TRANSACTIONAL MEMORY: AN OVERVIEW

Tim Harris
Microsoft Research

Adrián Cristal
Osman S. Unsal
Barcelona Supercomputing Center

Eduard Ayguade
Barcelona Supercomputing Center and Universitat Politècnica de Catalunya

Fabrizio Gagliardi
Burton Smith
Microsoft

Mateo Valero
Barcelona Supercomputing Center and Universitat Politècnica de Catalunya

WRITING APPLICATIONS THAT BENEFIT FROM THE MASSIVE COMPUTATIONAL POWER OF FUTURE MULTICORE CHIP MULTIPROCESSORS WILL NOT BE AN EASY TASK FOR MAINSTREAM PROGRAMMERS ACCUSTOMED TO SEQUENTIAL ALGORITHMS RATHER THAN PARALLEL ONES. THIS ARTICLE PRESENTS A SURVEY OF TRANSACTIONAL MEMORY, A MECHANISM THAT PROMISES TO ENABLE SCALABLE PERFORMANCE WHILE FREEING PROGRAMMERS FROM SOME OF THE BURDEN OF MODIFYING THEIR PARALLEL CODE.

The advent of shared-memory multicore microprocessors has created an immense opportunity to exploit thread-level parallelism. In most applications, parallel thread execution requires synchronization or ordering mechanisms for accessing shared data. Traditional multithreaded programming models usually offer a set of low-level primitives, such as locks, to guarantee mutual exclusion; ownership of one or more locks protects access to shared data. Locks are complex to use and error prone—especially when a programmer is trying to avoid deadlock situations or to achieve better scalability on highly parallel hardware by using fine-grained locking. Consequently, the programming and computer architecture communities are concerned that a parallel-programming productivity and performance wall might be looming.

Transactional memory (TM) is a promising mechanism for tackling this problem by abstracting some of the complexities associated with concurrent access to shared data.[1] With TM, multiple threads can simultaneously try to access shared memory locations in an atomic way, so all the accesses of a specific thread succeed or none, within the scope of what we call an *atomic transaction*. TM has its roots in transaction management in concurrent database systems. However, although it uses mechanisms similar to classical database transactions, TM's main purpose is different: to handle concurrency challenges in shared-memory chip multiprocessors.

Transactions replace locking with atomic execution units, so that the programmer can focus on determining where atomicity is necessary, rather than on the mechanisms that enforce it. For example, the following code segment shows an example atomic region in a simple kernel that computes the histogram of an array:

```
atomic {
        hist[array[i][j]]++;
}
```

With this abstraction, the programmer identifies the operations that form a critical section, while the TM implementation determines how to run that critical section in isolation from other threads.

Typical TM implementations optimistically run transactions in parallel, assuming

Published by the IEEE Computer Society.

that the transactions won't perform conflicting memory accesses. Generally, "conflicting" means in violation of a temporal order. Most often, a load (read) operation from an ongoing transaction has failed to use the result of a store (write) operation from a previous transaction. Here, "previous" usually has a serial temporal-ordering sense, although other logical orders are also possible.

If the transactions don't conflict, the optimism has paid off. The transactions did not have to contend for a common mutual exclusion lock for the data they updated. And, in many implementations, a transaction making read-only accesses to shared data allows all the data to remain in the core's data cache in shared mode, helping scalability.

However, if transactions do attempt conflicting accesses, the optimism has not paid off. The TM must abandon the work of one of the conflicting transactions, ensuring that any side effects of their attempted work are not visible to other threads, before reexecuting the abandoned transactions. The mechanisms by which the TM detects conflicts and contains the effects of abandoned transactions are the focus of the implementation techniques and case studies discussed in this article.

Compared with TM, locks are pessimistic. With mutual exclusion locks, only one thread can hold a lock at a time, whereas in most TM implementations, more than one thread can access a critical section simultaneously. Because actual conflicts are rare in many programs, the optimistic TM approach makes more sense as a future programming model. TM has the following advantages over locks:

- TM provides a higher-level abstraction for writing concurrent programs, letting the programmer concentrate on the algorithm instead of complex mechanisms such as locks.
- TM provides a better trade-off between scaling and implementation effort. Although algorithms using fine-grained locking can scale well, they are notoriously difficult to design.
- TM is inherently deadlock free. Livelock can be a problem, but it is easier to deal with than deadlock.
- As a side benefit, TM provides failure atomicity.

Of course, TM is not without disadvantages. First, as with many high-level programming abstractions, a carefully designed algorithm using lower-level primitives can outperform an algorithm using TM. Second, there are many questions about how to expose TM to programmers—exactly what kind of abstractions to provide and what kind of performance tuning and debugging tools to develop.

Basic TM concepts

A transaction is a sequence of instructions, including reads and writes to memory, that either executes completely (commits) or has no effect (aborts). When a transaction commits, all its writes become visible, and other transactions can use those values. When a transaction aborts, the system discards all its speculative writes.

To support transaction execution, a TM system needs a data-versioning mechanism to record the speculative writes. The system should discard this speculative state on an abort or use it to update the global state on a successful commit. The two usual approaches to implementing data versioning are to use either an undo log or buffered updates. In using an undo log, a transaction applies updates directly to memory locations, while logging the necessary information to undo the updates in case of an abort. In contrast, approaches using buffered updates keep the speculative state in a transaction-private buffer until commit time. If the commit succeeds, the buffer drops the original values before the store instructions and commits the transaction's speculative stores to memory.

A transaction's instruction sequence can be explicitly or implicitly delimited. Some high-level programming languages include constructs that explicitly define the extent of transactions—for example, the "atomic" statement shown earlier—whereas others provide lower-level operations to explicitly start and end transactions. In other cases,

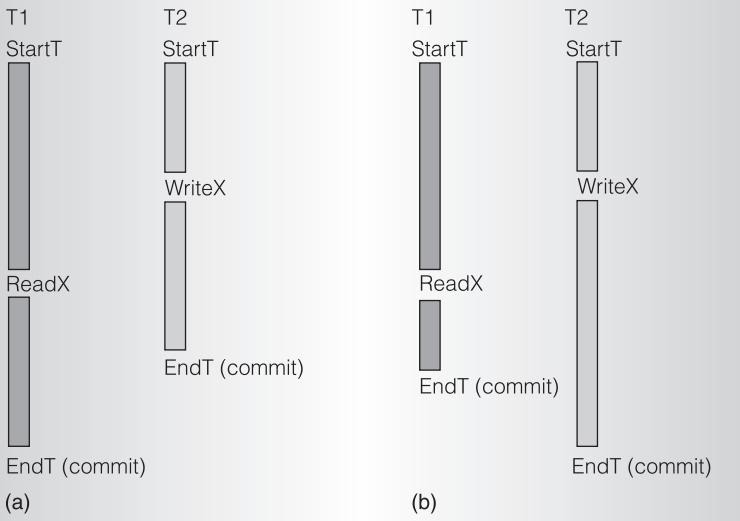

Figure 1. Execution timelines for two pairs of transactions. Both eager and lazy conflict detection would detect a conflict in the first transaction pair (a). In the second pair (b), eager conflict detection would detect the conflict, but lazy conflict detection would allow both transactions to commit.

transactions start implicitly after execution of a transactional read or write operation or immediately after the commit of another transaction in the instruction stream.

A TM system can abort transactions either explicitly by executing an abort instruction or implicitly because of data conflicts with concurrent transactions. Two issues are related to conflicts: detection and resolution. To detect and handle conflicts, each running transaction is typically associated with a *read set* and a *write set*. Inside a transaction, the execution of each transactional load instruction adds the memory address to the read set. Each transactional store instruction adds the memory address and value to the write set.

Conflict detection can be either eager or lazy. Eager conflict detection checks every individual read and write to see if there is a conflicting operation in another transaction. Eager conflict detection requires a transaction's read and write sets to be visible to all the other transactions in the system. On the other hand, in lazy conflict detection, a transaction waits until it tries to commit before checking its read and write sets against other transactions' write sets. Figure 1 illustrates conflict detection with two sequences of memory accesses. In both sequences, eager conflict detection would detect the conflict at the read of location X (ReadX) by T1 because T2 has already written to it (WriteX). In Figure 1a, lazy conflict detection would detect a conflict because T2 tries to commit first, implying that T1 should have used the result of T2's WriteX operation. In Figure 1b, however, both T1 and T2 can commit because T1 commits first, and its ReadX need not use the result of T2's WriteX.

Different TM systems detect conflicts at different levels of granularity—for example, the level of objects (arrays, lists, and so forth), individual fields in a data structure, or cache lines. This choice introduces tradeoffs in terms of time and space overheads and false sharing.

