to the address the mirrors offset (ie. the base address of the mirrors). As a result, for full-mirroring, each time

**Table 1** Per-core transactional write footprint for each application

| Application | Per-core trans. write footprint (bytes) | Total log space (bytes) |
|---|---|---|
| Redblack | 256 | 4096 |
| Skiplist | 64 | 1024 |
| Vacation | 320 | 3072 |
| Genome | 192 | 5120 |
| Kmeans | 256 | 4096 |

an address is saved, we need to pay two extra cycles for saving that address (one for reading the original content of the address and one cycle for writing it to the mirroring address). For the distributed logging, each time an address is saved, we need to traverse the log first. Based on our benchmarks profiling, a core's log in a bank can have up to 5 entries at a time, so we need 5 extra cycles to traverse the log in the worst case.

In case of abort, the total restoration time clearly depends on the write footprint of the target application: the higher the number of writes within a transaction, the bigger the size of the logs or the number of the saved mirrors, that we need to restore. For each address that needs to be restored, 2 cycles are spent, one for reading the original value from the log space and one for writing it back to its original address.

In case of commit, we do not need to restore any data, since both full-mirroring and distributed logging are eager versioning mechanisms, so the transactional data is already in place upon commit. This way commits are very fast.

### 5.2 Performance Characterization

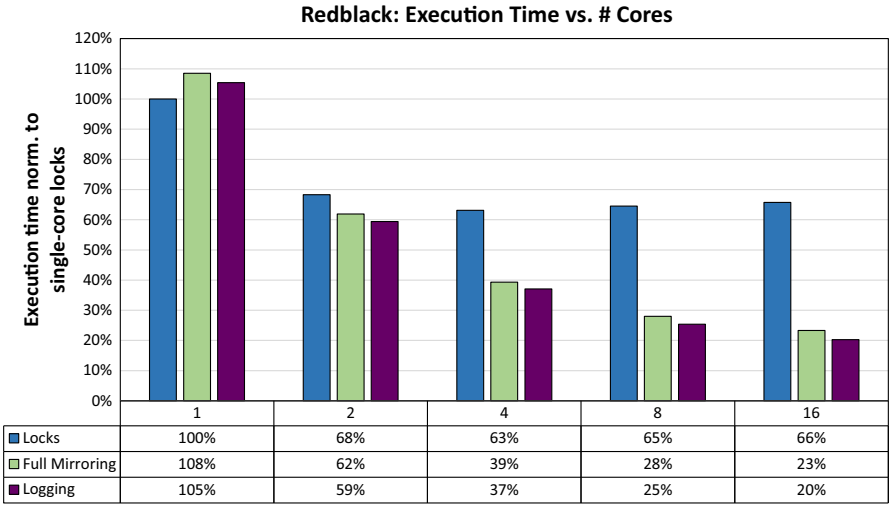
In this section we evaluate our proposed transactional memory design both for the full-mirroring and the distributed logging scheme and compare it with the use of a conventional locking scheme.

For each benchmark, we ran experiments using 1, 2, 4, 8 and 16 cores and measured the total execution time in cycles. As mentioned in Sect. 4.3, we chose to follow a *requester-abort* policy for managing conflicts. This is a basic approach also chosen in previous works for transactional memory. In the transactional retry process we incorporated exponential backoff. When a core aborts, it does not retry the transaction immediately. Instead it halts the execution of the transaction, restores the original register values and then waits for a random backoff period, after which it begins re-executing the transaction. The range of the backoff period is tuned according to the conflict rate. The first time a conflict occurs in a particular transaction, the core waits for an initial random period ($<$ 100 cycles) before restarting. If the transaction conflicts again, the backoff period is doubled, and will continue to double each time until the transaction completes successfully. This way, we avoid the scenario of a sequence of conflicts happening repeatedly between cores that retry the same aborted transaction.

