TM^{2}C: a Software Transactional Memory for Many-Cores

Vincent Gramoli
EPFL, Switzerland
vincent.gramoli@epfl.ch

Rachid Guerraoui
EPFL, Switzerland
rachid.guerraoui@epfl.ch

Vasileios Trigonakis
EPFL, Switzerland
vasileios.trigonakis@epfl.ch

Abstract

Transactional memory is an appealing paradigm for concurrent programming. Many software implementations of the paradigm were proposed in the last decades for both shared memory multi-core systems and clusters of distributed machines. However, chip manufacturers have started producing many-core architectures, with low network-on-chip communication latency and limited support for cache-coherence, rendering existing transactional memory implementations inapplicable.

This paper presents TM^{2}C, the first software Transactional Memory protocol for Many-Core systems. TM^{2}C exploits network-on-chip communications to get granted accesses to shared data through efficient message passing. In particular, it allows visible read accesses and hence effective distributed contention management with eager conflict detection.

We also propose FairCM, a companion contention manager that ensures starvation-freedom, which we believe is an important property in many-core systems, as well as an implementation of elastic transactions in these settings. Our evaluation on four benchmarks, i.e., a linked list and a hash table data structures as well as a bank and a MapReduce-like applications, indicates better scalability than locks and up to 20-fold speedup (relative to bare sequential code) when running 24 application cores.

Keywords Transactional Memory; Many-Cores; Concurrent Programming; Contention Management

General Terms Design, Languages, Performance

Categories and Subject Descriptors C.1.4 [*Processor Architectures*]: Parallel Architectures—Distributed architectures; D.1.3 [*Programming Techniques*]: Concurrent Programming

EuroSys'12, April 10–13, 2012, Bern, Switzerland.

1. Introduction

Although not a silver bullet, Transactional Memory (TM) [20, 39] is an appealing paradigm to leverage the availability of multi-processor systems. TM allows the programmer to define a sequence of commands, called a transaction, and then to execute it atomically. In its software form, called STM, the paradigm can be implemented without requiring any specific hardware support [11, 12], at least in principle. Indeed, this is not entirely true for STMs do typically assume multi-core architectures and rely on an underlying cache-coherent system. Recently, manufacturers have started producing many-core processors [22], with the idea of increasing the number of cores placed on a single die while decreasing their complexity for enhanced energy consumption [6]. Contemporary many-cores consist of up to 100 cores, but they are soon expected to scale up to 1000. In such systems, providing full hardware cache-coherence is not affordable because of memory and time costs [3].

In this many-core context, and since the programming model is message passing, one might be tempted to apply what has been called Distributed Transactional Memory (DTM), namely, implementations of the transaction paradigm on distributed clusters of machines. The communication on such platforms being particularly expensive, classical DTMs try however to enforce as much as possible data and node locality. The setting is fundamentally different from the network-on-chip one of many-cores: messaging latencies among cores (and the memory) differ, but the size of magnitude is insignificant compared to clusters.

Perhaps more importantly, existing DTMs fail to provide strong progress guarantees. We argue that it is particularly important to ensure starvation-freedom in a many-core TM system, so that continuous contention does not repeatedly abort the same transactions. In fact, such systems are usually foreseen to support cloud applications where each individual client request, typically executed on a separate core, must eventually complete. Starvation-freedom ensures that the termination of a client request does not depend on the termination of others and it avoids livelocks.

We present in this paper TM^{2}C, the first TM system tailored for non-coherent many-core processors. TM^{2}C capitalizes the low latency of on-die message passing by being

the first starvation-free DTM algorithm. To this end, TM^{2}C exploits visible reads and allows the detection of conflict whenever a transaction attempts to override some data read by another transaction, hence anticipating the conflict resolution, otherwise deferred to the commit phase of the reading transaction. In contrast with many-cores, high latency systems require generally to pipeline asynchronous reads (inherently invisible) to achieve reasonable performance. Visible reads allow us to utilize contention management in a way similar to STMs, yet fully decentralized, to provide starvation-freedom. TM^{2}C comes with FairCM, a companion distributed contention manager that ensures the termination of every transaction and the fair usage of the TM system by each core.

We exploit the large amount of cores by assigning the transactional application and the DTM services of TM^{2}C to different partitions of the cores so that no more than a single task is allocated per core. More precisely, two disjoint groups of cores run each of these two services, respectively. This decoupling benefits the communication load by limiting message exchanges between cores of distinct groups only. In addition and for a particular workload, TM^{2}C reduces communication further by trading read-access requests with a lightweight in-memory read validation to implement a weaker transactional model: elastic transactions [13].

We evaluate TM^{2}C on the Intel®'s Single-chip Cloud Computer (SCC), a non-coherent message passing processor. The SCC is a 48-core experimental processor relying on a 6×4 two-dimensional mesh of tiles (two cores per tile) that is claimed to be "arbitrarily scalable" [29]. On a hash table data structure and a MapReduce example application TM^{2}C performs up to 20 and 27 times better than the corresponding bare (non-transactional) applications running on a single core. We also evaluated the importance of fair contention management by comparing our FairCM scheme with alternative ones on various workloads. Particularly, FairCM performs up to 9 times better than the others on a workload with a single core running long conflict prone transactions. Last but not least, we also elaborate on the portability of TM^{2}C to cache-coherent architectures. We show that TM^{2}C is also efficient on multi-cores and we conjecture on the possible causes of performance difference when running it on multi-cores and many-cores.

The rest of the paper is organized as follows. Section 2 presents the many-core system model. Section 3 presents the services at the core of TM^{2}C and Section 4 describes the contention management policies we applied to TM^{2}C to make it starvation-free. Section 5 presents the results obtained on the Intel®'s SCC and Section 6 illustrates how the elastic transactions benefit TM^{2}C when adequately implemented. Section 7 introduces some preliminary work on porting TM^{2}C on a multi-core and Section 8 discusses privatization and portability. Section 9 positions TM^{2}C to the related work and Section 10 concludes the paper.

2. The Many-Core Model

We consider a *many-core*, a processor that embeds from tens to thousands of simpler cores than a multi-core to maximize overall performance while minimizing energy consumption [6]. The backbone of the many-core is the network-on-chip which interconnects all cores and carries the memory traffic. Every core has a private cache, however, a many-core has either a limited or no hardware cache-coherence at all. Therefore, this on-die interconnection network provides the programmer with efficient message passing. In order to increase the memory bandwidth, a many-core is connected to multiple memory controllers [1]. These controllers comprise both the private and the non-coherent shared memory of the cores.

The system is thus modelled as a fully distributed system whose *nodes*, which represent cores, are fully connected and can communicate with each other using asynchronous messages. We assume that the communication links between nodes are reliable: every message sent is eventually delivered and the links do neither duplicate nor forge new messages. In addition, we assume that nodes are non-faulty in that they respect their code specification but do not crash, and that each of them has a unique identifier. Note also that this model is sufficiently general to capture both homogeneous and heterogeneous many-cores [6].

Our aim is to guarantee that a concurrent program executing in this model is consistent (i.e., safe) and can terminate (i.e., live). To this end, we assume that a concurrent program is correctly written as a transactional program in which regions of sequential code that must appear as atomic are adequately delimited within *transactions* and that there are no infinite loops within a transaction.

Generally, a *Transactional Memory* (TM) protocol is responsible of ensuring the *atomic consistency* (i.e., opacity [15]) of transactions by wrapping all accesses delimited within the transactions and by detecting conflicts. Upon conflict detection a *Contention Manager* (CM) is called to resolve it by possibly aborting, delaying, or resuming the conflicting transactions. Our model is weakly atomic [28] in that transactional accesses are not isolated from non-transactional accesses. We do not support side effects within transactions, yet one could extend our code with irrevocable transactions that ask exclusive accesses to all responsible nodes before executing pessimistically.

