On Extending TM Primitives using Low Level Semantics

Mohamed M. Saad, Roberto Palmieri, Ahmed Hassan, and Binoy Ravindran

{msaad, robertop, hassan84, binoy}@vt.edu

ECE Dept., Virginia Tech, Blacksburg, VA 24061, US

Abstract

Transactional memory has recently emerged as an optimistic concurrency control technique that isolates concurrent executions at the level of reads and writes, and therefore provides an easy programming interface. However, such an isolation could be conservative from the application level perspective. In this work, we propose an extension to the classical TM primitives (read and write) to infer more semantics from the program code while maintaining the same level of abstraction at the programmers side. Practically, our proposed extension can be implemented either as a complete compiler pass, which maximizes programmability and allows backward compatibility, or as an extension to TM APIs to be exploited by expert programmers. We deployed this extension on two state-of-the-art STM algorithms and showed a speedup up to 4× better on different applications.

Categories and Subject Descriptors D.1.3 [*Software*]: Concurrent Programming; H.2.4 [*Systems*]: Transaction processing

Keywords Transactional Memory, Low Level Semantics

1. Introduction

Transactional Memory (TM) is a programming abstraction for accessing shared memory data without exposing any lock interface to the application so that implementation difficulties and drawbacks like deadlock, livelock, or priority inversion are prevented. With TM, programmers organize blocks of code that access shared memory addresses as atomic sections (or *transactions*), in which reads and writes appear to take effect instantaneously. As a common pattern, each transaction maintains its own read-set and write-set to detect conflicts with other concurrent transactions. When that happens, a contention manager [31] resolves the conflict by aborting all but one of the transactions and allowing the remaining one to proceed to commit, yielding (the illusion of) atomicity. TM was originally proposed in hardware (called HTM [19]), later in software (called STM [32]), and subsequently in a hardware/software combination (called HybridTM [20]).

In order for a TM implementation to be generic, conflicts are usually detected at the level of memory addresses. The TM abstraction can be expressed using four instructions: `TM_BEGIN`, `TM_END`, `TM_READ`, and `TM_WRITE`. The first two identify the transaction boundaries while the last two define the barriers for every memory read and write that occurs within those boundaries. TM algorithms differ in the way those instructions are implemented. Although some frameworks may add other features (such as allowing external aborts, non-transactional reads and writes, or irrevocable operations), the above four instructions are used to form the body of most TM solutions.

Despite TM's high programmability and generality, its performance is still not as good as (or better than) optimized manual implementations of synchronization. To overcome that, researchers investigated various approaches with different design choices. Regarding STM, they mostly varied the internal granularity of locking and/or validation of accessed memory addresses. Examples of those solutions include coarse-grained mutual exclusion of commit phases, as used in NOrec [8]; compact bloom filters [4] to track accesses, as used in RingSTM [33]; and fine-grained ownership records, as used in TL2 [11]. For HTM, given that current HTM processors are classified as "best effort" because transactions are not guaranteed to progress in HTM (even if they are executed alone without any actual concurrency), an efficient software "fallback" path is needed (i.e., hybrid TM) when hardware transactions repeatedly fail [12]. Recent literature proposes many compelling solutions that make the fallback path fast under different conditions [6, 7, 12, 23, 28].

The key commonality of all the aforementioned approaches is that they do not challenge the main objective of TM itself, which is providing generality at the application level. This is also the reason why those smart and advanced solutions still retain some of the fundamental inefficiency of TM. On the other hand, providing high performance in multi-threaded applications before the advent of TM (i.e., when thread synchronization was manually done using fine-grained locks and/or lock-free designs) closely depended upon the specific application semantics. For example, identifying the critical sections and the best number of locks to use are design choices that can be made only after deeply knowing the semantics of the application itself (i.e., what the application does).

A related question that arises in this regard is: *Is there some room for including semantics in TM frameworks without sacrificing their generality?* If the answer is "yes", which is what we claim and assess in this paper, then we will finally be able to overcome one of the main obstacles that has existed alongside TM since its early stages and accordingly boost its performance. Recent literature provides a few semantic-based concurrency controls, which will be detailed in Section 2. However, they either solve specific application patterns [30], break the generality of TM [13, 18], or are orthogonal and not integrable with TM [16, 17].

Motivated by the above question, we propose TM extensions that include semantics into existing and well-known TM frameworks while preserving the same generality as the original TM proposal. To accomplish this goal, we first aim at identifying which semantics can be included without impacting the generality of the TM abstraction (we name these semantics as *TM-friendly*); then we show how to modify TM algorithms to allow such semantic-based operations.

By TM-friendly semantics, we mean semantic optimizations that can be potentially decoupled from the application layer. Specifically, this paper focuses on optimizations on conditional operators (e.g., $>$, $<$, $\neq$), bitwise operators (e.g., &, |), and increments/decrements (e.g., $++$, $--$), which are commonly used in legacy applications. In practice, those semantics can be integrated into TM frameworks in two different ways, both of them having a minimal effect on TM programmability. The first way is to implement them entirely as *compiler passes*, thus providing backwards-compatibility with existing applications. Alternatively, they can be implemented as new TM interfaces that are translated by the TM framework (or the compiler), which gives conscious programmers an opportunity to better exploit the new interfaces while developing concurrent applications. For the purpose of this paper, we implemented them as TM interfaces, and we leave full integration with compilers as a future work. More details about those semantics are presented in Section 3.

To develop TM-friendly semantics into existing state-of-the-art STM (and HTM algorithms), we start by classifying them according to the technique used for validating execution. STM algorithms can be roughly classified as either *version-based*, where each memory location keeps a version number that is used to identify memory changes, or *value-based*, where the content of each location is itself leveraged to detect memory modifications. For value-based algorithms, we propose *semantic-validation* as a generalization of value-based validation, allowing TM frameworks to define a specific validator for the semantic-based instructions. For version-based approaches, we propose a methodology for adapting them to allow a hybrid (i.e., version/semantic) validation mechanism. In practice, in Section 4.1, we show how to modify NOrec (as an example of the former) and TL2 (as an example of the latter) to include semantics.

Modifying HTM-based algorithms is harder because compilers do not have full control over transaction execution given that it happens entirely in hardware and current hardware gives very limited control over HTM transactions [5, 27]. To circumvent that, we propose (in Section 4.2) the design of two orthogonal approaches that enable semantics; namely, injecting semantics into the software fallback path and modifying the order of instructions in TM blocks at compilation time to allow semantic enhancements. The details of these approaches are left as future work.