Table 2 shows the experimental setup of the VSoC platform that we used in our experiments. As shown in Table 2, the TCDM has 16 banks. The $t_{hit}$ and $t_{miss}$ values

**Table 2** Experimental setup for VSoC platform

| Parameter | Value |
|---|---|
| Main memory latency | 200 ns |
| Main memory size | 128 MB |
| Core frequency | 200 MHz |
| # Cores | 1, 2, 4, 8, 16 |
| TCDM size | 256 KB |
| # TCDM banks | 16 |
| I$ size | 4 KB |
| I$ $t_{hit}$ | 1 cycle |
| I$ $t_{miss}$ | $\geq$ 50 cycles |




| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| Locks | 100% | 68% | 63% | 65% | 66% |
| Full Mirroring | 108% | 62% | 39% | 28% | 23% |
| Logging | 105% | 59% | 37% | 25% | 20% |

**Fig. 8** Redblack: performance comparison between locks and transactions for different number of cores

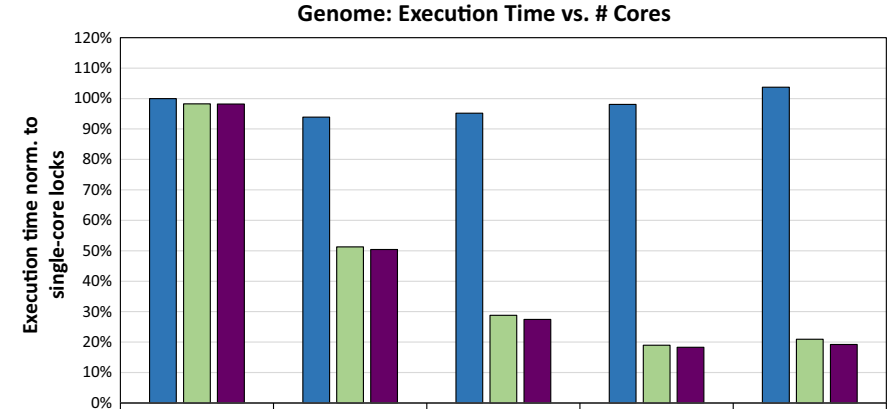
represent the instruction cache access times in case of a hit or a miss respectively. Accesses to the off-cluster main memory take 200 ns, significantly longer than accesses to the on-cluster TCDM that take only 4 ns in total. Access to the off-cluster main memory is assisted through a DMA with 0.5 GB/s bandwidth.[2]

We first run experiments for *redblack*, *skiplist*, *genome*, *kmeans* and *vacation*. The results of our experiments are depicted in Figs. 8, 9, 10, 11 and 12. The figures show the results for running the applications using spin-locks in comparison with our proposed transactional scheme using the full-mirroring or the distributed logging scheme respectively, for different number of cores. For each benchmark, we show the percentage change in execution time relative to a baseline execution time of a single core with locks. We make the following observations: First, for the locking scheme,

[2] Since we are assuming a DMA to assist in the data transfer, access time per word will not take the full 200 ns.


| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| Locks | 100% | 73% | 67% | 68% | 68% |
| Full Mirroring | 106% | 59% | 36% | 25% | 19% |
| Logging | 105% | 58% | 35% | 23% | 17% |

