Invyswell: A Hybrid Transactional Memory for Haswell's Restricted Transactional Memory

Irina Calciu
Brown University
Providence, RI, USA
irina@cs.brown.edu

Justin Gottschlich
Intel Labs
Santa Clara, CA, USA
justin.e.gottschlich@intel.com

Tatiana Shpeisman
Intel Labs
Santa Clara, CA, USA
tatiana.shpeisman@intel.com

Gilles Pokam
Intel Labs
Santa Clara, CA, USA
gilles.a.pokam@intel.com

Maurice Herlihy
Brown University
Providence, RI, USA
mph@cs.brown.edu

ABSTRACT

The Intel Haswell processor includes restricted transactional memory (RTM), which is the first commodity-based hardware transactional memory (HTM) to become publicly available. However, like other real HTMs, such as IBM's Blue Gene/Q, Haswell's RTM is best-effort, meaning it provides no transactional forward progress guarantees. Because of this, a software fallback system must be used in conjunction with Haswell's RTM to ensure transactional programs execute to completion. To complicate matters, Haswell does not provide escape actions. Without escape actions, non-transactional instructions cannot be executed within the context of a hardware transaction, thereby restricting the ways in which a software fallback can interact with the HTM. As such, the challenge of creating a scalable hybrid TM (HyTM) that uses Haswell's RTM and a software TM (STM) fallback is exacerbated.

In this paper, we present Invyswell, a novel HyTM that exploits the benefits and manages the limitations of Haswell's RTM. After describing Invyswell's design, we show that it outperforms NOrec, a state-of-the-art STM, by 35%, Hybrid NOrec, NOrec's hybrid implementation, by 18%, and Haswell's hardware-only lock elision by 25% across all STAMP benchmarks.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Algorithms; Design

Keywords

Hybrid Transactional Memory; Transactional Memory; Intel Haswell Restricted Transactional Memory

PACT'14, August 24–27, 2014, Edmonton, AB, Canada.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2809-8/14/08 ...$15.00.
http://dx.doi.org/10.1145/2628071.2628086 .

1. INTRODUCTION

Traditionally, locks have been the predominant mechanism used to synchronize shared memory in multithreaded programs [16]. Yet, developing software that correctly and efficiently uses locks is notoriously challenging, even for the most seasoned programmers. Transactional memory (TM) has been proposed as an alternative to locks, where much of the mechanical complexity of synchronization is managed by the underlying system, not the programmer [15, 30].

Experience with software transactional memory (STM), where transactions are implemented entirely in software, has demonstrated the simplicity of transactional programming, but has raised challenging performance issues. Modern STMs tend to be scalable at high thread counts [9], meaning that beyond a certain point (and up to a limit), adding more threads typically increases throughput for many benchmarks, yielding performance that is often competitive with fine-grained locking. Unfortunately, these STMs tend to perform poorly at low or medium thread counts, because of non-amortized transactional overhead, resulting in performance that is not competitive with fine-grained locking.

To improve the performance of transactions, hardware vendors such as Intel and IBM have included support for hardware transactional memory (HTM). One such example is Intel's Haswell processor [18], which includes restricted transactional memory (RTM), a cache-based HTM design that uses the microarchitecture's existing cache coherence protocol to manage transactional conflicts. Yet, it is unclear how RTM can be most effectively used by software. One cannot simply substitute hardware transactions for software transactions, because RTM, like other HTMs, such as IBM's Blue Gene/Q [31] and System z [19], is *best-effort*, providing no progress guarantees. [1] Whether a transaction succeeds depends on whether its data set fits in the processor's cache, whether the transaction finishes without interruption, and a myriad of other architectural and platform-specific limitations best hidden from the programmer.

It has been recognized that effectively integrating best-effort HTM with the software that uses it requires an intermediate software fallback when hardware transactions fail. Such a system is called hybrid transactional memory (HyTM) [6,

[1] Although System z supports constrained transactions, which are guaranteed to commit, we believe this does not present a generalized mechanism for HTM forward progress as constrained transactions are size-restricted.

4, 7, 22], where hardware and software transactions execute under the umbrella of a single TM system. In this paper, we present a novel HyTM, called *Invyswell*, that uses hardware transactions from Haswell's RTM in conjunction with software transactions from a heavily modified design of InvalSTM [11], an STM designed to provide scalability and performance for large transactions with notable contention.

Invyswell enables the concurrent execution of both hardware and software transactions with the aim of being performant for all transaction sizes and degrees of contentions. Haswell's RTM performs best for small transactions with low contention, as it imposes no instrumentation overhead, but is limited to a "requester-wins" contention policy. InvalSTM performs best for large transactions with high contention, because it can make highly informed contention management decisions through its commit-time invalidation process. Yet, challenges remain in finding an efficient solution for the "transactional twilight zone" - midsize transactions that are small enough to successfully execute in hardware but have a non-trivial degree of contention. Furthermore, even after designing a TM that addresses the unique challenges of each of these categories, that system must ensure that each individual component does not negatively impact the overall performance by mismanaging transactions for which it was not intended. Invyswell addresses this by using a sophisticated design that employs several hardware and software modes of execution. This gives the system the flexibility to trade execution overhead for precision in conflict detection.

Haswell's RTM does not support *escape actions*, non-transactional instructions executed within transactions [24]. This limitation complicated our design, especially with respect to opacity [12], a correctness conditions that guarantees consistency of eventually-aborted transactions. Another challenge we encountered was designing Invyswell's *contention manager* (CM), a decision-making process aimed at improving throughput, due to the different isolation properties for hardware and software transactions. The lack of escape actions further complicated this issue, as well, as it restricts the way a hardware transaction can abort a software transaction before the hardware transaction itself commits.

We evaluate Invyswell's performance using the STAMP benchmark suite. Invyswell's performance compares favorably to that of pure software, pure hardware, and hybrid solutions. Invyswell is 35% faster than NOrecSTM [5], a state-of-the-art software transactional memory, and 18% faster than NOrecHy [4], a state-of-the-art hybrid transactional memory, as shown in Figure 1. It also outperforms Haswell's native *hardware lock elision* (HLE) [17, 25], a hardware mechanism that attempts to elide locks by executing critical sections as transactions and supports transactional re-execution with single global lock fallback implemented purely in hardware. Although on the average Invyswell is only 25% faster than HLE, the performance difference is significant for some benchmarks with large transactions, where Invyswell outperforms HLE by 2× to 5.4×.

This paper makes the following contributions:

1. We present a hybrid transactional memory for Haswell RTM, called Invyswell, which features a novel design based on InvalSTM and an adaptive system supporting five transaction types.
2. We propose a novel method for precise conflict detection between hardware and software transactions using Bloom filters, which works well even for HTMs *without* support of escape actions.

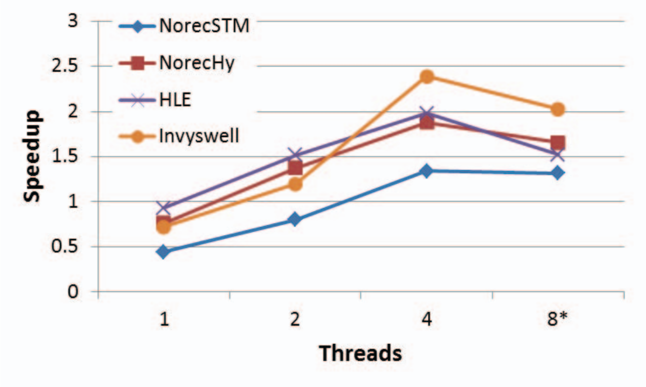


Figure 1: STAMP Performance Differential by Geometric Mean. *Hyperthreading is enabled for 8 threads. (Note: NOrec and Hybrid NOrec are abbreviated as NorecSTM and NorecHy, respectively, in the legend)

3. We demonstrate that achieving the best performance requires HyTM even in presence of HLE. Our evaluation on the STAMP benchmark suite shows that Invyswell outperforms HLE by up to 3.5× (25% on average) and state-of-the-art software and hybrid solutions by more than 18%.
4. We describe challenges and design tradeoffs for a Haswell HyTM, paving the way for even better systems in the future.