Another fundamental design choice is how to resolve a conflict once it is detected. Usually, a system must resolve a conflict by aborting one of the transactions involved in it. The decision of which transaction to abort is complex. Consider the example shown in Figure 2. Assume that the TM system uses eager conflict detection. When T1 performs ReadX, the system detects a conflict with WriteX in T2, and one transaction must be aborted. At this point, if the conflict resolution policy decides to abort T2, later T1 will conflict again with T3 (ReadY and WriteY), and another transaction abort will be necessary. However, if the resolution policy had decided to abort T1, both T2 and T3 could have finished, with only one transaction aborted instead of two.

Two other important TM concepts are blocking and choice. Blocking is the explicitly expressed mechanism that aborts a transaction, and choice is a set of transactional actions undertaken as an alternative to blocking. Providing proper hardware, runtime, and language support to implement blocking and choice are practical design issues that all TM systems must address.

TM implementations

The two main TM implementation styles are hardware based and software based. Historically, the earliest design proposals

were hardware based. Researchers proposed software transactional memory (STM) to address, among other things, inherent limitations of earlier forms of hardware transactional memory (HTM), such as a lack of commodity hardware with the proposed features and a limited number of locations that a transaction can access.

Beyond these two approaches, two mixed approaches are undergoing active research. Hybrid transactional memory (HyTM) supports HTM execution but falls back on STM transactions when hardware resources are exceeded.[2,3] Hardware-assisted STM (HaSTM) combines STM with new architectural support to accelerate parts of the STM's implementation.[4,5] These designs provide very different performance characteristics. HyTM provides near-HTM performance for short transactions but encounters a performance cliff when falling back to STM. In contrast, HaSTM provides performance somewhere between HTM and STM.

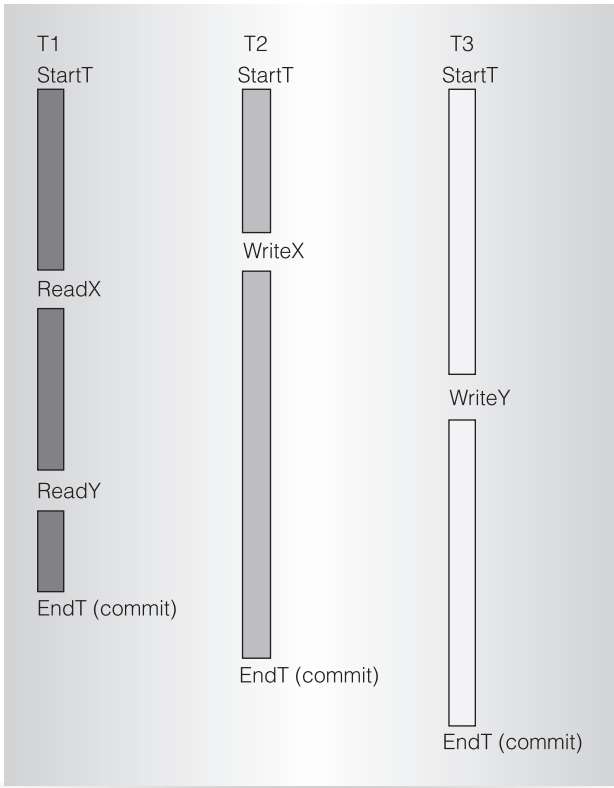

Figure 2. Execution timeline for three transactions. Transaction 1 conflicts with Transaction 2 (ReadX and WriteX) and later with Transaction 3 (ReadY and WriteY). If the conflict resolution policy aborts Transaction 2 at the first conflict, it will have to abort another transaction at the second conflict. If the policy aborts Transaction 1 at the first conflict, Transactions 2 and 3 can both commit.

Hardware transactional memory

The first HTM designs were minimalist (but fully functional and expressive) approaches based on modifying the cache consistency protocols and complementing the instruction set architecture (ISA) with a small set of new instructions. These designs keep the speculative state in an extended or partitioned cache (or a buffer) until the transaction either commits or aborts. These two modifications are sufficient for basic HTM.

Several proposed alternative approaches require minimal or no ISA support or cache modifications. For example, thread-level speculation relies on speculation past locks to admit multiple threads in the same critical section.[6,7] Another approach that doesn't require ISA modifications is implicit transactions.[8]

API design: ISA additions. The conceptual support required at the ISA level to delimit transactions is start transaction (STR) and end transaction (ETR) instructions. In addition, special versions of load (TLD) and store (TST) instructions for transactional read and writes are necessary. However, depending on the hardware implementation, some of these instructions might not be needed. For example, Herlihy's and Moss's implementation doesn't use STR; instead, when a transaction executes its first TLD or TST operation, a flag is set at the core indicating that the core is engaged in a transaction.

Adding special instructions for abort (ABR) and validation (VLD) of a transaction makes several optimizations possible. For example, the system can use ABR to select a victim transaction for aborting under program control. Usually, the implementation of abort policies involves a trade-off between performance (total throughput) and fairness. Similarly, VLD can provide energy savings. For example, consider a very long-running transaction that has a conflict early in its execution timeline. Waiting until

its commit instruction to check and roll back this transaction will waste a lot of energy and work. Using VLD throughout this transaction's execution timeline will catch the conflict early, and the transaction can roll back without wasting energy.

Data versioning and conflicts: cache or buffer modifications. HTM systems keep transactions' speculative state mostly in the data cache or in a hardware buffer area. Therefore, unlike STM systems, which often detect conflicts at the granularity of programming-language objects, HTM systems work at the word or cache line level. The systems keep transactional loads and stores in a separate transactional cache or in conventional data caches augmented with transactional support.[9] In either case, the modifications are minimal because transactional support relies on extending existing cache coherence protocols, such as MESI (modified, exclusive, shared, invalid), to detect conflicts and enforce atomicity.

One of the designs described by Herlihy and Moss keeps the transaction's read set and write set in the data cache and keeps an extra version of the transaction's tentative updates. For this design, just two extra bits per cache line (assuming cache line granularity) are sufficient to indicate whether the line is to be discarded on commit (for lines holding unmodified data) or on abort (for speculatively modified lines). The protocol extensions specify whether the version is to be kept or dropped. A conflict means that a load has read invalid data and the associated transaction must abort. In that case, the write set of the aborting transaction (associated with the tentative store instructions) is dropped. On the other hand, if the transaction commits, the version of the original values before the store instructions are dropped, and the transaction's speculative stores are committed to memory.

In one variant of the design, the system keeps the original state in main memory and the speculative state in the data cache. Just one bit per cache line is sufficient for this design. The bit is set when a transaction accesses the line. Upon commit and abort, this bit is cleared; for abort, the modified lines with this bit set are also invalidated.

Software transactional memory

The extensive research on software TM implementations goes back to Shavit's and Touitou's first use of the term in 1995.[10] Here, we look at the basic case of non-nesting transactions that make updates to shared memory within a single multithreaded process, focusing on the main problems that an STM must tackle: It must provide separate per-thread views of the heap as transactions execute, and it must provide a mechanism for detecting and resolving conflicts between transactions. Larus and Rajwar explore this topic in greater depth.[1]

API design: Managing transactional state. Whereas traditional HTM systems use data caches to buffer transactions' tentative state, an STM implementation must provide its own mechanism for concurrent transactions to maintain their own views of the heap. Such a mechanism allows a transaction to see its own writes as it continues to run and allows memory updates to be discarded if the transaction ultimately aborts.

A high-level distinction between STM implementations is how they organize data in memory. One approach separates transactional data and ordinary data, introducing a distinct memory format for transactional objects. An alternative approach allows data to retain its ordinary structure in memory, and the STM uses separate structures to maintain its own metadata. There are advantages and disadvantages to each approach.

Examples of the first type of design are dynamic STM (DSTM), object-based STM (OSTM), and adaptive STM (ASTM) systems.[11–13] These systems identify transactional data by references to object headers, which must be opened to gain transactional access to the object body. For example, the STM implementation uses the object header to track which transactions are currently accessing the object. In OSTM, for example, the programming API provides operations to open an object header, returning

a reference to a copy of the object's body for the transaction to use:

```
o_body *OpenForReading(tx
  *tx, o_header *o);
o_body *OpenForWriting(tx
  *tx, o_header *o);
```

The API must differentiate read and write accesses because multiple concurrent transactions can share the same object body as long as they are all reading from it. If a transaction must update an object, OpenForWriting returns a private shadow copy of the object body. A program using OSTM must not update object bodies that have been opened only for reading, and when working on pointer-based data structures, it must always add an extra level of indirection by storing references to object headers rather than direct references to object bodies.

The OSTM runtime system maintains a read set with the objects that a transaction has read from and a write set with the objects that it has updated. Aborting a transaction simply means discarding any shadow copies that have been created for it. Committing a transaction means 1) atomically checking that no conflicting transaction has updated objects in the read set or the write set, and 2) updating the object headers for objects in the write set, thus publishing the private shadow copies as the object's new contents.

