Extending TM Primitives using Low Level Semantics

Mohamed M. Saad
Virginia Tech
msaad@vt.edu

Roberto Palmieri
Virginia Tech
robertop@vt.edu

Ahmed Hassan
Virginia Tech
hassan84@vt.edu

Binoy Ravindran
Virginia Tech
binoy@vt.edu

ABSTRACT

Transactional Memory (TM) has recently emerged as an optimistic concurrency control technique that isolates concurrent executions at the level of memory reads and writes, therefore providing an easy programming interface. However, such transparency could be overly conservative from an application-level perspective. In this work, we propose an extension to the classical TM primitives (read and write) to capture program code semantics (e.g., conditional expressions) while maintaining the same level of programming abstraction. We deployed this extension on two state-of-the-art STM algorithms and integrated it into the GCC compiler and the RSTM software framework. Results showed speedups of up to 4× (average 1.6×) on different applications including micro benchmarks and STAMP.

Keywords

Transactional Memory, Semantics, GCC

1. INTRODUCTION

Transactional Memory (TM) is a programming abstraction for accessing shared memory data without exposing any lock interfaces to the application so that difficulties and drawbacks such as deadlock, livelock, and 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 a conflict happens, a contention manager [22] resolves it by aborting one transaction and allowing the other to proceed to commit, yielding (the illusion of) atomicity. TM was originally proposed in hardware (HTM) [21] and later in software (STM) [35].

In order for a TM implementation to be generic, conflicts are usually detected at the level of memory addresses. For this reason, 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 frameworks may add other features, such as allowing external aborts, non-transactional reads/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 yet as good as (or better than) optimized manual implementations of synchronization. To overcome that, researchers have 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 [11]; compact bloom filters [6] to track memory accesses, as used in RingSTM [36]; and fine-grained ownership records, as used in TL2 [12]. On the other hand, current HTM processors [30, 7] have a "best effort" nature because transactions are not guaranteed to progress in HTM (even if they are executed alone without any actual concurrency). That is why an efficient software "fallback" path is needed (i.e., hybrid TM) when hardware transactions repeatedly fail [13]. Recent literature proposes many compelling solutions that make the fallback path fast under different conditions [10, 31, 8, 13, 25, 4].

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, when thread synchronization was manually done using fine-grained locks and/or lock-free designs, 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 boost its performance accordingly. Recent literature provides a few

Published at the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2016)

semantic-based concurrency controls, which will be detailed in Section 2. However, they either solve specific application patterns [33], break the high abstraction of TM [20, 15], or are orthogonal to TM [19, 18].

Motivated by the above question, this paper provides three major contributions. First, we identify a set of semantics that can be included in TM frameworks without impacting the generality of the TM abstraction (we call them *TM-friendly* semantics), and we extend the existing TM API to include such semantics. Second, we show how to modify STM algorithms to exploit such semantic-based APIs. Finally, we illustrate how we embedded those extensions in compiler passes (using GCC) so that the application developing experience will not be altered.

Regarding the first point, with TM-friendly semantics we mean those optimizations that can be decoupled from the application layer. In particular, this paper focuses on optimizing conditional statements (e.g., `if x > 0`) and increments/decrements (e.g., `x++`), which are commonly used in legacy applications. More details about those semantics are presented in Section 3.

The second contribution involves deploying those TM-friendly semantics with existing state-of-the-art STM algorithms. Roughly, STM algorithms can be classified into two groups according to the technique used for validating transactions. The first group uses *version-based* validation, where each memory location keeps a version number that is used to identify memory changes. The second group uses *value-based* validation, where the content of each location itself is 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 Section 4, we show how to modify NOrec [11] (a value-based algorithm) and TL2 [12] (a version-based algorithm) to include semantics. Then, in Section 5, we discuss the correctness of those new algorithms.

Our last contribution is to integrate semantic APIs and their corresponding STM algorithms into current TM frameworks. We propose two approaches to achieve that:

- The first approach is to implement semantic extensions entirely as a *compiler pass*, thus not exposing any API additions to the programmer. This approach has the advantages of being entirely transparent and retaining backward compatibility with existing applications that leverage GCC's transactional API.
- The second approach involves exposing the new semantic APIs as TM interfaces. These new APIs give conscious programmers an opportunity to better exploit semantics while developing concurrent applications. Clearly, this approach increases the chance of achieving higher performance, with the cost of reducing, although marginally, the programmability level.

Since each of those two solutions fits specific interests, we assess both of them in this paper. We assess the latter solution (which is easier to implement) by enriching the API of the RSTM [24] framework with our semantics. Regarding the former, we show in Section 6 how we modified the compilation passes of GCC to provide full compilation support with limited overhead in terms of both compilation process and execution time. In Section 7, we evaluated our semantic-based TM (using both RSTM and GCC) with the following applications: Bank, a benchmark that simulates a multithreading application where threads mostly perform money transfers; LRU-Cache, a benchmark that simulates a software cache with the least-recently-used replacement policy; a hash-table benchmark; and the STAMP benchmark suite [28]. The results show that enabling semantics boosts performance consistently, yielding a peak of 4× improvement when semantics is highly exploited. Also, contrasting the performance trend of GCC experiments with that of RSTM experiments allows understanding the consequences of moving the whole TM framework, including our semantic extensions, into the compiler level.

All the implementations used in this paper, including the new version of GCC and RSTM, are available as open-source projects at http://www.hyflow.org.

2. RELATED WORK

Not surprisingly, the trials to include semantics in TM started in the literature as early as TM itself. In fact, the potential objective of the first TM proposal, as can be easily inferred from the title of the first TM paper [21], was providing architectural support for lock-free data structures. However, the approach proposed in that paper, as well as subsequent approaches, was fairly general because its main objective was improving programmability. As a result, the performance of TM could not compete with handcrafted (i.e., very optimized) fine-grained and lock-free designs.

In the last decade, involving semantics to improve TM performance has been an important topic, addressed by approaches such as open nested transactions [29], elastic transactions [15], specialized STM [14], and early release [20]. 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 transparent as possible from the programmer's standpoint, the practical adoption of the above approaches remained limited. 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 TM implementations.

Another research direction focused on developing collections of transactional blocks (essentially data structure operations) 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 [19, 18], consistency oblivious programming [3, 5], semantic locking [16], and partitioned transactions [38] are examples of that direction. Despite the promising results, those approaches remain isolated from TM as synchronization abstractions and appear as standalone components designed mainly for data structures.