We implemented our *semantic-enabled* TM algorithms and integrated them into RSTM [21] framework. Integration with compilers (e.g., GCC, LLVM) is kept as future work. In Section 5, we evaluated those algorithms on the following applications: Bank, a benchmark that simulates a multithreaded application where the threads mostly perform money transfers; LRU-Cache, a benchmark that simulates a software cache with a least-recently-used replacement policy; a hash-table benchmark; and the STAMP [25] benchmark suite. Results are very promising: enabling semantics for both NOrec and TL2 lead to a throughput improvement of up to 4$\times$.

2. Related Work

Not surprisingly, the trials to include semantics in TM started in literature as early as TM itself. In fact, the potential objective of the first TM proposal, as it can be easily inferred from the title of the first TM paper [19], was providing an architectural support for lock-free data structures. However, since the first and all the subsequent approaches were general because they aimed at improving programmability, their performance could not compete with handcrafted (i.e., very optimized) fine-grained and lock-free designs.

In the last decade, before the advent of commodity HTM processors, STM dominated the literature, and involving semantics to improve performance was an important topic, which has been addressed by approaches such as open nested transactions [26], elastic transactions [13], and early release [18]. The main downside of all those attempts is that they move the entire burden of providing optimizations to the programmer, and propose a modified framework to accept those programmer modifications. Since TM has been mainly proposed to make concurrency control as much transparent as possible from the programmers standpoint, the practical adoption of the above approaches remained limited. Flexibility deteriorated further after the release of processors with HTM support because the approaches proposed for STM could not work anymore given the limited APIs and control provided by HTM. The innovations presented in this paper overcome those issues by providing solutions that preserve the generality of TM, do not give up optimizations and semantics, and cope with the current state-of-the-art of TM in both software and hardware.

Another recent research direction focused on developing collections of transactional blocks (essentially data structures) that perform better than the corresponding "naive" TM-based counterparts (i.e., when the sequential specification of a data structure is made concurrent using TM). Methodologies like transactional boosting [16, 17], consistency oblivious programming [1, 2], semantic locking [14], and partitioned transactions [35] are examples of that direction. Despite the promising results, those approaches remain isolated from TM as a synchronization abstraction and appear as standalone components that are not integrable with generic (and existing) TM frameworks.

Involving compilers in TM's concurrency control is currently becoming mandatory given the enhanced GCC release [34], which includes TM support. However, to the best of our knowledge, very few works addressed the issue of detecting TM-friendly semantics at compilation time similar to what we propose in this paper. Among them, one recent approach proposes a new read-modify-write instruction to handle some programming patterns in TM [30]. However, those approaches still address specific execution patterns and do not generalize the problem like what we attempt to do in this paper, which rather pushes more in the direction of abstracting the problem and providing a comprehensive solution to inject semantics into existing TM frameworks.

3. TM-Friendly API

In this section we overview the proposed semantics that can be injected into TM frameworks without hampering the generality of TM itself. Then, in the following section, we show our proposed design on how to inject such semantics in practice.

As sketched before, TM defines two language/library constructs for reading (`TM_READ`) and writing (`TM_WRITE`) memory addresses. These constructs enforce a conservative conflict detection between concurrent transactions (recall that two transactions conflict if they access the same address and one access is a write).

As an exemplifying example, Algorithm 1 illustrates a code snippet that produces conflicting transactions. When T_1 executes its first line, existing TM algorithms save the value of x and y locally. Starting from this point, for the TM algorithm to preserve consistency, any concurrent change in x and y forces T_1 to abort. This abort can be triggered during the validation of T_1's next read (e.g., in NOrec), when T_1 tries to commit (e.g., in TL2), or immediately (e.g., in Intel HTM processors). In that specific example, since T_2 writes to x and y and commits before T_1 reaches its commit phase, most TM implementations force T_1 to abort. However, at the semantic level T_1 has no real issue and can safely commit since the boolean result of the conditional expression still holds,

Algorithm 1 Two transaction conflicting at the memory level but not at the semantic level.

```
Initially x = y = 5

TM_BEGIN($T_1$)
if x > 0 || y > 0 then
    // Do some reads and writes ...
                                        TM_BEGIN($T_2$)
                                        x++
                                        y- -
                                        TM_END
end if
TM_END
```

which means that the conflict triggered by the TM framework is a "false conflict" at the semantic level.

`TM_GT(address, value | address)`	tx greater than
`TM_GTE(address, value | address)`	tx greater than or equals
`TM_LT(address, value | address)`	tx less than
`TM_LTE(address, value | address)`	tx less than or equals
`TM_EQ(address, value | address)`	tx equals
`TM_NEQ(address, value | address)`	tx not equals
`TM_INC(address, {+/-}value)`	tx increment/decrement
`TM_AND(address, value | address)`	tx bitwise and
`TM_OR(address, value | address)`	tx bitwise or

Table 1: Extended TM Constructs.

Examples like the above motivated us to design extensions to the traditional transactional constructs that enrich the TM programming model. Those constructs are classified according to their semantic level and are summarized in Table 1. The first category includes *conditional operators*, which take two operands and return a boolean state of the conditional expression. The operands in this category can be two addresses or an address and a value. At the memory level, a traditional execution of those constructs inside transactions implies calling one or two `TM_READ` constructs (depending on the operands). Alternatively, using our constructs, we consider the whole expression as one semantic operation and the safety of the enclosing transaction is preserved by only validating that the return value of the condition remains the same until the transaction commits. The second category includes increment/decrement operations, which take an address and an offset. A traditional handling of those operations, unlike the first category, include both `TM_READ` and `TM_WRITE`. Consequently, as we show in detail in Section 4, handling those operations semantically should consider the write operation and its possible propagation to the next operations in the transaction. The third category includes bitwise operations. Currently, we optimize only bitwise operations inside conditional expressions. For example, in the conditional statement `if(x & 1 == 0)`, instead of reading `x` transactionally using `TM_READ`, we read `x` non-transactionally, apply the bitwise operation, and check whether the condition is `true` or `false`. Accordingly, this category is handled similarly to the first category in our framework. As a future work, we plan to include bitwise operations in assignment statements as well (e.g., `x |= 1`), which requires a different handler that considers the writing part of the operation.