In contrast, the Bartok STM system makes no low-level distinction between ordinary and transactional data.[14] This implementation holds the metadata that the STM uses for concurrency control in separate structures, with the STM using a function to map a word's address to a particular transactional metadata word (TMW) that manages that data. This means that the transactional API includes functions to open the TMWs for the data that it will read from or write to:

```
void OpenForReading(tx *tx,
   tmw *t);
void OpenForWriting(tx *tx,
   tmw *t);
```

```
atomic {
   hist[index]++;
}
```
(a)

```
atomic {
   …
   OpenForReading(tr, TMW_FOR(index));
   OpenForWriting(tr, TMW_FOR(hist));
   int *addr = &hist[STMRead(tr, &index)];
   STMWrite(tr, addr, STMRead(tr, addr) + 1);
   …
}
```
(b)

Figure 3. Translation of an atomic region into calls on a Bartok-STM-like API: original source code (a); expansion of memory accesses to use the STM API (b). Function TMW_FOR maps an address to the location of the TMW used for the address's concurrency control—for example, an extra header word on the object holding the address.

and functions for accessing the data that directly refer to memory addresses:

```
word STMRead(tx *tx, word *a);
void STMWrite(tx *tx, word *a,
   word d);
```

As in OSTM, OpenForReading and OpenForWriting build up the read and write sets that are used by the STM for concurrency control. However, Bartok STM differs from OSTM in that this interface means that data structures in memory are built using ordinary pointers rather than with a level of indirection via object headers.

As Figure 3 illustrates, the compiler's role in translating code inside an atomic block into STM operations can be quite straightforward. The figure does not show two more subtle compilation techniques: First, if the atomic block contains method invocations, the compiler must translate the code in these methods to use STM operations on its memory accesses. This can lead to code duplication if the same methods are used both inside and outside transactions. Compiler optimizations to improve the resulting code remain the subject of research.[14,15] For instance, if the same data is read more than once, only the first OpenForReading operation is needed on the data. Second, the atomic block itself must be translated into

calls to library functions to start and commit transactions and to reexecute the transaction if the commit fails.

There are several ways to implement these STM APIs, each with different performance trade-offs. Broadly, these choices come down to two ways of managing tentative updates. In designs that use buffered updates, a transaction keeps a private shadow copy of all the memory words it updates, much as an HTM keeps private copies of them in the local data cache. Calls to STMRead must consult the shadow copies so that they will see earlier writes by the same transaction. Hashing can accelerate this look-up (mapping an address to a slot in the current transaction's shadow table to avoid searching it). Also, Bloom filters, which quickly determine that a given item is not in a list, can detect that a given location is guaranteed not to be in the transaction's shadow table.

An alternative design uses in-place updates. STMWrite directly updates the heap so that calls to STMRead will see earlier updates without needing to search a table. In this case, STMWrite must maintain an undo log of all values that it overwrites, so that the transaction can roll back its changes if it aborts. A disadvantage of in-place updates is that they can introduce contention between transactions—only one transaction can be granted write access to the same TMW at the same time.

Detecting and resolving conflicts in STM. Classical HTM systems detect conflicts between transactions through extensions of the MESI cache protocol. The eviction of a line holding transactional data causes the transaction to abort. Aborting or committing a transaction is straightforward because all of the state involved is held in the local cache. Detecting and resolving conflicts can become complicated in STM implementations in which only individual memory operations are atomic. Nonetheless, researchers have explored a wide variety of approaches: pessimistic schemes based on automatically locking data that transactions are accessing, optimistic nonblocking schemes, and numerous hybrids. Again, Larus's and Rajwar's book provides a good contemporary survey,[1] and so, in this overview, we focus on the broad design space.

One approach, used in the Argus programming language, employs strict two-phase locking of objects.[16] A compiler can readily automate this approach, which acquires locks as a transaction executes and holds them until it commits. However, this approach scales poorly on current multiprocessor hardware because it introduces contention in the memory hierarchy. Acquiring a lock in shared-read mode usually means fetching the cache line holding the lock in exclusive mode.

An alternative is to use a nonblocking, atomic multiword update to commit a transaction. OSTM does this by performing an atomic multiword update across the header contents of the objects in the read and write sets. It checks that there has been no update to objects in the read set and updates the object headers in the transaction's write set to publish the transaction's shadow copies. This design avoids the poor scalability of read locks because the transaction's read set is validated purely by memory read operations.

Earlier work often aimed at nonblocking behavior as an explicit goal. Nonblocking algorithms provide a robust guarantee that progress cannot be obstructed by threads that are not actively executing. This avoids problems of priority inversion or of threads being descheduled while holding locks.

Work by Herlihy, Luchangco, and Moir on obstruction-free synchronization gives a good introduction to the design of practical nonblocking algorithms and the underlying definitions.[17] However, such algorithms are notoriously complicated, in terms of both software engineering and the number of atomic compare-and-swap operations used at runtime. Consequently, researchers have explored support in the operating system or a language runtime system to provide robustness. In that setting, a hybrid approach combines optimistic and pessimistic schemes by using versioned mutual-exclusion locks, which support normal mutual-exclusion semantics and provide access to a version number counting the number of times the lock has been acquired and released.[14] The design uses

mutual exclusion for pessimistic concurrency control to grant write access to the data the lock protects. The version number allows optimistic concurrency control for read access. That is, a transaction records the version number before it first reads from an object and then, at commit time, checks that the version number is unchanged, meaning that no one has updated the object concurrently with the transaction.

This general approach is flexible; it can serve as the basis of several STM designs. Locks can support buffered updates in two ways. A transaction can acquire locks eagerly as it runs (to prevent conflicts, even though the transaction still makes updates to a private log). Or it can acquire locks only when it tries to commit (preventing transactions that abort from causing conflicts).

Locks can also support data management. In Bartok STM a transaction acquires locks eagerly on objects that it wishes to update, makes the updates themselves directly to the objects, and maintains an undo log to allow the updates to be reverted if the transaction aborts. The locks protect the data from conflicting writes from concurrent transactions, and making updates in place means that transactional reads will see earlier values written by the same transaction. This approach aims to make read operations as fast as possible, on three assumptions:

- Reads will outnumber writes in transactional workloads (as they do ordinarily).
- Reading data's version number during a transaction and checking it at commit time will be faster than acquiring and releasing a lock on the data.
- Transaction conflicts will usually be rare, so it is better to accelerate transactions running without contention and committing, at the expense of extra work in cases where conflicts do occur. In other words, it is better to "apologize" occasionally than to "ask permission" frequently.[18]

These factors, of course, might change as our experience of transactional workloads grows and instruction set performance characteristics change.

Case studies

In this section, we discuss selected HTM and STM implementations in detail. Each provides insights into different TM design challenges.

TCC

The Transactional Memory Coherence and Consistency model (TCC) is a form of HTM in which the atomic transaction is the basic unit of parallel work, communication, and memory coherence.[19] Each transaction maintains its write set until it attempts to commit. Upon commit, the transaction asks for permission to write its data. When the bus grants this permission, the transaction broadcasts its write set to other processors, which can snoop the store addresses to detect dependence violations and roll back in case of conflict. Only one processor can write its write set at one time.

Instead of imposing some order between individual memory references, TCC imposes a serializable sequential order between transaction commits. Therefore, all memory references from a transaction that commits earlier appear to come before all memory references from a transaction that commits later. This is true even if the references are actually executed in an interleaved fashion. When the transaction commits, the processor notifies all processors of the status changes. During this process, the other processors perform invalidations or updates, depending on whether only the address or both the address and data were sent. A processor is forced to roll back if it reads data updated by another processor that has committed its transaction. So the snooping mechanism must check whether any address sent by the committing transaction is in the associated processor's read set. In that case, that processor must roll back its transaction.

Programmers using the TCC model must first divide the program into transactions that run concurrently on different processors. They must mark the parallel sections but need not guarantee the independence of these regions; the hardware will take care of that. Second, they can optionally specify the

Table 1. Comparison of TCC and LogTM.

Feature	TCC	LogTM
Commit	Slower	Faster
Abort	Faster	Slower
Coherence mechanism	Bus-based	Directory-based
Conflict detection	Lazy	Eager
Conflict resolution	Abort self	Oldest time stamp wins
Write visibility	At commit	Immediate
Always in transaction	Yes	No
Nesting	Yes	Yes

order between transactions to enforce a program order. Many parallel applications have points at which certain transactions must complete before others. To deal with these barriers, TCC adds hardware-managed phase numbers to each transaction. Only transactions with the oldest phase number are allowed to commit at any time. A third, performance-tuning step provides information about where violations occur, which programmers can use to improve performance. Transactions should be chosen to maximize parallelism and minimize the number of conflicts. Large transactions are preferable for amortizing startup and commit, but small transactions are better when violations are frequent or available space for write and read sets overflows.

LogTM

LogTM is an HTM design based on two observations: First, commits are typically much more frequent than aborts. Second, most existing HTM systems use lazy version management by using old values in place and new values in a slower structure. This makes commits slow but aborts fast.[20]

Because commit is the common case, LogTM proposes that new values be used in place and old values be placed in a slower log structure. When a transaction commits (the common case), new values become nonspeculative, a much faster operation with LogTM. In the case of abort, old values should be restored to their in-place position, requiring a traversal of the log table to undo the transaction's effects, a much slower operation. However, overall performance increases because aborts are relatively rare.