As described in the remainder, our TM protocol, namely TM^{2}C, wraps any of the shared memory accesses of a transaction into a communication protocol that requests the access grant for the appropriate memory bytes. Our distributed CM, namely FairCM, assigns a priority to each transaction that totally orders them and eventually rotates the highest priority among all cores. Hence, even if each core executes an infinite amount of transactions our protocol is *starvation-free*: every transaction is guaranteed to terminate.

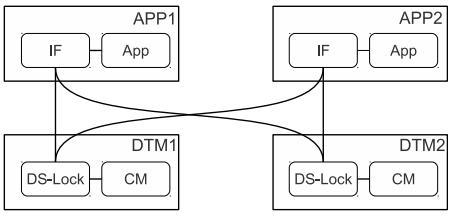


Figure 1. TM^{2}C system's architecture and communication paths.

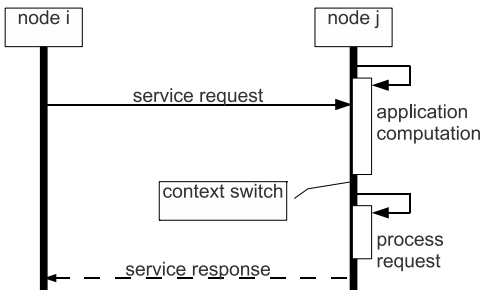


Figure 2. Multitasking – An example where the scheduling of node j affects the execution of node i.

3. TM^{2}C, Transactional Memory for Many-Cores

This section introduces the first *Transactional Memory for Many-Cores* (TM^{2}C). TM^{2}C allows programmers to exploit the inherent parallelism of a many-core through a simple interface. Its main novelty lies in guaranteeing starvation-freedom without the need of an underlying hardware cache-coherence, usually required to handle contention between cores. The immediate benefit is the scalability to foreseeable processors comprising a large number of cores, where a chip-wide coherence is non-affordable due to false conflicts and cache miss overheads. To achieve this result, TM^{2}C exploits the low network-on-chip message latency to implement a distributed contention management arbitrating contention cleverly between cores.

Specifically, TM^{2}C provides two services as depicted in Figure 1: (i) the application service (APP) interfaces the transaction with the application and hosts the transactional runtime; (ii) the Distributed TM (DTM) service grants a data access to the requesting transactions through the distributed locking (DS-Lock) which may call the contention manager (CM) upon conflict detection. First we describe how the two services can be deployed on the nodes, then we detail their roles.

3.1 TM^{2}C Deployment

The application and the DTM services are independent as the former is responsible for executing the transaction by requesting data accesses and the latter is responsible for deciding whether an access can be granted. Both services are fully distributed and could either be deployed on the same cores, all exploiting each core but at different time slots, or deployed on distinct cores, exploiting different cores but at the same time. The former deployment strategy thus leads to multitasking while the latter leads to dedicating roles to cores.

Multitasking. Our initial design used multitasking to allow both the application and the DTM system to run on every core. A user-space library, called libtask, was used for this implementation. We preferred libtask over POSIX threads (pthreads) because it has significantly cheaper context switches. Libtask is a simple coroutine library that gives the programmer the illusion of threads, but the operating system sees only a single kernel thread. As a result, libtask does not support preemption.

Yet, the multitasking still suffers from an important limitation: the scheduling of node j can potentially affect the execution of node i, where $j \neq i$. One such case is represented in Figure 2. Node j is executing some application code while node i tries to execute a service request that involves node j. The request cannot be served prior to j completing its local computation. Therefore, there is a waiting period that increases the latency of the service operation.

Dedicated service cores. As a many-core provides the application programmer with a large amount of simple cores, assigning a dedicated role to each core better exploits parallelism. As a follow-up to this observation, we engineered a second deployment strategy in which disjoint sets of cores are dedicated to hosting distinct services.

Such a dedicated strategy overcomes the above issue by avoiding timing dependencies between the application and the DTM services. In addition and as depicted in Figure 1, this strategy presents another significant advantage. In fact, the cores running the same service do not need to communicate with each other. This leads to complete decoupling among the DTM cores and the application cores respectively. The advantage is actually twofold. First, the number of messages in the system decreases. Second, the communication paths among the application cores can be exploited by an application that utilizes both the TM^{2}C in addition to direct messaging. Recall that TM^{2}C supports weak atomicity [28], hence the transactionally accessed data should not be concurrently accessed by non-transactional code.

3.2 Distributed Lock Service

The distributed lock (DS-Lock) component is at the heart of the DTM. It provides a service for acquiring multiple-readers/single-writer revocable locks. The operations exposed by the DS-Lock incorporate the transactional semantics, and are thus non-blocking. The transactional semantics comprises of *Read After Write (RAW)*, *Write After Read (WAR)*, and *Write After Write (WAW)* conflicts. Whenever the DS-Lock detects conflicts between two transactions ask-

Algorithm 1 Read-lock acquire operation

```
1:  dsl_read_lock(id, obj):
2:      enemy_tx ← obj.writer
3:      if enemy_tx ≠ NULL ∩ enemy_tx ≠ id then
4:          // if there is a writer different than id, read after write conflict, call CM
5:          cm ← contention_manager(id, enemy_tx, RAW)
6:          if cm = RAW then // CM aborted current transaction
7:              return RAW
8:      // no writer, or CM aborted enemy
9:      add_reader(obj, id)
10:     return NO_CONFLICT
```

ing for conflicting access grants, it calls the contention manager (described in Section 4). The contention manager is responsible for the conflict resolution.

The DS-Lock service is distributed among multiple nodes. Consequently, each node running a part of the DS-Lock service is responsible for controlling the accesses to a partition of the shared memory. In this context, the DS-Lock service is similar to some directory-based cache-coherence solutions [24, 26]. Devising a sophisticated way of allocating the data to the DS-Lock nodes is out of the scope of this work. A memory location, which in the case of TM^{2}C is a memory byte, is mapped to its responsible DS-Lock node by hashing.

The operations the DS-Lock implements are basically read-lock acquire/release and write-lock acquire/release. Notice that these operations are not explicitly called by the application code. The application calls the read and write wrapper functions which perform the appropriate message passing in order to implicitly trigger the corresponding DS-Lock service operations.

Read-lock acquire/release. The read-lock acquire operation attempts to acquire the read-lock corresponding to the input memory object for the requesting node identifier. It may be unsuccessful due to a RAW conflict. Algorithm 1 illustrates the pseudo-code for this operation. The read-lock release operation removes the corresponding node from the readers set of the memory object.

Write-lock acquire/release. The write-lock acquire operation attempts to acquire the write-lock corresponding to the input memory object for the requesting node identifier. It may be unsuccessful due to a WAW, or a WAR conflict. Algorithm 2 presents the pseudo-code for this operation. The write-lock release operation simply resets the writer of the memory object.

3.3 Transactions

A transaction is a delimited block of sequential code, whose shared accesses are redirected through transaction wrappers. Existing compilers wrap the shared accesses automatically[1]. The programmer could potentially benefit from these compilers as TM^{2}C respects the simple standard TM interface [23], even though it hides the underlying complex message passing implementation. Using the interface is the only way an application can interact with the DTM system. We discuss further interface extension to support elastic transactions in Section 6.

[1] The Intel® C/C++ compiler and gcc support it.

Algorithm 2 Write-lock acquire operation

```
1:  dsl_write_lock(id, obj):
2:      enemy_tx ← obj.writer
3:      if enemy_tx ≠ NULL ∩ enemy_tx ≠ id then
4:          // if there is a writer different than id, write after write conflict, call CM
5:          cm_1 ← contention_manager(id, enemy_tx, WAW)
6:          if cm_1 = WAW then
7:              return WAW
8:      // no writer, or CM aborted enemy
9:      enemy_list ← obj.readers
10:     if ¬is_empty(enemy_list) then
11:         // write after read conflict, call CM
12:         cm_2 ← contention_manager(id, enemy_list, WAR)
13:         if cm_2 = WAR then
14:             return WAR
15:     // no readers, or CM aborted enemies
16:     obj.writer ← id
17:     return NO_CONFLICT
```

The interface includes operations to start and commit a transaction, which are the delimiters of the transaction. Start simply creates a new transaction, while commit (`txcommit`) tries to commit a transaction. The commit operation has to acquire the necessary write-locks, persist the changes in the shared memory, and finally release all the acquired locks. The pseudo-code of `txcommit` is depicted in Algorithm 3.

