Taking the Heat off Transactions: Dynamic Selection of Pessimistic Concurrency Control

[1,2]Nehir Sönmez [3]Tim Harris [1]Adrián Cristal [1]Osman S. Ünsal [1,2]Mateo Valero
[1]BSC-Microsoft Research Centre
[2]Departament d'Arquitectura de Computadors - Universitat Politècnica de Catalunya, Spain
[3]Microsoft Research, Cambridge, UK
{nehir.sonmez, adrian.cristal, osman.unsal, mateo.valero}@bsc.es, tharris@microsoft.com

Abstract

In this paper we investigate feedback-directed dynamic selection between different implementations of `atomic` *blocks. We initially execute* `atomic` *blocks using STM with optimistic concurrency control. At runtime, we identify "hot" variables that cause large numbers of transactions to abort. For these variables we selectively switch to using pessimistic concurrency control, in the hope of deferring transactions until they will be able to run to completion. This trades off a reduction in single-threaded speed (since pessimistic concurrency control is not as streamlined as our optimistic implementation), against a reduced amount of wasted work in aborted transactions. We describe our implementation in the Haskell programming language, and examine its performance with a range of micro-benchmarks and larger programs. We show that our technique is effective at reducing the amount of wasted work, but that for current workloads there is often not enough wasted work for an overall improvement to be possible. As we demonstrate, our technique is not appropriate for some workloads: the extra work introduced by lock-induced deadlock is greater than the wasted work saved from aborted transactions. For other workloads, we show that using mutual exclusion locks for "hot" variables could be preferable to multi-reader locks because mutual exclusion avoids deadlocks caused by concurrent attempts to upgrade to write access.*

1. Introduction

Along with other researchers we have been investigating the design and implementation of `atomic` blocks for building thread-safe shared memory data structures. The idea of `atomic` blocks as a programming language construct is often conflated with the use of *Transactional Memory* (TM [13]) as an implementation mechanism: TM provides a mechanism for performing a series of memory accesses atomically, and so an `atomic` block can be built by using TM for the memory accesses enclosed within it. In this paper we examine the use of different implementations of `atomic` blocks, and the selection between them in runtime based on dynamic feedback.

In principle many alternatives are possible (Section 2). This, in part, is the beauty of `atomic` blocks. Concretely, however, we combine an *optimistic* mode of execution, in which an `atomic` block runs using software transactional memory (STM [25]) and a *pessimistic* mode, in which locks are associated with data items and the `atomic` block must acquire these locks before accessing the data.

Selecting between these alternatives is a complicated trade-off that is dependent both on the application (are the atomic blocks long or short?), the workload it is given (do atomic blocks frequently conflict with one another?) and on the resources allocated to the process (are threads running concurrently on different cores, or are they time-multiplexed on a single core)? Furthermore, when writing a library component, the "best" form of concurrency control to use may depend on the library's caller, and indeed the choice may vary between different callers in the same application. We therefore believe that the choice between these alternatives must be made at run-time, when all of this information is available, rather than being made by the programmer at compile-time.

The intuition behind our approach comes from our previous case study [26] which suggests that many of the conflicts that occur within a program using `atomic` blocks are triggered by a small number of "hot" variables; by treating these variables carefully we hope to prevent conflicts occurring. We discuss a concrete example in Section 3.

The key novelty in our approach is to distinguish between different kinds of data that transactions might access, rather than just selecting between different implementations for a complete transaction, or different TM implementations for the complete system. This can let us run the majority of a transaction using a streamlined implementation, while handling the "hot" variables carefully.

Section 4 details the design and implementation of our system. We base our work on our existing implementation of STM in Haskell [15]; a mature functional programming language. Although the functional core of Haskell looks very different to a mainstream language like C++, all of the accesses to shared mutable variables are written in an imperative style [22]. The STM is entirely conventional. Haskell is unusual in that the implementation of `atomic` blocks has been publicly available for several years and so we have example programs which were not written by the language implementers.

In Section 5 we compare the performance of our approach with (i) the baseline implementation of atomic blocks using STM with optimistic concurrency control, and (ii) with the baseline implementation extended to use the Polka contention management algorithm [24] that employs a form of priority-based randomized back-off to reduce the conflict rate between a set of transactions. As we show, there are various cases where the original, optimistic baseline implementation performs best of all, and others where Polka fits well or the pessimistic approach achieves overall improvements over the others.

We discuss related work in Section 6 and future work in Section 7.

2. Implementations of atomic blocks

In this section we discuss a number of different implementations of `atomic` blocks and the trade-offs between them.

Transactional memory with fine-grained commit (Tx-FG). The existing implementation of `atomic` blocks in Haskell works in this way. This baseline implementation provides system-wide progress because a transaction can only be forced to roll-back by a conflict with a concurrent transaction that has committed successfully; live-lock is not possible. This implementation provides disjoint-access parallelism, meaning that non-conflicting transactions can run and commit in parallel. This is built using optimistic concurrency control for the variables that the transaction has read from, and commit-time two-phase locking for the updates.