LogTM uses eager conflict detection; instead of waiting until the end of the transaction, the system checks for and detects conflicts on every read and write. To detect conflicts on the fly, LogTM uses directory-based cache coherence. The processor doing a read or write operation issues a coherence request to the directory, which forwards the request to other processors. A responding processor detects the conflict and tells the requesting processor about it. Although this scheme might affect the performance of each read or write, it detects conflicts earlier. Table 1 compares the features of TCC and LogTM.

Implicit Transactions using kilo-instruction processors

The Implicit Transactions approach translates TM concepts directly into hardware.[21] This provides benefits in multiprocesssor systems, even those running ordinary nontransactional applications, and establishes the basic support that transactional software needs.

The proposed multiprocessor system uses implicit transactions as the basic unit of parallel work. Transactions are implicit because they are transparent to the programmer, and, unlike most previous efforts, require no change of the programming model.

Implicit transactions are implemented with low complexity by appropriately leveraging the key mechanisms in a kilo-instruction processor, mainly its multicheckpoint mechanism.[8] The instructions between two consecutive checkpoints are considered part of one implicit transaction,

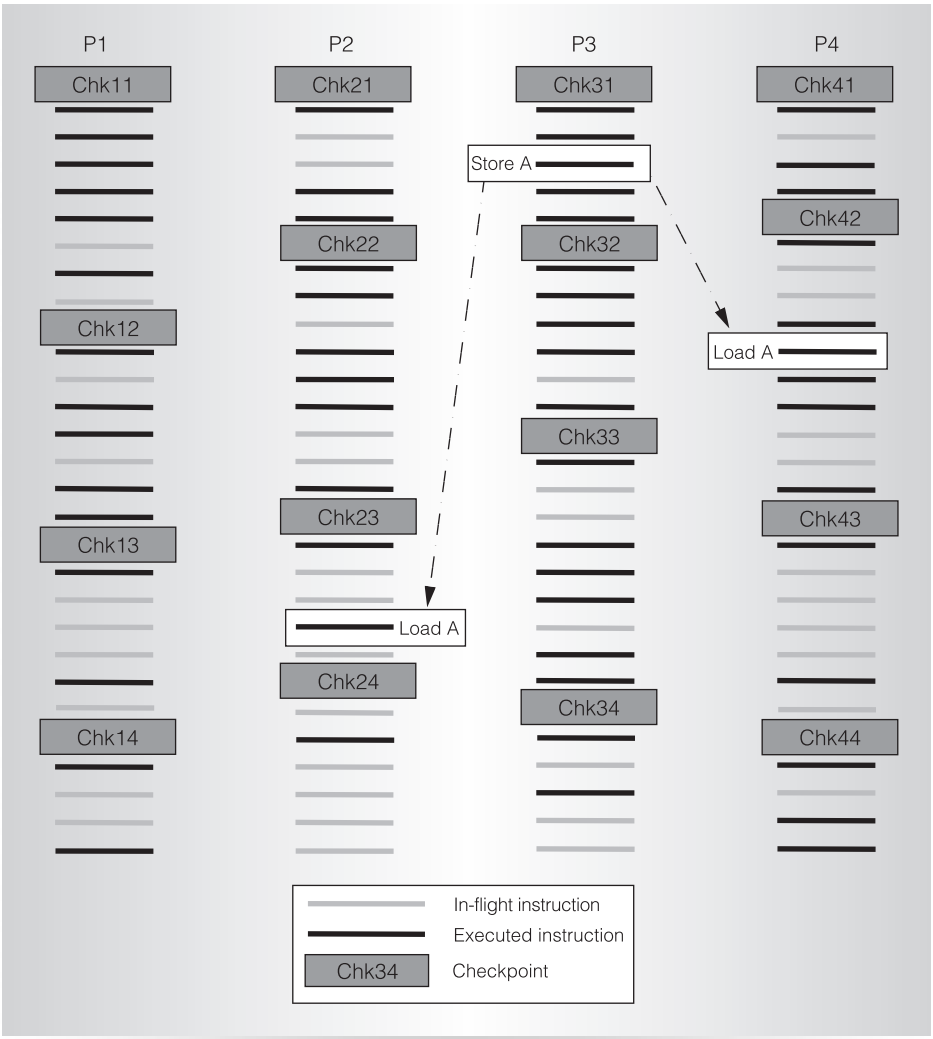

Figure 4. The Implicit Transactions mechanism for checkpointing and conflict resolution.

which appears to the rest of the system as a single memory transaction. In particular, the system manages memory updates (store instructions) associated with a transaction as a group and performs them globally and atomically when the corresponding checkpoint commits.

Figure 4 shows an example of the execution flow for four processors and their respective checkpoints. A processor's oldest checkpoint can commit when all of its corresponding instructions—the ones that come after the checkpoint and before the next checkpoint—are finished. For example, processor P3 can commit checkpoint Chk31 when all instructions up to Chk32 have completed. Meanwhile, a general speculation substrate buffers the speculative read set and searches for conflicts with any committed update, causing a processor to roll back in case of violation. This mechanism guarantees correct memory consistency. For example, in Figure 4, the broadcast of a store (st) to memory location A conflicts with two other processors that have already speculatively loaded from location A, and the loads have not yet committed. In this example, P2 rolls back to Chk23, causing instructions from Chk24 to Chk23 to be discarded. Also, P4 rolls back to Chk42, forcing its newest instructions to be discarded.

Obviously, the size of transactions depends on the point at which checkpoints are taken, and the system carries out the decision dynamically to avoid frequent rollback scenarios. If there is no conflict for a substantial time period, the size of the implicit transactions increases. Conversely, if there are frequent rollbacks, transaction size is halved.

Implicit transactions overcome a key limitation of HTM systems: large transactions. In case of overflow, a large transaction splits into multiple smaller implicit transactions without affecting correctness.

Haskell STM: Composable memory transactions

To illustrate the design and implementation of STM systems in more detail, we examine the system used in the Glasgow Haskell compiler (GHC) runtime system.[22] Because our focus is on lower levels of TM systems rather than higher-level language design issues, we use a simple, imperative style of pseudocode to illustrate the Haskell STM interface, rather than full Haskell programs.

The Haskell STM provides transactional access to distinguish transactional variables called TVars. It uses a buffered update design. While a transaction is running, it builds a log recording all the TVars it has accessed, the values those TVars held at the point they were accessed, and the new value (if any) that the transaction wishes to write to the TVar. The Haskell system ensures that only transactional code can access TVars, and that the only kinds of side effect possible inside a transaction are reads and writes to TVars. Each TVar contains three fields: the TVar's actual contents (a pointer to an ordinary piece of Haskell data), a TMW holding a versioned mutual-exclusion lock used by the Haskell STM implementation, and a wait queue used for condition synchronization within transactions.

As in other buffered-update systems, aborting a Haskell STM transaction simply means discarding the log holding its buffered updates. However, there is one subtlety. The semantics provide that a transaction rolls back if an exception propagates outside it, for instance in pseudocode:

```
try {
  atomic {
   throw new Exception(); //T1
  }
} catch (Exception e) {
  // C1;
}
```

This requires care: The language's designer must reconcile the idea that the transaction rolls back with the expectation that the exception allocated at T1 must survive to be caught at C1. GHC takes the approach that only updates to TVars roll back; object allocations are retained, and so the exception persists until at least C1.

At commit time, Haskell STM uses the transaction's log of buffered updates along with the TMWs to attempt to atomically perform the transaction's TVar accesses at the point at which it commits. Figure 5 shows how the commit operation proceeds, starting from 5a, which shows the buffered updates and the state of the TVars. First, Haskell STM logs the version number from each TVar that the transaction has read from but not updated and compares the value in the transaction's log with the current value in the TVar (Figure 5b). If the values don't match, the transaction is aborted and reexecuted. Second, the STM acquires the mutual-exclusion locks on the TVars that the transaction has updated (Figure 5c). If that succeeds, the point at which the STM finishes acquiring locks becomes the transaction's linearization point, where the transaction appears to execute atomically. If the thread fails to acquire all the locks, again it aborts and reexecutes the transaction.

