Composable Partitioned Transactions *

Lingxiang Xiang   Michael L. Scott

Department of Computer Science, University of Rochester

{lxiang, scott}@cs.rochester.edu

1. Introduction

Twenty years after the initial proposal [4], hardware transactional memory is becoming commonplace. All commercial versions to date—and all that are likely to emerge in the near future—are *best effort* implementations: a transaction may abort and retry not only because of an actual data conflict with some concurrent transaction, but also because of limitations on the instructions that can be executed, the time that can be consumed, or the size or associativity of the space used to buffer speculative reads and writes.

As programmers begin to write programs with transactions—particularly large transactions—scaling problems will be inevitable, and we can expect growing demand for programming techniques that minimize transaction conflicts and hardware overflow. This abstract introduces one such technique. It exploits the common pattern in which (1) a transaction spends significant time "figuring out what it wants to update" before actually making the updates, and (2) the decision as to what to update can be checked for correctness more easily than it could be computed in the first place. A transaction that satisfies these properties can then be *partitioned* into a planning phase and an update phase: the planning phase determines what to update; the update phase double-checks the correctness of the plan and performs it. The planning phase can often be performed in ordinary code; the update phase remains a true transaction. Information is passed between the two in the form of a *validator object* that encapsulates a description of the desired update (the plan) and whatever information is needed to confirm its continued correctness.

Partitioned transactions (ParT) complicate the creation and use of concurrent objects, but not dramatically so. Consider a sorted linked list, where each operation begins by scanning the entire sublist from the head to the neighborhood of a given key. In a naive implementation, a transaction will abort if another thread updates any element of the already-scanned sublist. If every element indicates the list to which it belongs, however, then the neighborhood of a key is self evident once found, and we can effect a dramatic reduction in conflicts by separating the scan from the update. Transactional calls to L.insert(x), for example, would be replaced by L.insert_plan(&v, x); L.insert_update(v). With a bit of care, the planning phase can be performed in ordinary (nontransactional) code. It initializes the validator object v, which contains both x and a pointer to a node in the to-be-modified neighborhood. The update phase (a transaction) subsequently confirms that the pointer still indicates the right place in the right list, and then makes the appropriate changes.

In a more computational vein, consider a transaction that sets y = f(a, b, c); x = y, where a, b, c, and x are shared variables, and f is a time-consuming function. We can significantly reduce the time in which the transaction is subject to conflict by computing y = f(a, b, c) nontransactionally; passing a validator containing a, b, c, and y; and making the update transaction the equivalent of an n-word compare_and_swap. If it is essential to the correctness of f that a, b, and c be consistent, we can put the three reads in a transaction.

For common operations, partitioning can lead to dramatically lower abort rates and higher scalability. We have confirmed these benefits on a 16-core TM-capable IBM zEnterprise EC12 mainframe and a 4-core Intel Haswell machine. Our experience further indicates that a key advantage of transactional synchronization—namely, composability—continues (with certain caveats) to hold for partitioned transactions.

Unlike *early release* [5], *elastic transactions* [1] and *composing relaxed transactions* [3], partitioned transactions are compatible with existing HTM. They induce a model in which the programmer highlights the data that a transaction needs, rather than the data that it doesn't. Mechanically, partitioned transactions resemble the *split hardware transactions* of Lev et al. [8], but instead of a low-level mechanism to accommodate nested transactions, they offer a high-level (program-specific) technique to reduce transaction aborts. For our part, we view partitioned transactions as a follow-on to manual speculation [12]. Where that work sought to reduce critical path length in programs based on mutual exclusion, the current work aims to increase success rates in programs based on speculation.

2. Partitioning

Plan-then-update operations, which cluster most of their writes toward the end of the operation, are ubiquitous in multithreaded programs, and can be expected to comprise many of the transactions on an HTM-capable machine. *Partitioned Transactions(ParT)* exploit this programming pattern by observing that the planning phase need not be executed in the same transaction as the update, so long as the plan is still valid when the update occurs.