Involving compilers in TM's concurrency control is currently becoming mandatory given the enhanced GCC release [37], 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 [33]. However, that approach still addresses specific execution patterns and does not generalize the problem like our attempt 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 show the proposed semantics that can be injected into TM frameworks without hampering the generality of TM itself. As mentioned before, TM defines two language/library constructs for reading (`TM_READ`) and writing (`TM_WRITE`) memory addresses. In most cases, these constructs enforce a "conservative" conflict resolution policy; two concurrent transactions are said to be conflicting if they access the same address and at least one access is a write. Algorithm 1 gives an example that shows why such a policy may be too conservative due to lack of semantics.

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

```
                      Initially x = y = 5
TM_BEGIN(T1)
if x > 0 || y > 0 then
    // Do reads/writes ...
                                  TM_BEGIN(T2)
                                  x++
                                  y- -
                                  TM_END
TM_END
```

In this example, when T_1 executes its first line, existing TM algorithms save x and y in the read-set. Starting from this point, in order to preserve consistency, most TM implementations force T_1 to abort as soon as any concurrent change in x or y occurs. 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, T_1 has no real issue at the semantic level and can safely commit since the boolean result of the conditional expression still holds, which means that the conflict triggered by the TM framework is a "false conflict" at the semantic level.

`TM_GT(address, value|address)`	greater than
`TM_GTE(address, value|address)`	greater or equals
`TM_LT(address, value|address)`	less than
`TM_LTE(address, value|address)`	less or equals
`TM_EQ(address, value|address)`	equals
`TM_NEQ(address, value|address)`	not equals
`TM_INC(address, value)`	increment
`TM_DEC(address, value)`	decrement

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 semantics into two categories (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 a transaction implies one or two calls to `TM_READ` (depending upon the type of operands). Using our constructs, we consider the whole expression as one semantic operation, and the safety of the enclosing transaction is preserved by 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 as arguments. Unlike the first category, the traditional way of handling transactional increment/decrement involves both `TM_READ` and `TM_WRITE`. In our solution, leveraging semantics means invoking one semantic operation that performs the actual read only at commit time, which allows for more concurrency.

Including those semantic operations in TM frameworks is appealing for two reasons. First, they are commonly used in applications, as we show later with some examples. Second, the integration can be entirely done at compilation time, where the compiler can detect semantic operators and translate them.

An interesting feature of the semantic operations listed in Table 1 is that they can compose by having more than one operator and/or more than one variable in the conditional expression. For example, the scenario shown in Algorithm 1 can be further enhanced if the whole conditional expression (i.e., `TM_READ(x) > 0 || TM_READ(y) > 0`) is considered as one semantic read operation. In this example, if the condition was initially `true` and then a concurrent transaction modifies only one variable, either x or y, to be negative, considering the clause as a whole avoids aborting T_1 given the `OR` operator. A similar enhancement consists of allowing complex expressions in conditional statements (e.g., `x + y > 0`), where modifications on multiple variables may compensate each other so that the return value of the overall expression remains unchanged. Although supporting such complex expressions is appealing because it may save additional aborts, integrating them into algorithm designs and GCC may add overheads in terms of compilation process and execution time. For that reason, we currently do not support those complex expressions, and we plan for further investigation on them. A more detailed discussion about those operations is in the technical report [34].

3.1 TM-friendly semantics in action

To further support the need of injecting semantics into the classical TM abstraction, we now show examples from real benchmarks and applications whose performance can be enhanced by our semantic TM extensions. These examples are clearly not exhaustive, but they are representative of programming patterns 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. 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, concurrent changes to the accessed cells will abort the probing transaction. Considering semantics through our proposed extensions avoid such aborts. Algorithm 2 depicts pseudocode of the probing method and its transformed semantic version.

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

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

Algorithm 3 Using our 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;
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
```

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 4 Using our semantic constructs to enhance reservations in the 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;
reservation = tablePtr.find(max_id);
                    ▷ Using our constructs: TM_INC(res.numFree, -1)
TM_WRITE(res.numFree, TM_READ(res.numFree) - 1));
TM_END
```

Vacation. This application is included in the STAMP suite [28] and 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 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 semantic extensions, 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, which also explains well the intuition behind our proposal, is that a reservation does not use the exact value of price or the amount of available resources, it just checks if the price is in the right range and resources are still available.

Kmeans. Kmeans is another STAMP application, which implements a clustering algorithm that iterates over a set of points and groups 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 our semantic constructs to enhance the 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]));
TM_END
```

4. SEMANTIC-BASED TM ALGORITHMS

The first step towards injecting semantics into STM algorithms is to find an abstract way to define them. The semantic operations listed in Table 1 can be seen as the implementation of two abstract methods:

```
bool cmp(operator, address, val)
void inc(address, delta)
```

where `cmp` and `inc` represent the semantic actions that replace the normal TM behavior (`delta` can be positive or negative to support increment and decrement). In this abstraction, we restrict `cmp` operations in Table 1 to those that have an address and a value as arguments. However, as we show in Section 6, our compilation pass also detects the address-address case and translates it to a specific API call. Extending the STM algorithms presented in this section to cover the address-address case is straightforward, thus we do not include it to simplify the presentation.

In this section, we show how we integrate the above two abstract methods into two state-of-the-art STM algorithms: NOrec [11] and TL2 [12].

4.1 S-NOrec

NOrec is an STM algorithm that exploits value-based validation to eliminate the need for fine-grained locks. A transaction stores the values it reads as metadata in a local read-set and validates this read-set before every read, as well as at the commit phase of writing transactions. The commit phase is protected by a single global timestamped lock. The validation procedure succeeds if all accessed addresses have the same values as what is saved in the read-set.

We extend NOrec to support our constructs as shown in Algorithm 6 (we call the new algorithm S-NOrec), mainly by executing `cmp` and `inc` using additional procedures. The main difference between `read` and `cmp` is that `read` appends the normal address/value pair to the read-set (line 41) while

Algorithm 6 S-NOrec