2. RELATED WORK

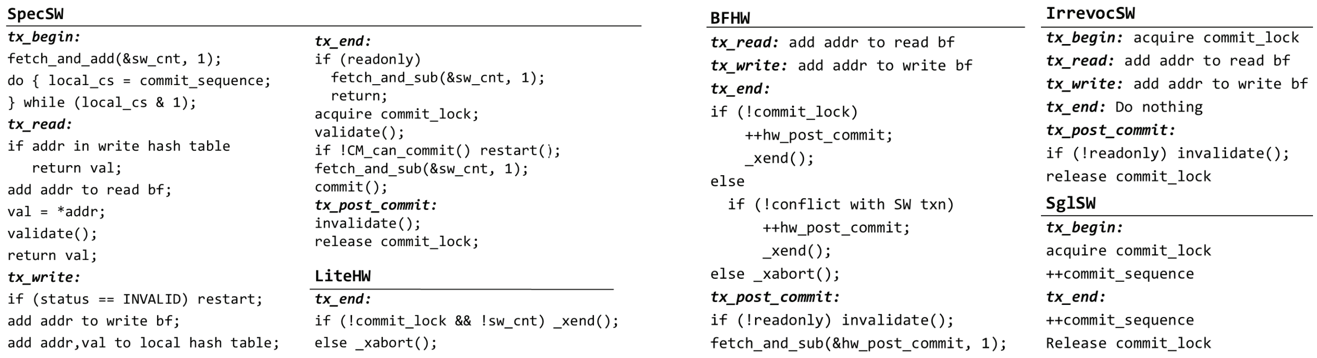
TM systems [13] fall into three rough categories: *software* (STM), *hardware* (HTM), and *hybrid* (HyTM). Most of the research literature concerns STM systems [1, 8, 11, 14, 23, 26, 29]. There are too many such systems to discuss them individually, but in this paper, we compare our HyTM to NOrec [5], a state-of-the-art STM that uses value-based validation, deferred update and lazy conflict detection.

Recently, Intel [18] and IBM [31, 19] announced new processors with hardware support for transactions, and it seems likely that others will follow. Like Herlihy and Moss's original TM proposal [15], these systems rely on modified cache coherence protocols to achieve atomicity and isolation. Haswell also supports *hardware lock elision* [25], a scheme where annotated lock-based critical sections are executed speculatively, but are retried pessimistically if speculation fails.

HyTM schemes promise to provide the best of both worlds: the efficiency of HTM with the scalability of STM. The first papers to articulate this point are from Damron et al. [6] and Kumar et al. [21]. Later work in this area includes PhTM [22], intended for Sun's Rock architecture and Riegel et al.'s work [27] intended for AMD's proposed Advanced Synchronization Facility (ASF). More recently, Wang et al. [31] proposed a HyTM for IBM Blue Gene/Q's best-effort HTM, based on a Single-Global-Lock fallback. In this paper, we compare Invyswell to Hybrid NOrec [4], a state-of-the-art hybrid transactional memory.

3. OVERVIEW OF INVALSTM

One of the key differences between InvalSTM and other STMs is that it performs commit-time invalidation [11]. This approach requires that a transaction identify and resolve



```
SpecSW
tx_begin:
fetch_and_add(&sw_cnt, 1);
do { local_cs = commit_sequence;
} while (local_cs & 1);
tx_read:
if addr in write hash table
   return val;
add addr to read bf;
val = *addr;
validate();
return val;
tx_write:
if (status == INVALID) restart;
add addr to write bf;
add addr,val to local hash table;

tx_end:
if (readonly)
  fetch_and_sub(&sw_cnt, 1);
  return;
acquire commit_lock;
validate();
if !CM_can_commit() restart();
fetch_and_sub(&sw_cnt, 1);
commit();
tx_post_commit:
invalidate();
release commit_lock;
```

```
LiteHW
tx_end:
if (!commit_lock && !sw_cnt) _xend();
else _xabort();
```

(a) SpecSW and LiteHW Events.

```
BFHW
tx_read: add addr to read bf
tx_write: add addr to write bf
tx_end:
if (!commit_lock)
    ++hw_post_commit;
    _xend();
else
  if (!conflict with SW txn)
      ++hw_post_commit;
      _xend();
else _xabort();
tx_post_commit:
if (!readonly) invalidate();
fetch_and_sub(&hw_post_commit, 1);
```

```
IrrevocSW
tx_begin: acquire commit_lock
tx_read: add addr to read bf
tx_write: add addr to write bf
tx_end: Do nothing
tx_post_commit:
if (!readonly) invalidate();
release commit_lock
```

```
SglSW
tx_begin:
acquire commit_lock
++commit_sequence
tx_end:
++commit_sequence
Release commit_lock
```

(b) BFHW, IrrevocSW, and SglSW Events.

Figure 2: Transactional Events for Invyswell's Five Different Transaction Types.

conflicts with all other in-flight (i.e., concurrently executing) transactions during its commit phase. InvalSTM achieves this by storing read and write sets in transaction-specific Bloom filters so it can perform conflict detection using constant-time set intersection. With commit-time invalidation, InvalSTM has complete knowledge of all conflicts between a committing transaction and other in-flight transactions, allowing it to make informed decisions on how to best mitigate contention. *All* InvalSTM transactions perform validation to achieve opacity in $O(N)$ total computational complexity, where N is the number of read elements, which is notably faster than the $O(N^2)$ overhead incurred by incremental validation and can drastically reduce the opacity cost for large transactions. Additionally, read-only transactions commit without incurring any commit-time serialization overhead.

For these reasons, InvalSTM naturally complements Haswell's RTM. Haswell's RTM can be used for small transactions and low thread counts, while InvalSTM can be used for large transactions and high thread counts. Moreover, Haswell's RTM can leverage InvalSTM's use of Bloom filters for conflict detection by augmenting Haswell's hardware transactions with Bloom filters to enable many hardware transactions to execute concurrently with many software transactions. These Bloom filters are a good fit for Haswell's cache-based HTM design because they can be structured for constant-sized cache line alignment, thereby minimizing the negative impact of introducing hardware-to-software conflict detection into an already restricted HTM space. Finally, because InvalSTM's read-only transactions do not introduce any serialization in their execution, the performance overhead for transactions is transparent to Haswell RTM's faster executing hardware transactions. This enables Haswell's RTM to perform without interference when read-only software transactions are executing within InvalSTM, regardless of their size.

4. INVYSWELL'S DESIGN

In this section, we describe Invyswell, a HyTM that supports the concurrent execution of multiple hardware and multiple software transactions while guaranteeing forward progress. Invyswell uses Haswell's RTM [18] and a modified version of InvalSTM [11]. In InvalSTM, when a transaction is ready to commit, it marks conflicting in-flight transactions as invalid. InvalSTM uses Bloom filters for fast conflict detection between software transactions. Invyswell also uses Bloom filters at times, but not always, for conflict detection between hardware and software transactions.

Because Haswell's RTM does not support escape actions, the communication between in-flight hardware and software transactions is essentially impossible without introducing conflicts between them. For example, if a software transaction writes to memory shared by a hardware transaction, the latter will abort. Yet, communication between hardware and software transactions might be useful to improve the precision of conflict detection between them, thereby increasing throughput in cases when conflicts do not occur.

To manage this space, Invyswell generally performs conflict detection between a hardware and a software transaction *after* the hardware transaction has committed. This enables increased throughput in cases where no conflicts exist while minimizing the chance of aborting a hardware transaction because of communication with in-flight software transactions.

Furthermore, Invyswell exploits the observation that hardware transactions do not need to check for conflicts with software transactions until just before committing, a mechanism called *lazy subscription*, which was introduced by Dalessandro et al. in their NOrec HyTM system [4]. By using lazy subscription, Invyswell reduces the "window of vulnerability" in which a write to a software transaction's conflict detection metadata (e.g., its read set, its execution lock, etc.) will abort a non-conflicting, in-flight hardware transaction.

Invyswell supports five transactions types, motivated by the need for progress guarantees and adaptability to different types of workloads. Two types are in hardware, lightweight (LiteHW) and bloom filter-based (BFHW), and three types are in software, speculative (SpecSW), irrevocable (IrrevocSW), and single global lock (SglSW). The pseudocode for these transaction is shown in Figure 2. Invyswell's state transitions between them are shown in Figure 3.