2.1 The Programming Model

A partitioned transaction comprises three components:

(1) a **Planning** method that "figures out what to do" and embeds the result in a *validator* object. The plan may execute either in ordinary (nontransactional) code—as in manual speculation [12]—or in an HTM transaction. In the former case, the programmer must assume responsibility for issues arising from potential data races, stale pointers, or ABA problems. Since plans are speculative, and may fail or be executed multiple times, they must not change the (high-level, semantic) value of shared data.

(2) a **Validation** method for the validator object that confirms the continued efficacy of the plan that the object embodies. The choice of validation strategy is flexible and algorithm-dependent; we describe several common patterns in Section 2.2.

(3) an **Update** method, executed as a transaction, that effects the actual changes to shared data. To ensure the atomicity of the overall operation, the update phase begins by calling the validation method. If validation fails, the update phase must abandon the plan and switch to a fallback code path.

Consider the general search-then-modify operation in Figure 1a, in which the goal of search is to locate the position (pos) at which to make a modification. Instead of placing the whole operation in a single transaction, we can partition the transaction as shown in Figure 1b. The planning phase (search_plan) places position information (pos) in validator object v, which has an is_valid method to confirm whether pos is still the right place for modification. If validation succeeds in the update phase (atomic region), pos is modified (line 9); otherwise, a fallback path is taken (line 11).

* This work was supported in part by an IBM Canada CAS Fellowship and by the US National Science Foundation under grants CCR-0963759, CCF-1116055, and CNS-1116109.

```
1  atomic {
2    pos = search()
3    modify(pos)
4  }
```

(a) Original transaction

```
5  validator v;
6  search_plan(&v)
7  atomic {
8    if (v.is_valid())
9      modify(v.pos)
10   else
11     ... // fallback path
12 }
```

(b) ParT

Figure 1: A transaction (a) and its ParT equivalent (b).

As long as is_valid is correctly implemented, the code in Figure 1b is semantically equivalent to the original version: by returning true at line 8, is_valid asserts that pos is the position that would be found by a fresh call to search.

2.2 High-level Validation

Several validation strategies are possible in ParT. Perhaps the most obvious is to double-check the values of all data read in the planning phase. This is essentially the strategy of *Split Hardware Transactions* [8]. Conceptually, it transfers the whole read set of the planning phase over to the update transaction. Consistency is trivially ensured at the implementation level. For compute-heavy planning phases, this strategy may be entirely appropriate: for these, the main purpose of ParT is to reduce the temporal window in which the transaction is vulnerable to spurious (e.g., false sharing) conflicts. Unfortunately, for search-heavy planning phases, passing the full read set may make the update transaction as likely to abort as the original monolithic transaction.

Complete consistency of all read locations is not always required for high-level correctness, however: the data to be modified may depend on only a portion of the read set, or on some property that does not always change when the values of the data change. In search structures, for example, it is often possible to verify, locally, that a location is the right place to make an update: how one found the location becomes immaterial once it has been found. Local verification may even succeed when local values have changed. Again in a sorted list, if the planning phase suggests inserting key k after node X, and passes a pointer to X in the validator object, the update phase may succeed even if another node has already been inserted after X, so long as k still belongs between the two.

In other cases, application-specific version numbers can be used to concisely capture the status of natural portions of a data structure. We found such numbers effective, at various granularities, in our work on manual speculation [12]. Seen in a certain light, application-specific version numbers resemble the ownership records (ORecs) of a software TM system. They protect the entire read set, but allow rapid validation by employing a data-to-Orec mapping specifically optimized for each individual structure.

2.3 Applying ParT to HTM

Partitioned Transactions allow us to overcome the limitations of best-effort HTM in important situations. To realize this potential, the following issues and trade-offs must be considered.

Software vs. hardware speculation. The planning phase of ParT can be done either in software or in a hardware transaction. Doing it in software is often preferable, as it avoids placing the touched data in HTM's read set, sidestepping HTM's capacity limits and thereby reducing aborts. However, pure software-based planning requires expert program knowledge [12] to handle potential data races between nontransactional planning and transactional updates, erroneous behaviors such as infinite loops, and ABA problems.