```
1:  procedure VALIDATE(TRANSACTION TX)
2:      time = global_lock
3:      if (time & 1) != 0 then go to 2 end if
4:      for each (addr, operation, val ) in reads do
5:          if ! (addr OP val) then                ▷ Semantic validation
6:              Abort()
7:      if time != global lock then go to 2 end if
8:      return time
9:  end procedure
10: procedure READVALID(ADDRESS ADDR, TRANSACTION TX)
11:     val = *addr
12:     while snapshot != global lock do
13:         snapshot = Validate(tx)
14:         val = *addr
15:     return val
16: end procedure
17: procedure RAW(ADDRESS ADDR, TRANSACTION TX)
18:     if writes[addr].type = INCREMENT then
19:         val = ReadValid(addr, tx)              ▷ Promote increment
20:         reads.append(address, val, EQUALS)
21:         writes[addr] = ( entry.value + val, WRITE )
22:     return writes[addr].value
23: end procedure
24: procedure START(TRANSACTION TX)
25:     do
26:         snapshot = global_lock
27:     while (snapshot & 1) ≠ 0
28: end procedure
29: procedure COMPARE(ADDRESS ADDR, OPERATION OP, VALUE
    OPERAND, TRANSACTION TX)
30:     if writes[addr] ≠ ϕ then
31:         return RAW(addr, tx) OP operand
32:     val = ReadValid(addr, tx)
33:     result = (val OP operand)
34:     reads.append(addr, operand, result ? OP : Inverse(OP))
35:     return result
36: end procedure
37: procedure READ(ADDRESS ADDR, TRANSACTION TX)
38:     if writes[addr] ≠ ϕ then
39:         return RAW(addr, tx)
40:     val = ReadValid(addr, tx)
41:     reads.append(addr, val, EQUALS)
42:     return val
43: end procedure
44: procedure INCREMENT(ADDRESS ADDR, VALUE DELTA, TRANSACTION
    TX)
45:     if writes[addr] ≠ ϕ then
46:         writes[addr] = ( entry.value + delta, entry.type )
47:     else
48:         writes[addr] = ( delta, INCREMENT )
49: end procedure
50: procedure WRITE(ADDRESS ADDR, VALUE VALUE, TRANSACTION TX)
51:     writes[addr] = ( value, WRITE )
52: end procedure
```

`cmp` saves the conditional expression (or its inverse if the condition is `false`) in the read-set (line 34). To simplify the `validate` procedure, we consider `read` as a semantic `TX_EQ` operation. Consequently, the `validate` procedure (lines 1-9) becomes a generalization of the original NOrec that uses a semantic validation instead of the original value-based one.

Both `read` and `cmp` read the address using a special procedure, called `readValid` (lines 10-16), that performs a read-set validation (if the global timestamp changed) to ensure the consistency of the current state of the read-set.

Supporting `inc` operations requires storing the delta (i.e., incremented or decremented value) in the write-set, and applying it at commit time. In practice, we support `inc` by overloading NOrec's write-set. In particular, a flag is added to each write-set entry to indicate whether it stores a standard write or an increment.

The cases where a variable is read/written (either semantically or non-semantically) by two different operations in the same transaction are handled by S-NOrec as follows:

- *write after write:* If an `inc` is preceded by a `write` or an `inc`, the new delta is accumulated over the entry's value without changing the entry's flag (line 46). If a `write` is preceded by a `write` or an `inc`, it just overwrites the value and changes the flag to indicate a `write` operation (line 51).
- *read after write:* Both `compare` and `read` check the write-set first for read-after-write conflicts (lines 31 and 39). If the write-set entry is an increment, the `inc` is promoted to traditional `read` and `write` operations (see lines 19-21). The read part of the promotion is also done using the `readValid` procedure (line 19).
- *write after read:* This case is inherently covered because the value of the address will be validated anyway at commit time (because of the read) before the write takes place. It does not matter if the read/write operations are semantic or non-semantic.
- *read after read:* We add two different entries in the read-set for each read. Although this approach looks redundant and may nullify the gain of adopting a semantic validation if one read is semantic and the other is non-semantic, the overhead of discovering duplicates may not be negligible in the normal cases.

S-NOrec is the first STM algorithm, to the best of our knowledge, that supports `cmp` operations. For `inc` operations, a recent approach discusses supporting a pattern similar to our proposal [33]. Interestingly, in contrast with [33], S-NOrec maintains the same privatization and publication properties [26] 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 with respect to both processing time and memory occupied, as it only adds the read-set operation type and the write-set flag to the algorithm's metadata.

4.2 S-TL2

TL2 is an STM algorithm that maps shared memory locations to a table of ownership records (`orecs`). Writing transactions lock the `orecs` of their write-set entries at commit instead of acquiring a global lock as in NOrec. Because of that, writing transactions can commit concurrently as long as they access different `orecs`, and hence TL2 is known to scale better than NOrec. To validate reads, TL2 leverages: *i)* a global timestamp, which is atomically incremented by each writing transaction at commit; *ii)* a *start_version* for each transaction, which is set at the beginning of the transaction by snapshotting the global timestamp; and *iii)* an *orec_version* for each `orec`, which is modified by the writing transaction at commit time. This way, validation is done simply by ensuring that the *orec_version* of a newly read address is less than the *start_version* of the transaction, and revalidating the *orec_versions* of the whole read-set at commit time (only if the transaction is a writing transactions).

Algorithm 7 depicts our extended version of TL2 (called S-TL2). The write-set handlers (`inc`, `write` and `raw`) are similar to Algorithm 6, so we did not show them in Algorithm 7. On the other hand, supporting the `cmp` operation in

Algorithm 7 S-TL2

