On Justifying and Verifying Relaxed Detection of Conflicts in Concurrent Programs

Position paper

Omer Subasi, Serdar Tasiran
Koç University
{osubasi,stasiran}@ku.edu.tr

Tayfun Elmas
University of California, Berkeley
elmas@cs.berkeley.edu

Adrián Cristal, Osman Unsal
Barcelona Supercomputing Center
{adrian.cristal,osman.unsal}@bsc.es

Tim Harris
Microsoft Research
tharris@microsoft.com

Rubén Titos-Gil
University of Murcia
rtitos@ditec.um.es

Abstract

Transactional Memory (TM) simplifies concurrent programming by providing atomic, compositional blocks within which programmers can reason sequentially. Many transactions have an access pattern where, initially, a large portion of the shared data is read, local computations is performed, and, later, a small portion of shared data is updated. Such transactions conflict frequently and result in poor performance. TM implementers have developed mechanisms for programmer-directed relaxed conflict detection. The programmer indicates to the TM to ignore certain conflicts. As a result, the frequency of conflict is reduced and performance approximates hand-crafted fine-grain concurrency. However, the clean, sequential, atomic compositional block simplicity of TM is lost. In this paper, we present a method for abstracting in a quantified manner the accesses on which conflicts are ignored. This abstraction, and a soundness theorem we state and prove, allow the programmer to reason sequentially on an abstracted version of the transaction. Properties proved on this abstract but sequential version of the transaction carry over to the original program.

1. Introduction

Transactional memory (TM) [11] simplifies shared-memory concurrent programming greatly by providing atomic blocks as a composable programming construct while, transparently to the programmer, allowing concurrency. Within atomic blocks, programmers are able to use sequential reasoning and sequential verification tools. Invariants about the program need only be proven using the atomic blocks as coarse-grain actions.

A typical TM implementation maintains read and write sets – the sets of memory locations that a transaction reads from and writes to. Transactions proceed optimistically, but when the write set of one transaction has a non-empty intersection with the read or write sets of another concurrent transaction, one of the transactions is aborted and retried. Careful contention management results in acceptable performance for many programs.

For atomic blocks with a certain access pattern, however, the straightforward conflict-detection approach results in poor performance. These programs are characterized by long-running transactions that start with a "read phase" where a possibly-large portion of shared data is read by the transaction. The transaction then carries out local computation and proceeds to its "commit phase," where it updates shared state. The commit phase may contain both read and write accesses to the shared state. Stated informally, these transactions have the property that if the following conditions are met, their updates to the shared state are valid:

1. The read phase reads a consistent but possibly outdated (stale) snapshot of the relevant part of the shared data,
2. The accesses in the commit phase do not conflict with the writes of other concurrent transactions

Examples of this access pattern can be found in the STAMP transactional programming benchmark suite [3, 13]. In the `Genome` benchmark, multiple concurrent transactions perform insertions into a sorted linked list. All insertions read prefixes of the linked list. Consider two transactions, one performing an insertion closer to the head of a long linked list, and the other one closer to the tail. The prefix read by the latter transaction will be modified by the former one. Yet, both transactions should be able to commit as they will be performing insertions correctly, despite this "Write-After-Read (WAR)" conflict. In the `Labyrinth` benchmark, concurrent transactions try to route wires connecting pairs of points in a three-dimensional grid. Each transaction first makes a local copy of the grid state and then tries to find a path. As long as the path found does not intersect with another one found by a concurrent transaction, the labyrinth router functions correctly. In these two examples, invariants of the program, i.e., sortedness of the list or non-overlapping of paths, hold true despite the fact that transactions may be operating on a stale but consistent snapshot.

In order to provide reasonable performance in these cases, TM packages have devised programmable mechanisms that allow transactions to ignore certain conflicts [5, 9, 12, 13]. For instance, [13] allows the programmer to use "!WAR" annotation for part of the transaction, with the interpretation that writes by other transactions conflicting with reads performed in the !WAR portion of the transaction will be ignored. These studies have demonstrated that the use of relaxed conflict detection greatly improves performance. Unfortunately, so far, this has come at the expense of losing the simplicity of sequential reasoning within a transaction. The programmer currently has no clear theory or tools using which she can reason about and verify that her program will run as intended under such TM with relaxed detection of conflicts.