If these problems prove too daunting, planning can be placed in its own hardware transaction. Doing so sacrifices certain benefits, such as unbounded space and avoided conflicts on no-longer-needed data, but programming becomes quite a bit simpler, and given an appropriate validator, performance may still be significantly better than it was with the original, monolithic transactions. Once transactional planning has been summarized by a validator, conflicts on much of the read set may no longer be a threat to the success of the update transaction. If the validator has a low probability of failure, an aborted update transaction may even be able to retry without re-running the planning phase.

```
1  atomic {
2    operation0
3    operation1
4    ...
5    operationN
6  }
```

(a) One big transaction

```
7  op0_plan(&v0)
8  ...
9  opN_plan(&vN)
10 atomic {
11   if (v0.is_valid())
12     op0_update(v0)
13   else
14     ... // fallback path 0
15   ...
16   if (vN.is_valid())
17     opN_update(vN)
18   else
19     ... // fallback path N
20 }
```

(b) CParT transaction

Figure 2: Composing speculation work for a big transaction through Composable Partitioned Transactions.

Validation granularity. Fine-grain, local validation minimizes the chance of spurious aborts. Coarse-grain (e.g., version-number-based) validation may execute more quickly, because it has less to check. Application-specific knowledge and experimentation may be required to find the best point in the design space.

Fallback strategy. When validation fails, two kinds of fallback are possible. One immediately terminates the update transaction and jumps back to the planning phase, in the hope that a retry is likely to succeed; we call this *aggressive fallback*. The other executes the entire operation—both the planning and update phases—within current transaction; we call this *conservative fallback*. If failed validation is uncommon, the aggressive strategy is probably better: the second try is likely to succeed, and to benefit from partitioning.

3. Composition

Transactions in real applications often comprise multiple operations, which need to *compose* with one another. Most previous efforts[2, 10] to optimize composable operations are incompatible with current HTM. ParT *does* work with HTM, and (with certain caveats) supports composability.

Consider a transaction that needs to perform N operations, in some order. For the moment, suppose that no operation reads locations written by a predecessor. The obvious way to achieve atomic execution is to enclose these operations in a big hardware transaction (Figure 2a).

When executing on current best-effort HTMs, such a big transaction may suffer from several problems. Though there may be sufficient hardware resources to finish each individual operation, there may not be enough to accommodate the read and write sets of all the operations together. The long execution window also increases the likelihood or aborting due to external events such as context switches and interrupts. Even on an unbounded system like VTM[11], the length of the transaction would likely increase the rate of conflict with other threads.

Given a big transaction containing multiple operations, each of which has a ParT implementation, we can extract and combine the planning phases into a single planning phase, followed by a

single update phase. We call this idiom *Composable Partitioned Transactions* (CParT). To pass information from the planning phase to the update phase, we employ a set of validators, one for each operation. Returning to the example of Figure 2a, we can use ParT implementations of the constituent operations to construct the code of Figure 2b, which contains a composed planning phase (lines 7–9) and a composed update transaction (lines 10–20). Every validator in the update transaction is verified and backed by a separate fallback path.

3.1 Benefits

Shrunken read/write set. If the read sets of N transactions are (mostly) disjoint, the read set of their naive composition will be N times the average size M of any one of them alone. In CParT, with application-specific (typically local) validation, we can often reduce M to some small constant c, leading to a composed footprint that is linear in N. Since M often varies with input, while c does not, the read set size in a CParT program may also be more predictable, making it easier to model and analyze performance.

Lower overall conflict rate. The smaller memory footprint and shorter temporal duration of a ParT transaction, relative to the monolithic transaction it replaces, can provide a significant reduction in the likelihood of conflict. This benefit is compounded by composition. A monolithic transaction T that includes operation O is vulnerable to abort if any other thread makes a conflicting access to O's data during any part of T. In CParT, O's planning phase (if executed in a transaction) can abort and retry independently of planning for the rest of T. The "shared vulnerability" of O and its peer operations is limited to the update transaction, which is typically much shorter than the original version of T. Moreover, if an update transaction aborts due to an external condition, CParT can retry it without re-executing the planning phase.

Fast re-execution. When validation of the plan for operation O fails within the CParT update transaction for composed operation T, we can generally arrange, in the fallback path, to re-execute only the planning phase of O before trying the update again. Shorter turn-around times in turn are likely to increase throughput. A simple example can be seen in transactions that end with a reduction (e.g., the update of a global sum). Unlike a monolithic composed transaction, a CParT update transaction that aborts due to conflict on a reduction variable can generally salvage the planning phases of everything else in the transaction. In effect, CParT allows us to separate time-consuming low-contention work from any fast, high-contention work that follows it, while still maintaining atomicity.