Transactional memory with coarse-grained commit (Tx-CG). This uses a single global lock to serialize transactions' commit operations: the bodies of transactions can still run in parallel, and conflicts are still detected at the level of individual data items. Compared with Tx-FG, this streamlines the single-threaded performance of a commit operation (since fewer locks are used), but serializing commit operations limits scalability. Consequently it is an appropriate choice in applications with a small number of threads, or applications where commit operations are rare.

Transactional memory with pessimistic concurrency control (Tx-Pes). STM implementations can perform concurrency control during execution, rather than just at commit time [20]. One approach, based on pessimistic concurrency control, is to use reader/writer locks to grant access to locations: conflicting transactions are delayed until a lock that they require is available, and deadlock-avoidance is necessary. This is an appropriate choice in workloads where the optimistic implementations of transactional memory experience high conflict rates. However, pessimistic concurrency control does not work well for low-conflict workloads because the implementation of locking typically introduces contention in the memory system (e.g. when updating counts of reading transactions, or lists of readers).

Lock inference (LI). In other contexts researchers have investigated using static analyses to associate locks with pieces of data that are accessed in transactions [5, 9, 14]. This has the potential to reduce the number of locks acquired or released when compared with Tx-Pes, although the difference is hard to predict quantitatively because the inference algorithm is typically conservative (i.e. some non-conflicting transactions become serialized). It is not clear how to combine lock inference with condition-synchronization.

In this paper we study the choice between Tx-FG and Tx-Pes; we discuss extension to further alternatives in Section 7.

3. Motivating Example

To illustrate our approach, consider the example in Figure 3 taken from a C# version of the Genome program from the STAMP STM benchmark suite [18]. In this example, a number of threads run the outer-most loop with each thread taking a disjoint set of `i_start`...`i_stop` values. Within the inner loop each thread computes a hash value of part of the input genome sequence and adds details to a hashtable referred to by an element of the `uniqueSegments` array. Superficially this approach looks scalable because the only conflicts will be between the threads with both the same `hashNumber` and the same `hashId`.

However this scalability can be lost when using a library hashtable implementation if each `Add` operation on a table increments a shared counter of the number of elements. To retain scalability, the hashtable implementation must typically be re-implemented for STM to avoid using a shared counter; this is undesirable in general because of the additional work it involves.

This example would not be handled well by the fine-grained STM implementation in Haskell: the first thread to commit would update a hashtable's `count` field, and concurrent transactions that have read the field would be forced

```
for (int i = i_start; i < i_stop; i += 12) {
  atomic {
    int ii_stop = (i_stop < (i + 12)) ? i_stop : (i + 12);
    for (int ii = i; ii < ii_stop; ii++) {
      String _value = segments[ii];
      uint hashId = hashString(_value);
      uint hashNumber = hashId % (uint)Config.threadNumber;
      if (!uniqueSegments[hashNumber].ContainsKey(hashId)) {
        uniqueSegments[hashNumber].Add(hashId, _value);
} } } } }
```

Figure 1. Example code from Genome

to abort and re-execute. It would also not perform well with typical implementations based on lock-inference. This is because locks are only acquired at the *start* of an atomic block, and so possible accesses to a common `count` field would prevent any concurrency between these `atomic` blocks. Furthermore, this example would be unlikely to perform well with implementations based solely on pessimistic concurrency control. This is because the single-threaded overhead of acquiring and releasing locks would slow down the initial portion of the `atomic` block that is accessing "cold" data that is not involved in conflicts.

Our approach in this paper is to dynamically identify the "hot" transacted variables – such as a shared `count` field – and to switch the management of these to use pessimistic concurrency control. In effect we introduce locking from the first access to each "hot" variable, until the end of the `atomic` block that contains it. We aim to continue to use optimistic concurrency control for the bulk of the data that a transaction accesses, so that we do not slow down the transaction's overall single-threaded execution time.

4. Implementation: Supporting pessimistic concurrency control

We first describe the baseline STM-Haskell system (Section 4.1), then our implementation of programmer-controlled selection between pessimistic and optimistic concurrency control (Section 4.2), the mechanism for dynamic selection between them (Section 4.3), and then our implementation of a Polka-style contention management policy for comparison with our approach (Section 4.4).

4.1. Background: STM in Haskell

The Glasgow Haskell Compiler (GHC) 6.6.1 provides a compilation and runtime system for Haskell 98, a pure, lazy, functional programming language. STM can be expressed elegantly in such a declarative language, where Haskell's type system guarantees that transactionally-managed state can only be accessed from within a transaction [10]. The type system also guarantees that a transaction cannot perform "non-transactional" operations, like calling I/O functions with unknown side-effects.