After the STM has acquired all the locks that the transaction needs, it continues by checking the version numbers against those it has just recorded. If they are all unchanged, that guarantees that no other transaction has committed an update to any of these TVars since the versions were recorded. If any version number is out of date, the transaction aborts. Finally, if the versions match, the STM writes the transaction's buffered updates to the TVars (Figure 5d) and releases the mutual-exclusion locks, atomically incre-

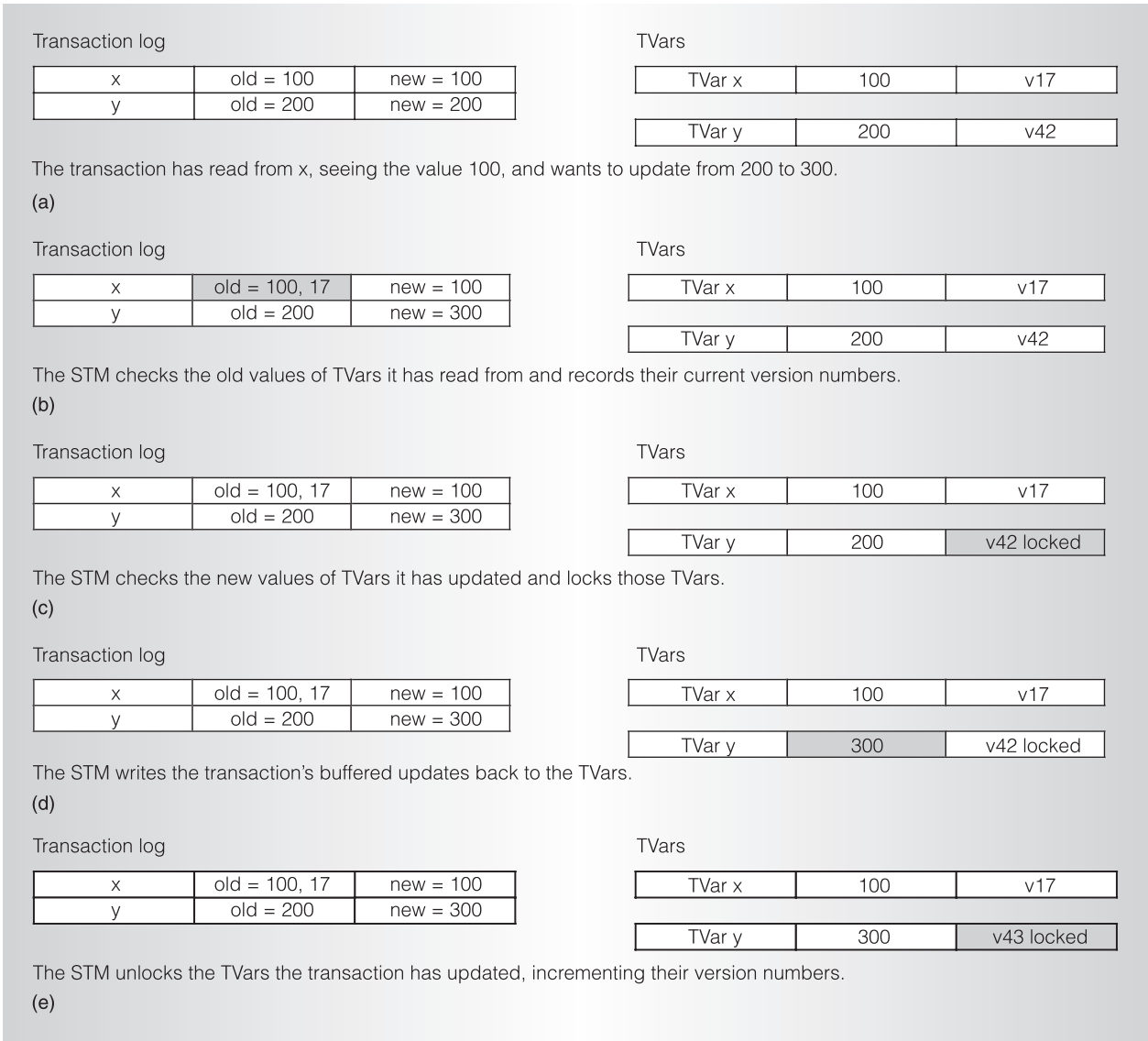

Figure 5. Committing a transaction in Haskell STM. The shaded boxes show the locations accessed at each step.

menting the TVars' version numbers (Figure 5e).

A further notable aspect of the Haskell STM design is the integration of composable blocking with atomic updates. The design uses this condition synchronization in place of condition variables: A language construct retry means "abort the current transaction, and reexecute it." For example, the following code would wait until TVar x holds a nonzero value:

```
atomic {
  if (ReadTVar(x) == 0) retry;
}
```

This is called composable blocking because the call to retry can occur in a function called anywhere within the atomic block. It is not necessary to hoist checks up to the start of the block (although, from a performance point of view, it is preferable to decide whether to retry sooner rather than later).

Although the Haskell STM design defines the retry semantics in terms of reexecution, this is not a good implementation strategy. The condition that caused the code to call retry is likely to still hold, so the thread will keep spinning until an update is made to the heap. This observation can be

exploited to make a more efficient implementation. When a thread calls retry, it confirms that it has seen a consistent view of the heap and atomically adds itself to the wait queues of all the TVars from which its transaction has read, before blocking in the GHC user-level thread scheduler. Then, when another thread commits a transaction that updates one of these TVars, the committing thread iterates through the wait queue, waking any threads blocked there.

The Haskell STM design provides the retry operation alongside a second operation, orElse, which allows a program to attempt a series of alternative transactional operations. This operation also has the goal of composability. For instance, if a programmer builds shared queues that call retry to wait for items to be deposited in the queue, the caller can use orElse to compose access to a series of queues, taking items from whichever queue first supplies one.

Challenges

Although TM provides a potential solution to the programmer productivity problem in a many-core environment, it poses several challenges to designers. Some of these challenges are specific to HTM or STM, and others apply to both types.

General challenges

TM research has investigated many challenges. Among them are open and closed nesting as a way TM can help expose more parallelism; I/O, which has been an issue starting with the earliest TM research; and mixing TM with programming models such as the Open Multiprocessing (OpenMP) interface or the Message-Passing Interface (MPI).

Open and closed nesting. An important TM property is the possibility of calling a function within a transaction that itself contains a transaction. This can help programmers write efficient code and use libraries. A nested transaction is a transaction that begins and ends within the scope of a surrounding transaction. There are two types of nested transactions: closed and open. In a closed-nested TM system, either all or none of the transactions in a nested region commit. In contrast, in an open-nested TM, when an inner transaction commits, its effects become visible for all threads in the system.

The easy way to deal with closed-nested transactions is the flattening model. This model includes all nested transactions in the outermost transaction; that is, all the involved transactions share one read set and one write set. Even this easy solution incurs some complexity. HTM systems must use counters to implement flattening. Each STR instruction increments a counter, and each ETR instruction decrements it, committing the transaction to memory only if the counter is zero. If a conflict occurs, the transaction must roll back to the outermost transaction. This mechanism is often inefficient because each time the transaction rolls back, it loses work that could be preserved. A more efficient mechanism is to allow each nested transaction to have its own read and write sets, so that when the transaction commits, the read and write sets merge with the read and write sets of the next level out. In case of abort, the innermost conflicting transaction rolls back to its STR but not to the top level.

Open-nested transactions can unleash more concurrency than closed-nested transactions. When an open-nested transaction commits, its write set becomes visible to all other transactions, so other transactions can see modifications sooner and work with the modified data. This is different from closed-nested transactions, which make all modifications visible only when the outermost transaction commits. However, open-nested transactions increase the programmer's burden. Compensating actions are needed when the outermost transaction commits and when one of the surrounding transactions aborts. Handling this compensating code can be complex, and the programmer must have an expert grasp of the code's semantics.[23]

The ongoing challenge is to support a rich nesting model with minimum hardware-software complexity. Even the simplest closed-nesting flattening model incurs some hardware complexity while limiting concurrency and performance. Proposed open-nested transaction models expose more concurrency but increase complexity for programmers, who must explicitly write commit and abort handler

codes to support these models. Moreover, these models are especially challenging to use with libraries because the library API might require changes to support the handler code. Efficient solutions to these problems are open research questions.

To highlight some of the issues, Figure 6 shows a simple example of code that performs preprocessing (in function someprocess), and then tries to obtain work from a work queue (getwork), and process the resulting work item (process). Suppose that there are two threads, P1 and P2, and that transactions are validated at the end of the transactions.

In the original code without nesting, if there is a conflict between two threads, at least one should abort. As Figure 7a shows, P1 finishes and detects a conflict with P2 because both have modified workqueue.head inside function getwork; in this case, P2 rolls back.

In closed-nested transactions, if the model allows partial rollbacks, at least the work done in someprocess can be saved (assuming there are no conflicts with the data accessed in that part of the code), as Figure 7b shows. In this case, when P1 finishes, it detects the conflict with P2, which simply rolls back the nested transaction. With the flattening model, the transaction is exactly the same as with no nesting.

To express open-nested transactions, the following construct is added to the language:

```
atomic_open {statements
        commit {statements}
        abort{statements} }
```

The commit executes when the outermost atomic block commits—in our example, free(work) in P1 (Figures 7c and 7d) The abort executes when a surrounding transaction aborts—in our example, the call to insertwork(workqueue,work) in Figure 7d.