3.2 Dependences

Independent operations compose trivially: if operations A and B share no data, then A_{update} and B_{plan} will safely commute with one another, and (A_{plan} A_{update}) (B_{plan} B_{update}) will be equivalent to (A_{plan} B_{plan}) (A_{update} B_{update}). Composition is slightly more difficult if B_{plan} depends on the results of A_{plan}: here we must write as A_{plan}(&v); B_{plan}(v.results); atomic $\{A_{update}\ B_{update}\}$.

The complicated case arises when B_{plan} depends on A_{update}. One possible strategy is to require A's validator to capture the intended new state of A's object, in a form amenable to querying by B_{plan}. The principal disadvantage here is the need to instrument any reads in B_{plan} that might need to see the new state—arguably a violation of abstraction. Alternatively, we might provide an undo operation for A_{update}, and then replace B_{plan} with atomic $\{A_{update}\ B_{plan}\ A_{undo}\}$. This strategy avoids the need for query operations inside B_{plan}, but it involves both extra coding effort and extra run-time overhead. While neither of these strategies is perfect, we note that CParT is already a manual programming technique, so a certain amount of programmer effort in the interest of composition may be acceptable.

3.3 Contention Management

3.3.1 Fallback Strategy

Every validation in an update transaction needs a fallback path in case of failure. When A_v fails (the is_valid method returns false), the aggressive strategy aborts the update transaction and returns to A_{plan}, in the hope that partitioning will work the next time around. The simpler, conservative strategy performs A_{plan} A_{update} in place, within the update transaction (preserving both earlier plans and earlier updates). This strategy works well if most operations are independent of one other.

If there exists a long dependence chain, where each plan depends on the output of one or more predecessors, a failed validation may negate all subsequent planning work. Under a conservative strategy, all plans in the chain will have to be re-executed inside the update transaction, and no benefit will be obtained from CParT. To preserve as much work as possible, a mixed strategy can be adopted: for operations close to the front of the chain (the beginning of the update transaction), aggressive fallback is used to restart a fresh planning phase, since little work is wasted by the abort; for operations close to the tail of the chain, as the work lost to an abort increases , conservative fallback is used instead.

3.3.2 Planning Transactions

We may execute the planning phase(s) of one or more CParT operations in a hardware transaction in order to avoid the complexity of software sandboxing (inconsistency tolerance). With the aid of undo operations, we may also execute update, planning, and undo operations together in a transaction to accommodate dependences. Unlike an update transaction, a planning transaction should make no semantically visible changes to shared data. In fact, it need not even commit in some cases: an incorrectly initialized validator will simply lead the update transaction down a fallback path.

The software surrounding most HTM systems falls back to a global lock in response to repeated transaction aborts. Once a thread acquires the lock, its operation becomes *irrevocable*. To ensure the atomicity of irrevocable operations, hardware transactions typically add the lock to their read set, forcing them to abort if a peer acquires the lock. Given the expense of the aborts and of subsequent serialization, a small planning transaction may simply choose to "give up" after a certain number of failures. That is, rather than abort all current hardware transactions to become irrevocable, it can defer to the fallback path of its update phase. The choice between the irrevocable and "give up" options could again be made dynamically, based on run-time statistics.

4. Experiments

4.1 Platform

We have evaluated ParT on a TM-capable IBM zEnterprise EC12 mainframe server, in a virtual machine with 16 dedicated cores [6].

The TM run-time library uses global-lock-based irrevocable execution as a software fallback. When a transaction aborts for a reason the hardware deems "non-persistent," we retry up to 8 times before switching to irrevocable mode. On "probably persistent" aborts, we retry no more than 5 times. Planning transactions are handled in a similar way, except that the whole transaction is skipped after a fixed number of retries for the reason described in Section 3.3.2.

4.2 Microbenchmarks

For controlled assessment of basic overheads, we evaluated ParT on sorted doubly-linked lists and red-black trees.