**Fig. 9** Skiplist: performance comparison between locks and transactions for different number of cores

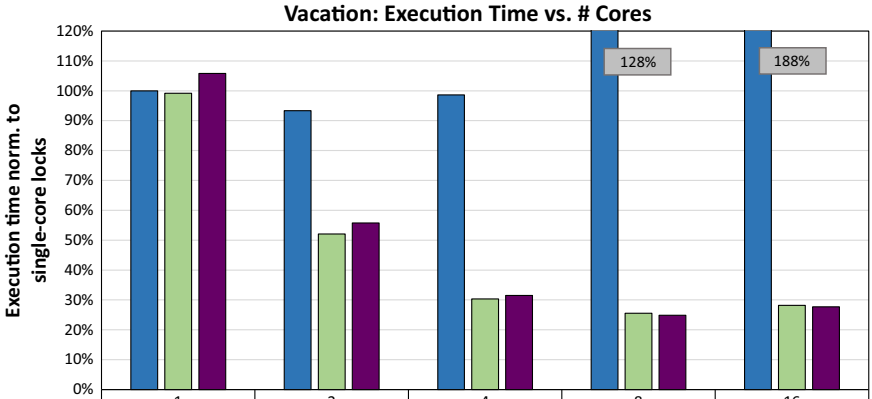



| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| Locks | 100% | 94% | 95% | 98% | 104% |
| Full Mirroring | 98% | 51% | 29% | 19% | 21% |
| Logging | 98% | 50% | 27% | 18% | 19% |

**Fig. 10** Genome: performance comparison between locks and transactions for different number of cores

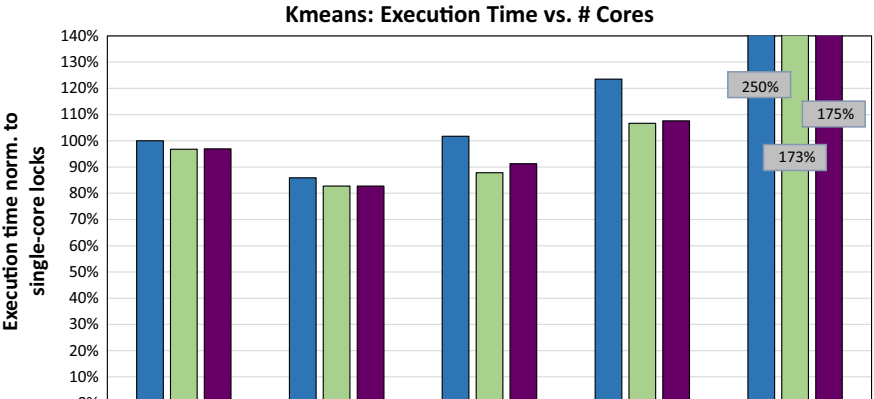
even though the performance improves as we scale from 1 to 2 cores, it does not show significant improvement and in most cases it gets worse, as we move above 4 cores. This means that the performance scaling that we hope to achieve using parallel execution does not follow the scaling of the number of cores. This is expected since in a standard locking scheme, the cores spend a lot of time spinning on the locks before entering the critical section. As a result, execution of the critical sections is serialized and this effect becomes worse as lock contention increases.

The second observation we make from the figures is that the transactional memory configurations always achieve better performance than the standard locking scheme.




| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| Locks | 100% | 93% | 99% | 128% | 188% |
| Full Mirroring | 99% | 52% | 30% | 26% | 28% |
| Logging | 106% | 56% | 31% | 25% | 28% |

**Fig. 11** Vacation: performance comparison between locks and transactions for different number of cores




| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| Locks | 100% | 86% | 102% | 123% | 250% |
| Full Mirroring | 97% | 83% | 88% | 107% | 173% |
| Logging | 97% | 83% | 91% | 108% | 175% |

**Fig. 12** Kmeans: performance comparison between locks and transactions for different number of cores

This is because, in the transactional memory scheme, the cores execute critical sections speculatively assuming a real data conflict will not occur; locks, on the other hand conservatively assume a conflict will occur and thus effectively serialize all accesses to critical sections. If the abort rate is not significant, then the overall performance improves tremendously. However, even in the event of aborts, transactions are restarted after an exponential backoff period, assuring that the retrying cores will not conflict repeatedly.