4.1 SpecSW: An HTM-Friendly InvalSTM

Invyswell's first type of transaction is the speculative software transaction (SpecSW), which is similar to an InvalSTM transaction, and is shown in Figure 4. It tracks its read and write locations in transaction-specific Bloom filters and stores its write set's values in a hash table for deferred update during its commit phase. Note that a memory barrier is necessary after inserting a memory location in a read bloom

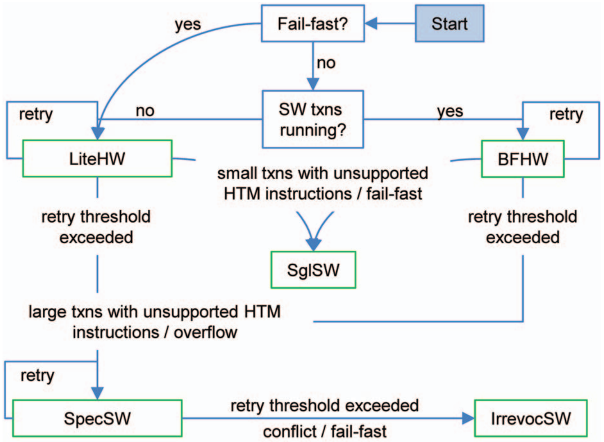


Figure 3: Invyswell's State Machine Describing the Transitions Between the Different Transaction Types.

filter and before reading the value from memory. At commit-time, a SpecSW performs invalidation, where it compares its write Bloom filter against all other in-flight SpecSWs' read Bloom filters. If a conflict is found, it consults the contention manager (CM) on how to proceed. The CM then either aborts the committing transaction or permits it to commit. If permitted to commit, the SpecSW transaction updates all write locations and then marks all conflicting in-flight transactions as invalid. During a SpecSW's execution, it checks to see if it has been marked as invalid prior to each read and write and prior to committing. If it has, it aborts and it retries again as a SpecSW or another type as illustrated in Figure 3.

A key difference between Invyswell and InvalSTM is that SpecSWs perform invalidation *after* committing changes to memory, unlike InvalSTM, which performs invalidation *before*. The reason for doing this is the following. In InvalSTM, new transactions acquire an in-flight lock to insert their transaction ID into an in-flight linked list. If Invyswell did the same, hardware transactions would have to read this lock before committing, to ensure correctness in their conflict detection. However, reading such a lock could subsequently cause many unnecessary hardware transaction aborts because whenever a new SpecSW was added to the list the in-flight lock would be acquired, automatically aborting all hardware transactions that previously read it.

To avoid this behavior, Invyswell performs invalidation after committing SpecSW's changes to memory and uses a slotted array for the in-flight SpecSWs, rather than a linked list. The combination of these changes results in Invyswell's elimination of the InvalSTM in-flight lock, thereby reducing the likelihood of unnecessary hardware transaction aborts. Instead, if a new transaction starts while the committing transaction is updating memory, it will be detected by the invalidation phase of the committing transaction, which will follow the memory update phase. Alternatively, if the new transaction starts after the memory was already updated, it could be missed by the invalidation phase. However, this new transaction is guaranteed to only read consistent states because the committing transaction has finished updating the memory, making the bloom filter check unnecessary for this transaction.

Initially, this modification results in the loss of opacity for SpecSWs, however, we restore opacity for SpecSWs by adding inexpensive validation to each read as described in Section 4.7. This change makes SpecSWs compatible with hardware transactions that can invalidate in-flight SpecSWs and it permits Invyswell to eliminate the need for an in-flight lock and the per-transaction locks that are required by InvalSTM.

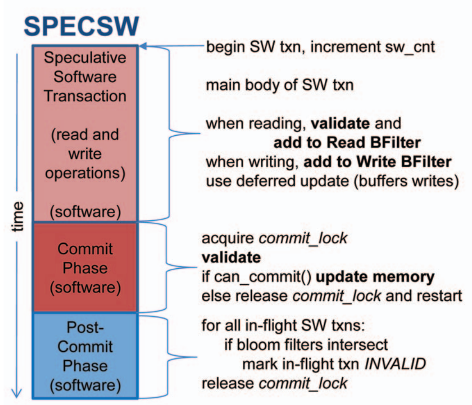

SPECSW
time
Speculative Software Transaction
(read and write operations)
(software)
Commit Phase (software)
Post-Commit Phase (software)
begin SW txn, increment sw_cnt
main body of SW txn
when reading, validate and add to Read BFilter
when writing, add to Write BFilter
use deferred update (buffers writes)
acquire commit_lock
validate
if can_commit() update memory
else release commit_lock and restart
for all in-flight SW txns:
if bloom filters intersect
mark in-flight txn INVALID
release commit_lock

Figure 4: Speculative Software Transaction (SpecSW).

4.2 BFHW: Hardware-Software Conflict Detection

Invyswell's second type of transaction is the Bloom filter hardware transaction (BFHW). BFHWs execute in hardware and, like SpecSWs, record the memory locations they read and write in transaction-specific software Bloom filters.

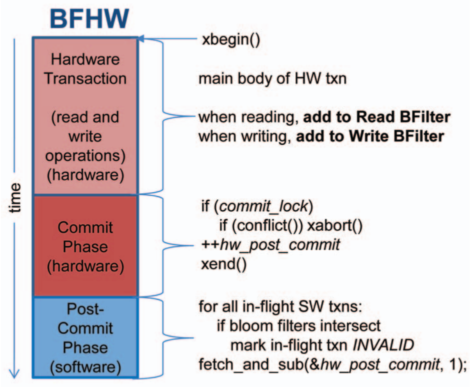
At commit time, if a BFHW sees the software `commit_lock` is free, it increments the `hw_post_commit` counter, which subsequently prevents SpecSWs from committing or reading new values while its value is non-zero, and then commits its speculative writes to memory and performs post-commit invalidation on all in-flight SpecSWs, where all conflicting transactions are marked as invalid. The BFHW then decrements the `hw_post_commit` counter to indicate its post-commit phase has completed, allowing software transactions to again commit, as shown in Figure 5.

The `hw_post_commit` counter is necessary because there is a window of vulnerability after a BFHW has committed, but before it has finished executing the invalidation phase, when SpecSWs can read inconsistent values written by the BFHW. Without the `hw_post_commit` counter these SpecSWs will be marked as *invalid* by the BFHW during its invalidation phase, but they could still execute momentarily returning inconsistent reads, causing SpecSWs to lose their opacity.

Alternatively, if the `commit_lock` is taken when a BFHW enters its commit phase, this means a SpecSW is committing. In this scenario, the simplest option is for the BFHW to abort, because there may be a conflict with the committing SpecSW. However, because BFHWs track their read and write accesses, Invyswell can instead perform conflict detection between the committing BFHW and the committing SpecSW via Bloom filter set intersection. If an overlap is found, the BFHW is aborted. Otherwise, no conflict exists between the BFHW and the SpecSW, and, because their respective read and write sets are immutable during their commit phases, the BFHW is permitted to commit.

When a SpecSW commits, it releases the `commit_lock` before clearing its read and write sets. This ensures that if the SpecSW commits before a committing BFHW performs conflict detection against the committing SpecSW, that the BFHW is automatically aborted because the write performed by the SpecSW to the `commit_lock` would trigger a hardware conflict with the BFHW from its prior read.

Note that if a BFHW transaction aborts, its `hw_post_commit` counter increment never becomes visible, because it is part of its speculative write set. Moreover, the new counter value becomes visible only when the hardware transaction commits. If a SpecSW reads this counter after it has been written to by a BFHW, but before the BFHW has committed, the BFHW will be automatically aborted by Haswell RTM's strong isolation property, thereby avoiding a race.


BFHW
Hardware Transaction
(read and write operations) (hardware)
Commit Phase (hardware)
Post-Commit Phase (software)
time
xbegin()
main body of HW txn
when reading, add to Read BFilter
when writing, add to Write BFilter

```
if (commit_lock)
   if (conflict()) xabort()
++hw_post_commit
xend()
```

```
for all in-flight SW txns:
   if bloom filters intersect
      mark in-flight txn INVALID
fetch_and_sub(&hw_post_commit, 1);
```

Figure 5: Bloom Filter Hardware Transaction (BFHW).

4.3 LiteHW: Optimizing for Small Transactions

Although BFHWs enable the concurrent execution of hardware and software transactions, they come with added overhead because each load and store requires an associated Bloom filter insert operation. Invyswell addresses this limitation with its third type of transaction, the LiteHW.

LiteHWs are lightweight hardware transactions, which execute without read or write annotations. They can only commit if there are no in-flight software transactions when they begin their commit phase. Unfortunately, because LiteHWs do not maintain read or write set metadata, if a software transaction is in-flight when a LiteHW enters its commit phase, Invyswell must assume a conflict exists between the LiteHW and the software transaction and, therefore, must abort the LiteHW. LiteHWs determine if there is an in-flight software transaction by reading the `commit_lock` and the software transaction counter, `sw_cnt`, prior to committing. Because LiteHWs do not perform conflict detection against software transactions, they require no post-commit phase.

4.4 IrrevocSW: Progress Guarantees

InvalSTM guarantees forward progress by using transaction-specific priorities that are incremented each time a transaction is aborted. Using this mechanism, a continuously aborted transaction will eventually yield the highest priority and is guaranteed to commit. Invyswell's BFHWs, however, deviate from this model and instead commit memory changes first and perform invalidation second, at which point all conflicting software transactions are aborted. Because of this change, there is a danger that BFHWs could repeatedly abort high-priority SpecSWs, resulting in their starvation.