The following paragraphs describe the operations to transactionally read and write from/to a shared memory location.

Algorithm 3 Transaction commit (`txcommit`) operation

```
1:  txcommit():
2:      tx_metadata ← get_metadata()
3:      // try to write-lock all objects in the write-buffer
4:      while (item ← get_item(write_buffer)) ≠ NULL) do
5:          // node responsible for obj locking
6:          nId ← get_responsible_node(item.obj)
7:          // similar to an RPC-like call on node nId, but uses message passing
8:          response ← write_lock(nId, id, tx_metadata, item.obj)
9:          if response ≠ NO_CONFLICT then
10:             // conflict and CM aborted current
11:             txabort(response)
12:         append(item, writes_locked)
13:     // all locks acquired, persist the write-set to the memory
14:     writeset_persist(writes_locked)
15:     // release all locks and update metadata
16:     wlock_release_all(id, writes_locked)
17:     rlock_release_all(id, read_buffer)
18:     update_metadata(tx_metadata)
```

Algorithm 4 Transactional read (`txread`) operation

```
1:  txread(obj):
2:    obj_buf ← get_buffered(obj)
3:    if obj_buf ≠ NULL then
4:      // if memory object is in write or read buffer
5:      return obj.value
6:    // node responsible for obj locking
7:    nId ← get_responsible_node(obj)
8:    tx_metadata ← get_metadata()
9:    // similar to an RPC-like call on node nId, but uses message passing
10:   response ← read_lock(nId, id, tx_metadata, obj)
11:   if response = NO_CONFLICT then // acquired the read-lock
12:     value ← shmem_read(obj)
13:     add_read_buffer(obj, value)
14:     return value
15:   else // else conflict and CM aborted current
16:     txabort(response)
```

Visible reads. The transactional read (`txread`) is the operation used to read a memory object within the context of a transaction. Algorithm 4 contains the pseudo-code describing the steps taken for this operation.

Transactional reads work with early lock acquisition and therefore the system operates with visible reads. Early acquisition suggests that a transaction has to acquire the corresponding read-lock before proceeding to the actual read. The visibility of reads is an outcome of the early acquisition: every transaction is able to "see" the reads of the others because of the read-locks. The motivation behind this design decision is twofold.

Firstly, every many-core processor provides a fast message passing mechanism. Taking this into account, the overhead from performing synchronous read validation is acceptable. On a cluster, the messaging latency is significantly higher, hence such a synchronous solution would be prohibitive. Additionally, visible reads are often cited as problematic for affecting the cache behavior of the system. In TM^{2}C this is not the case due to the message passing. The visibility of reads does not require changing some local memory objects (e.g., locks), but using the locking service to acquire the corresponding locks by communicating with a remote node.

Secondly, visible reads are necessary for contention management. Without the read visibility, the WAR conflict detection is deferred to a validation phase typically before the commit. If a conflict is detected, it is too late to perform conflict resolution, since the writing transaction has already committed the new values.

Deferred writes. Transactional write is the operation used to write to a memory object within the context of a transaction. Transactional writes work with lazy lock acquisition and deferred writes[2]. Every write operation is buffered and only in the commit phase the actual locks are acquired.

We chose lazy write acquisition for one main reason. If two transactions conflict, one has to be writing on the memory object. Therefore, if a transaction holds a write-lock for a long time, it increases the possibility that a conflict[3] may appear. Lazy write acquisition helps reducing the time that the write-locks are being held. For an experimental comparison of lazy against eager write acquisition see Section 5.2. Moreover, it allows the implementation of write-lock batching: requesting the locks for multiple memory objects in a single message, which can significantly reduce the number of messages.

4. Distributed Contention Management

In this section we present Contention Managers (CMs) that are fully decentralized and ensure transaction termination in the message passing model. Existing contention managers are generally centralized [14, 37, 38] and not applicable to our model as they either rely on a global counter (e.g., Greedy, PublishedTimestamp, Timestamp), randomization (e.g., KinderGarten), or on constantly changing priorities that become inaccurate when conflict resolution gets propagated in an asynchronous system (e.g., Eruption, Karma, Polka).

Our aim is to guarantee starvation-freedom so that each transaction eventually commits, hence precluding situations in which two transactions block each other (*deadlocks*) or where some transactions get repeatedly aborted (*livelocks*). Many-cores are foreseen to support cloud applications where independent client requests may contend on accessing the same service and it is highly desirable that a client request does not get repeatedly restarted because of concurrent requests from other clients. Existing DTMs usually target weaker progress properties than starvation-freedom, like lock-freedom, as it is easier to guarantee that at least one request progresses at any time.

4.1 Preliminaries

Each transaction is assigned a *priority* $\in \mathbb{Z} \times I$ allowing the contention manager to compare concurrent transactions using some identifier in I as a tie-breaker. Upon conflict between multiple transactions, the contention manager compares the priority of the conflicting transactions and aborts all of them but the highest priority one. The status of such an aborting transaction is atomically switched from pending to aborted by the contention manager. An aborted transaction is immediately restarted if not specified otherwise by the contention manager. The transaction's *lifespan* captures the period between the time the transaction starts and the time it commits, be it aborted several times in between.

[2] The strategy of deferring writes is also known as write-back.

[3] This conflict can be of type RAW or WAW.

The following property indicates a sufficient set of rules a CM should adhere to provide starvation-freedom on the TM^{2}C system.

PROPERTY 1. *On TM^{2}C, a contention manager satisfying the three following rules:*

a. the priority of a transaction does not change during its lifespan,
b. the priorities define a total order on the set of current transactions and
c. the priority of a transaction should be strictly lower than the priority of the preceding committed transaction initialized by the same node

ensures termination of every transaction.

The intuition of Property 1 is that every CM node has always (upon a conflict) up to date information about the conflicting transactions. Assume that a conflict of transactions t and t' is detected on node i. If t' performed the operation that caused the conflict, then node i has the correct data for t' since they were piggybacked in the request. Moreover, t has earlier performed an operation on node i. Node i has the correct data for t, because t's priority can only change if t has committed (rule *(a)*), in which case it would have already released the lock. Consequently, whenever there is a conflict the CM of the corresponding node will have up-to-date priorities for all the conflicting transactions. This implies that even if there are simultaneous conflicts of two (or more) transactions, the distributed CM will take a coherent decision.

Due to rule *(b)* it is guaranteed that at least one of the i's conflicting transactions, say a transaction on node j, will be able to commit its transaction, thus reducing the priorities of j's next transactions (rule *(c)*). After a finite number of transactions, transactions on j will stop having the highest priority and some other node becomes the highest priority one. This process repeats a finite number of times until node i has the highest priority among the conflicting nodes.

We now consider four contention managers in addition to the default policy, denoted by *no-CM*, that simply consists of aborting and restarting a transaction that detects a conflict. We first present two contention managers that are livelock-prone before presenting two contention managers that ensure starvation-freedom.

4.2 Back-off-Retry

The *Back-off-Retry* contention manager lets the transaction that detects the conflict abort and wait a period of time whose expected duration increases. More precisely, the waiting duration is chosen by tossing an integer that is lower than a upper bound that increases exponentially each time the same transaction aborts. When this transaction commits and a new transaction starts, the upper bound is reset to its initial value. Using TM^{2}C with the Back-off-Retry contention manager may lead to livelock as the same transaction may repeatedly detect all the conflicts it is involved in, yet in practice transactions often terminate thanks to its randomization.