Summarizing, using our new constructs, a transaction is able to capture application semantics more by treating comparison operators, bitwise operations, and increments (or decrements) differently. Including those semantic operations in TM frameworks is appealing for two reasons. First, they are commonly used in applications, as we show in later examples. Second, including them in TM frameworks can be done entirely at compilation time, where the compiler can detect and/or translate those "semantic operators". As a result, programmers are not required to make significant changes to their code, which does not affect generality of TM by any means.

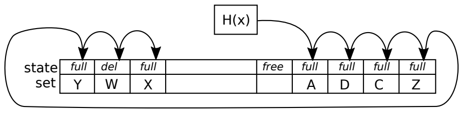


Figure 1: Probing a hash table with open addressing

An interesting feature of the semantic operations listed in Table 1 is that they compose. For example, the scenario shown in Algorithm 1 can be further enhanced if we consider the whole conditional expression (i.e., `TM_READ(x) > 0 || TM_READ(y) > 0`) as one semantic read operation in order to avoid further false conflicts. In this example, if the condition was initially `true` and then a concurrent transaction modifies only one variable (i.e., either x or y) to be negative in that expression, considering the clause as a whole avoids aborting T_1 given the `OR` operator. This enhancement can also be entirely accomplished by the framework without any need to modify the application.

3.1 TM-friendly semantics in action

Now we show examples from real benchmarks and applications that can be enhanced by our semantic-based TM proposal. These examples are clearly not exhaustive, but they are representative of programming patterns that are used in concurrent programming.

Hashtable with Open Addressing. Operations in such a hash table usually start by probing the table in order to find a matching index for a given hash value. Figure 1 depicts an example of this probing. This function can be enhanced by our approach because it consists of a chain of conditional expressions that check specific semantics and do not impose certain values of `state` or `set` (e.g., it may only require the checked cells to be not free and either flagged as removed or having a different value from the hashed one). On the other hand, when using the classical read/write TM constructs, many concurrent changes to the accessed cells (e.g., deleting 'D' and inserting 'B' instead) will abort the probing transaction. Considering semantics through our proposed extensions avoids such aborts. Algorithm 2 depicts pseudocode of the probing method and how we modify it.

Queues. Any efficient concurrent queue implementation should let an `enqueue` operation execute concurrently with a `dequeue` operation if the queue is not empty. However, this case is not allowed using traditional TM constructs because the `dequeue` operation compares the head with the tail in order to detect the special case of an empty queue. Algorithm 3 shows how we re-enable this level of concurrency in an array-based queue using our constructs.

Algorithm 3 Using the semantic constructs to enhance dequeue operation.

```
TM_BEGIN
                              ▷ Using our constructs: If (TM_EQ(head, tail))
if TM_READ(head) != TM_READ(tail) then
    return false;
end if
item = array[TM_READ(head) % array_size];
                              ▷ Using our constructs: TM_INC(head, 1);
TM_WRITE(head, TM_READ(head) + 1)
return true;
TM_END
```

Vacation. This application is one of the STAMP [25] benchmark applications that simulates a travel reservation system. The workload consists of clients' reservations; each client uses a coarse-grained transaction to execute its session. Vacation has two main operation profiles: making a reservation and updating offers (e.g., price changes). Although the reservation profile checks the common attributes of the offer (e.g., the number of free slots and the

Algorithm 2 Using the semantic constructs to enhance hash table probing.

```
TM_BEGIN
        ▷ Using our constructs: while (TM_EQ(state[index], DELETED) || (TM_NEQ(state[index], EMPTY) && TM_NEQ(set[index], value))
while TM_READ(state[index]) == DELETED || (TM_READ(state[index]) != EMPTY && TM_READ(set[index]) != value) do
    index = (index + probe)
end while
        ▷ Using our constructs: TM_EQ(state[index], EMPTY) ? -1 : index;
return TM_READ(state[index]) == EMPTY ? -1 : index;
TM_END
```

range of price), most of those checks are semantic and do not seek specific values. Using the classical (more conservative) TM model, any update on offers will conflict with all concurrent reservations because of those conditional statements. Using our proposed TM, as depicted in Algorithm 4, the reservation will not abort as long as the outcomes of the comparison conditions hold (e.g., `number of free slots > 0` and `price > max_price`). The key idea is that a reservation does not care about the exact value of price or the amount of available resources, it just cares about if the price is in the right range and resources are still available. Moreover, more concurrency is possible given that other transactions that update the same resources are allowed.

Algorithm 4 Using the semantic constructs to enhance reservations in Vacation benchmark.

```
TM_BEGIN
for n = 0; n < ids.length; n++ do
    res = tablePtr.find(ids[n]);
        ▷ Using our constructs: TM_GT(res.numFree, 0)
    if TM_READ(res.numFree) > 0 then
        ▷ Using our constructs: TM_GT(res.price, max_price)
        if TM_READ(res.price) > max_price then
            max_price = TM_READ(res.price);
            max_id = id;
        end if
    end if
end for
reservation = tablePtr.find(max_id);
        ▷ Using our constructs: TM_INC(res.numFree, -1)
TM_WRITE(res.numFree, TM_READ(res.numFree) - 1));
TM_END
```

Kmeans. Kmeans is another STAMP application that implements a clustering algorithm iterating over a set of points and grouping them into clusters. The main transactional overhead is in updating the cluster centers, which can be enhanced using our *TM_INC* operation, as shown in Algorithm 5.

Algorithm 5 Using the semantic constructs to enhance Kmeans benchmark.

```
TM_BEGIN
        ▷ Using our constructs: TM_INC(*new_centers_len[index], 1);
TM_WRITE(*new_centers_len[index], TM_READ(*new_centers_len[index]) + 1);
for j = 0; j < nfeatures; j++ do
        ▷ Using our constructs: TM_INC(new_centers[index][j], feature[i][j]));
    TM_WRITE(new_centers[index][j], TM_READ(new_centers[index][j]) + feature[i][j]));
end for
TM_END
```

4. Design

The challenges of injecting semantics into STM algorithms and HTM algorithms are very different. Concurrency controls in STM are entirely performed and integrated into the software framework itself. That allows any sort of modification to transactional execution, including embedding our proposal of defining new semantic constructs and calling them instead of classical ones (i.e., TM_READ and TM_WRITE). On the other hand, the current HTM releases [5, 27] leverage hardware for detecting conflicting executions and give very limited chances for optimization to the TM framework. For instance, they leave no control for modifying the granularity of the speculation in HTM transactions; in other words, every memory access within the boundaries of an HTM transaction is monitored by the hardware itself (exploiting an enhanced cache coherency protocol). As a result, executing HTM transactions means preventing any straightforward solution for replacing the basic TM_READ/TM_WRITE constructs with semantic calls (as can be done for STM).