To address this problem, Invyswell introduces a fourth transaction type, the IrrevocSW, a direct update irrevocable transaction type that cannot be aborted. To ensure irrevocability, IrrevocSWs acquire the `commit_lock` as soon as they begin their execution and hold it until they have committed. To enable conflict detection with other transactions, an IrrevocSW transactions records its read and write locations in Bloom filters. An IrrevocSW needs no commit phase, because its writes are in-place. Its post-commit phase invalidates conflicting in-flight SpecSWs. While an IrrevocSW is executing, SpecSWs are required to perform validation and are disallowed from committing. Furthermore, LiteHW transactions must abort if their commit phase overlaps with any part of an IrrevocSW's execution. However, BFHWs can execute concurrently with an IrrevocSW. Yet, to ensure correctness, a BFHW needs to check for conflicts with the IrrevocSW transaction prior to committing its changes to memory and it must abort itself if a conflict is found.

4.5 SglSW: Progress Guarantees with Reduced Overhead

Small transactions that execute instructions not supported by Haswell's RTM need to be executed in software. However, both SpecSWs and IrrevocSWs add transactional metadata that may be too expensive for transactions that only access a few memory elements. To address this need, Invyswell employs a final transaction type that uses a single global lock without any associated transactional metadata.

This transaction type, SglSW, uses direct update and is irrevocable. SglSW is fast, but it does not allow the concurrent execution of other software transactions. Because SglSW does not track its reads or writes, it cannot perform conflict detection. Instead, it uses a sequence lock to force all in-flight SpecSWs to abort and acquires the `commit_lock` when it begins its execution to prevent IrrevocSWs from starting. BFHW and LiteHW transactions abort if an SglSW is executing when they try to commit. However, SglSWs allows for some overlap in execution with BFHWs and LiteHWs, as long as the executing SglSW commits before the hardware transactions do, thereby ensuring that the hardware's strong isolation property aborts any BFHWs and LiteHWs that conflict with the SglSW.

4.6 Transitioning Between Transaction Types

Transactions are scheduled opportunistically, first as fast, high-risk hardware transactions, then as slower, low-risk software transactions as shown in Figure 3. Each transaction is first tried in hardware, as LiteHW or BFHW, depending on whether other software transactions are present. If the hardware abort status suggests that a transaction is unlikely to succeed in hardware, then it is retried as a SpecSW. If it fails again, it is either retried as a SpecSW or it is escalated to irrevocable status, preventing it from aborting and ensuring progress. The transitions between the different types are decided automatically at runtime based on a heuristic that is application-independent. [2]

[2] Due to limitations in Intel's first generation HTM (e.g., imprecision on a transaction's abort status and limitations of only four concurrent hardware threads, eight with hyperthreading) Invyswell's state transitions deviate slightly from that shown in Figure 3. In particular, we use a modified design that transitions to SglSW when SpecSWs fail.

4.7 SpecSW Validation

InvalSTM performs invalidation before committing a transaction's writes to memory. It uses a per-transaction `invalid` flag which is set to true when a committing transaction invalidates a conflicting in-flight transaction. For reasons described in Section 4.1, Invyswell departs from this design and performs invalidation after committing a SpecSW's writes to memory. Unfortunately, this change makes InvalSTM's approach to ensure opacity – using the transaction's `invalid` flag – insufficient for SpecSWs. Instead, on every new read that is not present in a SpecSW's write set, Invyswell inserts the new read location into the SpecSW's Bloom filter and only then is the SpecSW permitted to read the value. This ensures that a potential conflict will not be missed by another transaction's invalidation phase. Next, the SpecSW performs the validation process shown in Figure 6. This validation process is necessary because of the interactions SpecSWs can have with different transactions and the inconsistent reads they might cause, as we explain next.

SglSW.

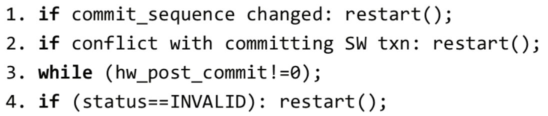
First, a SpecSW read could be inconsistent due to a concurrently executing SglSW. Because SglSWs do not store reads and writes using Bloom filters, conflict detection cannot be performed between them and a SpecSW. Thus, the SpecSW must abort if the `commit_sequence` has changed (Line 1 in Figure 6) after it was read at `tx_begin` (Figure 2).

IrrevocSw.

Second, a concurrently executing IrrevocSW or a committing SpecSW could cause an inconsistent read. Thus, the SpecSW read must check if the read location is in the Bloom filter of the transaction holding the `commit_lock` (Line 2 in Figure 6). If so, it must abort. If `commit_lock` changes during the read validation, the conflict may go unnoticed by the validation code. However, if the lock has changed, it means the transaction that released it must have finished the invalidation phase. Therefore, it is sufficient to check if the SpecSW has been invalidated in the meantime (Line 4 in Figure 6).

BFHW.

Finally, a SpecSW must wait for all committed BFHWs to finish invalidation (`hw_post_commit` to reach zero) before using a new read value (Line 3 in Figure 6). If the SpecSW is not marked as invalid, the read is safe (Line 4 in Figure 6).



```
1. if commit_sequence changed: restart();
2. if conflict with committing SW txn: restart();
3. while (hw_post_commit!=0);
4. if (status==INVALID): restart();
```

Figure 6: Overview of Invyswell's SpecSW Validation Process.

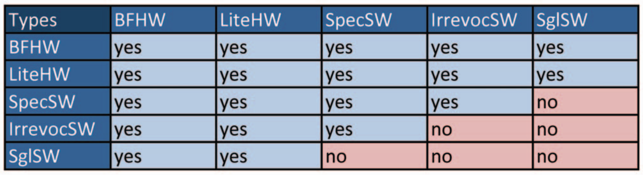
4.8 Contention Manager (CM)

SpecSWs consult the CM during the commit phase to acquire permission to commit. As in InvalSTM, the CM considers all in-flight transactions that would be aborted if the committing transaction was allowed to commit. Any CM policy can be used. Invyswell uses iBalanced [10], which makes decisions based on priority, read and write set sizes, and other factors.

Invyswell has trade-offs that the original InvalSTM design does not have. For example, InvalSTM's ability to make decisions based on complete knowledge of in-flight transactions is lost. Essentially, there is no CM for Invyswell's hardware transactions because Haswell's RTM does not support escape actions, and thus a hardware transaction has to abort all conflicting software transactions after the hardware transaction has committed. The side-effect of this approach is that, conceptually, hardware transactions are likely to scale to high thread counts only when there is little to no contention, even if mitigation of that contention could be possible with an intelligent CM. On the other hand, software transactions retain a complete knowledge of the CM decision-making process, enabling them to scale for high thread counts amidst high contention when the contention can be managed to provide wide transactional throughput.

5. CORRECTNESS

Figures 2 and 3 show the five types of Invyswell transactions and the transitions between them, respectively. In this section, we give an informal explanation why these five transaction types can run concurrently with one another without violating atomicity, as shown in Figure 7. However, atomicity by itself does not guarantee that aborted transactions are opaque; that is, that they only observe consistent states, a topic we discuss in Section 5.1.



Types	BFHW	LiteHW	SpecSW	IrrevocSW	SglSW
BFHW	yes	yes	yes	yes	yes
LiteHW	yes	yes	yes	yes	yes
SpecSW	yes	yes	yes	yes	no
IrrevocSW	yes	yes	yes	no	no
SglSW	yes	yes	no	no	no

Figure 7: Invyswell's Concurrent Execution Matrix.

LiteHW and BFHW vs. LiteHW and BFHW.

Haswell's hardware transactions are *strongly isolated*, meaning that their changes to memory become visible to other threads only on commit, whether those threads are executing a transaction or not. The hardware automatically detects conflicts between these types of transactions, and any conflict will abort at least one transaction. There is no need for additional mechanisms to synchronize concurrently executing LiteHWs and BFHWs with respect to each other.

LiteHW vs. Software Transactions.

LiteHWs can execute concurrently with Invyswell's software transactions, but they cannot commit while such software transactions are executing. A LiteHW that overlaps execution with a software transaction (SpecSW, IrrevocSW, or SglSW) can commit only after the software transaction has committed, otherwise the resulting execution may be not serializable. A LiteHW that tries to commit while a software transaction is executing will abort. Such behavior is detected by the `sw_cnt` counter and the commit lock (see Figure 2).

BFHW vs. SpecSW or IrrevocSW.