```
 1: procedure START(TRANSACTION tx)
 2:     tx.start_version = global_timestamp
 3: end procedure
 4: procedure COMPARE( ADDRESS addr, OPERATION op,
 5:                VALUE OPERAND, TRANSACTION tx)
 6:     if writes[addr] ≠ ϕ then
 7:         return RAW(addr, tx)
 8:     orec = getOrec(addr)
 9:     L1 = orec.version
10:     if tx.reads.isEmpty() then              ▷ Phase 1: No reads yet
11:         if orec.lock ∉ {tx, ϕ} then
12:             go to 8                             ▷ Wait until unlocked
13:         val = *addr
14:         L2 = orec.version
15:         if L1 ≠ L2 then
16:             go to 8                                      ▷ Retry read
17:         result = (val OP operand)             ▷ Add to compare-set
18:         compares.append(addr, operand, result ? OP : Inv(OP))
19:         if L1 > start_version then
20:             time = global_timestamp
21:             ValidateCompareSet()
22:             if time != global_timestamp then
23:                 go to 20                          ▷ Retry validation
24:             else
25:                 start_version = time          ▷ Extend start_version
26:     else              ▷ Phase 2: At least one pervious read occur
27:         if orec.lock ∉ {tx, ϕ} then
28:             Abort()
29:         val = *addr
30:         L2 = orec.version
31:         if L1 > start_version ∨ L1 != L2 then
32:             Abort()
33:         result = (val OP operand)             ▷ Add to compare-set
34:         compares.append(addr, operand, result ? OP : Inv(OP))
35:     return result
36: end procedure
37: procedure READ(ADDRESS addr, TRANSACTION tx)
38:     if writes[addr] ≠ ϕ then
39:         return RAW(addr, tx)
40:     orec = getOrec(addr)
41:     L1 = orec.version
42:     if orec.lock ∉ {tx, ϕ} then
43:         Abort()
44:     val = *addr
45:     L2 = orec.version
46:     if L1 > start_version ∨ L1 != L2 then
47:         Abort()
48:     reads.append(orec)                              ▷ Add to read-set
49:     return val
50: end procedure
51: procedure VALIDATEREADSET(TRANSACTION tx)
52:     for each (orec) in tx.reads do
53:         if orec.lock ∉ {tx,ϕ} ∨ orec.version > start_version then
54:             Abort()
55: end procedure
56: procedure VALIDATECOMPARESET(TRANSACTION tx)
57:     for each (addr, operation, val ) in tx.compares do
58:         current = *addr
59:         orec = getOrec(addr)
60:         if orec.version > start_version then
61:             if orec.lock ∉ {tx,ϕ} then
62:                 repeat until orec.lock = ϕ   ▷ Wait until unlocked
63:             if !(current OP val) then         ▷ Semantic validation
64:                 Abort()
65: end procedure
66: procedure COMMIT(TRANSACTION tx)
67:     AcquireWriteSetLocks(tx)
68:     time = global_timestamp
69:     if start_version ≠ time then
70:         ValidateCompareSet(tx)
71:     if !CAS(global_timestamp, time, time+1) then
72:         go to 68                    ▷ Retry compare-set validation
73:     if start_version + 1 ≠ time then
74:         ValidateReadSet(tx)
75:     WriteBack(tx, time + 1)
76:     ReleaseWriteSetLocks(tx)
77: end procedure
```

S-TL2 is more complex than S-NOrec. The first issue is that the actual addresses and their values are not saved in the read-set; only the corresponding `orecs` are saved. To solve this problem, we first define a separate *compare-set* for saving `cmp` operations whose structure is similar to S-NOrec's read-set. In particular, a `read` operation saves the `orec` of the address in the read-set (line 48), and a `cmp` operation saves the actual address along with the information about the compare operation in the compare-set (lines 18 and 34).

The second problem is that we now have two ways of validating reads: the first relies on value-based validation (for `cmp` operations), while the second relies on the relation between the *read_version* of an `orec` and the *start_version* of the enclosing transaction (for `read` operations). To address this issue in an efficient way, we split the execution into three phases. The first phase starts from the transaction begin until the first `read` operation. The second one starts from the first `read` until right before commit. The last phase is the commit phase.

In the first phase, before the first `read` operation, `cmp` operations can be optimized similar to S-NOrec (lines 10-25): the transaction's *start_version* is not used, and rather the compare-set is validated after each `cmp` operation (line 21). If this validation succeeds, the transaction's *start_version* is extended (line 25). This way, we allow semantic validations as long as no `read` operation is executed yet. Another optimization, although less important, is when the address's `orec` is observed to be locked by a concurrent transaction. In this case, the `cmp` operation waits until the `orec` is unlocked instead of aborting the transaction (line 62). In those cases, we employ a timeout mechanism (not shown in the algorithm) to avoid starvation. This optimization makes sense only for `cmp` operations. This is because for `read` operations, observing a locked `orec` means that its *orec_version* will likely be updated before it is unlocked, which also means that the read will be invalidated and the transaction will be aborted anyway. The same optimization is made when a concurrent transaction changes the *orec_version* while reading the variable (line 16), and also when the global-timestamp is changed during the compare-set validation (line 23).

In the second phase, after the first `read` operation, `cmp` operations have to preserve consistency with previous reads, and therefore the transaction's *start_version* cannot be extended anymore. That is why, in this phase, both `read` (lines 37-50) and `cmp` (lines 26-34) validate that the newly read address (even if inside a `cmp` operation) is consistent with the previous reads, by comparing the *orec_version* with the transaction's *start_time* as the original TL2 does.

The commit phase is depicted in lines 66-77. In TL2, the commit phase starts by locking the writes' `orecs` and atomically incrementing the global timestamp. Then the reads are re-validated. If validation succeeds, writes are published and then locks are released. The commit phase of S-TL2 differs from that of TL2 in two points: *i)* the way reads are validated; *ii)* the way the global timestamp is incremented.

Regarding the first point, the read-set and the compare-set are validated differently using `ValidateReadSet` (lines 51-

55) for the former and `ValidateCompareSet` (lines 56-65) for the latter. Specifically, when the *read_version* of an `orec` is greater than the *start_version* of the transaction, which means that the value of the address may have been changed, `ValidateReadSet` aborts the transaction (line 54), while `ValidateCompareSet` re-computes the expression and aborts only if the return value changes (line 64).

Second, if a concurrent transaction starts its commit phase during `ValidateCompareSet`, the compare-set has to be re-validated. This is important because the return value of one `cmp` operation may be affected by this new commit, and thus `ValidateCompareSet` procedure may return an incorrect result. Lines 68-72 depict how we achieve that (the order of lines is important here): the global timestamp is snapshotted, `ValidateCompareSet` is called, and then the global timestamp is incremented using `CAS` instead of using `AtomicFetchAndAdd`. If the `CAS` fails, validation is retried. It is worth to note that this mechanism is not needed for `ValidateReadSet` because it conservatively aborts the transaction if any `orec` in the read-set has changed.

S-TL2 requires adding a compare-set as well as a flag in the write-set in addition to the original metadata of TL2. An additional source of overhead is that `cmp` operations may involve calling `ValidateCompareSet`, whose execution time is linear with respect the size of the compare-set itself. However, as we show in the evaluation, those overheads are mostly dominated by the performance gain due to avoiding unnecessary aborts.

5. CORRECTNESS

The correctness of a TM algorithm is usually inferred by proving that all histories it generates are *opaque* [17]. We infer the correctness of S-NOrec and S-TL2 in the same way.

We start by roughly recalling some definitions related to opacity, borrowed from [17], to make the presented intuitions self-contained. A history $\mathcal{H}$ is a sequence of operations issued by transactions on a set of shared objects. Intuitively, we say that a history $\mathcal{H}$ is *sequential* if no two transactions are concurrent. A sequential specification of a shared object *ob*, called *Seq(ob)*, is the set of all sequences of operations on *ob* that are considered correct when executed sequentially. A sequential history is *legal* if, for every shared object *ob*, the subsequence of operations on *ob* in $\mathcal{H}$ is in *Seq(ob)*. Two histories are *equivalent* if they contain the same transactions with the same operations and the same return values. Given a history $\mathcal{H}$, $Complete(\mathcal{H})$ indicates the set of histories obtained by committing or aborting every commit-pending transaction in $\mathcal{H}$, and aborting every other live transaction in $\mathcal{H}$. A history $\mathcal{H}$ is *opaque* if any history in $Complete(\mathcal{H})$ is equivalent to a legal sequential history $\mathcal{S}$ that preserves the *real-time order* of $\mathcal{H}$.

The definition of opacity is general enough to be applied on shared objects with generic APIs, as long as every shared object has a well-defined *sequential specification* based on those APIs. However, as we mentioned before, most TMs consider the *read-write register* abstraction with two APIs: a `read` operation, which takes no argument and returns the current state of the register, and a *write* operation, which takes a value v as argument and always returns *ok*. This simple scheme implies a trivial sequential specification for each register x, as defined in [17]: *the set of all sequences of read and write operations on x such that every read operation returns the value given as an argument to the latest preceding write operation (the initial value is the default).*

Although this simple scheme best fits shared memory models, it does not match the API of our *semantic-based* TM because it does not distinguish between reads/writes that are made within a comparison/increment expression and all other reads/writes. That is why the first step towards proving the correctness of our algorithms is to define the new abstraction for our TM. Our TM algorithms (S-Norec and S-TL2) implement a TM whose shared *registers* export four operations: `read`, which takes no argument and returns the current state of the register; `write`, which takes a value v as an argument and returns always *ok*; `inc`, which takes a value d as an argument and returns always *ok*; and `cmp`, which takes a value *v_cmp* and an operator type *Op* whose value is given from the enum $\{==, !=, >, >=, <, <=\}$ as arguments and returns *true* or $false$.

The sequential specification of a register x in our TM, $Seq(x)$, is defined as follows: the set of all sequences of `read`, `write`, `cmp`, and `inc` operations on x, such that:

- every `read` operation returns $v + \sum d$, where v is the value given as an argument to the latest preceding `write` operation, w, and $\sum d$ is the sum of the values given as arguments to every `inc` between the `read` operation and w;
- every `cmp` operation returns the boolean value of the expression (v *Op v_cmp*), where v is the return value of the corresponding `read` operation.

Algorithm 8 A history that is opaque with our API.