Third, we observe that our proposed transactional scheme, both in the full-mirroring and the distributed logging configuration, achieves the intended scaling that we expect

from using multiple cores instead of a single core (the only exception to this is *kmeans*, which we explain later). In most cases, performance keeps doubling as we continue to double the number of cores in the cluster. As the number of cores increases beyond 4 cores though, the performance scaling is slightly reduced and in some cases it levels off (*genome*, *vacation*). This is due to the main memory accesses that generate large delays and end up masking the benefits of running on a larger number of cores. These accesses exist in all runs, independent of the number of cores we use, but their effect becomes more pronounced for the 8 and 16 cores configurations since the total execution time is further reduced as we increase the number of cores. In addition, not all benchmarks exhibit similar benefit from parallelism. For example, *kmeans* is a benchmark that suffers from very large abort rates as the number of cores is increased, which end up hurting performance. This is a benchmark that seems to benefit from parallelism only up to 2 cores. Even though some scaling in performance is achieved as we increase the number of cores beyond 4, this scaling is decreased due to the large number of main memory accesses. However, in most cases performance is still improved compared to using fewer cores and in all cases it is better than the performance of the locking scheme.

Next, we examine how the full-mirroring design compares with the distributed logging design in terms of performance. We observe that for *redblack*, *skiplist* and *genome*, the distributed logging scheme always outperforms the full mirroring schemes, while for *vacation*, it is worse for a small number of cores and it gets better as we increase the number of cores. The *kmeans* application is the only one in which the distributed logging scheme shows worse performance than the full-mirroring configuration. As we discussed in Sect. 5.1, the distributed logging scheme incurs a slightly bigger overhead in saving the logs than the full-mirroring design. This would normally result in the distributed logging scheme being worse in performance compared to the full-mirroring scheme. At the same time, the distributed logging scheme uses memory more efficiently, allowing a larger quantity of data to be stored in memory. In contrast, for full-mirroring a larger number of main memory accesses may be required for refilling data. The data refill overhead masks the difference in the log saving overhead. Applications *redblack*, *skiplist* and *genome* have a large number of refill accesses for the full-mirroring scheme because of its inefficient use of the TCDM. As a result, in those benchmarks we see performance being worse for the full-mirroring design. On the other hand, *kmeans* and *vacation* have smaller data footprints, hence only a small number of refill accesses to the main memory are necessary. As a result the performance difference caused by the log saving process in the distributed logging scheme now becomes visible. Specifically, it is even more pronounced for a smaller number of cores, since fewer cores mean less but bigger logs, hence bigger log traversing overhead.

Overall, we see that our transactional memory scheme achieves significant performance improvements compared to standard locks. Specifically, we show a maximum improvement of 80% for *redblack*, 83% for *skiplist*, 82% for *genome*, 17% for *kmeans* and 75% for *vacation* over the baseline single-core lock configuration, depending on the number of cores. Locks, in comparison, cannot effectively exploit the extra parallelism offered by adding additional cores so performance improvement lags far behind. We conclude that a transactional memory support scheme, when designed carefully

based on the needs of the target architecture, can achieve significant performance improvements.

### 5.3 EigenBench

To further evaluate and compare our two TM implementations, we use in this section the *EigenBench* exploration tool. EigenBench [9] is a simple microbenchmark for evaluating TM systems that allows for exploration of several *eigen-characteristics*, i.e., a set of orthogonal characteristics of TM applications that form a basis for all TM applications (similar to how a basis in linear algebra spans a vector space). The benchmark allows to decouple the eigen-characteristics from each other and vary them independently, enabling the evaluation of corners of the application space not easily reached by real programs.

Specifically, we focus here on three characteristics that are relevant to our TM designs:

(a) *Working-set size* This parameter represents the size of the memory accessed within transactions. Since our design requires explicit DMA transfers for TCDM management, when the transaction's memory footprint increases beyond the size of the TCDM we will experience a performance drop, due to the DMA transfers;
(b) Contention This parameter represents the probability of conflicts for a transaction. Since the roll-back mechanism that takes place upon conflict is different for the two TM implementations, we expect this to have an impact on the performance when the conflict rate is high;
(c) *Predominance* This parameter represents the fraction of cycles spent in memory operations within transactions to cycles spent in memory operations outside transactions. It thus represents a measure of the overhead for handling transactional reads/writes (i.e., for data versioning). When the predominance factor is low, we expect this overhead to be negligible, while for high predominance this could be important.