Unlike LiteHWs, BFHWs use software Bloom filters to keep track of the memory locations they access. By performing explicit conflict detection with these Bloom filters, BFHWs can commit in the presence of software transactions. If a committing SpecSW conflicts with an in-flight BFHW, then the BFHW will automatically be aborted by the hardware when the SpecSW writes its speculative data to memory. If a committing BFHW conflicts with an in-flight SpecSW, the SpecSW will be aborted during the BFHW's post-commit invalidation phase. Moreover, BFHWs' use of lazy subscription means it is sufficient to compare the Bloom filters of BFHWs and SpecSWs at the end of the hardware transaction.

Postponing conflict detection to the end of the BFHW's execution narrows the window in which it will be aborted by false conflicts. Moreover, SpecSWs' Bloom filters do not change while it is committing, so a BFHW can read them without being aborted due to metadata interference (i.e., non-transactional interference). Note that SpecSWs that are doomed to abort after a BFHW invalidates them could read inconsistent memory before they notice they were aborted, generating faulty behavior. For this reason, atomicity by itself is not the only TM correctness property that Invyswell guarantees, an issue we discuss in Section 5.1.

SpecSW vs. SpecSW.

Conflict detection between multiple SpecSWs uses invalidation. A committing SpecSW checks for conflicts with other in-flight SpecSWs and, if conflicts are found, the committing SpecSW either aborts itself or invalidates the SpecSWs it conflicts with. No SpecSW can commit during another SpecSW's invalidation process because the committing SpecSW holds the commit lock.

IrrevocSW vs. Software Transactions.

An IrrevocSW acquires the commit lock as soon as it becomes active, ensuring that no other software transaction can become irrevocable (i.e., other IrrevocSWs and SglSWs cannot start) or commit. When an IrrevocSW commits, it invalidates in-flight conflicting SpecSWs.

SglSW vs. Everything.

When an SglSW begins, it acquires the commit lock and aborts all other concurrently executing transactions. While it holds that lock, SglSWs and IrrevocSWs are prevented from starting, and LiteHWs and BFHWs cannot commit. The SglSW also updates the `commit_sequence` lock at the transaction's start and end, aborting all concurrently executing SpecSWs and BFHWs.

5.1 Opacity and Sandboxing

Opacity is a correctness property that ensures that aborted transactions do not observe inconsistent states [12]. The principal challenge to achieving opacity for Invyswell occurs when a hardware transaction and a software transaction conflict. Haswell's hardware transactions are strongly isolated, but InvalSTM's software transactions are not, so care must be taken when managing their interaction.

Invyswell's initial modification to InvalSTM's design permits doomed SpecSWs, i.e. SpecSWs that are guaranteed to abort, to observe inconsistent states because committing SpecSWs perform invalidation *after* writing their changes to memory. To prevent these transactions from observing inconsistent states, Invyswell performs validation at commit-time and before each new read as described in Section 4.7.

Unlike SpecSWs, Invyswell's IrrevocSWs and SglSWs cannot observe inconsistent states because these transactions are never aborted and are, therefore, never doomed. Finally, Haswell's shared memory writes executed by a hardware transaction become visible only when the transaction commits, and writes by aborted transactions never become visible. Moreover, Haswell's transactions are (mostly) sandboxed, meaning that faulty behavior caused by inconsistent reads will cause the transaction to abort. Unfortunately, however, there is one leak in the Haswell sandbox, described in detail in the next section.

5.2 Hardware Sandboxing Limitations

For the most part, hardware sandboxing ensures that no consistency violation within a hardware transaction can affect other transactions. There is, however, one vexing "loophole", an unlikely sequence of events in which (1) mutually inconsistent reads cause a spurious memory write, (2) which overwrite an address later used as the target of an indirect jump in that same transaction, (3) thereby causing a jump to a location that happens to contain either an `_xend` (commit transaction) instruction, or immediate data that looks like one. Executing this instruction without the final commit lock check could prematurely commit an inconsistent set of changes.

This hazard, however unlikely, presents a challenge for any HyTM system implemented in an unmanaged language. Broadly speaking, without escape actions, hardware transactions cannot guarantee transactional consistency if they execute concurrently with either in-place update software transactions or with the commit phase of a deferred update software transaction.

To address this hazard, Invyswell's hardware transactions check the `commit_lock` before doing an indirect jump using function pointers. Simple optimizations can reduce the cost of such a policy. For example, there is no need to check the lock if the transaction has an empty write set, because it could not have corrupted the jump address. If a transaction makes multiple indirect jumps, it suffices to check the lock before the first jump, because once read, the `commit_lock` remains in the transaction's read set, and the transaction will be aborted if the lock is changed externally.

In the results presented in Section 7, we performed these optimizations by hand. For some benchmarks, we found that early checking slightly improved performance, probably because transactions with indirect jumps are often longer, hence less likely to succeed in hardware, and more likely to benefit from a quicker fallback to software.

In the long term, there is a trend toward compiler support to help with this issue. The danger posed by indirect jumps in transactions is similar to the danger posed by common security threats such as buffer overflow in general-purpose programs. The security literature has many examples of compiler techniques to protect jump addresses, such as moving vtables and return addresses in a separate memory space [2] marked as read-only. The latest GCC supports security functionality to check vtable integrity.

Static validity checking for function pointers is difficult, in general, but feasible for common special cases, such as initializers. GCC uses devirtualization and inlining for the most likely target for indirect pointers for optimization levels -O2 or higher. When inlining is possible, GCC can make indirect jumps direct. A transactional compiler could be more aggressive about eliminating or protecting indirect jumps.

6. OPTIMIZATIONS

In this section, we describe the modifications that we made to Invyswell's original design to improve its performance. We found these optimizations to be effective for the first-generation Intel Haswell RTM processor, however, some optimizations are designed specifically for performance of low thread counts (as indicated by the * below) and may degrade performance as thread counts increase. As a result, when Intel's RTM scales to higher thread counts, these "low thread count" changes should be eliminated.

Hardware Transactions.

Hardware transactions are retried with exponential backoff. Before starting a hardware transaction, the `commit_lock` and the software transaction counter, `sw_cnt`, are read non-transactionally to increase the likelihood of finding these data cached, and to optimize for the case when only hardware transactions are active.

Validation.

Consider two SpecSWs, T_A and T_B. Assume that T_A has entered its commit phase and T_B is about to validate a read. Furthermore, assume that T_B has higher priority than T_A and that they conflict with one another. When T_B performs its validation, it could notice that T_A has acquired the commit lock and abort because of the conflict it identifies. At the same time, T_A could consult the CM and abort because T_B has a higher priority, resulting in both transactions aborting because of each other. A similar situation could also occur between a committing SpecSW and a committing BFHW.

To avoid such scenarios, we introduce two global flags, `hw_check` and `sw_check`, in addition to the `commit_lock`, to indicate the different phases of a SpecSW's commit phase. At the highest level, these flags are used to ensure that SpecSWs and BFHWs are only aborted by a SpecSW that is guaranteed to commit. These flags change the SpecSW and BFHW commit process in the following way.

At commit, a SpecSW, called T_C, acquires the `commit_lock` and then consults the CM to receive permission to commit. If permitted to commit, T_C sets the `hw_check = true` to signal to BFHWs that it is committing its writes to memory. With this approach, BFHWs only read the `hw_check` flag at commit-time, instead of the `commit_lock`, which ensures that a BFHW can only be aborted by a SpecSW that will eventually commit, rather than reading the `commit_lock`, where a BFHW could be aborted by a SpecSW that has only started its commit phase but may eventually be aborted by the CM.

Next, T_C waits for the `hw_post_commit` counter to reach zero and, once it has, it checks if it was invalidated by a concurrently committing BFHW. If still valid, T_C sets the `sw_check = true`, which informs other SpecSWs about to read new memory to perform conflict detection against T_C's Bloom filters. At this point, T_C and many concurrently reading SpecSWs may perform simultaneous conflict detection on each other. If conflicts are found, the reading SpecSWs are aborted. If no conflicts are found between reading SpecSWs and T_C, the reading SpecSWs subsequently check their `valid` flag to ensure they were not invalidated by T_C, which may have performed conflict detection before the reading SpecSWs had, and subsequently cleared its Bloom filters before the reading SpecSWs could identify conflicts with them. Any reading SpecSWs that are still valid are permitted to continue their execution. Without the `sw_check` flag, the scenario of conflicting transactions T_A and T_B might occur. With it, a reading SpecSW's validation can only fail if it conflicts with a concurrently executing SpecSW that is guaranteed to commit.