Transactional execution with `TVars`. In STM-Haskell `atomic` blocks are written using explicit operations to read and write from designated "transactional variables" (`TVars`). This follows the way in which mutable reference cells are accessed explicitly in the rest of the language.

Figure 2 shows the operations involved. `newTVar v` allocates a new `TVar` holding the value of `v`. `readTVar x` returns the value held in `x`. `writeTVar x v` writes the value `v` to `x`. The function `atomically` takes a series of these operations and executes them in an atomic step. The functions `retry` and `orElse` are used for composable blocking: calling `retry` indicates that the current atomic action is not ready to run to completion, and `orElse x y` provides an alternative action `y` to execute in case the first choice `x` is not ready to run. Earlier work has provided an operational semantics for these constructs [10].

For example, a program to increment the value stored in a `TVar` would be:

```
increment :: TVar int -> IO ()
increment x =
    atomically do v <- readTVar x
                  writeTVar x (v+1)
```

The existing implementation of atomic blocks in STM-Haskell uses optimistic concurrency control. As a transaction runs, it builds up a log of the `TVars` it has read from (its read-set) and the updates that it wants to make (its write-set). When it tries to commit, it checks that there have been no intervening conflicting updates committed to these `TVars`. If there have been conflicts then it re-executes immediately. If there have not been any conflicts, then its updates are written back. This approach has been termed *lazy version management* and *lazy conflict detection* [19].

Running STM Operations	**Transactional Variable Operations**
`atomically :: STM a -> IO a`	`data TVar a`
`retry :: STM a`	`newTVar :: a -> STM (TVar a)`
`orElse :: STM a -> STM a -> STM a`	`readTVar :: TVar a -> STM a`
	`writeTVar :: TVar a -> a -> STM ()`

Figure 2. Haskell STM Operations

User-mode scheduling in Haskell. Haskell threads are scheduled in user-mode over a pool of OS threads. To encourage locality, the runtime system associates a list of Haskell threads with each OS thread, and provides each OS thread with a separate allocation buffer for the objects that it creates.

Switching from one Haskell thread to another occurs entirely in user mode. Typically, the user-mode scheduler will not switch between Haskell threads within a transaction (avoiding the problem of *pseudo-parallelism* [6]).

4.2. Pessimistic `TVar`s

We extend the programming API to include a new operation `newPesTVar` for explicitly creating a `TVar` that will be managed by pessimistic concurrency control. This new kind of `TVar` is identical from the point of view of the programmer; it can be read and written by the same `readTVar` and `writeTVar` operations. However, it is treated differently from a normal `TVar` by the runtime system. We do not intend this `newPesTVar` operation to be used by programmers; we provide it for use in testing and benchmarking.

At runtime a `PesTVar` extends the structure of a normal `TVar` with a multi-reader single-write lock implementation built using an explicit list of reading transactions, or a single writing transaction. The lock is acquired in the appropriate mode when a transaction attempts to read or write the `PesTVar`, and it is released when the transaction has finished (committed, aborted, or blocked using `retry`).

If a lock is not available then we use a deadlock avoidance heuristic to identify whether or not the Haskell thread may block. Currently, we conservatively assume that deadlock is possible if there are any threads waiting for access to `PesTVar`s that the current thread has locked. (In Section 5.1 we discuss the impact of this choice, rather than a less conservative approach, e.g. [16].)

If deadlock is impossible, then the Haskell thread blocks and the user-mode scheduler switches to another Haskell thread if one is available. Conversely, if deadlock may have occurred, then the transaction is condemned (prevented from committing), the locks that it already holds are released, and then it waits on the single `PesTVar` that it was attempting to access.

4.3. Dynamic selection

When a transaction fails to commit, we *blame* the `TVar`s whose values have changed during the transaction. We use a simple per-TVar blame-threshold to trigger conversion from an ordinary optimistic `TVar` to a `PesTVar`.

In our prototype, this adaptation just involves updating the object's header word to indicate the change in the object's type; of course, this change must be done in unsafe C code in the GHC runtime system, rather than in Haskell.

Although this approach is simple, it means that an ordinary `TVar` must hold the additional fields necessary to support pessimistic operation. In Haskell-STM this seems acceptable for the workloads that we have studied; most data is immutable functional objects, and so the structure of these is unchanged. The overhead on normal `TVar`s could be avoided by marking candidates for adaptation and expanding them to a larger `PesTVar` structure during garbage collection.

4.4. The Polka contention manager on GHC

Many STM implementations use a *contention manager* to attempt to resolve conflicts between transactions, and to delay transactions' re-execution in the presence of contention. This approach was originally proposed for obstruction-free systems [11] and used with DSTM [12]. It is particularly important in that setting because an obstruction-free system only guarantees progress to operations that run without experiencing contention: the role of the contention manager is to arrange that such runs occur, preventing live-lock.