I/O. The relationship between I/O operations and transactions is a significant research challenge. For example, suppose that inside a transaction, a system call attempts to output a character to the terminal. One solution is to execute the system call immediately; however, that would be very problematic if this transaction aborted later. Trying to undo the I/O by deleting the character upon an abort would obviously lead to a wobbly system. In some cases, even executing the I/O operation might be difficult if the data is buffered in HTM.

Another option is to defer the I/O operation, and wait until commit to output the character to the terminal. However, this can lead to problems if the user is expecting the I/O to take some action. Particular problems emerge in deferring I/O in real-

```
atomic  {
    someprocess(shared_data);
    work=getwork(workqueue);
    process(work,shared_data);
}

node *  getwork(workqueue_t &workqueue)
{
    node *work;
    work=workqueue.head;

    if (work!=NULL)
         workqueue.head=work->next;
    return work;
}
```

(a)

```
atomic  {
    someprocess(shared_data);
    atomic {
      work=getwork(workqueue);
    }
    process(work,shared_data);
}
```

(b)

```
atomic  {
    someprocess(shared_data);
    atomic_open {
      work=getwork(workqueue);
      commit {
         free(work);
      }
      abort {
         insertwork(workqueue,work);
      }
    }
    process(work,shared_data);
}
```

(c)

Figure 6. Example code for open- and closed-nested transactions: original code (a), code with closed-nested transactions (b), and code with open-nested transactions (c).

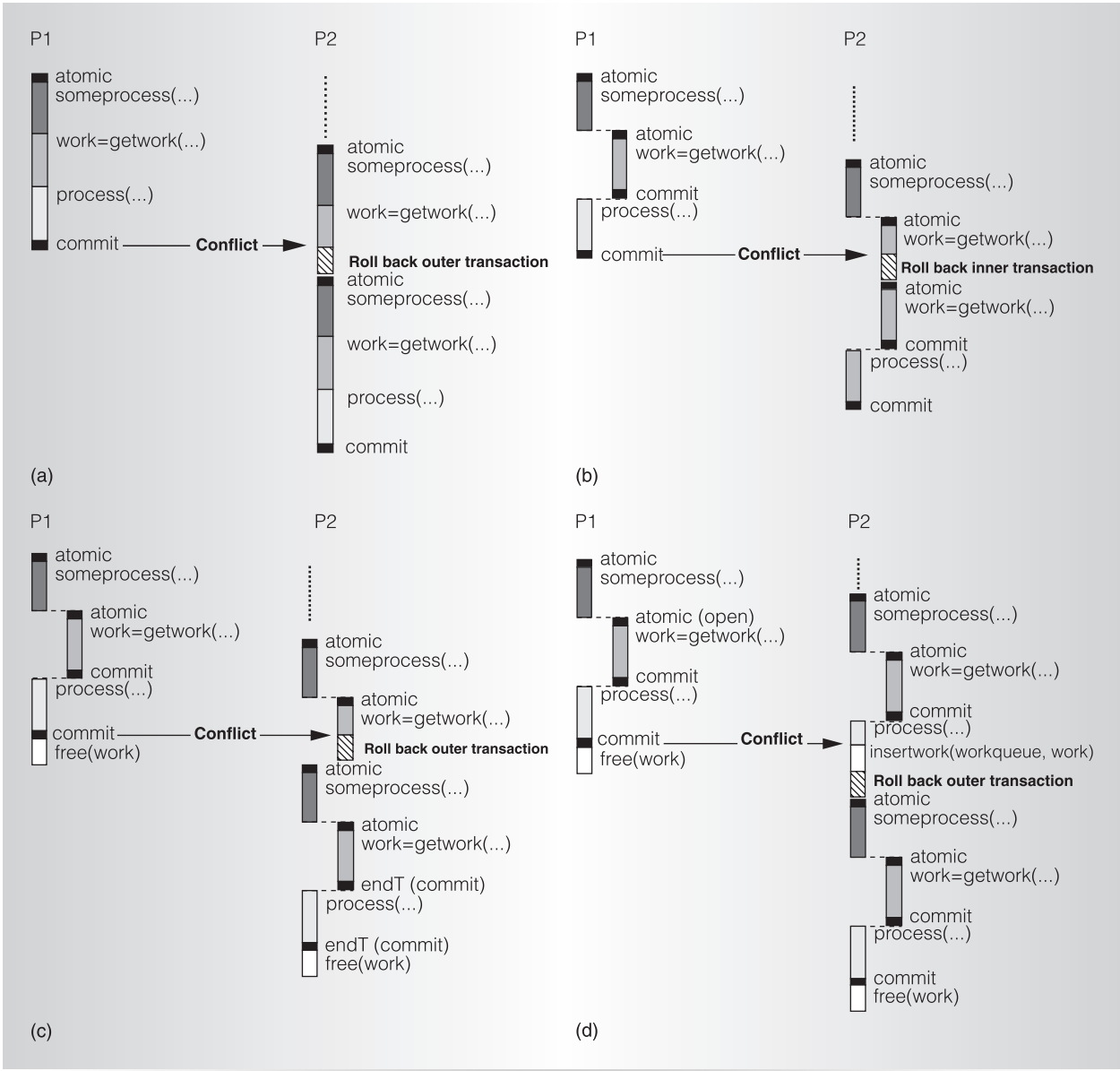

Figure 7. Example comparing open- and closed-nested transactions: no nested transactions (a); closed-nested transaction (b); open-nested transaction—conflict before the open-nested transaction commits (c); open-nested transaction—conflict after the open-nested transaction commits (d).

time systems, which must process sensor inputs as soon as they are received to avoid deadline misses.

Still another approach is to forbid I/O operations from within transactions—or at least to restrict them to particular forms of I/O that can themselves become transactional, such as access to a transactional database or file system.

A final approach to solving the I/O problem is to categorize inputs and outputs according to their abortive properties. By definition, an I/O call is undoable if its effects can be rolled back, which implies that its effects are self-contained to the I/O operation only. In this approach, the challenge is to allow maximum programmer expressiveness while avoiding overly com-

plex implementations. Programmers must be aware of types of I/O operations that don't make sense to transparently perform as part of a single atomic transaction—for instance, prompting the user for input and then receiving and acting on the input.

Harris discusses some of the issues described here in the context of atomic blocks.[24]

Programming models and TM. Some recently proposed programming models, such as Sun's Fortress,[25] IBM's X10,[26] and Cray's Chapel,[27] include an atomic statement to define conditional or conditional atomic blocks of statements. In some cases, atomic can also be an attribute for variables; thus, any update to them in the code is treated as if the update is in a short atomic section.

Other established programming models, such as OpenMP and MPI, widely used for parallelizing scientific and engineering applications, were not designed with TM in mind. Exploring how TM can make parallel programming with these models affordable is another challenge.

OpenMP has become the industrial standard for writing parallel programs on shared-memory architectures for C, C++, and Fortran. With OpenMP the programmer must insert pragmas to express, in a portable way, the opportunities to exploit parallelism, distribute work among threads, and synchronize their execution. However, OpenMP was initially designed for scientific applications in which the main source of parallelism is in loops. The new specification of OpenMP (3.0) includes task parallelism. It allows the programmer to define tasks as blocks of work that can enter a queue of tasks and execute in parallel with other tasks.

A major complexity in writing OpenMP applications is the use of critical regions (locks), atomic regions, and barriers to synchronize the execution of parallel activities in threads. The simplest way to mix Open MP and TM is to replace critical and atomic regions with transactional code regions, making parallel applications easier to program, understand, and maintain. However, TM possibly can provide further advantages to the OpenMP programming model. For example, the violation of barriers, both explicit and implicit, in certain OpenMP work distributors, is a source of potential speculative parallelism. The inclusion of tasks in OpenMP 3.0 also adds more complexity in specifying the synchronization of parallel activities performed by the tasks, increasing TM's potential for applications based on tasking.

Supporting OpenMP parallelism within transactions introduces challenges that researchers have not yet considered in detail. For example, extant TM implementations don't permit multiple threads to run in parallel in the same transactional state. Transparently to the programmer, TM can also implement the runtime library supporting the code generated by the OpenMP compiler, which relies on extensive use of shared memory to implement the programming model.

Although mixing TM with nonshared-memory, message-passing programming models such as MPI seems unpromising at first glance, TM's failure atomicity might offer opportunities. Mixing the two models might offer advantages in game application domains in which CPU and graphics processing unit (GPU) chips communicate through message passing and in which the communication channel is prone to faults or errors. Although fault-tolerant versions of MPI, such as MPI-FT, exist, they require a new API. For MPI with TM however, programmers could achieve fault tolerance by wrapping each standard MPI directive inside a transaction with handler code to handle aborts (in the case of faults) or commits. One challenge would be to properly implement the mixed model to prevent cascading aborts.

Hardware transactional memory challenges

Limited on-chip resources present many challenges for HTM design. The main problem is that HTM-imposed rules (such as limited transaction size) decrease programming expressiveness and ease.