**Bloom Filters.*

In principle BFHWs and IrrevocSWs enable more concurrency than LiteHWs or SglSWs, yet, in practice the overhead associated with BFHWs' and IrrevocSWs' Bloom filters can negate their concurrency benefits. This is especially true at low thread counts where there is not enough concurrency to justify such overhead. Because of this, we use SglSWs, rather than IrrevocSWs, as the fallback from SpecSWs for our experiments (see Figure 3), as SglSWs do not employ Bloom filters. However, once RTM becomes available with higher core counts, we plan to reinstate IrrevocSWs as the fallback for SpecSWs because they enable SpecSWs to execute alongside them, while SglSWs do not.

To reduce the overhead of BFHWs, we optimize away their read set Bloom filters. This optimization is possible because BFHWs only invalidate SpecSWs – SpecSWs never invalidate BFHWs – thereby only requiring write-write and write-read conflict detection for BFHWs invalidation phase. [3] However, this change prohibits BFHWs and SpecSWs from committing concurrently, which the original Invyswell design permitted. For low thread counts, however, we have found this change to only positively impact performance. Yet, for higher thread counts, this change will likely degrade performance and, therefore, it would be advisable to revert back to Invyswell's original Bloom filter design.

**Fail-Fast.*

When there is contention, many SpecSWs will repeatedly abort before reaching their retry threshold and falling back to SglSWs. The amount of wasted work that this process can incur could be substantial if contention is consistent, or even bursty, throughout the entire benchmark.

To address this, we add a counter to count the number of high-priority software transactions aborted during the invalidation phase. Whenever a thread notices that this number is over a threshold, it increments a racy shared counter. Once this counter reaches a pre-defined threshold, our optimized system switches to Fail-Fast mode, which only uses LiteHWs and SglSWs. We have found this optimization to be efficient because it identifies the cases when STMs are wasting work with too many retries, which eventually fail to irrevocable mode. In these cases, we have found it is better to use irrevocable software transactions immediately.

Read-Only.

We employed optimizations for both read-only SpecSWs and BFHWs. Read-only SpecSWs can commit when they reach commit phase without acquiring the commit lock, even if they were invalidated. First, the validation process in the read annotations ensures that the transaction's read set was consistent at the time of the last read. Second, read-only SpecSWs, as well as read-only BFHWs do not need to perform invalidation, as they can be serialized before conflicting in-flight software transactions.

[3] BFHWs can be aborted by other hardware transactions, but that is handled automatically by the hardware.

7. EXPERIMENTAL RESULTS

Our experimental results were gathered on an Intel Haswell four-core processor (Core i7-4770) with RTM and HLE support, running at 3.40GHz. Each core has a 32KB L1 cache, and a total of 8GB RAM shared across all cores. We enabled hyperthreading to collect data for up to eight threads. Because of L1 cache sharing due to hyperthreading, we noticed that at eight threads some hardware transactions that previously executed without failure began to abort due to overflow, thereby degrading performance. We used the GCC 4.8 compiler with -O3 optimizations for all benchmarks.

We used the STAMP benchmark suite [3] to measure the speedup that Invyswell provides relative to sequential execution. We compare this speedup against NOrec, which we call NorecSTM, Hybrid NOrec, which we abbreviate as NorecHy, and Haswell's HLE. For each of these systems, we executed each STAMP benchmark five times and present the median result as shown in Figure 8. Variance was generally low, except for Bayes.

Invyswell Details.

We instrumented the STAMP code using its macros to use a thread-local transaction type indicator for choosing which code path to execute. This instrumentation incurs a run-time performance penalty. A compiler could generate different code paths for these transaction types, but it would not need to generate a code path for each type. In particular, LiteHW and SglSW have similar read/write annotations, as do BFHW and IrrevocSW. Moreover, the overhead incurred for manual instrumentation is higher than the overhead incurred by compiler instrumentation.

Hardware transactions are retried N times, where $N = 10$ for our experiments, unless the abort status indicates that the transaction is unlikely to succeed in hardware, in which case the transaction is immediately retried in software. SpecSWs are retried M times, where $M = 4$, and used SglSW as a fallback if the number of retries is exceeded. Invyswell was configured to use 1024 bits and the spooky-hash function [20] for its Bloom filters. Outside of normal Bloom filter trade-offs of precision versus size, there is an additional trade-off with Bloom filters for Invyswell's hardware transactions between their precision and the aborts they cause by overflow. [4] We found 1024 bits to be a good balance across all benchmarks. For example, the Yada benchmark emits many Bloom filter false positives and makes this tradeoff apparent. Increasing the Bloom filters' size improves SpecSW performance but degrades BFHW, as it causes more aborts.

Hybrid NOrec and Invyswell.

Hybrid NOrec has many variants, many of which require nonspeculative loads. [27] requires both nonspeculative loads and nonspeculative stores. These variants cannot be implemented using TSX, and are not considered in this paper. The version of NOrec evaluated in this paper uses the two location variant and the sw_exists filter described in [4]. [5] Hybrid NOrec has two types of transactions, hardware and software. Both types can execute at the same time. To ensure hardware transactions do not see inconsistent memory states, they eagerly subscribe to the software transactions' commit lock as soon as they begin their execution. When a software transaction begins its commit phase, hardware transactions are automatically aborted. When a hardware transaction commits, it increments a shared counter, which notifies software transactions that they must perform value-based validation to ensure consistency. To perform validation, each software transaction maintains its own list of read memory locations. To reduce list insert computational overhead, each software transaction inserts new read element directly to the list's tail, even if the item is already in the list, resulting in $O(1)$ insert time complexity. A disadvantage of this approach is that the read list can become large if a software transaction reads many locations, thereby increasing the time it takes to perform validation, where the entire list must be walked. Each software transaction performs validation in $O(N)$ time, where N is the size of the read set, for every new read added to the transaction's read set after a software or hardware transaction has committed.

In contrast to Hybrid NOrec, Invyswell has two hardware transaction types, three software transaction types, and performs conflict detection using Bloom filters, not lists, which house the memory accessed by both hardware transactions (BFHW) and software transactions (SpecSW and IrrevocSW). With Bloom filters, Invyswell's conflict detection is performed in $O(1)$ time, yet, because Invyswell uses invalidation, it has additional overhead that Hybrid NOrec does not have, where invalidation is performed after committing a transaction's speculative writes to memory.

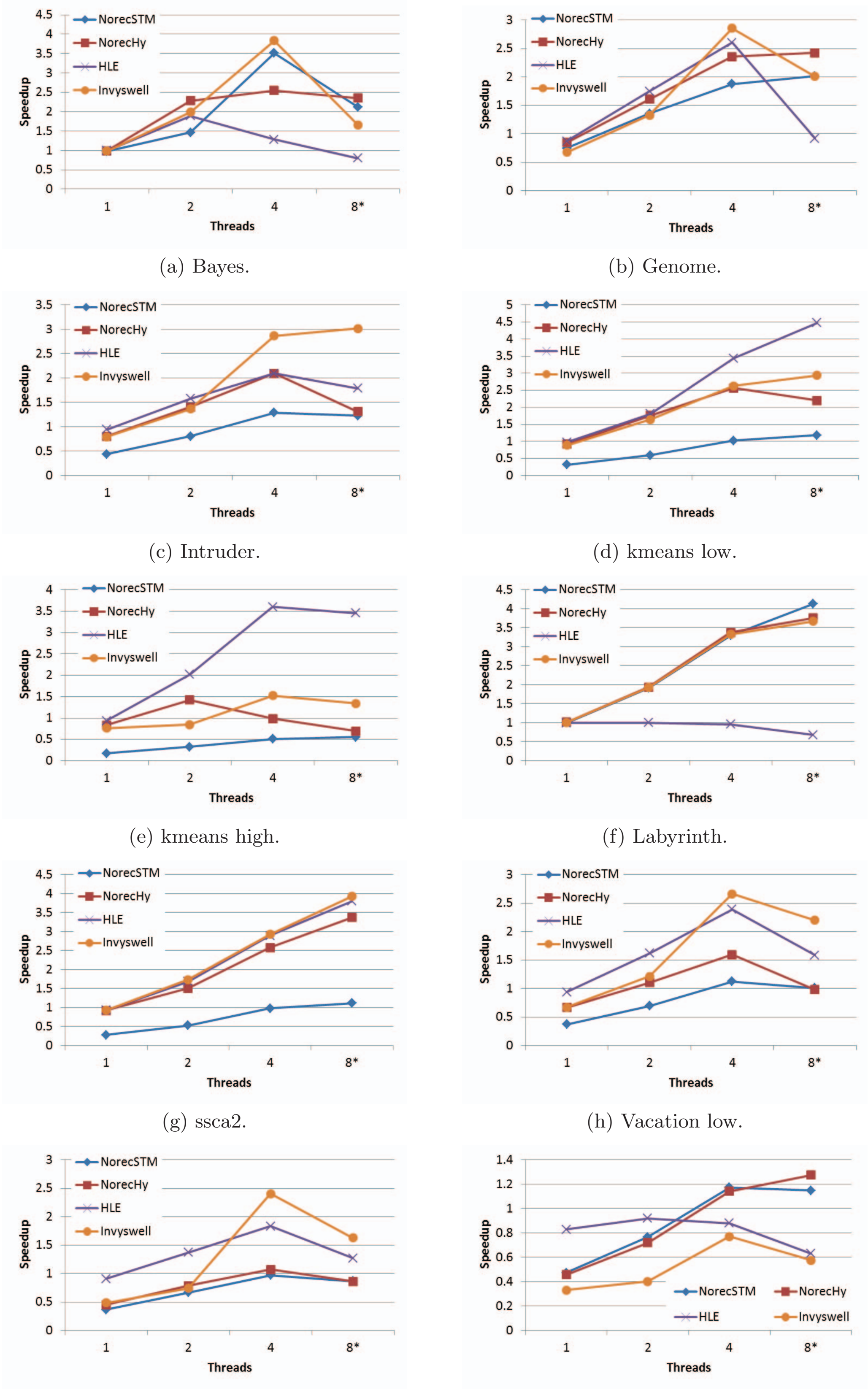
Invyswell's LiteHWs are similar to Hybrid NOrec's hardware transactions, but Invyswell's BFHWs have no Hybrid NOrec counterpart. Although BFHWs incur overhead not found in Hybrid NOrec's hardware transactions – the storing of read and write set data in Bloom filters – this overhead is amortized on large transactions because of the finer grained conflict detection that it enables. The improved precision of conflict detection enables wider transactional throughput between hardware and software transactions if they don't conflict (e.g., Figure 8f's benchmark).