The baseline Haskell-STM implementation does not use a contention manager because it cannot suffer from live-lock since one transaction can only be forced to roll-back because another transaction has successfully committed. However, to provide a point of comparison for our use of pessimistic synchronization, we have also implemented a contention manager as an alternative way to reduce contention on "hot" `TVar`s.

Many alternative contention managers have been studied [24, 23], and although no individual scheme performs uniformly best, the *Polka* algorithm is generally effective. This combines exponential back-off with "priority accumulation": a transaction accumulates priority as it adds objects to its read/write sets. In the case of a conflict, when an operation is obstructed by another with higher priority, it will back off, initially for a randomized amount of time, with back-off intervals increasing exponentially. Each back-off further increases the transaction's priority; only when its priority is greater than that of its obstructor will it abort it. Polka aims to minimize wasted work and memory interconnect contention [23].

To enable Polka to detect conflicts we added queues of current readers and writers to each `TVar`. We support a visible-reader mode (where the presence of readers is recorded), and an invisible-reader mode (where only the presence of writers is recorded). To support priority accumulation, a new field that indicates the size of the transaction's read and write sets was added. In case of an abort, the aborted transaction passes this information to the new transaction. We add an "abort request" field to each transaction record, so that a different transaction can signal that it has won a conflict. We use a simple timing loop for exponential back-off.

The Haskell version of Polka has some differences from the original. Incremental validation during execution is not necessary: even if a transaction becomes invalid then all it can do it access `TVars` (unlike in C++ where it might try to access de-allocated memory).

5. Results

We studied the performance of our implementation using a 4 dual-core Dell machine with 64-bit Xeon processors clocked at 3.2 GHz. We ran experiments on 1..8 cores; a single-threaded garbage collector is still used in this version of GHC which is known to limit the scalability somewhat [21]. All results are the mean of five runs. Benchmark runs were configured to last for several seconds.

Figure 3 shows the test cases we used. These are divided into two categories. First, there are synthetic tests and data-structure micro-benchmarks: a high-contention workload with concurrent threads incrementing a single shared integer, and mixed insert/delete/search workloads on linked lists, binary search trees, and a hashtable with external chaining.

The second category of tests are two application programs; neither of these was written explicitly as an STM benchmark. "Parallel Sudoku" is a parallel search algorithm for Sudoku puzzles. Parallelism is used to explore different alternative choices – e.g. different numbers that might be placed in the same square in the puzzle. `atomic` blocks are also used when updating a shared result structure that holds the complete solution that is obtained. "SPI-Calculus" is an interpreter for expressions written in a process algebra that models cryptographic operations [1]. Different parts of the expression are evaluated in parallel; they communicate through a shared heap.

We plot the performance of six runtime-system configurations for each program:

- *Opt* is the baseline implementation of STM using fine-grained commit-time locking.
- *Pes-0* uses multi-reader single-writer pessimistic concurrency control for all `TVars`.
- *Pes-3* uses optimistic concurrency control by default, and switches a `TVar` to pessimistic concurrency control after it has been blamed for 3 conflicts.
- *Pes-ME* uses pessimistic concurrency control with mutual exclusion for all `TVars`.
- *Pes-ME-3* uses optimistic concurrency control by default, and switches a `TVar` to pessimistic concurrency control using mutual exclusion after it has been blamed for 3 conflicts.
- *Polka* uses Polka-style contention management with invisible readers (the variant with visible readers performed uniformly worse; the overhead of managing visible reader information dwarfed the possible improvements in contention management).

For each configuration we plot the wall-clock time-to-completion for each program, and also examine the ratio of aborted transactions to the committed transactions during its execution.

Figure 4 shows the results for the data-structure micro-benchmarks, and Figure 5 shows the results for the whole applications. For each test case we plot the benchmark's execution time on 1..8 cores on the left hand graph (higher is worse), and the ratio of aborts to committs on the right-hand side. We normalize this against the single-thread performance using optimistic concurrency control.

Broadly speaking, these results suggest that there are not many cases where switching "hot" `TVars` to pessimistic concurrency control works for these workloads. The exceptions are the Sudoku solver and the linked-list micro-benchmark under higher contention loads.

In some cases the reason for this is clear (and entirely expected). For example, in the synthetic single-integer test, the `atomic` blocks are very short, and the window at which an optimistic implementation is "at risk" from a conflict is low: the cost of managing locks dominates the cost of the aborted work that is saved.