Bounded and unbounded transactions. One concern for HTM system programmers is the lack of sufficient buffer space. Transactions create additional copies of shared data

items. One version of the data item remains modified, and the other version keeps the original value in case a rollback occurs. This means that transactional state requires at least twice the space that conventional state does. Usually, transactions are small. For example, Hammond et al. report that 90 percent of transactions fit in less than 8 Kbytes for most applications the authors ran, and the rest fit in 64 Kbytes.[19] However, some applications have a very large transaction footprint. For example, javac, the Java compiler, needs more than 1.2 million lines.[28] As a result of large transaction size, as well as limited associativity, the transactional state might not fit into the cache.

There are two ways to deal with this problem. A simple approach is to roll back the HTM transactions and reexecute them serially to guarantee that they don't conflict. However, this approach can violate TM semantics. Nevertheless, this approach is attractive in systems using speculative lock elision, in which TM is effectively an optimistic implementation of traditional mutual-exclusion locks.

The other approach is to combine a bounded-HTM system with an unbounded-STM system. Transactions are first attempted with the fast HTM, and then, if they overflow the hardware limits, they are rolled back and reexecuted with STM. This approach can add overhead to the HTM transactions because they must watch for conflicts from STM transactions. Hybrid transactional memory (HyTM) systems use this method to provide unbounded transaction size support without the increased hardware design complexity that a full unbounded-HTM system entails.[2,3]

Providing predictable performance for applications using HyTM is very difficult because the switch from HTM to STM leads to a performance cliff. A further challenge is avoiding complex implementations while providing support for arbitrarily sized transactions. To address the last issue, Ananian et al. propose two different systems: the complex Unbounded Transactional Memory (UTM) and the simple, limited Large Transaction Memory (LTM).[28] UTM allows a very large memory footprint, close to virtual-memory size, and lets a transaction migrate, run indefinitely, and survive time-slice interrupts. Like LogTM, UTM is optimistic in the sense that it assumes conflicts rarely happen. UTM stores the original data in a transaction log and the modified data in place in memory. When a transaction commits, it must discard the transaction log, but when a transaction aborts it must write the transaction log to memory.

LTM is a similar system but has some constraints. It cannot support transactions of virtual-memory size, so it limits transaction size to physical-memory size. In LTM, transactions cannot survive time-slice interrupts and cannot migrate. Moreover, it uses cache memory to handle the different data versions and detects conflicts using the cache coherency mechanism. If the cache overflows, the system uses a hash table in main memory.

ISA support. Because HTM systems have not been deployed yet, many ISA extension proposals exist. They range from no ISA support whatsoever in Implicit Transactions[21] to recent detailed support mechanism,[29] demonstrating the evolution of the field. In Herlihy, Eliot, and Moss's proposal, ISA support is at a minimum; even the start transaction instruction is unnecessary in their system.[9] Still, their system could support or emulate many of the more complex models; for example, the flattening used in their system could support a version of closed nesting. In contrast, McDonald et al. provide explicit ISA support for two-phase transaction commit, closed- and open-nested transactions, and support handlers for transaction commit, conflict, and abort.[29] The challenge is to provide the right level of ISA support for TM (the issue is similar to the RISC-CISC debate).

Software transactional memory challenges

The main advantage of STM over HTM is the flexibility to implement a wide range of semantics to handle the issues discussed earlier: managing transactional state and detecting and resolving conflicts. This flexibility comes at a cost. STM's most obvious weakness is the runtime overhead introduced by transaction management.

However, the following are two less obvious challenges introduced by STM.

Atomicity and code interaction. In classic HTM designs, normal memory accesses interact cleanly with transactions. Nontransactional reads see only committed state, and conflicts are detected between nontransactional updates and concurrent transactions accessing the same data. However, this is not usually true of STM systems because concurrency control operations must be explicitly introduced.

One aspect of this problem occurs in systems such as OSTM, where transactional data structures use different runtime formats from ordinary data. Nontransactional code must be aware of this—for instance, when a thread creates a data structure that later becomes shared between threads through transactions.

Even if ordinary data structures represent transactional data in memory, casual sharing is not usually possible, and various problems can occur. For instance, if an STM makes updates in place, a direct read from transactional memory might see uncommitted data even if the transaction subsequently aborts. In contrast, if an STM buffers its updates, a direct read might not see an update by a transaction that has logically committed (in terms of the serialized order of transactions) but has not yet written its buffered updates back to the heap. Direct stores are problematic as well. Whether the STM uses in-place updates or buffered updates, a direct store won't cause a concurrent transaction to detect a conflict.

Systems that cannot use direct access and transactional access together are said to exhibit weak atomicity. This means that the programmer must select whether a data structure will be managed directly or transactionally—either by hand or with the aid of the language's type system. Conversely, systems in which direct accesses look like single-location transactions are said to exhibit strong atomicity.

Providing strong atomicity in an STM system appears challenging from a performance viewpoint. Strong atomicity requires that direct memory accesses check the metadata structures that the STM uses to coordinate transactions. Although a compiler or managed runtime system can automate the addition of these checks the performance cost would be high because of the vast increase in the number of memory accesses that the checks need.

A possible middle ground between strong and weak atomicity is to require some discipline in the programmer's use of atomic blocks. One programming discipline is single-lock equivalence, in which atomic blocks are considered—merely conceptually—as acquiring and releasing a single processwide transaction lock. If the resulting lock-based program is correctly synchronized, its original version using atomic blocks will run with strong atomicity. However, if the resulting lock-based program is not correctly synchronized, the original version can exhibit the kind of data races that weak atomicity permits. For example, the following pair of operations is not correctly synchronized because there is no concurrency control between the transaction executed by Thread 1 and the direct access in Thread 2:

```
// Thread 1
atomic {
  hist[index]++;
}

// Thread 2
hist[index] = 42;
```

We will need more experience of transactional workloads to decide what kind of atomicity model to provide. Although strong atomicity is tempting, using it in place of single-lock equivalence doesn't make many nontrivial programs work correctly. Instead, we should focus on reliably preventing or detecting the kinds of synchronization errors caused by weak atomicity. On the other hand, in many settings we still need some guarantees, even when a program mixes direct accesses and transactional accesses. Even if such programs are incorrectly synchronized, they should not, for instance, lead to type-safety violations in a managed runtime system.

Inconsistent reads and zombie transactions. The second problem particular to STM is controlling so-called zombie transactions.

These are transactions condemned to abort because another transaction has made a conflicting update, but the STM runtime system has not yet detected the conflict. The problem with zombies is that they might have read an inconsistent set of data. For instance, in OSTM, transaction T1 can read from object O1, transaction T2 can commit updates to O1 and O2, and then T1 can continue to read O2, leaving T1 a zombie.

How do we guarantee that zombies are discovered, and how do we prevent them from doing harm in the meantime? If conflicts are detected only at commit time, it is possible that inconsistent data read form O1 and O2 might cause zombie T1 to enter an infinite loop and never try to commit. We must make sure that the zombie doesn't perform any irrevocable operations before it aborts. For instance, T1 might raise a null pointer exception or attempt an out-of-bounds array access, both of which must be detected and rolled back.

One way to solve these problems is to use a managed programming environment, such as the Java Virtual Machine, Microsoft's .NET Common Language Runtime, or the GHC runtime system. The managed runtime system can periodically validate transactions to find any zombies and ensure that all memory accesses performed by a transaction go through the STM machinery so that the effects of zombie transactions can be rolled back.

The situation is more difficult without a managed runtime. For low-level languages, one approach is to push the problem onto the STM system user. The user must be aware of the problem and make sure that zombies are detected and remain benign—for instance, by checking array bounds accesses made by transactions. However, this eliminates many of the ease-of-programming advantages of using transactions.

An alternative with a more palatable programming model is to prevent zombies from arising in the first place. TM designs can do this either by using pessimistic concurrency control for all data accesses (that is, reads as well as writes) or, if using optimistic concurrency control, by ensuring that the read set remains valid. The DSTM system uses the latter approach by maintaining visible reader lists that record which transactions are reading from which objects. A transaction opening an object for writing can thereby detect which readers it might conflict with.

The Transactional Locking II design avoids explicit reader lists by assigning global version numbers to transactions and associating each memory location with a versioned write lock that records the version number of the last transaction that updates the location.[30] A new read is consistent with a transaction's earlier reads as long as the location's version number predates the global version at the time that the transaction began.

Transactional memory provides a graceful and natural mechanism for writing parallel programs. Recently, a flurry of research activity has endeavored to define and design hardware and software TM. We are entering an era in which the design of multicore chips will almost entirely be driven by software usage models such as TM. The greatest challenge is to make the programmer community's adoption of TM as smooth as possible through hardware and software design. Effective synchronization mechanisms are crucial to fulfilling the promise of improved application performance on future multicore chip multiprocessors. MICRO

Acknowledgments

This work was supported by a cooperation agreement between the Barcelona Supercomputing Center's National Supercomputer Facility and Microsoft Research. It is also supported by Spain's Ministry of Science and Technology, the European Union under contract TIN2004-07739-C02-01, and by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC).