4.3 Offset-Greedy

We now describe a distributed CM, namely *Offset-Greedy*, as an adaptation of an existing centralized CM, called Greedy [14], to illustrate the difficulty of ensuring starvation-freedom in a distributed system. Greedy prioritizes a transaction using a timestamp, representing the time it started. In case of conflict, the youngest conflicting transactions are aborted in favor of the oldest one.

In a distributed system, the lack of a global clock prevents us from implementing Greedy since different nodes of the system do not have a way of taking consistent timestamps. Typically, the transaction with the most advanced clock may obtain the lower priority even though it starts first.

To bypass this limitation we introduced Offset-Greedy that estimates timestamps based on time offsets. Offset-Greedy takes the following steps whenever a node performs a transactional operation:

1. The transaction uses the node's local clock in order to calculate the time offset since the beginning of the transaction.
2. The transaction sends the request to the responsible DTM node, piggybacking the offset calculated in step 1.
3. The DTM uses the offset from the request and its local clock to estimate the timestamp of the transaction according to its own local clock.
4. The request is normally processed.

However, Offset-Greedy does not guarantee starvation-freedom. Although it ensures rules *(a)* and *(c)* of Property 1, it does not guarantee rule *(b)*. The offset calculation technique does not take into account the message delay in computing the offset. As the DTM load impacts the message delay, nodes may happen obtaining inconsistent views of timestamps. As a result, if a conflict emerges concurrently on the two nodes with inconsistent views, both transactions might abort. This scenario could lead to livelocks, however, we did not experience such an issue in our experiments.

4.4 Wholly

To address the starvation problem of the above contention managers, we propose *Wholly*. Wholly guarantees that the nodes progress altogether. The priority is the inverse of the number of transactions that each application node has already committed. Upon a conflict, the node that has committed the most transactions is aborted. If two nodes have the same number of committed transactions, then their identifiers are used as tie-breakers.

PROPERTY 2. *Wholly guarantees starvation-freedom.*

Property 2 follows from the fact that Wholly satisfies the three rules of Property 1. Wholly clearly ensures rule *(a)*.

The combination of the number of committed transactions and the node identifier (if there is a tie) define a total order on the priorities, thus satisfying rule *(b)*. Finally, Wholly satisfies rule *(c)* because whenever a transaction commits, it reduces its priority. Consequently, according to Property 1, Wholly guarantees the termination of every transaction.

Unfortunately, Wholly does not promote short transactions over longer ones, hence, long transactions may reduce the overall throughput by causing a large amount of aborts due to numerous restarts.

4.5 FairCM

To promote short transactions over longer ones we propose a last contention manager, called *FairCM*, that is fair regarding the effective transactional time of each node. Instead of using the number of committed transactions, FairCM uses the cumulative time spent on successful transaction attempts (in addition to the identifier). So, if a transaction proceeds as follows:

$$\text{Start}\rightarrow\text{Abort}_1 \rightarrow\text{Restart}_1 \rightarrow\text{Abort}_2 \rightarrow\mathbf{Restart}_2\rightarrow\mathbf{Commit}$$

then only the duration from $\mathbf{Restart}_2$ to **Commit** will be added to the cumulative time. Upon a conflict, the transaction with the less cumulative time has higher priority. According to Property 3, FairCM guarantees starvation-freedom.

As described in Section 5.3, the particularity of promoting short transactions over longer ones may prove very important for the performance of the system. In particular, when some nodes tend to run long conflict-prone transactions. Without fairness, these nodes would degrade the overall throughput of the system.

PROPERTY 3. *FairCM guarantees starvation-freedom.*

Property 3 follows the exact same reasoning as Wholly's, using the effective transactional time for the priority instead of the number of committed transactions.

5. Evaluating TM^{2}C

In this section we evaluate TM^{2}C on the Intel®'s Single-chip Cloud Computer (SCC) (described in Section 5.1). More precisely, we use TM^{2}C with FairCM, its companion contention manager, to run a concurrent hash table benchmark (Section 5.2), and two concurrent applications: bank (Section 5.3) and MapReduce (Section 5.4). In addition, we compare the obtained performance against the one using Back-off-Retry, Offset-Greedy and Wholly. The elastic transaction evaluation is deferred to Section 6.

5.1 The Target Platform: Intel®'s SCC

The Single-chip Cloud Computer (SCC) [22] is an experimental many-core platform developed by Intel® that embeds 48 non-cache-coherent cores on a single die. Its architecture is designed to "scale, in principle, to 1,000 cores" [29] and represents a 6×4 2D mesh of tiles, each tile comprising two P54C x86 cores. Figure 3 provides an overview of SCC's architecture.

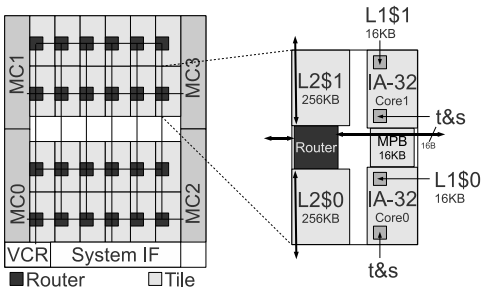


Figure 3. The SCC layout

Every core has 32KB of L1 cache (16KB instruction/16KB data), a separate on-tile 256KB L2 cache, and provides one globally accessible atomic test-and-set register. In addition, each tile has 16KB of SRAM, called the Message Passing Buffer (MPB), intended to be used for implementing message passing. Finally, the SCC processor includes 4 DDR3 Memory Controllers (MC), with a default of $4 * 8 = 32$GB of memory. Every core uses a partition of this memory as its local RAM and the remaining can be allocated as shared memory. An important characteristic of the SCC is the lack of any hardware cache-coherence protocol. The coherence of the shared memory and the MPB should be handled by software.

Settings. The SCC has the following five performance settings:

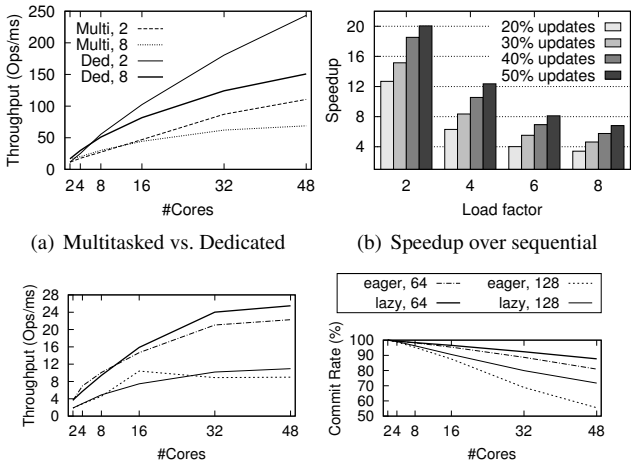
Setting	Tile	Mesh	DRAM
0	533	800	800
1	800	1600	1066
2	800	1600	800
3	800	800	1066
4	800	800	800

The different columns refer to the tile, the mesh, and the memory speed frequency settings (in MHz) respectively. Using a setting other than 0 proved problematic in some cases. Moreover, Intel® recommends using setting 0, consequently the data collected for this section were taken under this setting.

Except for the experiment comparing the multitasking and the dedicated DTM versions, all other measurements used the dedicated DTM TM^{2}C. Specifically, unless explicitly mentioned, the benchmarks use a 24 DTM / 24 application cores setting (the reasoning behind this allocation can be found on Section 5.3).

5.2 Hash Table

The hash table benchmark belongs to the synchrobench suite and supports three operations: a `contains` operation checks if an element exists in the hash table, an `add` inserts an el-



(a) Multitasked vs. Dedicated (b) Speedup over sequential

(c) Eager vs. Lazy write-lock acquisition

Figure 4. The hash table benchmark

ement in the hash table, and a `remove` deletes an element from the hash table. We designed two versions of the operations, a transactional and a sequential.