This paper is about verifying programs that use TMs with programmer-controlled relaxed conflict detection. We provide a way of modeling relaxed conflict detection, which, in turn, allows the programmer to use sequential reasoning on a more abstract but sequential version of the transaction. With this approach, static proofs of concurrent programs using TMs with relaxed conflicts are still much simpler than those for hand-crafted fine-grain concurrent programs. In order to allow sequential thinking within a transaction, for each variable access α, we identify the set of accesses $IgnoreConflicts(\alpha)$ in the program such that $\beta \in IgnoreConflicts(\alpha)$ iff the transaction ignores potential conflicts between α and β. We then abstract α to $\widetilde{\alpha}$ in a prescribed way. This abstraction makes $\widetilde{\alpha}$ express possible side effects of the conflicting accesses β on access α and allows transactions containing $\widetilde{\alpha}$ to be serialized in a sound manner. For instance, a read of a variable x, instead of returning the current value of x, can return a value for x that may be the result of one of the conflicting writes to x. Put differently, we add more behavior to relaxed accesses by replacing them with abstracted versions. As a result, we obtain a transaction with more abstract actions, but one that now can be treated sequentially. On this version of the program, invariants, pre- and post-conditions, and assertions can be proven using sequential reasoning. The soundness theorem for our approach guarantees that properties proven on the abstract sequential version hold of the original program—when it is run under TM with relaxed detection of conflicts.

The atomicity guarantee provided by TMs relies on the fact that, stated in Lipton's theory of movers [8], that actions are of the correct mover types (right-mover or left-mover) so that they can all be made contiguous around the commit action of the transaction. In terms of the access pattern described above, this translates to all read accesses needing to be right movers. Conflicting writes by other transactions, which are ignored due to the !WAR annotation, prevent reads from being right movers, and this is the reason we perform the abstraction as outlined above. To be able to prove anything interesting about programs using relaxed conflict detection, the programmer needs to provide a program-specific abstraction for read actions. The abstraction allows a read action commuted to the right of a conflicting write to still read the old (stale) value – the value before the conflicting write took place.

While our method requires manual work for the abstraction, we believe that the programmer who has directed the TM to ignore certain conflicts has in mind an intuitive model of possible interfering actions, therefore, it is not difficult for her to provide this abstraction.

In particular, for a read returning a value v, the programmer believes that, despite the interference by other concurrent writes, the value read still satisfies a certain property. By abstracting the read, the user provides us this guess. In the `Labyrinth` benchmark, for instance, this abstraction allows a read to non-deterministically see a free labyrinth cell as full to model interference from concurrent labyrinth transactions. In the `Genome` benchmark, while a thread is traversing a linked list, other threads may modify the link structure by inserting new nodes to the list. The abstraction of reads from the "next" pointers of list nodes allows a thread to jump over multiple nodes to ignore the writes to and thus to model the interference on the next pointers.

If the guess of the programmer about the safety of the abstraction is wrong, our method will not succeed, but it will tell the programmer that the guess was wrong, and why. Otherwise, the correctness of the guess is verified, and, using this guess, now the programmer can assume that she has a sequential atomic block and verify all relevant properties of the block she initially had. This can include assertions, pre- and post-conditions of blocks. In particular, let P be the original program and P_{Abs} be the program after applying the abstraction. If the abstraction in P_{Abs} is verified correct (sound), then our soundness theorem gives two guarantees. First, every concurrent execution of P_{Abs} is equivalent to an execution of P_{Abs} in which every transaction is serialized. This enables sound, sequential verification of any safety property in P_{Abs}. Second, the result of any sequential safety proof on P_{Abs} can be soundly carried on P.

Our method is compositional in the following way. When the program at hand is composed with others that contain transactions, the correctness of the existing parts of the programs can simply be checked by annotating the additional transactions with sequential assertions regarding their potentially interfering writes, and verifying these using sequential methods. This is not no cost, but, as is usually the case, improved performance requires somewhat reduced modularity.

To evaluate the usability of our technique, we verified pre- and post-conditions of transactions from `Genome` and `Labyrinth` benchmarks from STAMP [3] that are using TM with relaxed detection of conflicts. Titos et al. claimed in [13] that these transactions are intended to function correctly with !WAR annotation. Our work validates the claims in [13] about the correctness of these transactions and pro-

vides the systematic way of validating such claims for similar programs.

We performed the proofs in two steps. First, we assumed a TM that does not ignore any conflicts and did a standard, contract-based sequential proof. (Here, contracts include invariants, pre- and post-conditions and loop invariants.) For this, we used VCC [4] and HAVOC [7], state-of-the-art modular verification tools for C programs. Then, we applied the abstraction on the program annotated with contracts and applied the same tools. We found that the sequential verification of these benchmarks after the abstraction did not require extra or stronger contracts compared to the original sequential proof. This indicates that the arguments about why the transactions work under relaxed detection conflicts originates from the arguments about why these transactions work sequentially.