In other cases there are a number of different factors involved. First, the best-case speed-up that can be achieved over *Opt* is limited by the amount of wasted work that *Opt* performs. In the 4-core SPI-Calculus benchmark, fewer than 1 in 20 transactions re-executes. The results are

Micro-benchmarks	Description	# lines	# atomic
SharedInt	A corner-case program incrementing a shared integer variable for a total of 200,000 times.	82	1
Linked-List (LL)	Singly-linked list applications 10% inserting, 10% deleting and 80% searching random numbers. The maximum list length is 500 nodes.	265	7
Binary Tree (BT)	Binary trees inserting, deleting and searching random numbers. The steady-state size of the tree is 1,000 elements.	262	7
HashTable	A hash table implementation inserting, deleting and searching random numbers. The steady-state size of the table is 1,000 elements.	544	5

Applications	Description	# lines	# atomic
Parallel Sudoku	A parallel Sudoku solver.	282	7
SPI Calculus Interpreter	A parallel interpreter for the SPI-Calculus.	1867	14

Figure 3. Description of the test cases: number of lines of code and atomic blocks

broadly similar in the hash-table case. It is possible that our approach will be more significant under higher levels of contention – and, of course, with STM implementations where (unlike *Opt*) livelock can occur in the absence of contention management. Although Polka appears to perform well under the depicted executions of the linked structures (LL and BT), when ran with bigger transactions that use larger maximum list sizes than what is shown here, it can fall in livelock and fail to complete. Similar behavior was also observed running STAMP applications on DSTM2 [3].

The second important factor is the role of deadlock. This proved to be significantly more important than what we had anticipated. In particular, "hot" `TVars` are frequently ones where multiple threads perform read-modify-write operations. Deadlock occurs when multiple threads holds the lock in read-mode, and want to upgrade to write access – none of the threads can be granted write-mode access until the readers have aborted. This means that, unlike *Opt*, a transaction can be forced to roll-back without another transaction having made progress.

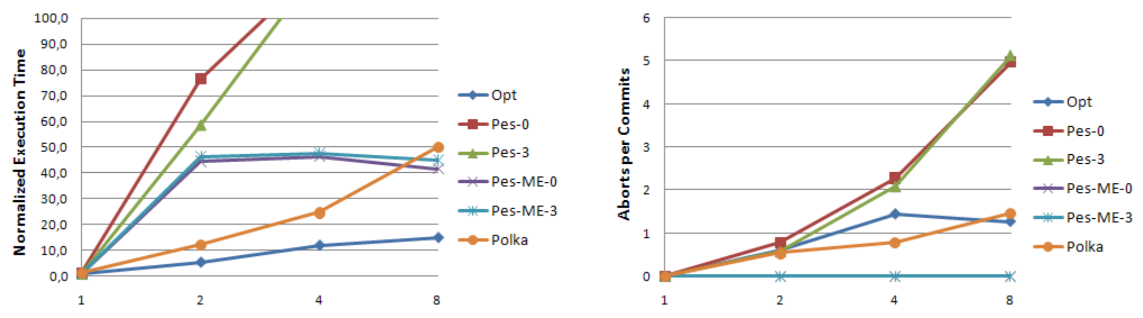
Note that we can distinguish two kinds of deadlock: *real* deadlock where the threads involved form a waiting-cycle, and *false* deadlock where our basic deadlock-detection heuristic is conservative. We constructed the `Pes-ME` implementation to help investigate the proportion of these two cases. This replaces all `TVars` with mutual exclusion locks, so precludes the concurrent-upgrade problem. We see almost no aborts to prevent possible deadlocks in this case; never more than 15 in any of these several-second runs. This suggests that we are seeing real deadlock via upgrades on the multi-reader locks, rather than just seeing false deadlocks caused by our simple deadlock-detection heuristic.

Although *Pes-ME* is very effective at avoiding wasted work it does, of course, serialize many transactions' operations (for example, the binary-tree results illustrate this). We also examined a hybrid scheme, switching from optimistic to pessimistic concurrency control with mutual exclusion locks after a given number of conflicts on a `TVar`. This is shown by the *Pes-ME-3* plots. In some cases this configuration manages to combine the low abort rate of *Pes-ME* with the better scalability of *Opt*.

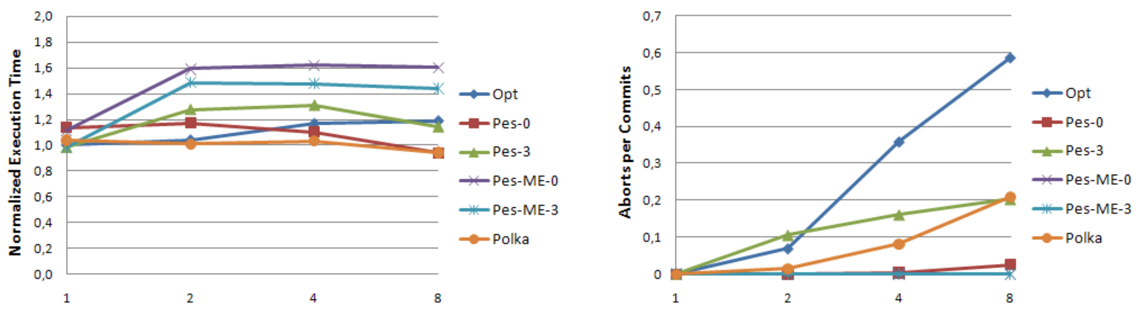
5.1. Summary

To summarize our conclusions from these results:

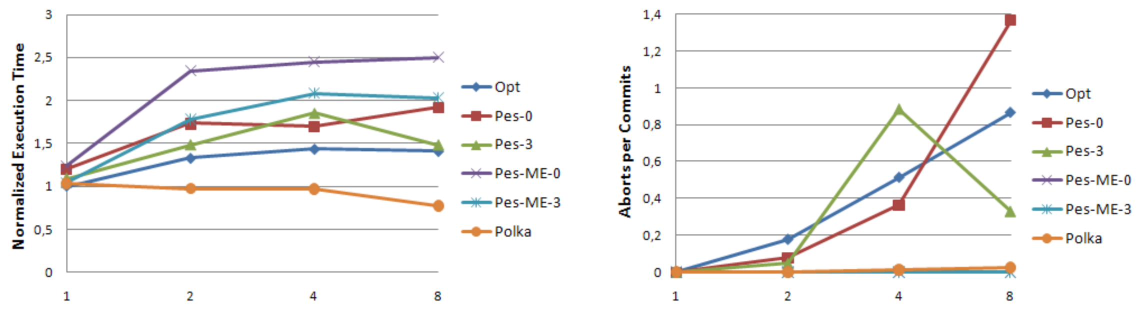
- Selectively using pessimistic concurrency control proves effective at reducing the amount of wasted work by reducing the number of transactions that abort.
- Using mutual-exclusion locks might be preferable to using multi-reader single-writer locks for some workloads; the implementation is simpler, and deadlock is reduced.
- The good performance of the baseline *Opt* implementation (and the lack of such clear impact of *Polka*, when compared with earlier work [23]) is because *Opt* can only cause a transaction to abort because a conflicting transaction has successfully committed; not because a conflicting transaction is in progress.
- With this implementation of *Opt* the amount of wasted work is often low, or the transactions are so short that direct re-execution is as fast as waiting.