All operations are given a random value and the update ratio indicates the amount of operations effectively modifying the hash table, while the failed updates count as read-only transactions. In addition, we tested different load factors that indicate the number of elements divided by the number of buckets.

Deployment strategies. Figure 4(a) indicates the performance of the TM^{2}C deployment strategy against the multitasking strategy as described in Section 3.1: unlike the default strategy that dedicates disjoint sets of nodes to run the application and the DTM, the multitasking strategy runs the application and the DTM on the same nodes. We tested these two deployment strategies for load factors 2 and 8 and update ratio 20%. The results outline the performance benefit of using dedicated cores for the DTM, thus confirming our initial thoughts.

Sequential speedup. Figure 4(b) depicts the improvement of TM^{2}C running the hash table over the bare sequential implementation for 20% to 50% update ratios. The transactional implementation runs on 48 cores, including 24 application cores and 24 DTM cores, and performs up to 20 times faster than its sequential counterpart running on a single core. Interestingly, we notice that the speedup decreases for higher load factors. The reason is that a higher load factor raises the duration of hash table operations, thus increasing the probability of conflicts.

Furthermore, a higher update ratios leads to lower performance, for both the sequential and the TM^{2}C versions. This is due to the additional contention induced by update operations. The performance drops for the sequential version is however more important than for the transactional one. The initial hash table resides only in one of the four memory controllers of the SCC, thus utilizing 25% of the memory bandwidth. During the benchmark execution, each core adding a new element stores it in its closest memory controller leading to a better balancing of the load as the update ratio increases.

Eager vs. lazy write-lock acquisition. Figure 4(c) corresponds to the throughput and the commit rate of the hash table benchmark using eager and lazy write-lock acquisition. As described in Section 3.3, we decided to use lazy write-lock acquisition on TM^{2}C. With eager acquisition a transaction asks for the write-lock of the memory location on-time, when the transactional write operation is called. For these tests we implemented a fourth operation on the hash table, namely `move`, which removes an element and inserts a new one. We ran our tests with 30% total updates, 20% of which were `move` operations. We picked this workload because it includes some write operations in the middle of the transaction, thus making the two schemes performing differently.

The results follow our expectations: both schemes perform similarly under low contention, however, when the number of conflicts increases, lazy acquisition outperforms eager. Lazy acquisition has the advantage of keeping the write-locks for a smaller amount of time, hence decreasing the number of detected conflicts and increasing the commit rate, as one can notice on the right graph of Figure 4(c).

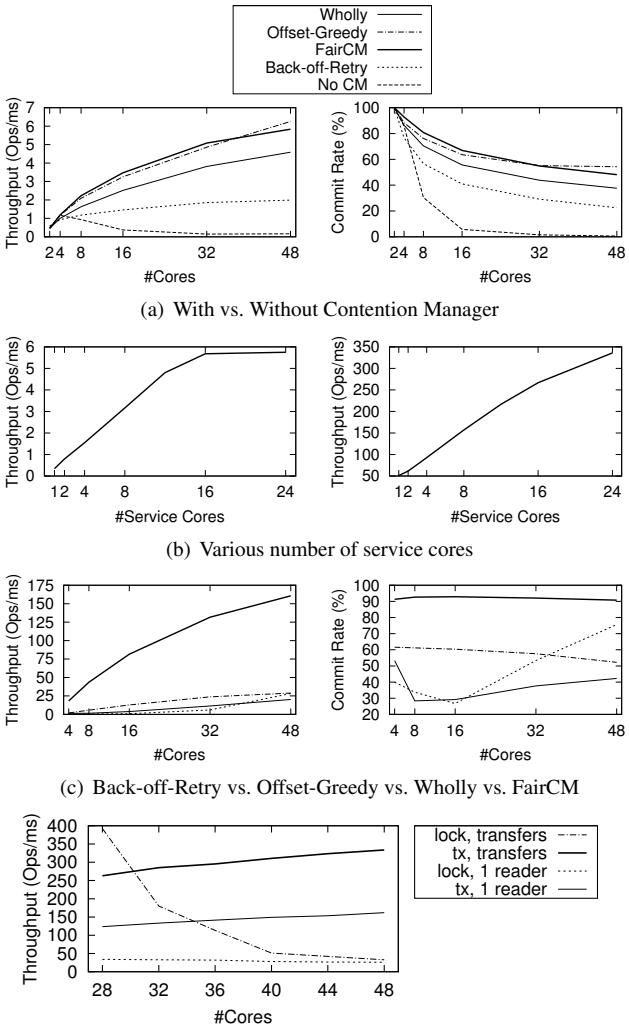
5.3 Bank

Bank is an application consisting of operations for transferring and computing the balance of bank accounts (1024 accounts, unless specified). It was first used to evaluate shared memory STMs [18] and is especially suited to evaluate the effect of livelocks on performance [17]. We compare TM^{2}C to locks using a single global lock as the SCC provides a limited number of test-and-set registers (one per core) which prevents us from using fine-grained locks. We saw two alternatives to this issue: either implementing lock-striping (medium-grained locks) or a hierarchical form of fine-grained software locks. Yet comparing TM, which is easy-to-use, to these approaches, which are difficult to use in non-cache-coherent machine, is unfair.

Contention management benefits. The left and right graphs of Figure 5(a) illustrate the TM^{2}C's throughput and commit rate, respectively, when each core performs 20% balance operations and 80% transfers with and without CM.

Without any CM, the performance drops because of a livelock. In fact, each balance operation acquires a read-lock on every account thus conflicting with concurrent update transactions. Such conflicts emerge either due to a balance operation (RAW conflict), or due to a transfer operation (WAR conflict). Using any of the four CMs performs and scales significantly better because they avoid livelocks.

Comparing different number of service cores. Due to the reasons described in Section 3.1, dedicating some cores to



(a) With vs. Without Contention Manager

(b) Various number of service cores

(c) Back-off-Retry vs. Offset-Greedy vs. Wholly vs. FairCM

(d) Locks vs. Transactions

Figure 5. The bank application

host the TM service is advisable on a many-core. This design decision generates the interesting question of how many cores should be allocated for the service. On TM^{2}C this is a system's parameter. We used the bank application to evaluate the performance of TM^{2}C under different number of service cores.

Figure 5(b) depicts the performance of the bank with 20% balance and 80% transfer operations (left) and with 100% transfer operations (right). The label on the x-axis indicates the number of the service cores, all remaining cores are hosting the application.

Both results explain why we selected to dedicate half of the cores for hosting the TM service in most of our experiments. The results are explained as follows. Firstly, bank[4] does not contain any actual local computations since it consists only of transactional operations. Therefore, the request load produced is very high. Secondly, the message passing on the SCC does not scale particularly well. As an example consider the average latency for a round-trip[5] message. In the case of 2 cores the latency is $5.1\mu s$, while with 48 it increases to $12.4\mu s$ (for more details see Section 7). Consequently, increasing the number of service cores does not entail a linear increase of the system throughput.

Comparing contention managers. Figure 5(c) illustrates in more detail the performance of TM^{2}C when running with each of the four CMs, presented in Section 4. All cores perform transfers, except one which runs balance operations repeatedly.

Offset-Greedy and Wholly exhibit similar performance: the balance and the transfer transactions are prioritized the same, hence the "balance core" degrades the overall throughput. By contrast, FairCM prioritizes the transactions based on the transactional time they consume, therefore the balance operations are significantly more expensive than the transfers. Consequently, FairCM scales well by keeping the abort rate lower than 10%, even for 48 cores, and performs up to 12 and 9 times better than Wholly and Offset-Greedy, respectively.

Back-off-Retry performs similarly to Offset-Greedy and Wholly, but exhibits an interesting behaviour. Up to 16 cores the commit rate drops, but for more than 16 it increases. This is because the core performing balance operations tends to starve. Increasing the number of cores performing transfers, increases the probability that while the "balance core" is scanning the bank will find a RAW conflict. Interestingly, FairCM diminishes the performance of one core to the benefit of the system's throughput by committing 44 balance operations per second as opposed to the 81 of Offset-Greedy. The performance difference of FairCM to the others increases as we increase the number of bank accounts.