In summary, the contribution of this work is to provide a modeling and static, tool-supported verification method for programs using transactions with relaxed conflict detection. This will enable the improved performance of relaxed conflict detection, with performance close to hand-crafted concurrency control, with a little bit of extra static verification work. This verification work is made into a recipe, and is significantly easier than the work needed to verify hand-crafted, fine-grain concurrency data structures.

2. Overview

2.1 Running example: StringBuffer pool

Figure 1 shows the (C++-like) pseudocode for operations of a data structure that implements a pool of `StringBuffer` objects. Such pools are widely-used in text processing (e.g., indexing and searching) applications that create a high number of temporarily-used strings. The pool is represented by a fixed-size array (1K in our case) of StringBuffer object pointers. We say that a cell in the pool array is *full* if that cell contains a non-`NULL` pointer, and *empty* if it contains the `NULL` pointer.

We imagine a program with multiple threads that share the `pool` array to store references to unused `StringBuffer` objects. In the code, we mark the lines that read from or write to the shared pool array as shaded; other lines only access local variables.

The `Allocate` operation returns a pointer to a `StringBuffer` object that was either in the pool or created freshly using the `new` operator. `Allocate` operate by traversing the pool array and examining cells one at a time. If the cell `pool[i]` is full, `Allocate` swaps `pool[i]` with `NULL` and returns the old value of the cell.

The `Free` operation is the dual of `Allocate`. `Free` takes a pointer to a `StringBuffer` object, and it either inserts the pointer to an empty cell in the pool array or deallocates the pointed `StringBuffer` object using the `delete` operator. If the cell `pool[i]` is empty, `Free` swaps `pool[i]` with `NULL` and returns the old value of the cell.

Let the following invariant express the programmer's intended correctness condition:

SBPoolInvariant: At any time, every `StringBuffer` object must either be in the pool, or it must have been returned by a call to `Allocate` and being used by the program. This implies that, a call to `Free` must either insert the given the pointer to the pool or deallocate the corresponding object.

2.2 Choosing a concurrency control mechanism

Notice that, we do not explicitly decorate the code in Figure 1 with any concurrency control primitive. In fact, this program can run correctly—i.e., satisfy **SBPoolInvariant**—with multiple concurrency control mechanisms. A trivial concurrency technique that guarantees correctness is to acquire (resp. to release) a common lock, say `Lock`, before starting (resp. ending) operations `Allocate` and `Free`. However, such a common-global lock does not permit any concurrency, resulting a very poor performance if many threads attempt to simultaneously call operations of the same pool. Transactional memory (TM) [11] provides a high degree of concurrency, still preserving the invariant. In this case, the programmer can declare each of `Allocate` and `Free` a transaction and obtain the atomicity guarantee for both operations without any more changes in the code.

However, this atomicity guarantee by TM still comes with a considerable performance cost, which increases rapidly with the contention between threads [6]. To see why, observe that before writing to cell, `Allocate` operates in two phases (`Free` operates similarly but in a dual manner): First, in the "read phase", `Allocate` traverses the `pool` array and check if each cell (`pool[i]`) is full. If it finds a full cell, in the "commit phase", it swaps the value of that cell with `NULL`. Consider a thread T1 running `Allocate`, which reads cells 1 through 100 (line 4) in its read phase and writes the 101^{st} cell (line 6) in its commit phase. While T1 is in its read phase, other threads may attempt to write to any cell $1 - 100$; for example, another thread, say T2, running `Allocate` may empty a cell before T1 reads from it, or running `Free` may fill a cell after T1 reads from it. In such cases the TM implementation would detect a conflict between the read of T1 and the write of T2, and force one of the transactions to rollback its transaction. However, we observe that such Write-After-Read (WAR) conflicts between two threads only cause a correctness flaw (violates **SBPoolInvariant**) when the read access involved in the conflict is performed in the commit phase of the transaction, i.e., T1 reads from `pool[i]` and gets prepared to write to `pool[i]`, then T2 writes to `pool[i]`, and finally T1 attempts to write to the same cell `pool[i]`. Otherwise, reading a cell later overwritten by another thread does not violate the correctness property dictated by **SBPoolInvariant**. In summary, the correctness would be established under any concurrency control that provides atomicity for the last successful iteration of `Allocate` and `Free` that ends at line 7; the reads of `pool`

```
StringBuffer* pool[] = new StringBuffer[1000];

1  StringBuffer* Allocate() {
2    StringBuffer* ptr;
3    for (int i = 0; i < 1000; ++i) {
4      ptr = pool[i];
5      if (ptr != NULL) { // check if full
6        pool[i] = NULL; // empty cell
7        return ptr;
8      }
9    }
10   // default operation
11   return new StringBuffer();
12 }
```

```
1  void Free(StringBuffer* buff) {
2    StringBuffer* ptr;
3    for (int i = 0; i < 1000; ++i) {
4      ptr = pool[i];
5      if (ptr == NULL) { // check if empty
6        pool[i] = buff; // fill cell
7        return;
8      }
9    }
10   // default operation
11   delete buff;
12 }
```