In this section, we first show how to inject semantics into STM algorithms and then we propose designs to still exploit HTM and semantics.

4.1 Software Transactional Memory

The first step towards injecting semantics into STM algorithms is to find an abstract way to define those semantics. The semantic operations listed in Table 1 can be seen as implementations of an abstract method

```
value operation(operator, operand1, operand2)
```

where `operation` represents the abstract action (e.g., conditional expression, increment statement) that replaces the normal TM behavior when validating/updating the operands. We say that `operation` is a *read-only* operation if the values of its (variable) operands do not change after executing it; and an *update* operation otherwise. For example:

- `TM_GTE(5,x)` implements an abstract read-only method "*boolean operation($\geq$, x, 5)*";
- `TM_INC(x, 1)` implements an abstract update method "*void operation(++, x, 1)*".

The composition of those semantic operations (e.g., `x > 0 || y > 0`) can also be generalized as an abstract method `operation(x,y, ...)`, where the number of arguments depends on the number of variables involved.

Based on this abstraction, it becomes possible to define a unified methodology to port all the proposed API extensions in STM algorithms by showing how STM algorithms will handle both *read-only* and *update* semantic operations. In this section, we show how we ported the new extension to two state-of-art STM algorithms: NOrec [8] and TL2 [11][1].

4.1.1 S-NOrec

NOrec is an STM algorithm that exploits value-based validation to eliminate the need for fine-grained locks. In more detail, a transaction stores its read value as metadata in its read-set. The validation procedure, which is called before every read as well as at the commit phase of a writing transaction, succeeds if all accessed addresses have the same values as what is saved in the read-set (an optimization is made by first checking if the global timestamp has not changed).

[1] For brevity, the implementation described here does not cover the handling of address-to-address extended comparison operations.

Algorithm 6 S-NOrec

```
 1: procedure VALIDATE(TRANSACTION TX)
 2:     time = global_lock
 3:     if (time & 1) != 0 then go to 5 end if
 4:     for each (addr, operation, val ) in reads do
 5:         if ! (addr OP val) then
 6:             Abort()                                ▷ Abort will longjmp
 7:         end if
 8:     end for
 9:     if time != global lock then go to 5 end if
10:     return time
11: end procedure

12: procedure READVALID(ADDRESS ADDR, TRANSACTION TX)
13:     val = *addr
14:     while snapshot != global lock do
15:         snapshot = Validate(tx)
16:         val = *addr
17:     end while
18:     return val
19: end procedure

20: procedure RAW(ADDRESS ADDR, TRANSACTION TX)
21:     if writes[addr].type = INCREMENT then
22:         val = ReadValid(addr, tx)
23:         reads.append(address, val, EQUALS)
24:         writes[addr] = ( entry.value + val, WRITE )
25:     end if
26:     return writes[addr].value
27: end procedure

28: procedure START(TRANSACTION TX)
29:     do
30:         snapshot = global_lock
31:     while (snapshot & 1) ≠ 0
32: end procedure

33: procedure COMPARE(ADDRESS ADDR, OPERATION OP, VALUE OPERAND,
    TRANSACTION TX)
34:     if writes[addr] then
35:         return RAW(addr, tx) OP operand
36:     end if
37:     val = ReadValid(addr, tx)
38:     result = (val OP operand)
39:     reads.append(addr, operand, result ? OP : Inverse(OP))
40:     return result
41: end procedure

42: procedure READ(ADDRESS ADDR, TRANSACTION TX)
43:     if writes[addr] then
44:         return RAW(addr, tx)
45:     end if
46:     val = ReadValid(addr, tx)
47:     reads.append(addr, val, EQUALS)
48:     return val
49: end procedure

50: procedure INCREMENT(ADDRESS ADDR, VALUE DELTA, TRANSACTION TX)
51:     if writes[addr] then
52:         writes[addr] = ( entry.value + delta, entry.type )
53:     else
54:         writes[addr] = ( delta, INCREMENT )
55:     end if
56: end procedure

57: procedure WRITE(ADDRESS ADDR, VALUE VALUE, TRANSACTION TX)
58:     writes[addr] = ( value, WRITE )
59: end procedure
```

We extend NOrec to support our constructs as shown in Algorithm 6 (we call the new algorithm S-NOrec). For *read-only* semantic operations (which are the conditional expressions and bitwise operations in Table 1), we execute them using a special `compare` operation (lines 33-41). The main difference between `read` and `compare` is that `read` appends the normal address/value pair to the read-set (line 47), while `compare` saves the conditional expression (or its inverse if the condition is `false`) in the read-set (line 39). To simplify the `validate` procedure, we consider `read` as a semantic `TX_EQ` operation. Consequently, the `validate` procedure (lines 1-11) becomes a generalization of the original NOrec algorithm that uses a semantic validation instead of the original value-based one.

Supporting *update* semantic operations (increment/decrement operations in Table 1) requires storing the delta (i.e., incremented or decremented value), and applying it at the commit time relative to the current value stored at the address at commit time. In our implementation, we support the increment/decrement operations by overloading the write-set with these values. In particular, a flag is added to each write-set entry to indicate whether it stores a standard write or an increment.

To preserve consistency, S-NOrec handles the cases where a single variable is read/written by two different operations in the same transaction as follows:

- If an `increment` is preceded by a `write` or another `increment`, the new delta is accumulated over the entry's value without changing the entry's flag (line 52).
- If a `write` is preceded by an `increment` or another `write`, it just overwrites the value and changes the flag to indicate a `write` operation (line 58).
- Both `compare` and `read` check first for read-after-write hazards (lines 35 and 44) and return any written values from the write-buffer. If the write-set entry is an increment, it is promoted to traditional read and write operations (see lines 22-24). During the promotion, S-NOrec performs a read-set validation to ensure the consistency of the recently-observed snapshot.

Although, to the best of our knowledge, S-NOrec is the first STM algorithm that supports `compare` operations, a recent approach discusses supporting some patterns similar to our proposed `update` operations [30][2]. Nevertheless, S-NOrec still provides two additional innovations. First, it generalizes the problem of handling semantic operations in STM algorithms instead of focusing on a certain pattern. Second, it maintains the same privatization and publication properties [24] of the original NOrec algorithm, since it still uses the global timestamp at commit time. In fact, there is no considerable overhead of S-NOrec over NOrec in both memory and processing, as it only adds the read-set operation type and the write-set flag to distinguish between writes and increments.