If Invyswell did not include BFHWs, nearly all of Labyrinth's transactions would execute as software transactions, because Invyswell's LiteHWs often get aborted by the long-running software transactions. However, with BFHW, hardware and speculative software transactions (SpecSWs) can execute concurrently and both types of transactions can commit, as there are not many conflicts. NOrec hardware transactions do not exhibit Bloom filter overhead but, instead, incur overhead on its software transactions, which must do value based validation, re-validating the entire read set after each transactional commit. As 50% of the transactions in Labyrinth cannot succeed in hardware, the performance of both HyTMs is similar to that of NOrec STM.

Another important difference between Invyswell and Hybrid NOrec is how fast software transactions execute for different transaction sizes. Invyswell's SpecSW transactions, which are similar to InvalSTM's transactions, are fast for large transactions, while NOrec's software transactions are

[4] The larger the Bloom filter, the better its precision, but the more likely a hardware transaction using such a Bloom filter will abort due to cache overflow, because the Bloom filter must be part of the hardware transaction's speculative state stored, in this case, in Haswell's L1D cache.

[5] Our implementation of Hybrid NOrec included all the optimizations used in [4]. In addition, we tried a variant of this algorithm that had hardware transactions lazily subscribe to the software commit lock, which also used the indirect jump annotations that we used for Invyswell. This version performed similarly to Hybrid NOrec's normal eager subscription, so we omitted the results for clarity.



(a) Bayes.

(b) Genome.

(c) Intruder.

(d) kmeans low.

(e) kmeans high.

(f) Labyrinth.

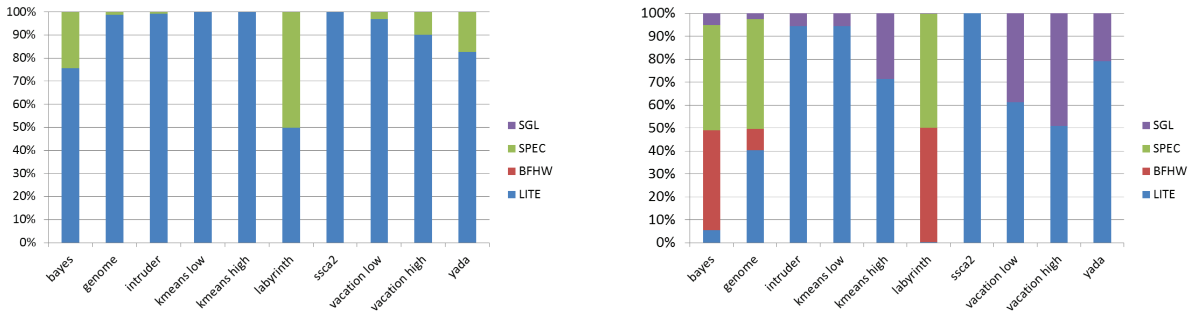
(g) ssca2.

(h) Vacation low.

(i) Vacation high.

(j) Yada.

Figure 8: Speedup on STAMP Benchmarks (Note: 8 threads using hyperthreading).



(a) Invyswell Transaction Types: 1-threaded execution.

(b) Invyswell Transaction Types: 8-threaded execution.

Figure 9: Percentage of Transaction Types for Invyswell for 1-Threaded and 8-Threaded Executions.

fast for small transactions without many reads to re-validate. Yet, because Haswell's RTM can successfully execute most smaller size transactions (those without unsupported instructions), we believe SpecSWs are the natural choice as a fallback mechanism for hardware transactions.

Nevertheless, there is an interesting effect that occurs in the presence of hyperthreading, where hardware transactions overflow at smaller sizes than they would without hyperthreading because of cache sharing between two hyperthreads on the same core. For example, in Genome (Figure 8b), at eight threads about 50% of hardware transactions spill to software, for both HyTMs, because of overflow. Because of this, Hybrid NOrec performs better than Invyswell for Genome at eight threads. However, we believe this is an artifact of hyperthreading, as Invyswell is notably faster than Hybrid NOrec for Genome at four threads, where significantly fewer hardware transactions spill to software. With this in mind, we expect Invywell to perform better as HTMs scale in core count, as only large transactions will overflow the cache, resulting in the use of Invysell's SpecSWs only in the cases in which they were intended.

NOrec and HyTMs.

STMs typically scale at higher thread counts, but often perform poorly at low thread counts, especially for small and mid-sized transactions. NOrec, referred to as NorecSTM in our figures, like any STM, incurs instrumentation overhead that limits performance for small (Ssca2, Kmeans) and mid-sized (Intruder, Vacation, Genome) transactions. For such benchmarks, Invyswell can outperform NOrec by a factor of 3.5× (8g). Hybrid NOrec also outperforms NOrec on these benchmarks, indicating that a hybrid is necessary over an STM. However, Invyswell can be twice as fast as Hybrid NOrec (8c) because of its more lightweight SglSWs, in which Hybrid NOrec has no software equivalent.

As expected, NOrec performs best for benchmarks with longer transactions, and bigger read and write sets, such as Bayes, Labyrinth and Yada (Figures 8a, 8f, and 8j, respectively). Hybrid NOrec closely approaches the NOrec's speedup, as most of the benefit in these cases comes from the software transactions. In Figure 8a, NOrec is 2.1× faster than sequential execution, while Invyswell is 1.6× faster. For completeness, we included results for Bayes, but its high variance suggests that these results should be interpreted with caution [28].

Labyrinth (Figure 8f) has long transactions, where the first portion of the transaction manipulates non-shared memory. For this benchmark, 50% of the transactions cannot complete in hardware, so HLE's performance degrades to that of a lock. In contrast, NOrec yields high throughput because it enables concurrency between its transactions. Because Haswell does not support non-transactional loads and stores, all local operations performed inside a transaction are also transactional, putting pressure on the cache. Therefore, both Hybrid NOrec and Invyswell are negatively affected, resulting in performance similar to NOrec.

Hardware Lock Elision (HLE).

HLE is implemented entirely in hardware and has no instrumentation overhead, but uses a non-scalable single global lock fallback when transactions fail. For large benchmarks, such as Bayes or Labyrinth, even at small thread counts, Invyswell outperforms HLE by a notable margin. This is because many transactions overflow the cache and fall back to software, being serialized by the lock used in HLE. For medium sized benchmarks, Invyswell also outperforms HLE. However, for small transactions, HLE benefits most from the lack of overhead, so it is faster than Invyswell on benchmarks such as Kmeans Low and Kmeans High. Ssca2 is also a benchmark with small transactions, but Invyswell and HLE perform similarly.