References

1. J. Larus and R. Rajwar, *Transactional Memory*, Morgan Claypool, 2006.
2. P. Damron et al., ''Hybrid Transactional Memory,'' *Proc. 12th Int'l Conf. Architec-*

tural Support for Programming Languages and Operating Systems (ASPLOS 06), ACM Press, 2006, pp. 336-346.
3. S. Kumar et al., ''Hybrid Transactional Memory,'' *Proc. 11th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming* (PPoPP 06), ACM Press, 2006, pp. 209-220.
4. B. Saha, A. Adl-Tabatabai, and Q. Jacobson, ''Architectural Support for Software Transactional Memory,'' *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 06), IEEE CS Press, 2006, pp. 185-196.
5. A. Shriraman et al., ''Hardware Acceleration of Software Transactional Memory,'' http://www.cs.rochester.edu/u/scott/papers/2006_TRANSACT_RTM.pdf.
6. R. Rajwar and J.R. Goodman, ''Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution,'' *Proc. 34th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 01), IEEE CS Press, 2001, pp. 294-305.
7. J. Martínez and J. Torrellas, ''Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications,'' *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, 2002, pp. 18-29.
8. A. Cristal et al., ''Kilo-Instruction Processors: Overcoming the Memory Wall,'' *IEEE Micro*, vol. 25, no. 3, May–June 2005, pp. 48-57.
9. M. Herlihy, J. Eliot, and B. Moss, ''Transactional Memory: Architectural Support for Lock-Free Data Structures,'' *Proc. 20th Ann. Int'l Symp. Computer Architecture* (ISCA 93), IEEE Press, 1993, pp. 289-300.
10. N. Shavit and D. Touitou, ''Software Transactional Memory,'' *Proc. 14th Ann. ACM Symp. Principles of Distributed Computing* (PODC 95), ACM Press, 1995, pp. 204-213.
11. M. Herlihy et al., ''Software Transactional Memory for Dynamic-Sized Data Structures,'' *Proc. 22nd Ann. Symp. Principles of Distributed Computing* (PODC 03), ACM Press, 2003, pp. 92-101.
12. K. Fraser, *Practical Lock-Freedom*, doctoral dissertation, UCAMCL-TR-579, Computer Laboratory, Cambridge Univ., 2004.
13. V.J. Marathe, W.N. Scherer III, and M.L. Scott, ''Adaptive Software Transactional Memory,'' *Proc. 19th Int'l Symp. Distributed Computing* (DISC 05), LNCS 3724, Springer, 2005, pp. 354-368.
14. T. Harris et al., ''Optimizing Memory Transactions,'' *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 06), ACM Press, 2006, pp. 14-25.
15. A. Adl-Tabatabai et al., ''Compiler and Runtime Support for Efficient Software Transactional Memory,'' *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 06), ACM Press, 2006, pp. 26-37.
16. B. Liskov, ''Distributed Programming in Argus,'' *Comm. ACM*, vol. 31, no. 3, Mar. 1988, pp. 300-312.
17. M. Herlihy, V. Luchangco, and M. Moir, ''Obstruction-Free Synchronization: Double-Ended Queues as an Example,'' *Proc. 23rd IEEE Int'l Conf. Distributed Computing Systems* (ICDCS 03), IEEE CS Press, 2003, pp. 522-529.
18. M. Herlihy, ''Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types,'' *ACM Trans. Database Systems*, vol. 15, no. 1, Mar. 1990, pp. 96-124.
19. L. Hammond et al., ''Transactional Memory Coherence and Consistency,'' *Proc. 31st Ann. Int'l Symp. Computer Architecture* (ISCA 04), IEEE CS Press, 2004, pp. 102-113.
20. K.E. Moore et al., ''LogTM Log-Based Transactional Memory,'' *Proc. 12th Int'l Symp. High-Performance Computer Architecture* (HPCA 06), IEEE Press, 2006, pp. 254-265.
21. E. Vallejo et al., ''Implementing Kilo-Instruction Multiprocessors,'' *Proc. Int'l Conf. Pervasive Services* (ICPS 05), IEEE Press, 2005, pp. 325-336.
22. S.L. Peyton Jones et al., ''The Glasgow Haskell Compiler: A Technical Overview,'' *Proc. UK Joint Framework for Information Technology (JFIT) Tech. Conf.*, JFIT 1992, pp. 249-257.
23. J.E.B. Moss, ''Open Nested Transactions: Semantics and Support,'' 2006, http://www.cs.utah.edu/wmpi/2006/final-version/wmpi-posters-1-Moss.pdf.
24. T. Harris, ''Exceptions and Side-Effects in Atomic Blocks,'' *Science of Computer Programming*, vol. 58, no. 3, Dec. 2005, pp. 325-343.

25. E. Allen et al., *The Fortress Language Specification*, Sun Microsystems, 2005.
26. P. Charles et al., ''X10: An Object-Oriented Approach to Non-Uniform Cluster Computing,'' *Proc. 20th Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA 05), ACM Press, 2005, pp. 519-538.
27. *Chapel Specification 0.4*, Cray Inc., 2005; http://chapel.cs.washington.edu/Specification.pdf.
28. C.S. Ananian et al., ''Unbounded Transactional Memory,'' *Proc. 11th Int'l Symp. High-Performance Computer Architecture* (HPCA 05), IEEE CS Press, 2005, pp. 316-327.
29. A. McDonald et al., ''Architectural Semantics for Practical Transactional Memory,'' *Proc. 33rd Int'l Symp. Computer Architecture* (ISCA 06), IEEE CS Press, 2006, pp. 53-65.
30. D. Dice, O. Shalev, and N. Shavit, ''Transactional Locking II,'' *Proc. 20th Int'l. Symp. Distributed Computing* (DISC 06), LNCS 4167, Springer, 2006, pp. 194-208.

Tim Harris is a researcher at Microsoft Research Cambridge where he works on the design and implementation of programming languages and runtime systems with a particular focus on the needs of multicore processors and systems problems. He received his BA and PhD from the University of Cambridge Computer Laboratory and is a professional member of the ACM.

Adrián Cristal is a senior researcher in the Barcelona Supercomputing Center. His interests include high-performance microarchitecture, multi- and many-core chip multiprocessors, transactional memory, and programming models. He received a PhD from the Computer Architecture Department at the Polytechnic University of Catalonia (UPC), Spain, and he has a BS and an MS in computer science from the University of Buenos Aires, Argentina.

Osman S. Unsal is a senior researcher at the Barcelona Supercomputing Center. His current research interests include computer architecture, reliability, and ensuring programmer productivity. He holds BS, MS, and PhD degrees in electrical and computer engineering from Istanbul Technical University, Brown University, and University of Massachusetts, Amherst, respectively.

Eduard Ayguade is a full professor in the Computer Architecture Department at Universitat Politècnica de Catalunya (UPC), Spain. He is also associate director for research on Computer Sciences at the Barcelona Supercomputing Center (BSC). His research interests cover the areas of multicore architectures and programming models and compilers for high-performance architectures. He has an Engineering degree in telecommunications and a PhD in computer science, both from UPC.

Fabrizio Gagliardi is Europe, Middle East, Africa, and Latin America director for scientific and technical computing in the Advanced Strategies and Policy Division at Microsoft. He joined Microsoft after a long career at CERN, the world leader laboratory for particle physics in Geneva, Switzerland. There, he served as director of the EU Grid project EGEE, director of the EU DataGrid project, head of mass storage services, leader of the EU project GPMIMD2, and cofounder and member of the International Advisory Committee of the Global Grid Forum. Gagliardi received a doctoral degree in computer science from the University of Pisa. He is a longtime member of the IEEE.

Burton Smith is a Technical Fellow at Microsoft. His main interest is general-purpose parallel computer architecture. Earlier in his career, he was a Fellow at the Supercomputing Research Center of the Institute for Defense Analyses in Maryland; Vice President, Research and Development at Denelcor; and chief architect of the HEP computer system. He received a BSEE from the University of New Mexico and an ScD from MIT. Smith is a fellow of both the ACM and the IEEE and winner of the 1991 IEEE-ACM Eckert-Mauchly award. He was elected to the National Academy of Engineering and received the Seymour Cray award, both in 2003.

Mateo Valero is a professor in the Computer Architecture Department at UPC and director of the Barcelona Supercomputer Center. His research interests include high-performance architectures. Valero has a PhD in telecommunications from UPC. He is an IEEE Fellow, an Intel Distinguished Research Fellow, and an ACM Fellow. He is a founding member of the Royal Spanish Academy of Engineering.

Direct questions and comments about this article to Osman S. Unsal, Barcelona Supercomputing Center, C/Jordi Girona, 29, Edificio Nexus II, 3ª planta, 08034 Barcelona, Spain; osman.unsal@bsc.es.