4.1.2 S-TL2

TL2 relies on a shared global timestamp and a shared lock-table of versioned locks. The global timestamp is used to determine the earliest snapshot seen by the transaction. Versioned locks are used to lock addresses at commit time and to distinguish different address versions. Unlike NOrec, the read-set stores only the accessed addresses (not their values), so in order to support semantic operations we need to add a compare-set to the transaction metadata. The compare-set stores the address, the operation type, and the compared value (the operand), similar to the read-set of S-NOrec.

Algorithm 7 depicts the extended version of TL2 algorithm (called S-TL2). TL2 defines the $start\_version$ of a transaction as the lowest version accessible by it, and it is set at the transaction start using the global timestamp (line 2). However, in S-TL2 this version can be advanced during the transaction execution. Specifically, if the transaction did not issue any `read` operation, S-TL2 exploits the values stored in the compare-set by revalidating them and extending the $start\_version$ if they are still valid. This allows the transaction to tolerate more concurrent updates, because validation before the first `read` operation will be only semantic. After the first `read` operation, it is no longer possible to advance the $start\_version$ and the transaction must preserve the snapshot identified by that version.

[2] It is worth noting that the other patterns discussed in [30] can be added to our framework in a similar way to `increment/decrement`.

Algorithm 7 S-TL2

```
 1: procedure START(TRANSACTION TX)
 2:     tx.start_version = global_lock
 3: end procedure

 4: procedure VALIDATECOMPARESET(TRANSACTION TX)
 5:     time = global_lock
 6:     for each (addr, operation, val ) in tx.compares do
 7:         current = *addr
 8:         lock = getLock(addr)
 9:         if lock.version $\leq$ start_version then        ▷ Unchange since last snapshot
10:             continue                                      ▷ Skip validation
11:         end if
12:         if lock.writer $\neq$ tx then
13:             repeat until lock.writer = $\phi$             ▷ Wait for concurrent write
14:         end if
15:         if ! (current OP val) then
16:             Abort()                                       ▷ Abort will longjmp
17:         end if
18:     end for
19:     if time != global lock then go to 5 end if            ▷ Concurrent commit
20:     return time
21: end procedure

22: procedure COMPARE(ADDRESS ADDR, OPERATION OP, VALUE OPERAND,
        TRANSACTION TX)
23:     if writes[addr] then
24:         return RAW(addr, tx)
25:     end if
26:     lock = getLock(addr)
27:     L1 = lock.version
28:     if lock.writer then                                   ▷ Concurrent write
29:         if tx.reads.isEmpty() then                        ▷ No reads yet
30:             go to 26                                      ▷ Wait for concurrent writes
31:         else
32:             Abort()
33:         end if
34:     end if
35:     val = *addr
36:     L2 = lock.version
37:     if tx.reads.isEmpty() then                            ▷ No reads yet
38:         if L1 $\neq$ L2 then                              ▷ Version got changed meanwhile
39:             go to 26                                      ▷ Retry read
40:         end if
41:     else if L1 $>$ start_version $\vee$ L1 != L2 then     ▷ Concurrent write
42:         Abort()
43:     end if
44:     result = (val OP operand)                             ▷ Execute the semanitc opeartion
45:     compares.append(addr, operand, result ? OP : Inverse(OP))
46:     if tx.reads.isEmpty() $\wedge$ L1 $>$ start_version then   ▷ Newer than snapshot
47:         start_version = ValidateCompareSet()              ▷ Try to extend snapshot
48:     end if
49:     return result
50: end procedure

51: procedure READ(ADDRESS ADDR, TRANSACTION TX)
52:     if writes[addr] then
53:         return RAW(addr, tx)
54:     end if
55:     lock = getLock(addr)
56:     L1 = lock.version
57:     if lock.writer then                                   ▷ Concurrent write
58:         Abort()
59:     end if
60:     val = *addr
61:     L2 = lock.version
62:     if L1 $>$ start_version $\vee$ L1 != L2 then
63:         Abort()
64:     end if
65:     reads.append(addr)
66:     return val
67: end procedure
```

More in detail, to preserve the transaction's consistency after its first `read` operation, each `read` or `compare` operation must check the $start\_version$ (lines 41 and 62) to ensure opacity [15] – an important TM correctness property. However, the `compare` procedure tries to maximize the gain of having no read operations yet. In this case, the transaction can afford concurrent changes to accessed objects (lines 13, 30 and 39). However, this makes the algorithm subject to live-lock, so we employ a timeout mechanism to avoid that (not shown in the pseudo-code). At the end of the `compare` operation, the transaction appends the address, operation, and value to its compare-set, and it may perform a compare-set validation if the compared address is newer than the observed snapshot (line 46).

The write-set handlers (`increment`, `write` and `raw`) are similar to Algorithm 6, so we did not show them in Algorithm 7. The commit procedure (also not shown in Algorithm 7) is similar to TL2: the transaction acquires locks over its write-set; then it validates both its $read\_set$ and its compare-set; and finally it publishes the write-set values to the main memory and releases the locks. The only difference is that the compare-set is validated semantically (call `validateCompareSet`), while the $read\_set$ is validated as usual by checking the versions of the addresses against the transaction's $start\_version$. To optimize the semantic validation, the $start\_version$ is checked first. If the lock version of an entry is less than the transaction start version, there is no need to semantically validate it (line 10).

S-TL2 requires adding a compare-set and a write-set flag in addition to the orignal meta-data of TL2. Another overhead over TL2 is that the `compare` procedure is more complex than `read` and may involve calling `validateCompareSet`, whose execution time-complexity is O(N) (where N is the size of the compare-set). However, as we show later in Section 5, those overheads are dominated in most cases by the performance gain due to avoiding *false conflicts* at semantic level.

4.2 Proposed Extensions for HTM

Although injecting semantics in HTM algorithms is harder than for STM, we propose a scheme that enables it. In particular, we propose two approaches, which can be adopted separately:

- Injecting semantics in the software fallback path of HTM transaction, similar to how we injected them in pure STM algorithms;
- Reordering instructions at compilation time to minimize false conflicts (i.e., conflicts at the memory level that do not impact application semantics).