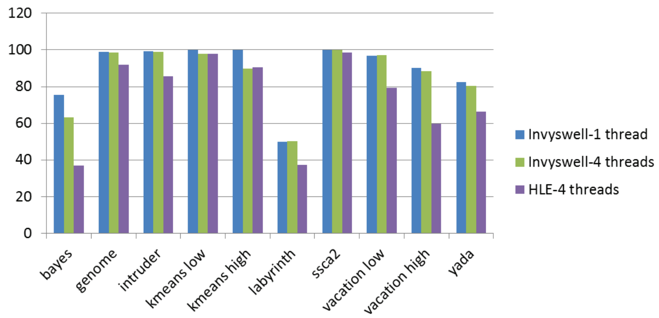


Figure 10: Percentage of Committed Hardware Transactions.

Figure 10 shows the percentage of committed hardware transactions for one thread and four threads for both Invyswell and HLE. The one-threaded execution indicates, in general, the percentage of transactions that fail in hardware because of unsupported instructions or overflow. This provides a baseline of the maximum number of hardware transaction commits that are possible for each benchmark. We also found that the number of HLE hardware transactions that begin is higher than the total number of committed transactions. This suggests that HLE also retries failed transactions before falling back to its global lock.

Invyswell's percentage of committed hardware transactions at four threads is similar to its percentage at one thread, and it is higher than HLE's percentage at four threads. This makes the argument that Invyswell generally makes more efficient use of hardware resources than the hardware (i.e., HLE) itself. Figures 9a and 9b show the breakdown of Invyswell's transaction types for one thread and eight threads executions. The eight-threaded execution suffers from the effects of hyperthreading, so the number of hardware transactions successfully committed is lower than for the one thread execution.

Overall, Invyswell outperforms HLE. For Yada, however, HLE is faster than Invyswell despite using fewer hardware transactions. This benchmark has large transactions and high contention, causing a lot of conflicts between transactions. In this case, Invyswell suffers from many false positives in its Bloom filter set intersection. We noticed an increase in performance for SpecSWs as we increase the size of the Bloom filters. However, as we previously explained, larger Bloom filters negatively impact BFHWs. Therefore, the size of the Bloom filters represents a tradeoff to balance the performance of SpecSWs and BFHWs.

Discussion.

In general, Invyswell outperforms prior methods across all STAMP benchmarks. Not only does Invyswell outperform HLE for all but the smallest transactions, it is inherently more flexible, because the programmer has explicit control over CM and failover policies. Although Invyswell is adapted from the earlier InvalSTM design, the existence of hardware transactions that bypass the CM means that the two systems are divergent, in terms of design and behavior.

Hardware transactions can fail for a variety of reasons, including resource exhaustion, timing anomalies, or illegal instructions. For future work, there is a need for better adaptive CM to identify when a particular approach is not working well, and when to switch to a more effective alternative.

8. CONCLUSIONS

We described Invyswell, a HyTM that combines Haswell's RTM transactions with software transactions from a heavily modified version of InvalSTM. We evaluated Invyswell on a 3.4 GHz 4-core Haswell processor capable of supporting up to eight hardware threads and compared it to to Haswell's native hardware lock elision (HLE), a state-of-the-art STM (NOrec), and a state-of-the-art HyTM (Hybrid NOrec).

Our main goals with Invyswell were to *(i)* improve performance for small- to medium-sized transactions, configurations where the instrumentation costs of STMs typically cause them to perform poorly and *(ii)* to extend InvalSTM's design to support the concurrent execution of both hardware and software transactions. We found that very small transactions are handled well by a simple combination of hardware transactions with fallback to a single global lock. The most interesting challenges were *(i)* modifying InvalSTM to provide some degree of precision in its conflict detection between concurrently executing hardware and software transactions and *(ii)* improving mid-size transaction performance, transactions that are small enough to benefit from hardware transactions, but too large to work well with a single global lock.

We evaluated a variety of transactional mechanisms, both hardware and software, on a range of STAMP benchmarks. As one might expect for such heterogeneous benchmarks, no single mechanism was best for every benchmark, but overall, Invyswell outperformed prior methods by more than 18%.

Haswell supports *hardware lock elision* (HLE), which allows an annotated critical section to be first executed speculatively as a hardware transaction, and then, if that transaction fails, to be re-executed non-speculatively using the original lock. HLE already provides some of the functionality of HyTM, so it is natural to ask whether Haswell needs HyTM at all. We find that HyTM is indeed needed: on average, Invyswell is about 25% faster than HLE across all benchmarks. Moreover, for benchmarks with large transactions, such as Bayes and Labyrinth, HLE does not scale and it is 2×-5.4× slower than Invyswell. The principal reason HLE does not eliminate the need for HyTM is that HyTM allows for better contention management. HLE follows a hard-wired policy of falling back to a lock after failure, but HyTM can make more intelligent and flexible decisions about resolving conflicts, taking advantage of software-based transactions, and making more effective transitions between speculative and various non-speculative synchronization mechanisms.

We tested alternative software mechanisms that trade overhead for precision. Conflict detection can be coarse and fast (SglSW) or more precise and slower (IrrevocSW and SpecSW). In the thread-count range supported by our platform, coarse-and-fast usually slightly outperforms precise-and-slower. We conjecture that precise conflict detection will become more attractive in future hardware platforms with more cores, where Invyswell is likely to perform well.

Any HyTM faces the challenge of providing opacity, which ensures that all transactions only observe consistent states. This is more difficult than it may seem, because the composition of two opaque mechanisms (for example, Haswell's RTM and InvalSTM) is not necessarily opaque. RTM's lack of escape actions complicated our task. Escape actions could make it substantially easier to ensure opacity, and to provide more effective conflict management. For example, a hardware transaction could invalidate software transactions during its commit phase, rather than after it, allowing, in some cases, for it to abort itself to improve overall throughput, as was the case in InvalSTM's original design.

Our experience suggests that hybrid mechanisms can improve the performance of small to mid-size transactions that can execute in hardware, compared to software-only or hardware lock-elision mechanisms. We conjecture that this difference will become even more pronounced when Haswell platforms with more cores become available.

Acknowledgements

We thank Andi Kleen, Michael Spear, and the anonymous reviewers for useful feedback. We have tried our best to address all of their excellent comments.

9. REFERENCES

[1] A.-R. Adl-Tabatabai, C. Kozyrakis, and B. Saha. Transactional programming in a multi-core environment. In K. A. Yelick and J. M. Mellor-Crummey, editors, *PPoPP*, page 272. ACM, 2007.

[2] P. Broadwell, M. Harren, and N. Sastry. Scrash: A system for generating secure crash information. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 19–19, Berkeley, CA, USA, 2003. USENIX Association.

[3] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[4] 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.

[5] 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.

[6] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pages 336–346, New York, NY, USA, 2006. ACM.

[7] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIV, pages 157–168, New York, NY, USA, 2009. ACM.

[8] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

[9] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why stm can be more than a research toy. *Commun. ACM*, 54(4):70–77, Apr. 2011.

[10] J. E. Gottschlich, M. P. Herlihy, G. A. Pokam, and J. G. Siek. Visualizing transactional memory. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 159–170, New York, NY, USA, 2012. ACM.

[11] J. E. Gottschlich, M. Vachharajani, and J. G. Siek. An efficient software transactional memory using commit-time invalidation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, April 2010.

[12] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, New York, NY, USA, 2008. ACM.

[13] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, Second Edition*. Morgan and Claypool, 2010.

[14] 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*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[15] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture*. May 1993.

[16] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier, Inc., 2008.

[17] Intel Corporation. Hardware lock elision in Haswell. Retrieved from `http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-win/GUID-A462FBC8-37F2-490F-A68B-2FFA8010DEBC.htm`.

[18] Intel Corporation. Transactional Synchronization in Haswell. Retrieved from `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`, 8 September 2012.

[19] C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for ibm system z. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 25–36, Washington, DC, USA, 2012. IEEE Computer Society.

[20] Jenkins, B. SpookyHash: a 128-bit non-cryptographic hash (2010). Retrieved from `http://burtleburtle.net/bob/hash/spooky.html`, 25 June 2014.

[21] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 209–220, New York, NY, USA, 2006. ACM.

[22] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, aug 2007.

[23] V. J. Marathe and M. Moir. Toward high performance nonblocking software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 227–236, New York, NY, USA, 2008. ACM.

[24] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logtm. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pages 359–370, New York, NY, USA, 2006. ACM.

[25] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO*, pages 294–305. ACM/IEEE, 2001.

[26] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 221–228, New York, NY, USA, 2007. ACM.

[27] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing hybrid transactional memory: the importance of nonspeculative operations. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 53–64, New York, NY, USA, 2011. ACM.

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

[29] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPOPP*. ACM SIGPLAN 2006, Mar. 2006.

[30] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Principles of Distributed Computing*. Aug 1995.

[31] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of blue gene/q hardware support for transactional memories. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 127–136, New York, NY, USA, 2012. ACM.