```
                    Initial values are 0
TM_BEGIN(T1)
if x >= 0 then
                                  TM_BEGIN(T2)
                                  x = 1
                                  y = 1
                                  TM_END
    z = y;
TM_END
```

Algorithm 8 gives an example that clarifies the importance of defining a new abstraction for our TM. Using the original *read-write register* abstraction, the corresponding history has two possible equivalent sequential histories ($T_1 \to T_2$ and $T_2 \to T_1$), and both of them are not legal because T_1 returns an illegal value for y in the former and an illegal value for x in the latter. However, using our (correct) abstraction, $T_2 \to T_1$ is an equivalent legal sequential history because x is read using `cmp` and the return value of this `cmp` is legal, which means that the history is opaque.

Another interesting example to assess the correctness of S-NOrec and S-TL2 is shown in Algorithm 9. The history of this example is not opaque even with the new API because the value of x at the moment of executing the `cmp` operation was different from its value when the transaction read y.

Based on the two cases in Algorithms 8 and 9, it is easy to understand the idea behind proving opacity of histories containing `cmp` operations, which is proving that: *i)* the address read inside each `cmp` is consistent with all the previous reads at the moment of computing the return value of the conditional expression, and *ii)* this return value does not change until the transaction commits (even if the value of the address becomes inconsistent).

Proving that a history containing `inc` operations is opaque is easier to infer. This is because the read part of `inc` can be

Algorithm 9 A history that is not opaque with our API.