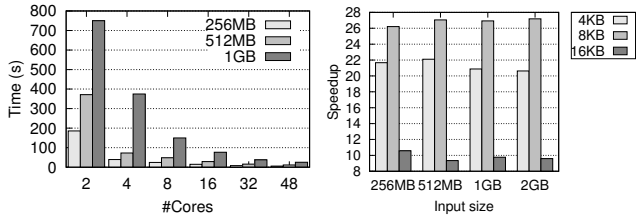
Comparing against locks. Figure 5(d) indicates the throughput of the bank implementation based on TM^{2}C and on locks under two different workloads. These experiments use 2048 bank accounts.

The first workload consists in every core executing transfer operations. Up to 28 cores, the lock-based version (lock, transfers) performs better than the transactional version (tx, transfers). This is not surprising as the sequential implementation of a transfer performs only four accesses to the shared memory. However, for more than 28 cores, the performance of the lock-based version degrades due to the contention on the lock, while the transactional version keeps scaling.

The second workload comprises a core that repeatedly performs balance operations, while all others transfer. In this case, the transactional implementation (tx, 1 reader) performs and scales better than the lock-based one (lock, 1 reader), regardless of the number of cores. This is expected

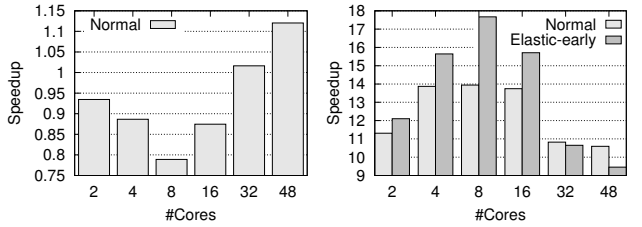
[4] The same applies to the hash table and linked list benchmarks.

[5] A round-trip consists in a request followed by a response.



(a) Duration (b) Speedup over sequential

Figure 6. The MapReduce application



(a) Normal vs. Elastic-early (b) Normal & Elastic-early vs. Elastic-read

Figure 7. The linked list benchmark

since in the lock-based implementation the core that executes the balance operations delays all other cores from executing the lighter transfer operations, while TM^{2}C handles this case properly.

5.4 MapReduce

To test TM^{2}C under a heterogeneous workload combining transactional and local computations, we developed a MapReduce-like application. The application takes a text file as an input and counts the number of occurrences of each letter in the file. Typical MapReduce implementations use a master node to coordinate the map and reduce phases. TM^{2}C takes here the role of allocating chunks of the file to cores and of updating the total statistics atomically thus removing the need for a master node.

Scalability and sequential speedup. Figure 6(a) indicates the experiment duration as the number of cores increases. Figure 6(b) indicates the speedup of TM^{2}C over the sequential implementation for different chunks sizes (4, 8, and 16 KBytes). Since the transactional load is low, only one core is dedicated to run the DTM service so that the 47 remaining cores run the application. Our evaluation reveals that using an 8KB chunk size leads to the best performance. This can be explained by the L1 cache size of each core. Each core has a 16KB data cache, but since it is shared between the operating system and the application it is not fully available to the latter.

6. Distributed Elastic Transactions on TM^{2}C

The *elastic transaction* model [13] is a variant of the classical transactional model particularly efficient when implementing search structures. Elastic transactions complement the classical transactions and can be optionally used instead to provide higher performance. They ensure atomicity of some high level operations while ignoring their false low level conflicts. An elastic transaction relaxes the atomicity between all the shared read accesses of its read-only prefix by requiring only that consecutive read accesses remain atomic. Consider the following sorted linked list example:

$$\textbf{head} \rightarrow \boxed{\text{n1}} \rightarrow \boxed{\text{n2}} \rightarrow \boxed{\text{n3}} \rightarrow \boxed{\text{n4}} \rightarrow \textbf{tail}$$

when a searching transaction reaches node 3, node 1 is no more relevant to the search, because even if it is modified by a concurrent transaction (producing a WAR conflict), the search will not be semantically affected. By ignoring these false conflicts, elastic transactions enable higher concurrency.

6.1 Implementations of the Elastic Transaction Model

The elastic transactional model can be implemented in various ways. We designed two implementations and a linked list data structure to evaluate it.

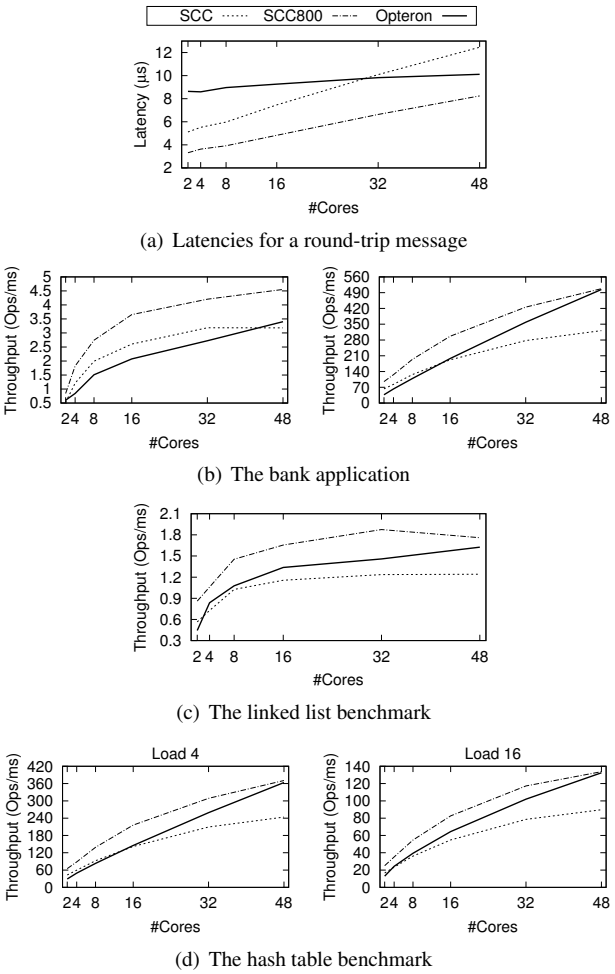
Our first implementation (elastic-early) employs an explicit `release` action, similar to the one used in DSTM [19], in order to discard a read entry from the transaction read set prior to commit time. In our case, this release action is used by a transaction to release one of its acquired read-locks immediately after acquiring new ones on subsequent data. Using such an early release, we can ensure that only consecutive read accesses are atomic as required by elastic transactions.

Our second implementation (elastic-read) was designed using read-validation. Instead of acquiring the relevant read-locks, the transaction performs read validation. This technique relies on the fact that if a concurrent transaction commits an update, the new value will be visible to a read validation, since the committed transaction can only write new/different values to the altered fields. For example, for the `contains` operation it is important to validate node i after stepping to node $i+1$. If the value of node i did not change, then the transaction proceeds normally, otherwise it has to be aborted.

6.2 Evaluating the Two Implementations

The linked list benchmark also comes from the synchrobench suite and exports the same operations as the hash table (Section 5.2). We used a 2048 element sorted linked list for this test. Each core performs 20% update operations (`add`, `remove`) and 80% `contains`.

Figure 7(a) depicts the improvement of the elastic-early version over normal transactions. The elastic transactions diminish the abort rate to less than 1%, even for 48 cores, so one would expect a better performance improvement. The



(a) Latencies for a round-trip message

(b) The bank application

(c) The linked list benchmark

(d) The hash table benchmark

Figure 8. TM^{2}C performance on many-core (SCC/SCC800) vs. multi-core (Opteron)

reason for the limited speedup is that each early release operation requires an extra message to the DTM, thus significantly increasing the communication load of the system and producing extra transactional overheads.