Note that there are no other instructions inside and outside transactions, besides the memory accesses described above. This is a default setting for the eigenbench for measuring the worst-case overhead of the TM system being evaluated. Figure 13 shows three plots that report the results for the above described eigen-characteristics. We measure the execution cycles for four configurations:

(1) *FM* Transactions are handled with the Full-Mirroring scheme;
(2) *LOGGING* Transactions are handled with the Logging scheme;
(3) *LOCKS* Transactions are protected with locks;
(4) *UNPROTECTED* Transactions are not protected and instead are allowed to run fully in parallel. While this is functionally not correct, it provides an upper bound on the achievable performance.

Next, we analyze the results shown in Fig. 13.

(a) *Working-set size* We show the speedup of the two TM systems versus locks. On the $x$-axis we show the transaction's working-set size (i.e., the footprint of

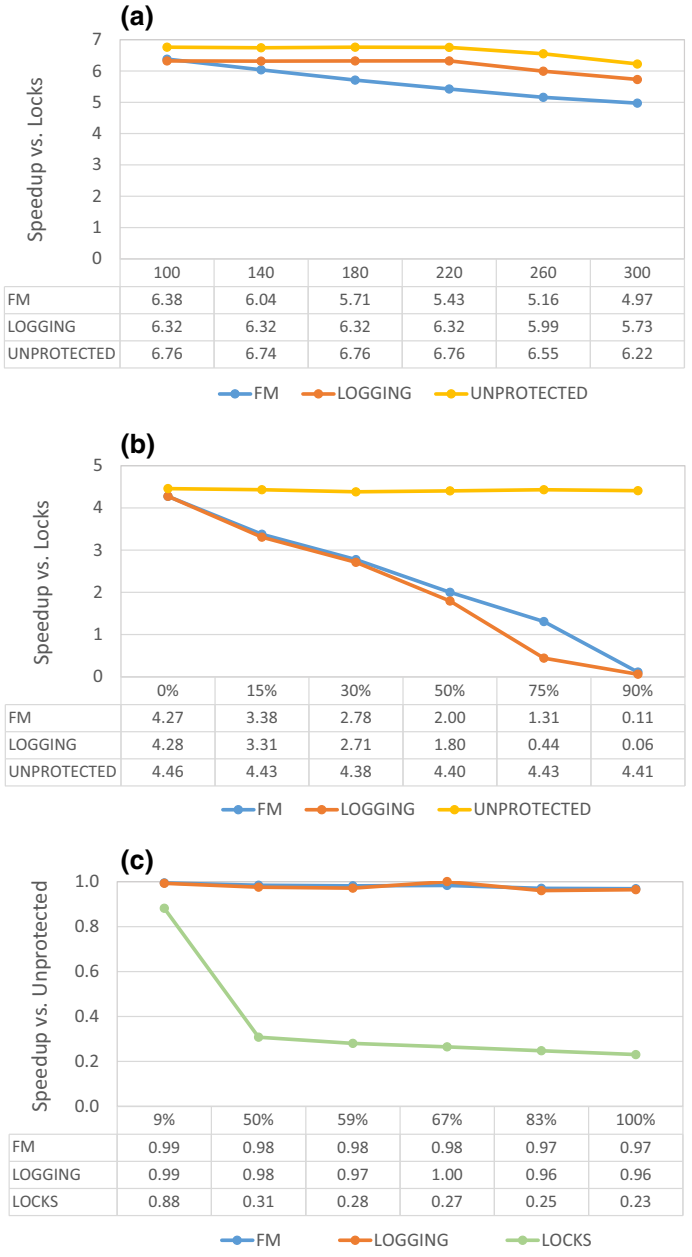



**Fig. 13** Results for the eigenbench evaluation methodology. The eigen-characteristics considered are **a** working-set size (KB), **b** contention (%) and **c** predominance (%). Note that the results presented are normalized with respect to locks for **a**, **b** and with respect to unprotected for **c**