Figure 1. StringBuffer pool example.

from the previous iterations do not have to be atomic with the last iteration.

We note that such concurrency control can be obtained by assigning a separate lock to each cell in the pool array, and acquiring the lock for `pool[i]` at each iteration `i` of the `for` loop. While this scheme can be improved for performance by using readers/writer lock, locking operations at each iteration would unnecessarily reduce the performance. In fact, the locking operations in the read phase of the transaction (for cells that will not be written) are unnecessary.[1] Clearly, for such programs, there is a need for concurrency control mechanisms that can not only always look for conflicts but also be configured to ignore some conflicts that do not affect the functionality of the program.

2.3 Relaxed detection of conflicts for StringBuffer pool

Titos et al. [13] have addressed the problem above by proposing a TM implementation with software-defined conflicts. By annotating a transaction with !WAR, the programmer indicates that the TM can ignore WAR conflicts, unless the conflicting read is followed by a write by the same transaction to the same variable. Recall that such reads indicate that the performing transaction is in its commit phase. It is reported in [13] that for STAMP benchmarks [3] annotated with conflict-defined transactions, the combination of relaxed detection of conflicts and appropriate hardware support is able to decrease the number of aborted transactions between 50% and 90% for up to 32-thread configurations, and consequently reduce execution time.

Despite this performance benefit, TM with relaxed detection of conflicts makes the verification of the program a challenge. The standard TM provides the atomicity guarantee to `Allocate` and `Free`, since no concurrent, conflicting accesses are allowed to happen during a transaction. That is, for every execution of the program in which operations `Allocate` and `Free` interleave with each other, there exists an equivalent execution containing the same calls to `Allocate` and `Free` but every call executing sequentially. Therefore, if `Allocate` and `Free` are treated transactions in a standard TM, one can verify **SBPoolinvariant** "sequentially", as if all calls to `Allocate` and `Free` are always run by a single thread. This enables one to carry out the reasoning using state-of-the-art program analysis and verification techniques originally developed for sequential programs.

However, TM with relaxed detection of conflicts does not provide this atomicity guarantee, since conflicting reads and writes from/to the `pool` array by different transactions are now allowed to be interleaved. The programmer should now ensure that such conflicting accesses do not cause harmful interference and lead the program to unintended states. Without proving the absence of such harmful interference, it is not sound to verify the program assuming always-atomic execution of `Allocate` and `Free`. Thus, the user has to apply techniques for concurrent programs. On the other hand, verification techniques for concurrent programs are fairly more complicated and tedious to use than those for sequential programs; further, model checking-based techniques scale poorly as the concurrent program gets larger and create more threads. Next, we present a technique that addresses this problem and allows user to still reason sequentially about the program.

2.4 Abstraction for sequential verification

We propose a two-step recipe, which is elaborated for our running example below:

1. Transform the original program P to a new program P_{Abs}, by (i) replacing reads from global variables with more abstract read operations and (ii) adding assertions to writes to the same global variables.
2. Verify "sequentially" the correctness of P_{Abs}, i.e., prove all the assertions in P_{Abs} including the ones added in 1.