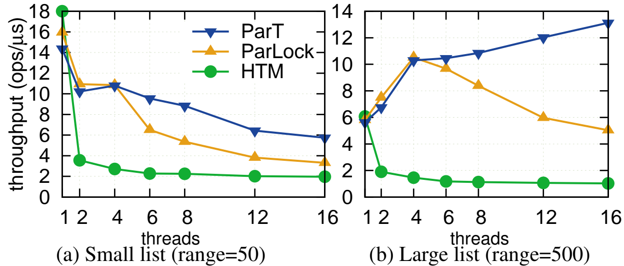

Figure 3: Scalability of different DList implementations with (a) small and (b) large key range on z. 50% insert, 50% remove.

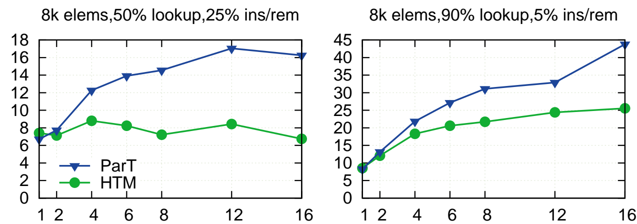

Figure 4: Throughput of red-black tree. The Y axis is the number of operations per μs.

In the list microbenchmark, the planning phases of insert and remove operations search speculatively for the position at which the key would belong, and save that position (a pointer to the last node whose key is less than the given one) in the validator. Performance results for different implementations appear in Figure 3. The three curves represent pure HTM, ParT, and a partitioned implementation in which the update phase comprises a lock-based critical section rather than a transaction (ParLock). With 4 or fewer threads, which run within a single processor, ParLock and ParT have similar performance. As soon as threads have to synchronize across chips, however, the lock can no longer compete. Pure HTM, for its part, ends up falling back to a global lock for many of its transactions, and doesn't do well at all.

In the red-black tree, insert and remove operations start with a tree search. To avoid the possibility that a speculative search will find itself on the wrong branch of the tree due to a rotation, we perform the planning phase in a transaction of its own. At the end of tree search, we keep in the validator a pointer to the last accessed node. Space limitations preclude a detailed explanation, but local validation based on this pointer is trickier than in the linked-list case, and can fail when it "ought" to succeed about 25% of the time, forcing execution down the fallback path. As shown in Figure 4, these "false negatives" cause ParT to lag slightly behind the baseline HTM in single-thread runs. It wins in parallel runs. Generally, the benefit of ParT increases with the height of tree (due to fewer data conflicts on the validator) and with the percentage of update operations (due to faster re-execution of aborted transactions).

4.3 Macrobenchmarks

To understand the potential benefit of ParT and CParT in larger applications, we chose four macrobenchmarks from the STAMP [9] and RMS-TM [7] benchmark suites. Genome, Intruder, and Vacation all contain composite transactions that we were able to recast with CParT. UtilityMine is amenable to optimization with ParT, but has no opportunities for composition. In effecting our transformations, we have taken care to preserve both the data structure layout and the algorithms of the original code base.

A summary of performance results can be found in Figure 5. In both Genome and Vacation, CParT results in significantly fewer

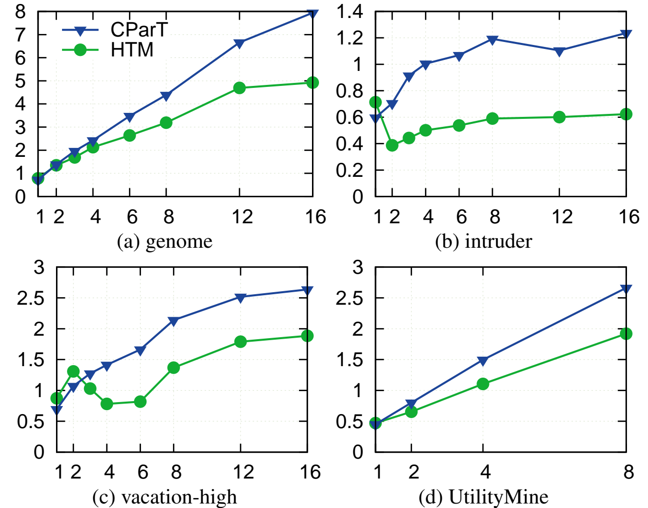

Figure 5: Performance results of TM macro-benchmarks on IBM zEC12. The Y axis is speedup over the sequential version.