Figure 7(b) depicts the speedup of the elastic-read implementation over the other two. Similarly to the elastic-early, elastic-read diminishes the abort rate. However, elastic-read additionally reduces the number of messages sent by increasing the number of accesses to the shared memory. On the SCC, a memory access is faster than a message delivery, therefore the read validation significantly increases the performance. The speedup drops for more than 8 cores due to memory congestion.

7. TM^{2}C on a Cache-Coherent Multi-Core

In order to verify the portability of TM^{2}C and evaluate its performance and scalability on a different platform we ported it on a cache-coherent multi-core machine.

7.1 Porting TM^{2}C to Multi-cores

The underlying message passing communication paradigm makes TM^{2}C easily portable to different architectures, including multi-core machines with cache-coherence support. We also ported the simple Back-off-Retry contention manager to obtain a common ground for comparison. Like the SCC, our multi-core machine also embeds 48 cores in total. More precisely, it consists of four 2.1 GHz 12-core AMD Opteron^{TM} processors and 32 GB of RAM running Linux (Ubuntu 10.04 64 bit, kernel version 2.6.32). The L1 cache size is 128 KB, the L2 cache is 512 KB, and each of the processors has a 12 MB L3 cache. To take benefit of the inherent hardware cache-coherence protocol provided by the multi-core machine, we used a message passing library similar to the one of Barrelfish [3] that translates cache lines into core-to-core communication channels. Additionally, we used the SCC on both its slowest and fastest performance settings (see Section 5.1), yet note that the clock frequencies of the many-core remain more than twice slower than the one of the multi-core.

Figure 8(a) illustrates the latency of message passing in TM^{2}C. Specifically, we use the dedicated service cores (one half dedicated to the DTM, the other half dedicated to the application services) and set each application core to send one million messages evenly distributed to all service cores. Upon reception of a message, a service core responds immediately, without performing any local computation. The results reveal that asynchronous message passing on the SCC does not scale well. This degradation stems from the software-based message passing implementation of the SCC. In order to be able to asynchronously receive messages, a core has to repeatedly poll a flag for any other core to be able to detect any incoming messages. However, the SCC on its fastest setting (SCC800) provides faster message passing than the messaging implementation used on the multi-core.

7.2 Experimental Comparison

We compare TM^{2}C on the multi-core and the many-core using the bank application, the hash table and the linked list benchmarks. On 48 cores, the multi-core and SCC800 performed similarly. SCC800 has slightly faster message passing but the multi-core has significantly faster processing speed. Since our benchmarks make heavy use of the DTM service, messaging is more important than the clock frequency.

We run TM^{2}C on the bank application (Figure 8(b)) under two workloads: the first consists of 20% balance and 80% transfer operations (left graph) and the second contains only transfer operations (right graph). The former workload reveals that the SCC behaves better under heavy contention. The latter, which is a low contention workload, follows the messaging latencies. We also run TM^{2}C both on the linked list (Figure 8(c)) and the hash table (Figure 8(d)) with an

initial size of 512 and 10% update ratio. Linked list is another high contention example. In this case, the multi-core performs better relatively to the bank. All operations of the linked list include a sequential search among its elements, creating some hotspots on the first elements. Consequently, caching improves the memory access latencies.

These results are preliminary and a more extensive evaluation is necessary to precisely assess the architectural artifacts that affect the observed performance. Yet, the observed results (in particular on the low contention hash table) confirm the difference of message latencies we observed on both architectures. Finally, a general observation is that TM^{2}C seems to scale almost linearly with the number of application cores on low contended workloads when the message passing scales accordingly, independently from the considered architectures.

8. Discussion

In this section, we discuss a possible extension to support privatization and we elaborate on the portability of TM^{2}C.

Privatization. The action of making data private to some thread is known as *privatization*. Privatization is appealing when using transactional memory to support legacy code or to avoid the overhead of transactional wrappers when accessing some private data.

The Intel®'s SCC allows the programmer to define barriers that can be employed to guarantee that after some execution point all transactions have completed. Such technique allows to delimit a part of the program where some data is shared among transactions, and a subsequent part where the same data is private to some thread. In TM^{2}C a more generic solution would be to implement barriers using the available message passing paths among the application cores. When the application reaches a barrier it sends a barrier-reached message to all the other application cores and blocks until it receives a message from each of them.

Portability. As our experience illustrates (see Section 7), TM^{2}C can be ported to platforms providing reliable asynchronous message passing. However, both versions (many-core and multi-core) utilize the existing shared memory of the platform. We have started implementing a Partitioned Global Address Space (PGAS) memory model for TM^{2}C. We expect the benefit from PGAS to be twofold. Firstly, PGAS will increase the portability of the system since message passing will be the only requirement. Towards this direction we are working on implementing a version of TM^{2}C running on clusters. Secondly, PGAS will act as a software-level cache-coherence protocol since the data will be locally cached on the residing node. We anticipate that the data caching will diminish the memory load and increase the performance.

9. Related Work

Transactional memory (TM) was originally proposed to simplify concurrent programming to avoid lock-related issues, like deadlocks [20, 39]. They were dedicated to shared memory systems, all relying on an underlying hardware cache-coherence [35, 36]. More recently, much effort was spent in implementing the TM abstraction on top of clusters of distributed machines, resulting in various distributed transactional memories (DTMs) [5, 25, 27, 33].This distribution unveiled new research challenges, whose prominent one is possibly to guarantee transaction termination despite message-based synchronization. Our solution is the first to exploit many-cores to provide efficient transactions that are guaranteed to terminate.

A first class of DTMs use a separate centralized service in order to arbitrate contention between transactions. Distributed Multi-Versioning (DMV) [27] is a replicated DTM that exploits multi-version concurrency control to minimize the number of aborts. DMV operates in two different modes. The first mode requires a global token to protect the broadcast of updates in order to keep the memory consistent, but suffers from livelocks. The second mode relies on a centralized master node, which may hamper the scalability by serializing all update transactions, even non-conflicting ones. DiSTM [25] is a framework for prototyping and testing software cache-coherence protocols for TMs. The underlying TCC protocol is described as a decentralized coherence protocol, yet it needs a single master node and a global ticket mechanism.

A second class of DTMs are *control-flow* in that they handle transactions that execute on a distributed set of data. The challenge of such a control-flow technique is to guarantee that the conflict resolution adopted at some place does not contradict another conflict resolution adopted at a remote place. Cluster-TM [5] is a DTM designed for large-scale clusters. It introduces techniques for minimizing the communication among nodes by exploiting data locality. Unfortunately, Cluster-TM suffers from livelock, being unable to guarantee that an issued transaction will eventually commit. Snake D-STM [34] utilizes local contention management where each node takes decision based on local information. Such a local decision does not avoid the creation a global cycle among the aborting relations, also leading to livelocks.

A third class of DTMs are *data-flow* in that they move data among processors executing transactions and rely on an underlying cache-coherence protocol to invalidate distant transactions upon conflict detection. The crux here is rather to ensure that communication asynchrony does not stale contention management. New directory protocols were accordingly designed to move and retrieve data in a cache-coherent way [2, 21, 41] but none of them proposes a full-fledged DTM protocol. Combine [2] guarantees termination of individual move and retrieval operations, not of transactional groups of moves/retrievals, and in a distributed envi-

ronment the Greedy contention manager needs nodes to synchronize their clock [40]. DecentSTM [4] is another data-flow STM that utilizes consensus on the cached copies but does not guarantee livelock freedom. As opposed to Snake D-STM, the Transactional Forwarding Algorithm (TFA) of RMI-DSTM [32] is a data flow algorithm that relies on an underlying directory protocol. It uses a modification of Lamport's clocks in order to have a synchronized timestamp to be used for object versioning. Although it guarantees strong progressiveness [16], it remains livelock-prone.