Regarding the first approach, we investigated the possible alternatives for implementing the software fallback path published in literature so far. Interestingly, we found that most of them are compliant with our former proposals on STM. For example, NOrec has been considered in literature [7, 23, 28] as one of the best STM candidates to integrate with HTM transactions because it has only one global lock as shared metadata, which minimizes the overhead of monitoring the metadata inside HTM transactions (recall any speculation on the metadata affects the number of cache lines occupied and may generate false conflicts). Among those proposals, RH-NOrec [23] is known to be one of the best-performing HTM algorithms. The main idea of RH-NOrec is to execute transactions in one of three modes: *fast-path*, in which a transaction is entirely executed in HTM; *slow-path*, in which the body of a transaction is executed in software similar to NOrec and the commit phase is executed using a small HTM transaction; and *slow-slow-path*, which is NOrec itself. These three modes are made consistent by speculating the global lock in the HTM transactions. We propose injecting semantics in the *slow-path* and the *slow-slow-path* of RH-NOrec in a similar way to what we did for NOrec.

The second approach for injecting semantics into HTM algorithms is to involve the compiler. Compilation-time solutions do not require modifying the execution pattern of HTM transactions at runtime (which is impossible in current HTM models), providing a good way of overcoming the limitations of HTM APIs.

	Hashtable		Bank		LRU		Vacation		Kmeans		Labyrinth		Yada		SSCA2		Genome		Intruder	
	base	*semantic*	*base*	*semantic*	*base*	*semantic*	*base*	*semantic*	*base*	*semantic*	*base*	*semantic*	*base*	*semantic*	*base*	*semantic*	*base*	*semantic*	*base*	*semantic*
Read	3440	0	22.5	0.05	173	12	14704	13714	25	0	176	4	142	135	2	1	84	84	28.5	28.5
Write	6.2	6.2	12.7	0	19.7	19.7	28.5	12	25	0	173	173	21.4	21.4	2	1	3	3	2.6	2.6
Compare	-	3440	-	10	-	161	-	989.5	-	0	-	172	-	7	-	0	-	0.06	-	0
Increment	-	0	-	12.7	-	0.03	-	16.7	-	25	-	0	-	0	-	1	-	0.01	-	0
Prompot	-	0	-	0.05	-	0.01	-	15.7	-	0	-	0	-	0	-	0	-	0	-	0

Table 2: Average Number of Operations per Transaction.

The order of executing reads/writes inside transactions is one of the main reasons for raising/avoiding conflicts at runtime. For example, if the conflicting reads/writes are shifted by the compiler to the end of the transaction, the probability of raising a conflict at runtime is minimized [9, 30]. We propose a further optimization on the compilation-level semantic operations defined in Table 1 by reordering them at compilation time (if possible). As a preliminary example, the increment/decrement operations can be delayed to the end of the transaction if the incremented/decremented variables are not read later in the transaction. Also, conditional branches can be "optimistically" calculated before a transaction starts, resulting in a deterministic branch selection inside that transaction, and then the optimistic branch selection can be revalidated at the end of the transaction, to preserve serializability [3]. The proposed compilation-time optimizations are valid for both STM and HTM, although HTM benefits more from that because of its limited APIs.

We mainly plan to investigate the different correctness guarantees that can be provided by altering operations at compilation time, such as opacity [15], serializability [3], and publication/privatization safety [22, 24]. Regarding the former examples, the optimistic brach selection clearly breaks opacity, generating issues like those related to the lazy subscription of the global lock as discussed in [10]. Also, shifting increments/decrements may break publication-safety in some cases, as discussed in [30]. Making an investigation on the *theoretical* and *practical* possibilities/impossibilities in that direction is one of the objectives of our future work.

5. Evaluation

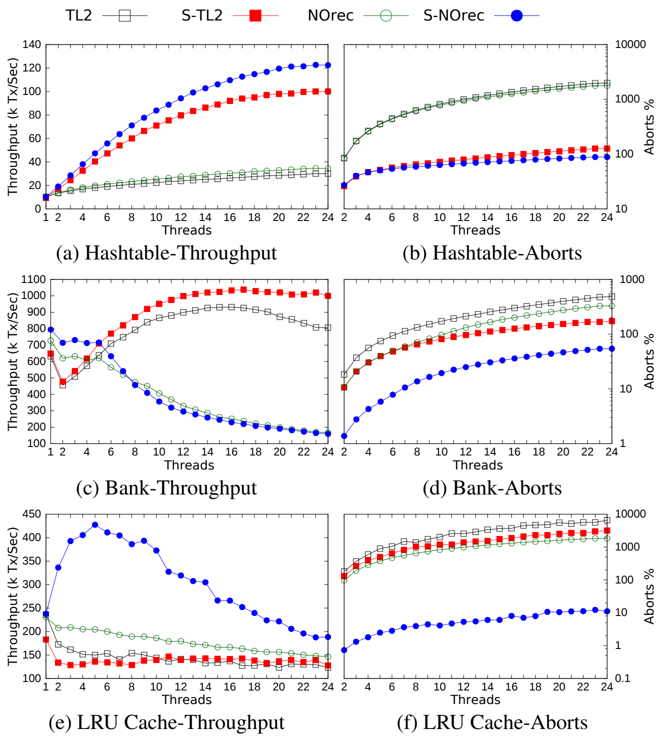
We conducted our experiments on an AMD machine equipped with 2 Opteron 6168 CPUs, each with 12-core running at 1.9 GHz. The total memory available is 12 GB and the cache sizes are 128 KB for the L1, 512 KB for the L2, and 12 MB for the L3. We reported the throughput for micro benchmarks and the application execution time for STAMP by varying the number of threads executing concurrently (the datapoint at 1 thread shows the performance of the single-threaded transactional execution).

Table 2 shows the average number of the different type of operations (measured at runtime) for each of the benchmarks. This gives an intuition about the amount of modifications we made for each benchmark by applying our semantic constructs relative to the transactions size.

5.1 Micro Benchmark

In our first set of experiments we considered three micro benchmarks: Hashtable with Open Addressing, Bank, and Least Recently Used (LRU) Cache. In each experiment, 0.5-5 million transactions were executed and both throughput and abort rate were calculated for both NOrec and TL2, with and without our semantic extensions.

Hashtable with Open Addressing. The workload in this experiment was a collection of *set* and *get* operations, where each transaction performed 10 set/get operations. Both S-NOrec and S-TL2 exploited our semantic extensions in the probing procedure, as depicted in Algorithm 2. As a result, as shown in Table 2, all `read`



(a) Hashtable-Throughput (b) Hashtable-Aborts
(c) Bank-Throughput (d) Bank-Aborts
(e) LRU Cache-Throughput (f) LRU Cache-Aborts

Figure 2: Micro Benchmarks.

operations were transformed into semantic `compare` operations. This reduced aborts by one order of magnitude (Figure 2b), which directly raised the throughput (3.5× speedup) in both algorithms (Figure 2a).