Figure 1 and 2 show P and P_{Abs}, respectively, for our running example. If all the assertions in P_{Abs} are verified sequentially in Step 2, then P_{Abs} is a sound, sequential reduction of P, and the result of this verification can be safely carried to P. To conclude this, we have proved that if the assertions in P_{Abs} are all valid sequentially, then (i) every transaction of P_{Abs} is serializable—i.e., for every interleaved execution of P_{Abs}, there exist a serialized, equivalent execu-

[1] We assume that the execution is sequentially-consistent even without the use of locks, i.e., accessing shared variables without any synchronization does not cause harmful race conditions.

```
1  StringBuffer* Allocate() {
2    StringBuffer* ptr;
3    for (int i = 0; i < 1000; ++i) {
4      assert Abs(i, pool[i]);
5      havoc ptr;
6      assume Abs(i, ptr);
7      if (ptr != NULL) { // check if full
8          assume ptr == pool[i];
9          assert Abs(i, NULL);
10         pool[i] = NULL; // empty cell
11         return ptr;
12     }
13   }
14   // default operation
15   return new StringBuffer();
16 }
```

```
1  void Free(StringBuffer* buff) {
2    StringBuffer* ptr;
3    for (int i = 0; i < 1000; ++i) {
4      assert Abs(i, pool[i]);
5      havoc ptr;
6      assume Abs(i, ptr);
7      if (ptr == NULL) { // check if empty
8          assume ptr == pool[i];
9          assert Abs(i, buff);
10         pool[i] = buff; // fill cell
11         return;
12     }
13   }
14   // default operation
15   delete buff;
16 }
```

Figure 2. Abstraction of StringBuffer pool example. Abstraction predicate `Abs(i,v)` is `true` for all `i` and `v`.

tion of P_{Abs}— and (ii) P_{Abs} is a sound abstraction of P, thus all the assertions satisfied by P_{Abs} are also satisfied by P (in the concurrent context). In the following, we focus on the steps to reduce P to its abstract form P_{Abs}, but, due to lack of space, we omit explaining the sequential proof of P_{Abs}.

We note that the aim of this changes is not to obtain an executable program but a program on which static verification can be carried out. While implementing the nondeterministic read operations can be tricky, the `havoc` and `assume` statements are suitable and precise enough to express such operations for the purpose of static verification.

Abstraction predicate. Our key idea is to incorporate the effects of the conflicting writes that are ignored by the TM on the global reads of a transaction. We incorporate these effects into the code statically by rewriting these reads. For this, we require that the user provide, for each global variable x that is subject to conflicting accesses, a predicate Abs_x, we call *abstraction predicate*. Intuitively, the set Abs_x represents the set of all feasible values that may be written to x by any transaction during an execution.

For our example, the cells of the `pool` array are the shared variables, and the abstraction predicate for the `pool` array is: `Abs(i,v)` $\equiv$ `true`. This means that, the data structure works correctly (satisfies **SBPoolInvariant**) even though the reads from `pool[i]` in the read phase are later overwritten by other transactions with any other value, including `NULL`.

While our running example requires the abstraction predicate `true`, we found that for several other programs (including `Genome` and `Labyrinth` of STAMP), stronger abstraction predicates is necessary for correctness. For example, the linked list example in `Genome` does not satisfy the post-condition of the list-insertion operation, if all the reads from the "next" pointers while traversing the linked list are replaced by abstract reads that can read any node pointer. Thus, for the linked list example, we needed to use stronger abstraction predicates that dictate that the nondeterministically read node is reachable from the "head" of the list.

We transform program P to P_{Abs} performing the following changes in P for every global variable x.

1. Replace every read action $l = x$ with an action that nondeterministically chooses a value from Abs_x.
2. Insert before every write action $x = l$ an assertion that checks if the value of l satisfies Abs_x.

The first modification replaces reads from x with a more abstract version of the action, which, instead of reading the current value of x, is now free to read any value that satisfies Abs_x. Thus, the read from `pool[i]` at line 4 of Figure 1 is replaced by lines 4-6 in Figure 2. The assertion at line 4 ensures that our abstraction is sound—i.e., the abstraction predicate is already satisfied by the value the original read would return. Note that, during the verification of P_{Abs}, the user also need to verify all assertions in Figure 2 in order to perform the reasoning about P_{Abs} sequentially and apply this result to P. In our case, the assertion is trivially valid, as `Abs(i,v)` simply evaluates to `true`.

Lines 5-6 implement the abstracted read action. We express this abstraction in the notation of the Boogie language [2]. The `havoc` statement assigns a nondeterministic pointer value to `ptr` and the `assume` statement introduces an assumption that the nondeterministic value assigned by line 5 satisfies the abstraction predicate for `pool[i]`.

The second modification inserts an assertion before writes to global variables. In Figure 2, the assertion is inserted at line 9 to check that the write to `pool[i]` satisfies the abstraction predicate for the cell `i`. Similar to the ones at line 4, the assertion at line 9 is trivially valid, as `Abs(i,v)` directly evaluates to `true`.