Finally and in accordance with [30, 31], there was lately an extensive work towards designing replication techniques for DTMs. To our knowledge, all existing techniques build on top of the Atomic Broadcast [10]; a rather strong and expensive (regarding communication) primitive. D^{2}STM [9] is a fully replicated fault-tolerant DTM which uses a certification scheme for guaranteeing the consistency among different nodes. D^{2}STM is also livelock-prone. Asynchronous Lease-Based Replication (ALC) [7] is a certification scheme for replicating STMs. A transaction needs to acquire the leases that correspond to its data-set in order to commit. In case of an abort and retry the transaction keeps the acquired leases, but there is a chance that these data do not coincide with the newly accessed data. For the aforementioned reason, ALC cannot guarantee livelock-freedom since every new transaction run may need a disjoint set of leases, hence it is not guaranteed it will be able to commit. The authors suggest that this problem could be bypassed if all transactions explicitly request for the whole set of leases (sort of a global lock), solution that hinders concurrency. SCert is a complement to ALC replication/certification scheme which inherits ALC's livelock problems [8].

To our knowledge TM^{2}C is the first TM protocol for many-core systems. It does not require any underlying cache-coherence protocol, thus avoiding data lookup, cache misses and false sharing. It detects conflicts eagerly by exploiting the low network-on-chip latency to rapidly grant shared read accesses to memory bytes. Once granted, the read access becomes visible to other transactions thus allowing conflicts to be detected at read time. Last but not least, all its transactions terminate. Upon conflict detection, any of the three companion contention managers resolve the conflict ensuring that there is no executions in which one transaction may repeatedly abort another.

10. Conclusions

We have proposed TM^{2}C, the first transactional memory for many-cores, the family of processors that promise to reconcile high performance with low energy consumption at the cost of trading hardware cache-coherence for message passing. TM^{2}C exports the standard transactional interface hiding the complex underlying on-chip communications from the programmer. It incorporates the first starvation-free distributed contention manager, FairCM, thus preventing continuous contention from repeatedly aborting the same transactions. Moreover, it ascertains the fair usage of the system by every core. We implemented and evaluated TM^{2}C on the Intel®'s SCC many-core processor, attesting that TM^{2}C exploits the scalability of many-cores even on irregular applications with many dependencies.

As for future work, we plan to introduce fault-tolerance to TM^{2}C. Contemporary many-cores consist of less than a hundred cores, thus the non-failure assumption is realistic. However, many-cores are expected to scale in the number of cores, hence on a single many-core node failures should become more frequent.

Another research direction is to automate the selection of the DTM service cores. Currently, the cores that host the TM^{2}C are statically predetermined. Under heterogeneous workloads it would be preferable for the system to vary the number of service cores depending on the transactional load.

Acknowledgments

We wish to thank our shepherd, Maurice Herlihy, and the anonymous reviewers for their fruitful comments, and the Intel® MARC Community for its support while programming on the SCC. Part of the research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement 248465, the S(o)OS project.

References

[1] Dennis Abts, Natalie D. Enright Jerger, John Kim, Dan Gibson, and Mikko H. Lipasti. Achieving predictable performance through better memory controller placement in many-core cmps. In *ISCA*, pages 451–461, 2009.

[2] Hagit Attiya, Vincent Gramoli, and Alessia Milani. A provably starvation-free distributed directory protocol. In *SSS*, pages 405–419, 2010.

[3] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schupbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP*, pages 29–44, 2009.

[4] Annette Bieniusa and Thomas Fuhrmann. Consistency in hindsight: A fully decentralized stm algorithm. In *IPDPS*, pages 1–12, 2010.

[5] Robert Bocchino, Vikram Adve, and Bradford Chamberlain. Software transactional memory for large scale clusters. In *PPoPP*, pages 247–258, 2008.

[6] Shekhar Borkar. Thousand core chips: a technology perspective. In *DAC*, pages 746–749, 2007.

[7] Nuno Carvalho, Paolo Romano, and Luís Rodrigues. Asynchronous lease-based replication of software transactional memory. In *Middleware*, pages 376–396, 2010.

[8] Nuno Carvalho, Paolo Romano, and Luís Rodrigues. SCert: Speculative certification in replicated software transactional memories. In *SYSTOR*, pages 10:1–10:13, 2011.

[9] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luís Rodrigues. D2STM: Dependable Distributed Software Transactional Memory. In *PRDC*, pages 307–313, 2009.

[10] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, pages 372–421, 2004.

[11] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *DISC*, pages 194–208, 2006.

[12] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*, pages 237–246, 2008.

[13] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic Transactions. *DISC*, pages 93–107, 2009.

[14] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *PODC*, pages 258–264, 2005.

[15] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *PPoPP*, pages 175–184, 2008.

[16] Rachid Guerraoui and Michal Kapalka. The semantics of progress in lock-based transactional memory. In *POPL*, pages 404–415, 2009.

[17] Derin Harmanci, Vincent Gramoli, Pascal Felber, and Christof Fetzer. Extensible transactional memory testbed. *JPDC*, 70(10):1053–1067, 2010.

[18] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA*, pages 253–262, 2006.

[19] Maurice Herlihy, Victor Luchangco, Mark Moir, and William Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, 2003.

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

[21] Maurice Herlihy and Ye Sun. Distributed transactional memory for metric-space networks. In *DISC*, pages 58–208, 2005.

[22] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *ISSCC*, pages 108–109, 2010.

[23] intel. Intel transactional memory abi. http://software.intel.com/file/8097, 2009.

[24] Leonidas Kontothanassis and Michael Scott. Software cache coherence for large scale multiprocessors. In *HPCA*, pages 286–295, 1995.

[25] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. DiSTM: A Software Transactional Memory Framework for Clusters. In *ICPP*, pages 51–58, 2008.

[26] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *ISCA*, pages 148–159, 1990.

[27] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP*, pages 198–208, 2006.

[28] Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5, 2006.

[29] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. The 48-core SCC processor: the programmer's view. In *SC*, pages 1–11, 2010.

[30] Paolo Romano, Nuno Carvalho, and Luís Rodrigues. Towards distributed software transactional memory systems. In *LADIS*, pages 4:1–4:4, 2008.

[31] Paolo Romano, Luís Rodrigues, Nuno Carvalho, and Joäo Cachopo. Cloud-tm: harnessing the cloud with distributed transactional memories. *SIGOPS Oper. Syst. Rev.*, pages 1–6, 2010.

[32] Mohamed Saad and Binoy Ravindran. Control flow distributed software transactional memory. In *SSS*, 2011.

[33] Mohamed Saad and Binoy Ravindran. Supporting STM in Distributed Systems: Mechanisms and a Java Framework. In *Transact*, 2011.

[34] Mohamed Saad and Binoy Ravindran. Transactional Forwarding Algorithm. Technical report, Virigina Tech, 2011.

[35] Bratin Saha, Ali-Reza Adl-Tabatabai, Anwar Ghuloum, Mohan Rajagopalan, Richard L. Hudson, Leaf Petersen, Vijay Menon, Brian Murphy, Tatiana Shpeisman, Eric Sprangle, Anwar Rohillah, Doug Carmean, and Jesse Fang. Enabling scalability and performance in a large scale CMP environment. In *EuroSys*, pages 73–86, 2007.

[36] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP*, pages 187–197, 2006.

[37] William Scherer and Michael Scott. Contention Management in Dynamic Software Transactional Memory. In *CSJP*, 2004.

[38] William Scherer and Michael Scott. Advanced contention management for dynamic software transactional memory. In *PODC*, page 240, 2005.

[39] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[40] Bo Zhang. *On the Design of Contention Managers and Cache-Coherence Protocols for Distributed Transactional Memory*. PhD thesis, Virginia Tech, 2009.

[41] Bo Zhang and Binoy Ravindran. Relay : A Cache-Coherence Protocol for Distributed Transactional Memory. In *OPODIS*, pages 48–53, 2009.