```
                    Initial values are 0
TM_BEGIN($T_1$)
z = y
                                   TM_BEGIN($T_2$)
                                   x = 1
                                   y = 1
                                   TM_END
if x >= 1 then
    z = 1
TM_END
```

deferred to the commit phase where the address is locked, and thus the whole `inc` operation is considered as a `write` operation during the transaction execution. The only exception for that is when the address accessed by an `inc` operation is also accessed by another operation in the same transaction. In the following two sections, we show how those cases are covered by S-NOrec and S-TL2.

5.1 Correctness of S-NOrec

Based on the opacity definition, the correctness of S-NOrec can be inferred if we identify the *legal* sequential history $\mathcal{S}$ that is equivalent to a generic history $\mathcal{H}$ generated at any point of its execution (after completing $\mathcal{H}$). Roughly, $\mathcal{H}$ may contain committed transactions (either read-only or writing) and live transactions (considering aborted transaction as live right before they trigger the abort call). $\mathcal{S}$ is identified, similar to NOrec, as follows: committed writing transactions are serialized when they CAS the global timestamp at commit; and both read-only and live transactions are serialized when the validation of their last finished `read`/`cmp` succeeds (i.e., after the committed writing transaction that sets the global timestamp with the value returned at line 8).

The history $\mathcal{S}$ remains legal with the existence of `cmp` operations because of the following reasons: *i)* every address is initially read consistently using `readValid` procedure in all `read`, `cmp`, and `RAW` calls; and *ii)* the semantic validation made after each read and during commit guarantees that the return values of each `cmp` remain the same.

The existence of `inc` operations also does not affect the legality of $\mathcal{S}$ because: *i)* at commit time, `inc` is handled exactly like `write`, which is safe because the transaction has an exclusive access to the address (in fact commit phases in NOrec are executed serially); and *ii)* live transactions that execute `inc` along with other operations on the same address are always consistent because the read operations (`read` and `cmp`) check the write-set first and promote the `inc` operation if needed, and the write operations (`write` and `inc`) properly override the write-set entry.

5.2 Correctness of S-TL2

The legal equivalent serialization of a history generated by S-TL2 is slightly different from TL2. In fact, handling `inc` operations does not affect the serialization because of the same reasons mentioned for S-NOrec. However, we identify two differences that arise due to `cmp` operation.

First, committed writing transactions are serialized in TL2 when they atomically increment the timestamp. In S-TL2, this atomic increment is replaced with a `CAS` operation which forms the new serialization point (line 71). Using `CAS` instead of `AtomicFetchAndAdd` is needed because it is not legal for a transaction to observe the writes of any transaction that increments the global timestamp after it. Note that it is guaranteed that the transaction observes the writes of all transactions that increment the timestamp before it, because the `orecs` of the write-set entries are locked before incrementing the timestamp (line 67) and the transaction waits until those `orecs` are unlocked.

Second, the serialization of read-only and live transactions depends on the phase they are executing. If the transaction is in the first phase, before any `read` operation, the serialization point is, similar to S-NOrec, when the validation during the last `cmp` operation succeeds (more specifically, when *start_version* is advanced at line 25). That is because all the committed writing transactions so far did not change the return value of all `cmp` operations. On the other hand, if the transaction is in the second phase, after the first `read` operation, it is serialized similar to TL2, namely before all writing transactions that commit with a timestamp greater than its *start_version*. That is legal because: *i)* all `cmp` operations in the first phase are consistent up to the current value of *start_version*; and *ii)* all `read` and `cmp` operations in the second phase use this *start_version* in validation.

6. INTEGRATION WITH GCC

GCC has supported STM since version 4.7 and HTM since version 4.9. The integration of TM into GCC resulted in adding `_transaction_atomic` to the constructs of C/C++ [2, 23]. Statements within a `_transaction_atomic` block are translated by GCC to the appropriate TM calls that follow an Application Binary Interface (ABI) similar to the TM ABI proposed by Intel [1]. Those calls are handled according to the TM algorithm chosen by the programmer. The implementation of those TM algorithms is encapsulated in the *libitm* library[1].

The first, straightforward, step we made towards embedding our semantic interfaces is adding three semantic operations to *libitm*'s ABI (see Table 2). The first two operations, `_ITM_S2R` and `_ITM_S1R`, handle `cmp` operation with address-address and address-value modes, while `_ITM_SW` handles `inc` operation. Then, we deployed our S-NOrec algorithm as an additional TM algorithm in the *libitm* library, and implemented the new ABI operations as described in Section 4.1. Due to lack of a TL2 implementation that matches the baseline we used to construct our S-TL2, we plan the integration of S-TL2 as a future work. Besides, in the TM algorithms currently existing in *libitm* library, those new operations are implemented by delegating their execution to the classical read and write handlers.

`_ITM_S2R`*type*	address-address semantic read operation
`_ITM_S1R`*type*	address-value semantic read operation
`_ITM_SW`*type*	semantic write operation

Table 2: Extended GCC ABI.

The next, more complicated, step is to detect the code patterns of our semantic operations (`cmp` and `inc`) during compilation. We do that after GCC generates the GIMPLE [27] representation of the program. GIMPLE is a language independent, tree-based representation that uses 3-operands expressions (except for function calls) in Static Single Assignment (SSA) [9] form. We chose GIMPLE to deploy our

[1]We use GCC 5.3 and *libitm* libraries from https://github.com/mfs409/transmem, which include NOrec.

optimization passes for two reasons. First, GIMPLE is both architecture and language independent, thus optimizing it is considered a transparent *middle-end* optimization. Second, GIMPLE uses temporary variables to put its expressions in 3-operand form, where every variable is assigned only once. This form simplifies dependency analysis.

The `tm_mark` pass is one of the optimization passes on the GIMPLE representation where statements that perform transactional memory operations are replaced with the appropriate TM built-ins. We extended this pass to detect the code patterns of `cmp` and `inc` operations as follows.