transactional accesses) in KB. The first thing to emphasize is that for a transactional data footprint smaller than 128 KB (the size of the TCDM that a program

can use in the FM scheme) both TM systems perform very closely to the ideal (UNPROTECTED) case. Beyond 128KB the FM system starts suffering from DMA transfers. The same happens for the LOGGING scheme when the transactional data footprint grows beyond 226 KB.

(b) *Contention* We show again the speedup of the two TM systems versus locks. On the $x$-axis we show the percent transactions conflict rate. Only at 0% conflict rate, both TM systems perform very closely to the UNPROTECTED case. As the conflict rate increases their performance starts dropping and at around 30% conflict rate the LOGGING scheme starts behaving slightly worse than FM. This, as expected, is due to the slightly costlier rollback. It is also relevant to notice that around 75% conflict rate both schemes start performing worse than locks.

(c) *Predominance* We show the slowdown of the two TM systems versus UNPROTECTED, as we assess how the overhead for transactional read/write logging causes our TM schemes to depart from the ideal performance. On the $x$-axis we show the percent predominance. We observe that for both TM schemes, performance is consistently very close to the UNPROTECTED case (where data versioning is not used). This means that the time overhead for handling logs and mirrors in our TM schemes is negligible and does not have an impact on the overall performance potential.

## 6 Conclusions and Future Work

In this paper we proposed the first HTM design for handling transactions within a cluster-based, many-core embedded architecture without caches and cache coherence support. We introduced the idea of distributing conflict detection and resolution to multiple Transaction Support Modules so that our solution is scalable. As a first step, we restricted transactions within a single cluster and focused on simple and fast transactional handling schemes. However, the proposed scheme is designed to be scalable and can be extended to multiple clusters.

We introduced two alternative data versioning designs, full-mirroring and distributed logging, and compared them in terms of memory overhead and performance over a range of benchmarks. While full-mirroring is a simpler design, it is wasteful and rather impractical in terms of memory overhead, using 50% of the TCDM for mirrors when only about 2% is required in the distributed logging scheme. Furthermore, through simulations we demonstrated that the full-mirroring design requires more main memory accesses for data refilling that can hurt performance making distributed logging a better choice in most cases. Performance results showed that both transactional memory designs achieve a significant improvement over conventional locks. The maximum performance improvement over the baseline single-core lock configuration ranges from 17 to 83% depending on the application.

The transactional support scheme designed in this work gives us a good understanding of how speculation can benefit performance on a cache-free many-core embedded architecture, when transactions are restricted within a single cluster. For future work, we would like to extend this design to multiple clusters, providing inter-cluster trans-

actional support. Also, we would like to experiment with alternative conflict resolution policies and bookkeeping designs that may provide better efficiency.