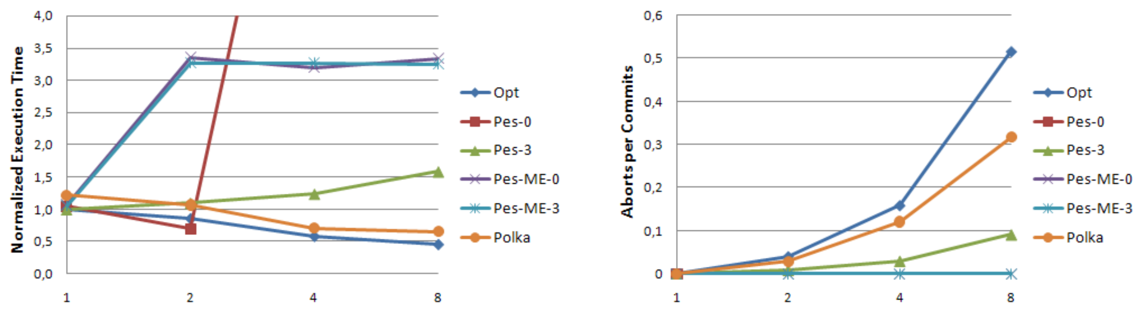
Shared-Integer



Linked-List

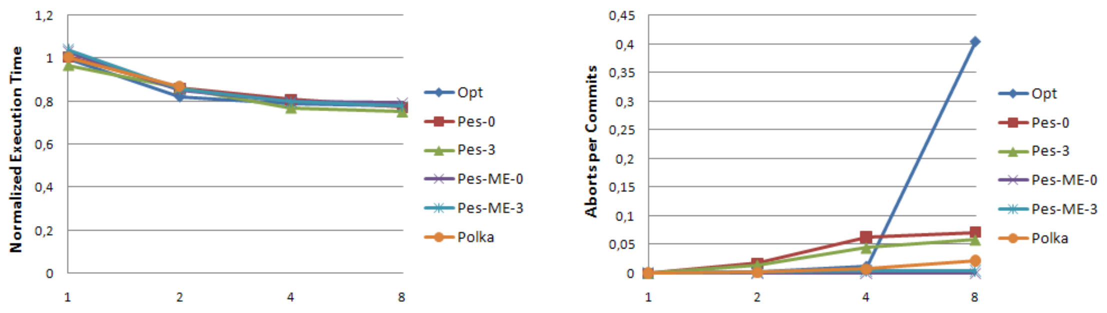


Binary Tree

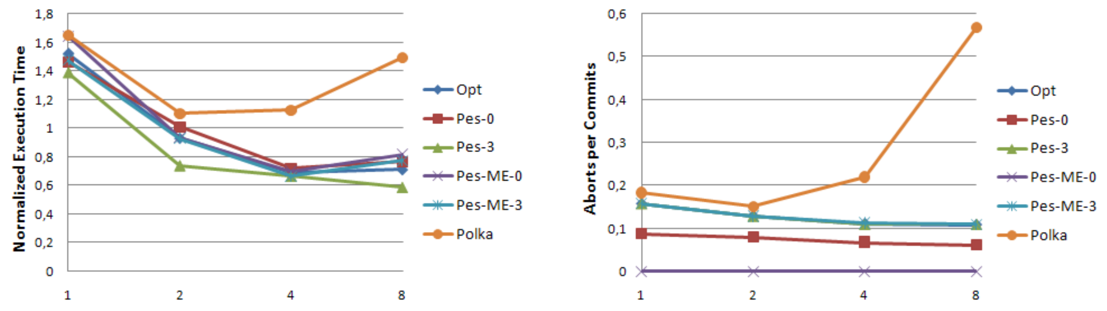


Hashtable

Figure 4. Data-structure micro-benchmarks, showing scaling relative to 1-thread STM baseline (left), and the ratio of the total number of aborted transactions to committed transactions (right).



SPI-Calculus Interpreter



Sudoku Solver

Figure 5. Whole-program benchmarks, showing scaling relative to 1-thread STM baseline (left), and the ratio of the total number of aborted transactions to committed transactions (right).

6. Related work

There has been a substantial amount of work on contention management techniques in general, and in STM in particular. Herlihy *et al.* introduced the idea of writing non-blocking algorithms by combining an obstruction-free algorithm with an out-of-band contention manager [11]. The obstruction-free algorithm guarantees progress in the absence of contention, and the contention manager attempts to create contention-free runs. Bounds on the behavior of this approach have been studied from a formal point of view [8].

Scherer and Scott examined a wide range of contention managers for use with transactional memory [24, 23], and showed how the best-performing choice depended on many factors. However, the Polka algorithm was identified as a promising general-purpose approach. Scherer's experiments [23] used a range of data-structure micro-benchmarks, and so the fact that different contention managers are appropriate in different cases suggests that, in a larger application built from a range of data structures, a single choice may not be appropriate for the whole system.

Ansari *et al.* [2] and Yoo *et al.* [30] examined the approach of adjusting the level of concurrency as a way of avoiding contention. For example, if there are frequent aborts, then reducing the number of transactions that are running concurrently may reduce the amount of wasted work. As with the selection of a process-wide contention management algorithm, this approach may harm the performance of "good" transactions (which do not suffer any conflicts) when they are running concurrently with a set of "bad" transactions (which conflict with one another).

In recent work Dolev *et al.* [6] describe a serializing contention manager which, as with our work, benefits from integration between the STM and the thread-scheduler. Their approach limits the number of times that any pair of transactions can conflict by serializing their execution in one order or the other after their first conflict. They allow the programmer to provide hints about transaction placement to reduce the likelihood of conflict.

Welc *et al.* [29] and Spear *et al.* [27] propose the use of an irrevocable transaction (i.e., one that can not be aborted) as a contention management mechanism. Typically only one such transaction can be supported in a system, but the implementation guarantees that that transaction will be able to run to completion. It would be interesting to combine this approach with ours, treating irrevocability as a dynamic choice made by the runtime system (e.g. in the face of frequent possible-deadlock detections).

The Intel STM Compiler [20] supports a mode that uses pessimistic two-phase locking for reads and writes, and an "obstinate mode" which allows one pessimistically-managed transaction to run while winning all conflicts it is involved in. These work on a per-transaction basis, rather than trying to target "hot" variables that cause conflicts.

In recent work, Waliullah and Stenstrom use individual conflicting transacted variables to set checkpoints for partially rolling back transactions in case of conflicts in hardware TM systems [28].

Furthermore, transaction processing systems (e.g., [7, 4, 17]) benefit from the observation that optimistic concurrency control seems to work better for data under low contention, while locking appears to work best for data under high contention.

7. Conclusions and future work

In this paper we have examined the use of dynamic adaptation between optimistic and pessimistic synchronization in an implementation of `atomic` blocks. Our results surprised us somewhat: the performance of the baseline optimistic implementation is frequently good because there is little wasted work. It is not yet clear whether this is a property of the particular workloads that we have studied (e.g. they may have been written to perform well on the current implementation), a property of our particular baseline STM implementation, or if it is true in general.

We did not initially anticipate that using mutual-exclusion locks might be effective for "hot" `TVars`. In some workloads we have studied this turns out to be the case because most transactions perform read-modify-write operations on these variables – MRSW-style locking leads to deadlocks when multiple readers wish to upgrade to write access. Mutual-exclusion locks avoid this, and dynamically switching to this form of synchronization could prove an effective way of reducing wasted work where it is present.