While the full abstraction of reads in the read phase of our example are acceptable for correctness, we cannot prove **SBPoolInvariant** if we abstract the last successful read in the commit phase, because then, for example, the `Free` operation can see a full cell `pool[i]` as empty and overwrite

it with the value of `buff`. Thus, we should incorporate the fact that while the TM ignores WAR conflicts during the read phase of each transaction, it does not ignore such conflicts in the commit phase. We use the `assume` statement at line 8 of Figure 2 for this purpose. If, for example, a transaction T running `Allocate` detects that `pool[i]` is full (and is going to commit by returning the pointer in `pool[i]`) , no other transaction is allowed to write to `pool[i]` between T's read from `pool[i]` and its write to `pool[i]` at line 10. By using the assumption at line 8, we indicate that `ptr` contains the current (stable) value of `pool[i]`, but not a nondeterministically-chosen value; in other words, the abstraction at lines 5-6 has not been applied to the last read. We found (also while proving the other benchmarks in the STAMP suite) that not abstracting the read in the commit phase are essential for the proof, since the correct functionality of the code depends on the atomicity of the commit phase.

3. Related work

Closely related to our work, there is research on sequential verification of correctness criteria for transactional memories. Attiya et. al. [1] provide a set of reduction rules and types to verify concurrent modules to be serializable. They argue that by only considering sequential interleavings, one can verify that implementations that is based on locking such as two-phase locking, tree locking and hand-over-hand locking, thereby, using only sequential reasoning. They prove that a concurrent module based on these locking protocols is serializable. The reductions show that if every non-interleaved execution of the concurrent module satisfies the locking protocol, then all executions of the module satisfy the locking protocol. This work does not consider concurrency control with relaxed detection of conflicts.

Researchers have started to relax conflicts for gaining more performance, less number of aborts and improve scalability. Titos et. al. [13] suggest conflict-defined blocks and some language construct to realize custom conflict definition where some types of conflict are superfluous. Programmer specifies the types of conflict that exist in real. They show that user-defined conflict types causes less number of transaction aborts and improves scalability. While [13] was written by relying on the informal and intuitive deduction of the programmer about the safety of real conflict scenarios, our work provides evidence that programs using TM implementations with relaxed detection of conflicts can be reasoned systematically and in a sequential way requiring minimal manual effort.

Michael L. Scott [10] suggests a sequential specification to provide the semantics of transactional memories. When providing the semantics, the author uses conflict function to define when two transactions cannot both succeed. In addition, by defining arbitration function, the author specifies which of the conflicting transaction fails. The author argues that sequential specifications enable one to construct correctness proofs easily as well as formal comparison of TM implementations. The article also considers the progress issues for TM implementations. The author suggests under what conditions a TM implementation admits livelock-freedom, starvation and non-blocking.

References

[1] H. Attiya, G. Ramalingam, and N. Rinetzky. Sequential verification of serializability. *SIGPLAN Not.*, 45:31–42, Jan. 2010.

[2] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. *FMCO*, 2005.

[3] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proc. of The IEEE Int. Symposium on Workload Characterization*, Sep. 2008.

[4] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. Vcc: Contract-based modular verification of concurrent c. In *ICSE-Companion 2009.*, pages 429 –430, May 2009.

[5] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proc. 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 207–216, 2008.

[6] Gokcen Kestor, Roberto Gioiosa, Tim Harris, Osman S. Unsal, Adrian Cristal, Ibrahim Hur, and Mateo Valero. Stm2: A parallel stm for high performance simultaneous multithreading systems. In *2011 Int. Conference onParallel Architectures and Compilation Techniques (PACT)*, pages 221 –231, Oct. 2011.

[7] Shuvendu Lahiri and Shaz Qadeer. Back to the future: revisiting precise program verification using smt solvers. *SIGPLAN Not.*, 43:171–182, Jan. 2008.

[8] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[9] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *Proc. of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 68–78, New York, NY, USA, 2007.

[10] Michael L. Scott. Sequential specification of transactional memory semantics. In *ACM SIGPLAN Workshop on Transactional Computing*. Jun. 2006.

[11] Nir Shavit and Dan Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[12] Travis Skare and Christos Kozyrakis. Early release: Friend or foe? In *Workshop on Transactional Memory Workloads*. Jun 2006.

[13] R Titos, M. E. Acacio, J. M. Garca, T Harris, A Cristal, O Unsal, and M. Valero. Hardware transactional memory with software-defined conflicts. In *High-Performance and Embedded Architectures and Compilation (HiPEAC'2012)*, Jan. 2012.