## References


1. Adapteva: Epiphany-IV 64-core 28nm microprocessor (E64G401). http://www.adapteva.com/epiphanyiv/ (2013)
2. Bit-tech.net: IBM releases "world's most powerful" 5.5GHz processor. http://www.bit-tech.net/news/hardware/2012/08/29/ibm-zec12/1, 8 Sept 2012
3. Bortolotti, D., Pinto, C., Marongiu, A., Ruggiero, M., Benini, L.: Virtualsoc: A full-system simulation environment for massively parallel heterogeneous system-on-chip. In: 2013 IEEE International Symposium on Parallel and Distributed Processing, pp. 2182–2187 (2013). https://doi.org/10.1109/IPDPSW.2013.177
4. Ferri, C., Marongiu, A., Lipton, B., Moreshet, T., Bahar, R.I., Herlihy, M., Benini, L.: SoC-TM: integrated HW/SW support for transactional memory programming on embedded mpsocs. In: CODES, pp. 39–48. Taipei, Taiwan (2011)
5. Ferri, C., Wood, S., Moreshet, T., Bahar, R.I., Herlihy, M.: Embedded-TM: energy and complexity-effective hardware transactional memory for embedded multicore systems. J. Parallel Distrib. Comput. **70**(10), 1042–1052 (2010)
6. Helmstetter, C., Joloboff, V.: SimSoC: a systemC TLM integrated ISS for full system simulation. In: IEEE Asia Pacific Conference, pp. 1759–1762 (2008)
7. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: ISCA, pp. 289–300 (1993). https://doi.org/10.1145/165123.165164
8. Hong, S., Oguntebi, T., Casper, J., Bronson, N., Kozyrakis, C., Olukotun, K.: Eigenbench: A simple exploration tool for orthogonal tm characteristics. In: Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10), IISWC '10, pp. 1–11. IEEE Computer Society, Washington (2010). https://doi.org/10.1109/IISWC.2010.5648812
9. Hong, S., Oguntebi, T., Casper, J., Bronson, N., Kozyrakis, C., Olukotun, K.: Eigenbench: a simple exploration tool for orthogonal TM characteristics. In: IEEE International Symposium on Workload Characterization (IISWC), 2010, pp. 1–11 (2010). https://doi.org/10.1109/IISWC.2010.5648812
10. Intel Corporation: Transactional Synchronization in Haswell. http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/, 8 Sept 2012
11. Kalray: MPPA 256—Programmable Manycore Processor. www.kalray.eu/products/mppa-manycore/mppa-256/
12. Kunz, L., Girão, G., Wagner, F.: Evaluation of a hardware transactional memory model in an NoC-based embedded MPSoC. In: SBCCI, pp. 85–90. São Paulo, Brazil (2010)
13. Melpignano, D., Benini, L., Flamand, E., Jego, B., Lepley, T., Haugou, G., Clermidy, F., Dutoit, D.: Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications. In: DAC, pp. 1137–1142. ACM (2012)
14. Meunier, Q., Petrot, F.: Lightweight transactional memory systems for NoCs based architectures: design, implementation and comparison of two policies. J. Parallel Distrib. Comput. **70**(10), 1024–1041 (2010)
15. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: International Symposium on Workload Characterization (2008)
16. Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A.: LogTM: log-based transactional memory. In: HPCA, pp. 254–265 (2006)
17. NVIDIA: NVIDIA's next generation CUDA compute architecture: Fermi. White paper, NVIDIA (2009)
18. Papagiannopoulou, D., Capodanno, G., Moreshet, T., Herlihy, M., Bahar, R.: Energy-efficient and high-performance lock speculation hardware for embedded multicore systems. ACM Trans. Embed. Comput. Syst. (2015). https://doi.org/10.1145/2700097
19. Papagiannopoulou, D., Marongiu, A., Moreshet, T., Benini, L., Herlihy, M., Bahar, R.: Playing with fire: transactional memory revisited for error-resilient and energy-efficient MPSoC execution. In: GLSVLSI (2015). https://doi.org/10.1145/2742060.2742090
20. Papagiannopoulou, D., Moreshet, T., Marongiu, A., Benini, L., Herlihy, M., Bahar, R.: Speculative synchronization for coherence-free embedded NUMA architectures. In: SAMOS, pp. 99–106 (2014). https://doi.org/10.1109/SAMOS.2014.6893200

21. Rajwar, R., Goodman, J.R.: Speculative lock elision: enabling highly concurrent multithreaded execution. In: MICRO, pp. 294–305 (2001). http://dl.acm.org/citation.cfm?id=563998.564036
22. Rajwar, R., Goodman, J.R.: Transactional lock-free execution of lock-based programs. In: ASPLOS, pp. 5–17 (2002). https://doi.org/10.1145/605397.605399