There are a number of possible directions for future work. First, we would like to identify sets of `TVars` that are accessed together and perform concurrency control on them as an aggregate: i.e. acquiring a single lock, rather than many locks. This may also help avoid needing to abort transactions because of possible deadlock. Second, we would like to introduce further alternative implementations of `atomic` blocks – e.g. selecting between in-place updates and deferred updates.

8. Acknowledgments

We would like to thank Simon Peyton Jones for making the SPI-Calculus application available to us, Wouter Swierstra for the parallel Sudoku solver, Edward Kmett for the hashtable library and Simon Marlow and Srdjan Stipic for help with building and modifying the GHC runtime system. We would also like to thank Sasa Tomic for comments on an earlier draft of this paper.

This work is supported by the cooperation agreement between the Barcelona Supercomputing Center – National Supercomputer Facility and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625, by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC) and by the European Commission FP7 project VELOX (216852).

References

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[2] M. Ansari, C. Kotselidis, K. Jarvis, M. Lujan, C. Kirkham, and I. Watson. Experiences using adaptive concurrency in transactional memory with Lee's routing algorithm. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 261–262, February 2008.

[3] M. Ansari, C. Kotselidis, M. Lujan, C. Kirkham, and I. Watson. Investigating contention management for complex transactional memory benchmarks. *In Proc. Second Workshop on Programmability Issues for Multi-core Computers (MULTIPROG'09)*, January 2009.

[4] M. S. Atkins and M. Y. Coady. Adaptable concurrency control for atomic data types. *ACM Transactions on Computer Systems*, 10(3):190–225, 1992.

[5] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. *Proc. 2008 ACM SIGPLAN conference on programming language design and implementation (PLDI 2008)*, 43(6):304–315, 2008.

[6] S. Dolev, D. Hendler, and A. Suissa. CAR-STM: Scheduling-based collision avoidance and resolution for software transactional memory. In *PODC '08: Proceedings of the twenty-seventh ACM symposium on Principles of Distributed Computing*, pages 125–134, August 2008.

[7] R. E. Gruber. Temperature-based concurrency control. *Proc. Third Int. Workshop on Object Orientation and Operating Systems*, pages 230–232, December 1993.

[8] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic contention management. In *Proc. of 19th International Symposium on Distributed Computing (DISC 2005)*, pages 303–323, September 2005.

[9] R. L. Halpert, C. J. F. Pickett, and C. Verbrugge. Component-based lock allocation. In *PACT '07: Proceedings of the 16th International Conference on Parallel*

Architecture and Compilation Techniques, pages 353–364, September 2007.

[10] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proc. 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, June 2005.

[11] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 522–529, May 2003.

[12] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, pages 92–101, July 2003.

[13] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–301, May 1993.

[14] M. Hicks, J. S. Foster, and P. Prattikakis. Lock inference for atomic sections. In *First ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2006)*, June 2006.

[15] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 1–55, June 2007.

[16] E. Koskinen and M. Herlihy. Dreadlocks: efficient deadlock detection. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in Algorithms and Architectures*, pages 297–303, June 2008.

[17] B. Liskov, M. Day, and L. Shrira. Distributed object management in Thor. *Distributed Object Management*, pages 79–91, 1993.

[18] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IEEE International Symposium on Workload Characterization (IISWC 2008)*, pages 35–46, September 2008.

[19] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proc. 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. February 2006.

[20] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Ollivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for C/C++. In *International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2008.

[21] C. Perfumo, N. Sönmez, S. Stipic, O. Unsal, A. Cristal, T. Harris, and M. Valero. The limits of software transactional memory (STM): dissecting Haskell STM applications on a many-core environment. In *CF '08: Proceedings of the 2008 conference on Computing Frontiers*, pages 67–78, May 2008.

[22] S. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. July 2000. Presented at the 2000 Marktoberdorf Summer School.

[23] W. N. Scherer III. *Synchronization and Concurrency in User-level Software Systems*. PhD thesis, University of Rochester, Department of Computer Science, 2006.

[24] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248, July 2005.

[25] N. Shavit and D. Touitou. Software transactional memory. In *Proc. 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, August 1995.

[26] N. Sönmez, A. Cristal, O. Unsal, T. Harris, and M. Valero. Why you should profile transactional memory applications on an atomic block basis: A Haskell case study. *In Proceedings of Second Workshop on Programmability Issues for Multi-core Computers (MULTIPROG'09)*, January 2009.

[27] M. Spear, M. Michael, and M. Scott. Inevitability mechanisms for software transactional memory. In *Third ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2008)*, February 2008.

[28] M. M. Waliullah and P. Stenstrom. Intermediate checkpointing with conflicting access prediction in transactional memory systems. *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, April 2008.

[29] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA '08: Proceedings of the twentieth annual Symposium on Parallelism in Algorithms and Architectures*, pages 285–296, June 2008.

[30] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in Algorithms and Architectures*, pages 169–178, June 2008.