Bank. Each transaction performs multiple transfers (at most 10) between accounts with an overdraft check (i.e., skip the transfer if account balance is insufficient). In the semantic version of the benchmark, the reads/writes were transformed into `compare` and `increment` operations. As shown in Figure 2c, exploiting semantics helps S-NOrec to outperform NOrec at low-contention (1-8 threads). However, when contention increases, both NOrec and S-NOrec degrade and perform similarly. This is mainly because the probability of having true conflicts increases, and transactions start to abort even if they are semantically validated. In TL2, concurrent commits are allowed so it scales better than NOrec. Similarly, S-TL2 benefits from the underlying semantics and performs 20% better than TL2, and incurs 2.5× fewer aborts.

LRU Cache. This benchmark simulates an $m \times n$ cache implementation with least-frequently-used replacement policy. The cache uses m cache lines, and each line contains n buckets. Each bucket stores both the data and the hit frequency. Each transaction either sets or looks up multiple entries in the cache. Table 2 shows that 93% of the `read` operations were transformed into `compare` operations. Accordingly, as shown in Figures 2e and 2f, S-NOrec reduced the aborts by two order of magnitude and achieved up to

2× speedup. S-TL2 was not improved much (only 25% speedup). The reason is that the non-transformed reads in S-TL2 prevented it from advancing its snapshot; thus, any compare operations had to preserve the snapshot identified by the $start\_version$ and the overall behavior became similar to TL2.

5.2 STAMP

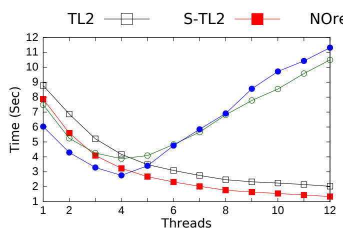
STAMP [25] is a suite of applications designed for evaluating in-memory concurrency controls. Figure 3 shows both execution time and abort rate in some of the STAMP benchmarks. We did not show the results of three applications (`Genome`, `Intruder`, and `SSCA2`) because we found that the semantic operations per transaction were very limited (see Table 2) and hence there was no difference in either abort rate or throughput. We also excluded `Bayes` because of its nondeterministic behavior. Since the performance saturated at a high number of threads in all tested applications, we show the results only up to 12 threads.

`Kmeans` is a clustering algorithm that iterates over a set of points and associate them to clusters. The main computation is in finding the nearest point, while shared data updates occur at the end of each iteration. As illustrated in Algorithm 5, updating the centroid is changed by transforming all `writes` into `increments`. S-NOrec and S-TL2 achieve 25%-40% speedup (Figure 3a). However, at a high number of threads both NOrec and S-NOrec saturate and start to degrade in performance, which indicates a high contention workload due to the coarse-grained locking. Consequently, starting from 8 threads, S-NOrec slightly performs worse than NOrec (see Figure 3a), because it adds an overhead that is not exploited to reduce the abort rate (see Figure 3b).

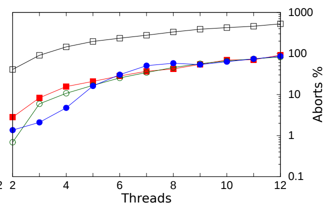
`Vacation` is a travel reservation system using an in-memory database. The workload consists of client reservations. This application emulated an OLTP workload. The reservation procedure was optimized as in Algorithm 4; however, only 7% of the reads were transformed into compares. This is because most of the `read` operations are part of the internal red-black tree operations. Additionally, almost all the `increment` operations were promoted to `read` and `write` operations because of an additional sanity check performed by the transaction. Although these two factors limited the gain of using the benchmark semantics, both S-NOrec and S-TL2 consistently outperformed the original algorithms. An exception to that was the single thread execution of S-TL2 because of the overhead of maintaining the semantic metadata (see Section 4.1.2).

`Labyrinth` is a multi-path maze solver. The maze is represented as a three-dimensional uniform grid, and each thread tries to connect input pairs by a path of adjacent maze points. Upon finding a path, it is highlighted at a shared output grid. Different checks along the routing path (e.g., isEmpty, isGarbage) were transformed into semantic `compare` operations, which allowed S-TL2 to outperform TL2 by 20%-50% speedup, and to reduce the aborts by 2× (see Figures 3e & 3f). Both S-NOrect and NOrec perform similarly, which indicates that transactions that fail in NORec's value-based validation also fail in the semantic validation of S-NOrec. In [29], an optimized version of Labyrinth was proposed. The optimized version moved some non-transactional operations (memory copy) outside the transaction, which in effect reduces the transaction size. Figures 3g & 3h shows the performance in this new version. Although S-TL2 still achieves a reduction in abort rate, the returned gain in performance became insignificant because most of the work became outside transactions.

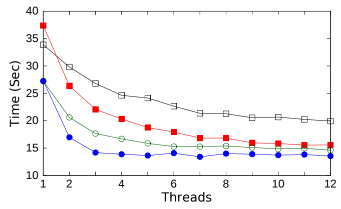
`Yada` is a mesh triangulation benchmark implementing Ruppert's algorithm. Threads iterate over the mesh and try to produce a smoother one by identifying triangles whose minimum angle is below some threshold. NOrec's behavior is similar to `Labyrinth`. Interestingly, although S-TL2 reduced the number of aborts by 3×, throughput was not affected (Figures 3i & 3j). Our measurements