- *`cmp`:* For any conditional expression we track the origins of its two operands along the GIMPLE tree. If one origin refers to a direct transactional memory access and the other refers to either a literal value or a local variable, then we replace the condition with a call to the `_ITM_S1R` built-in. If the two origins refer to direct transactional memory accesses, we use `_ITM_S2R`.
- *`inc`:* For any transactional write, we track the origin of its right hand side, and if it is calculated using a mathematical "+" or "-" equation, we track both its operands. If the origin of one of them is a transactional read to the same written address and the origin of the second operand is either a literal value or a local variable, we call the `_ITM_SW` built-in instead of generating the transactional write. At this stage, the original transactional read of the `inc` still exists and has to be removed. We did so by developing another optimization pass, named `tm_optimize`, that removes TM read calls for *never-live variables*, since the read part of every `inc` becomes one of those *never-live variables* reads after replacing the write part with our call. This pass is made in a conservative way; it does not remove a read if there is no guarantee that it is *never-live*.

A side optimization that our `tm_optimize` pass performs is removing any TM read that is part of a never-live assignment, even if it is not originally part of an `inc` operation. The current GCC version does not perform any *liveness optimization* or *dead assignment identification* on the transactional code, therefore it does not remove such reads.

Another important note is that cases where a shared variable is involved in both semantic (i.e., `cmp` and `inc`) and non-semantic (i.e., `read` and `write`) operations of the same transaction are handled by the TM algorithm as mentioned in Section 4 (see lines 31 & 46 of Algorithm 6, and line 7 of Algorithm 7), therefore it is not needed to detect those cases in the compiler passes.

One of the advantages of our optimizations is that they reduce the number of TM calls from two to one when using `_ITM_S2R` or `_ITM_SW`. Such reduction has a visible impact on application performance; in fact, TM calls are costly because GCC performs three indirect calls per TM call. Also, our pass does not require complex alias analysis for tracking the operands origin because we look for simple expression patterns that usually reside in the same basic block.

7. EVALUATION

We tested our extended semantic-based TM on a set of micro-benchmarks, as well as applications of the STAMP suite [28]. We conducted our experiments on an AMD machine equipped with 2 Opteron 6168 CPUs, each with 12 cores running at 1.9 GHz. The total memory available is 12 GB. We reported the throughput for the micro-benchmarks, and the application execution time for STAMP, by varying the number of threads executing concurrently. We reported the results for both RSTM and GCC implementations.

Table 3 shows the average number of invocations per operation type in the used benchmarks. They are measured at runtime using RSTM because it provides more flexibility to extract statistics than GCC. In this table, semantic and non-semantic algorithms are contrasted to give an intuition about the number of read and write operations saved by applying our semantic constructs. Reduction is substantial, which enables performance improvement as showed later.

7.1 RSTM-based implementations

In the following experiments, depicted in Figure 1, throughput/time and abort rate were computed for NOrec and TL2 in both their semantic and original (i.e., non-semantic) versions.

Micro benchmarks. In our first set of experiments we considered three micro benchmarks: Hashtable with Open Addressing, Bank, and Least Recently Used (LRU) Cache.

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, and as shown in Table 3, all `read` operations were transformed into semantic `cmp` operations. This reduced the number of aborts significantly (Figure 1b), which directly raised the throughput (up to 4× speedup) in both algorithms (Figure 1a).

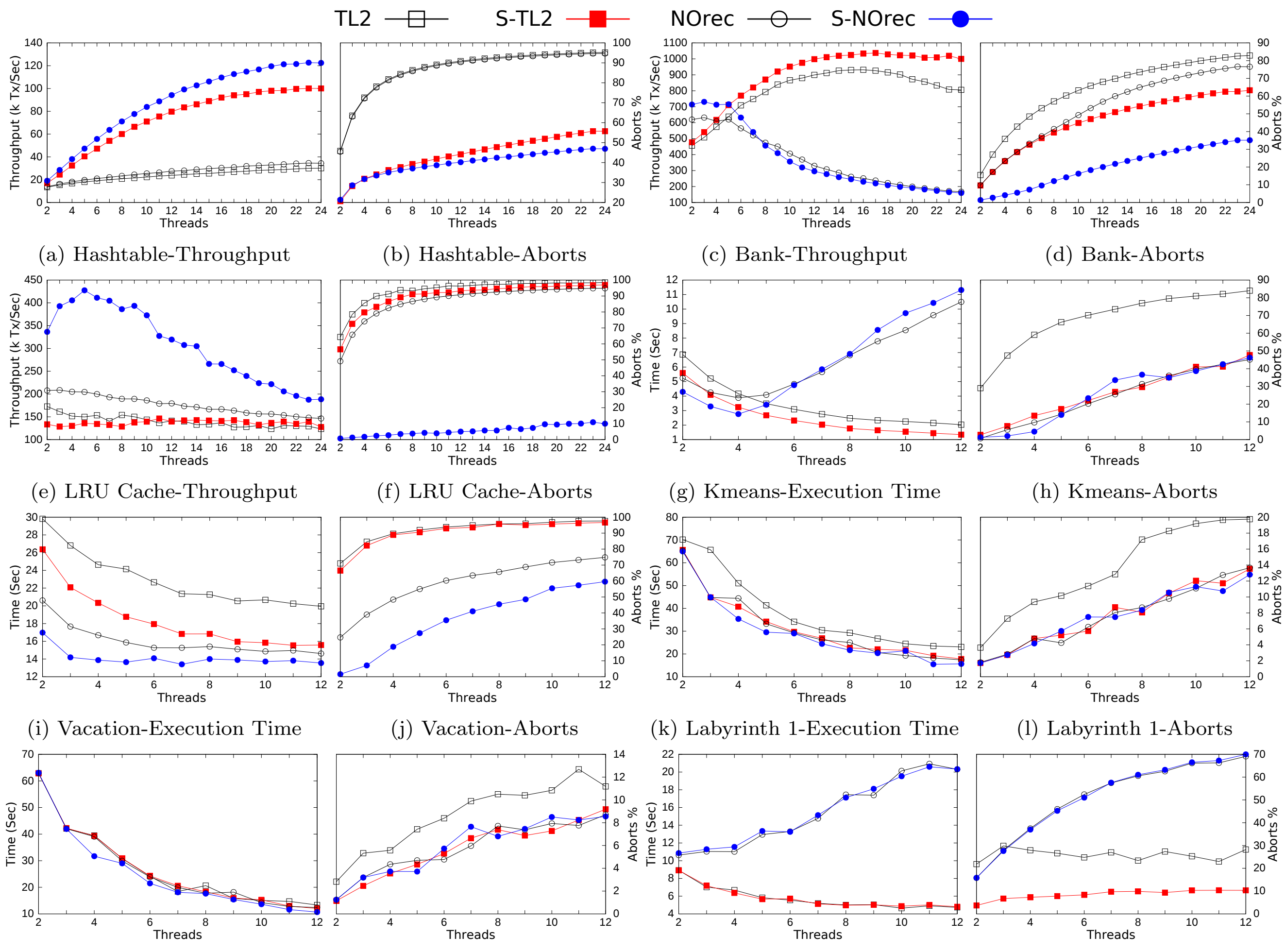
Bank. Each transaction performs multiple transfers (at most 10) between accounts with an overdraft check (i.e., skip the transfer if account balance is insufficient). In the semantic version of the benchmark, the reads/writes were transformed into `cmp` and `inc` operations. As shown in Figure 1c, exploiting 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. Moreover, due to the overhead of semantic validation, S-NOrec performs slightly worse in some cases. In TL2, concurrent commits are allowed, thus it scales better than NOrec. Similarly, S-TL2 benefits from the underlying semantics and performs 20% better than TL2, and incurs 25% fewer aborts.

LRU Cache. This benchmark simulates an $m \times n$ cache 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 3 shows that 93% of the `read` operations were transformed into `cmp` operations. Accordingly, as shown in Figures 1e and 1f, S-NOrec reduced the aborts dramatically 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 (i.e., it makes the first phase described in Section 4.2 shorter); thus, any compare operations had to preserve the snapshot identified by the *start_version* and the overall behavior becomes similar to TL2.

STAMP. STAMP is a suite of applications designed for evaluating in-memory concurrency controls. We did not show the results of three applications (`Genome`, `Intruder`,

	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
Promote	-	0	-	0.05	-	0.01	-	15.7	-	0	-	0	-	0	-	0	-	0	-	0

Table 3: Average Number of Operations per Transaction.



(a) Hashtable-Throughput (b) Hashtable-Aborts (c) Bank-Throughput (d) Bank-Aborts

(e) LRU Cache-Throughput (f) LRU Cache-Aborts (g) Kmeans-Execution Time (h) Kmeans-Aborts

(i) Vacation-Execution Time (j) Vacation-Aborts (k) Labyrinth 1-Execution Time (l) Labyrinth 1-Aborts

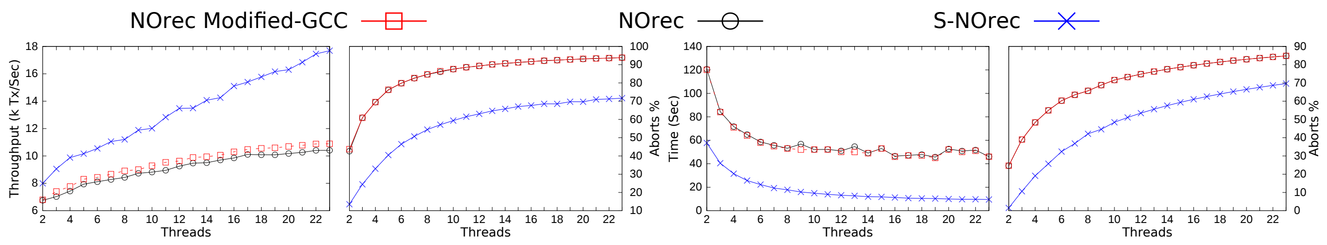
(m) Labyrinth 2-Execution Time (n) Labyrinth 2-Aborts (o) Yada-Execution Time (p) Yada-Aborts

Figure 1: Micro Benchmarks and STAMP Applications using RSTM.

and `SSCA2`) because we found that the semantic operations per transaction were very limited (see Table 3). Hence, there was no difference in both abort rate and throughput in those three applications, which is expected since both the overhead and the potential gain depend on having semantic operations. 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.

<u>`Kmeans`</u> 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 1g). 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 performs slightly worse than NOrec (see Figure 1g) because it adds an overhead that is not exploited to reduce the abort rate (see Figure 1h).

<u>`Vacation`</u> 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



(a) Hashtable-Throughput (b) Hashtable-Aborts (c) Vacation-Execution Time (d) Vacation-Aborts

Figure 2: Hashtable and STAMP Vacation using GCC.

red-black tree operations. Additionally, almost all the `inc` 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.