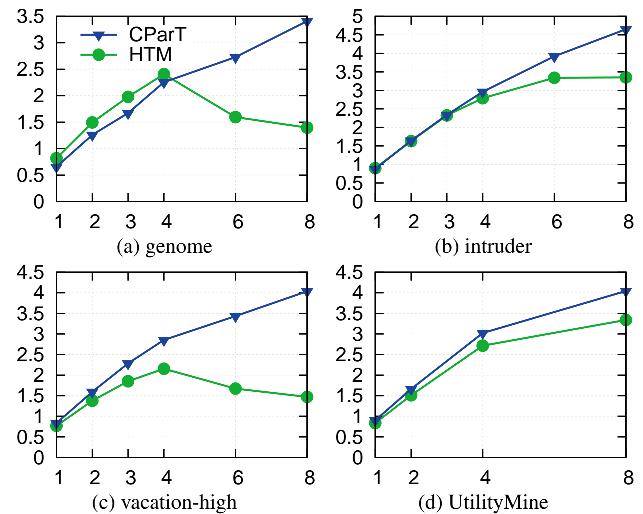

Figure 6: Performance results on Intel Core i7-4770.

hardware overflow aborts. In all four applications, ParT / CParT results in significantly fewer conflict aborts. Additional benefits results from the planning phase's tendency to warm up the cache for the update phase, which can then be even shorter (and thus less likely to suffer conflict aborts). Similarly, Figure 6 illustrates CParT's performance advantage over the baseline HTM on a 4-core Intel Haswell machine.

5. Conclusions

Based on our experiments to date, ParT and CParT seem likely to be valuable additions to the "TM programmer's toolkit." Our macrobenchmarks in particular suggest the possibility of dramatic improvements in throughput—at least for some applications—on state-of-the-art HTM. In future work, we are exploring a variety of issues, including integration with software and hybrid TM; compiler support for nontransactional planning phases, in the style of CSpec [12]; and dynamic choice of fallback strategies based on run-time statistics.

References

[1] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *Proc. of the 23rd Intl. Conf. on Distributed Computing (DISC)*, pages 93–107, Elche/Elx, Spain, Sep. 2009.

[2] G. Golan-Gueta, G. Ramalingam, M. Sagiv, and E. Yahav. Concurrent libraries with foresight. In *Proc. of the 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 263–274, Seattle, WA, June 2013.

[3] V. Gramoli, R. Guerraoui, and M. Letia. Composing relaxed transactions. In *2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1171–1182, 2013.

[4] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of the 20th Intl. Symp. on Computer Architecture (ISCA)*, pages 289–300, San Diego, CA, May 1993.

[5] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proc. of the 22nd ACM Symp. on Principles of Distributed Computing (PODC)*, pages 92–101, Boston, MA, July 2003.

[6] C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for IBM System z. In *Proc. of the 45th Intl. Symp. on Microarchitecture (MICRO)*, pages 25–36, Vancouver, BC, Canada, Dec. 2012.

[7] G. Kestor, V. Karakostas, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. RMS-TM: a comprehensive benchmark suite for transactional memory systems. In *Proc. of the 2nd Intl. Conf. on Performance Engineering*, pages 335–346, Karlsruhe, Germany, Mar. 2011.

[8] Y. Lev and J.-W. Maessen. Split hardware transactions: True nesting of transactions using best-effort hardware transactional memory. In *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 197–206, Salt Lake City, UT, Feb. 2008.

[9] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IEEE Intl. Symp. on Workload Characterization (IISWC)*, pages 35–46, Seattle, WA, Sep. 2008.

[10] J. E. B. Moss and A. L. Hosking. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming*, 63(2):186–201, Dec. 2006.

[11] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proc. of the 32nd Intl. Symp. on Computer Architecture (ISCA)*, pages 494–505, Washington, DC, 2005.

[12] L. Xiang and M. L. Scott. Compiler aided manual speculation for high performance concurrent data structures. In *Proc. of the 18th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 47–56, Shenzhen, China, Feb. 2013.