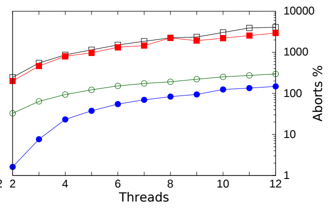
(a) Kmeans-Throughput



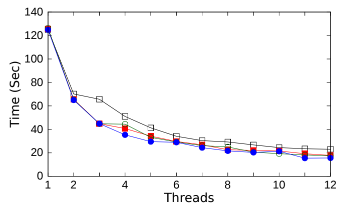
(b) Kmeans-Aborts



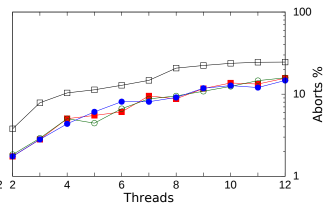
(c) Vacation-Throughput



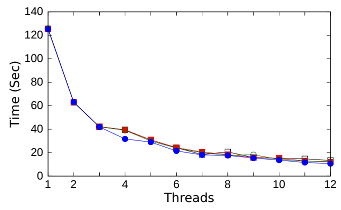
(d) Vacation-Aborts



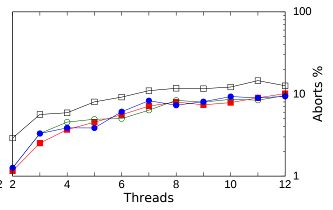
(e) Labyrinth v1-Throughput



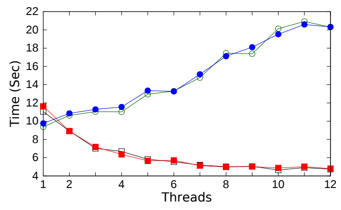
(f) Labyrinth v1-Aborts



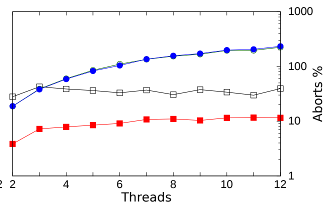
(g) Labyrinth v2-Throughput



(h) Labyrinth v2-Aborts



(i) Yada-Throughput



(j) Yada-Aborts

Figure 3: STAMP Benchmarks

revealed that the reason for this behavior is in the aborted transactions. Although resolving the semantic conflicts of transactions in S-TL2 allowed them to proceed with execution, true conflicts caused most of them to abort later. Therefore, the length of the aborted transactions in S-TL2 became longer than TL2 without real benefit from that (since transactions eventually aborted in both cases). This is similar to what happened in Bank.

6. Conclusions and Future Work

In this paper, we show that generality of TM is not always contradicting with applications semantics. We did so by identifying *TM-friendly* semantics and proposing an approach to inject them in the current TM algorithms and frameworks. Our experimental results on two different *semantic-enabled* STM algorithms depicted a promising improvement over the base algorithms. We plan to extend this line of research by investigating more on providing full compiler support, including HTM algorithms, and supporting more complex semantic patterns.

7. Acknowledgments

This work is partially supported by Air Force Office of Scientific Research (AFOSR) under grant FA9550-14-1-0187.

References

[1] Y. Afek, H. Avni, and N. Shavit. Towards consistency oblivious programming. In *OPODIS*, pages 65–79, 2011.

[2] H. Avni and B. C. Kuszmaul. Improving HTM scaling with consistency-oblivious programming. In *9th Workshop on Transactional Computing*, TRANSACT '14, 2014. Available: `http://transact2014.cse.lehigh.edu/`.

[3] P. A. Bernstein and N. Goodman. Serializability theory for replicated databases. *J. Comput. Syst. Sci.*, 31(3):355–374, 1985.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[5] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le. Robust architectural support for transactional memory in the power architecture. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 225–236, New York, NY, USA, 2013. ACM.

[6] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. In *9th Workshop on Transactional Computing*, TRANSACT '14, 2014. Available: `http://transact2014.cse.lehigh.edu/`.

[7] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 39–52, New York, NY, USA, 2011. ACM.

[8] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM.

[9] A. Dhoke, R. Palmieri, and B. Ravindran. An automated framework for decomposing memory transactions to exploit partial rollback. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 249–258, 2015.

[10] D. Dice, T. L. Harris, A. Kogan, Y. Lev, and M. Moir. Pitfalls of lazy subscription. In *WTTM*, 2014.

[11] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[12] N. Diegues and P. Romano. Self-tuning Intel transactional synchronization extensions. In *11th International Conference on Autonomic Computing*, ICAC '14. USENIX Association, 2014.

[13] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *Distributed Computing, 23rd International Symposium, DISC 2009, Elche, Spain, September 23-25, 2009. Proceedings*, pages 93–107, 2009.

[14] G. Golan-Gueta, G. Ramalingam, M. Sagiv, and E. Yahav. Automatic scalable atomicity via semantic locking. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 31–41, 2015.

[15] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[16] A. Hassan, R. Palmieri, and B. Ravindran. Optimistic transactional boosting. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 387–388, 2014.

[17] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 207–216, New York, NY, USA, 2008. ACM.

[18] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM, 2003.

[19] 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*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[20] S. Lie. Hardware support for unbounded transactional memory. Master's thesis, MIT, 2004. Available at: `http://dspace.mit.edu/handle/1721.1/28440`.

[21] V. Marathe, M. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. Scherer III, and M. Scott. Lowering the overhead of nonblocking software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.

[22] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable techniques for transparent privatization in software transactional memory. In *2008 International Conference on Parallel Processing, ICPP 2008, September 8-12, 2008, Portland, Oregon, USA*, pages 67–74, 2008.

[23] A. Matveev and N. Shavit. Reduced hardware NOrec. In *9th ACM SIGPLAN Workshop on Transactional Computing*, TRANSACT '14, 2014. Available: `http://transact2014.cse.lehigh.edu/`.

[24] V. Menon, S. Balensiefer, T. Shpeisman, A. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical weak-atomicity semantics for java stm. In *SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 14-16, 2008*, pages 314–325, 2008.

[25] 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.*, pages 35–46, 2008.

[26] Y. Ni, V. Menon, A. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007, San Jose, California, USA, March 14-17, 2007*, pages 68–78, 2007.

[27] J. Reinders. Transactional synchronization in Haswell. `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`, 2013.

[28] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 53–64, New York, NY, USA, 2011. ACM.

[29] W. Ruan, Y. Liu, and M. Spear. Stamp need not be considered harmful. In *Ninth ACM SIGPLAN Workshop on Transactional Computing*, 2014.

[30] W. Ruan, Y. Liu, and M. F. Spear. Transactional read-modify-write without aborts. *TACO*, 11(4):63:1–63:24, 2014.

[31] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *PODC '04: Proceedings of Workshop on Concurrency and Synchronization in Java Programs.*, NL, Canada, 2004. ACM.

[32] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[33] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. In *Proceedings of the*

ACM Annual Symposium on Parallelism in Algorithms and Architectures, SPAA '08, pages 275–284, New York, NY, USA, 2008. ACM.

[34] Transactional Memory Specification Drafting Group. Draft specification of transactional language constructs for C++, version 1.1, February 2012. Available `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3725.pdf`.

[35] L. Xiang and M. L. Scott. Software partitioning of hardware transactions. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 76–86, 2015.