<u>`Labyrinth`</u> 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 in a shared output grid. Different checks along the routing path (e.g., *isEmpty*, *isGarbage*) were transformed into semantic `cmp` operations, which allowed S-TL2 to outperform TL2 by 20%-50% speedup and to save half of the aborts (see Figures 1k & 1l). Both S-NOrect and NOrec perform similarly, which indicates that transactions that fail in NORec's value-based validation also fail in S-NOrec's semantic validation. In [32], an optimized version of Labyrinth was proposed, where some non-transactional operations (memory copy) are moved outside the transaction, which in effect reduces the transaction size. Figures 1m & 1n show the performance in this new version. Although S-TL2 still has lower abort rate, the returned gain in performance became insignificant because most of the work was moved outside transactions.

<u>`Yada`</u> 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, throughput was not affected (Figures 1o & 1p). Our measurements 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 (since transactions eventually aborted). This is similar to what happened in Bank.

7.2 GCC-based implementations

Now we discuss the results of the above experiments using our modified GCC instead of RSTM. As mentioned in Section 6, in these experiments we focus on NOrec and S-NOrec. We also added one more version of NOrec that uses our modified GCC API but does not handle them semantically. The only difference between this version and NOrec is that, in the former, the applications calls our semantic API and internally delegates them to the normal reads/writes of NOrec, while the latter calls NOrec's API directly.

The results in Figure 2 follow the same trends described above with RSTM (due to space limitations, we move some results to the technical report [34]). As before, our semantic extensions help improve performance in all benchmarks. Compared to RSTM, the actual throughput values decreased. This is mainly because GCC speculates every read and write within the `_transaction_atomic` blocks, while RSTM speculates only addresses accessed using its transactional `TM_READ` and `TM_WRITE` APIs. However, GCC algorithms scale better than RSTM algorithms, mainly because of the internal optimizations in GCC, such as using more efficient structures to store and handle metadata. Interestingly, using our modified GCC, even without exploiting semantics ("NOrec Modified-GCC"), we observe some performance improvement due to decreasing the overall number of TM calls.

8. CONCLUSIONS

In this paper, we show that generality of TM does not always contradict application semantics. We did so by identifying *TM-friendly* semantics and proposing an approach to inject them in current TM algorithms and frameworks. We also integrated our work in GCC and provide full compiler support for them. Our experimental results depicted a promising improvement over the base algorithms. We plan to extend this line of research by investigating more on including HTM algorithms and supporting more complex semantic patterns.

9. ACKNOWLEDGMENTS

Authors would like to thank Marina Sadini and Yuzhong Wen for their support in integrating our extensions into GCC. We also thank our anonymous reviewers for their valuable comments. This work is partially supported by Air Force Office of Scientific Research (AFOSR) under grant FA9550-14-1-0187.

10. REFERENCES

[1] Intel transactional memory compiler and runtime application binary interface. https://software.intel.com/sites/default/files/m/5/a/2/a/f/8097-Intel_TM_ABI_1_0_1.pdf, 2008.

[2] A.-R. Adl-Tabatabai, T. Shpeisman, and J. Gottschlich. Draft specification of transactional language constructs for c++, 2009.

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

[4] Y. Afek, A. Matveev, O. R. Moll, and N. Shavit. Amalgamated lock-elision. In *DISC*, pages 309–324, 2015.

[5] H. Avni and B. C. Kuszmaul. Improving HTM scaling with consistency-oblivious programming. In *TRANSACT*, 2014.

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

[7] 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 *ISCA*, pages 225–236, 2013.

[8] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. In *TRANSACT*, 2014.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL*, pages 25–35, 1989.

[10] 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 *ASPLOS*, pages 39–52, 2011.

[11] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *PPoPP*, pages 67–78, 2010.

[12] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, pages 194–208, 2006.

[13] N. Diegues and P. Romano. Self-tuning Intel transactional synchronization extensions. In *ICAC*, pages 209–219, 2014.

[14] A. Dragojevic and T. Harris. STM in the small: trading generality for performance in software transactional memory. In *EuroSys*, pages 1–14, 2012.

[15] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC*, pages 93–107, 2009.

[16] G. Golan-Gueta, G. Ramalingam, M. Sagiv, and E. Yahav. Automatic scalable atomicity via semantic locking. In *PPoPP*, pages 31–41, 2015.

[17] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP*, pages 175–184, 2008.

[18] A. Hassan, R. Palmieri, and B. Ravindran. Optimistic transactional boosting. In *PPoPP*, pages 387–388, 2014.

[19] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *PPoPP*, pages 207–216, 2008.

[20] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, 2003.

[21] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[22] W. N. S. III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC*, pages 240–248, 2005.

[23] V. Luchangco, M. Wong, H. Boehm, J. Gottschlich, J. Maurer, P. McKenney, M. Michael, M. Moir, T. Riegel, M. Scott, et al. Transactional memory support for c++. 2014.

[24] 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 *TRANSACT*, 2006.

[25] A. Matveev and N. Shavit. Reduced hardware norec: A safe and scalable hybrid transactional memory. In *ASPLOS*, pages 59–71, 2015.

[26] 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*, pages 314–325, 2008.

[27] J. Merrill. Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC DevelopersâĂŹ Summit*, pages 171–179, 2003.

[28] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, pages 35–46, 2008.

[29] 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 *PPoPP*, pages 68–78, 2007.

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

[31] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *SPAA*, pages 53–64, 2011.

[32] W. Ruan, Y. Liu, and M. Spear. Stamp need not be considered harmful. In *TRANSACT*, 2014.

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

[34] M. M. Saad, R. Palmieri, A. Hassan, and B. Ravindran. Extending TM primitives using low level semantics. Technical report, ECE Dept., Virginia Tech, May 2016. www.hyflow.org/pubs/spaa16-saad-TR.pdf.

[35] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[36] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. In *SPAA*, pages 275–284, 2008.

[37] 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.

[38] L. Xiang and M. L. Scott. Software partitioning of hardware transactions. In *PPoPP*, pages 76–86, 2015.