ABSTRACT

Transactional memory (TM) is a speculative shared-memory synchronization mechanism used to speed up concurrent programs. Most current TM implementations are software-based (STM) and incur noticeable overheads for each transactional memory access. Hardware TM proposals (HTM) address this issue but typically suffer from other restrictions such as limits on the number of data locations that can be accessed in a transaction.

In this paper, we present several new hybrid TM algorithms that can execute HTM and STM transactions concurrently and can thus provide good performance over a large spectrum of workloads. The algorithms exploit the ability of some HTMs to have both speculative and nonspeculative (nontransactional) memory accesses within a transaction to decrease the transactions’ runtime overhead, abort rates, and hardware capacity requirements. We evaluate implementations of these algorithms based on AMD’s Advanced Synchronization Facility, an x86 instruction set extension proposal that has been shown to provide a sound basis for HTM.

Categories and Subject Descriptors
D.1.3 [Programming Techniques]: Concurrent Programming

General Terms
Algorithms, Performance

Keywords
Transactional Memory

1. INTRODUCTION

Today’s multicore and manycore CPUs require parallelized software to unfold their full performance potential. Shared-memory synchronization plays a big role in parallel software, either when synchronizing and merging results of parallel tasks, or when parallelizing programs by speculatively executing tasks concurrently.

Until now, most concurrent programs have been programmed using lock-based synchronization. Yet, locks are considered difficult to use for the average programmer, especially when locking at a fine granularity to provide scalable performance. This is particularly important when considering that large classes of programs will have to be parallelized by programmers who are not well trained in concurrent programming. Transactional memory (TM) is a promising alternative for synchronization because programmers only need to declare which regions in their program must be atomic, not how atomicity will be implemented. Unfortunately, current software transactional memory (STM) implementations have a relatively large performance overhead. While there is certainly room left for further optimizations, it is believed by many that only hardware transactional memory (HTM) implementations can have a sufficiently good performance for TM to become widely adopted by developers.

Of the many published HTMs, only two designs have been proposed by industry for possible inclusion in high-volume microprocessors: Sun’s Rock TM [10] and AMD’s Advanced Synchronization Facility (ASF) [1]. While these HTMs have notable differences, they are both based on simple designs that provide best-effort HTM in the sense that only a subset of all reasonable transactions are expected to be supported by hardware. They have several limitations (e.g., the number of cache lines that can be accessed in a transaction can be as low as four) and have to be complemented with software fallback solutions that execute in software the transactions that cannot run in hardware. A simple fallback strategy is to execute software transactions serially, i.e., one at a time. However, this approach limits performance when software transactions are frequent. It is therefore desirable to develop hybrid TM (HyTM) in which multiple hardware and software transactions can run concurrently.

Most previous HyTM proposals have assumed HTMs in which every memory access inside a transaction is speculative, that is, it is transactional, isolated from other threads until transaction commit and will be rolled back on abort. In contrast, ASF provides selective annotation, which means that nonspeculative memory accesses are supported within transactions (including nonspeculative atomic instructions) and speculative memory accesses have to be explicitly marked as such.
2. BACKGROUND AND RELATED WORK

Our objective is to investigate the design of hybrid transactional memory algorithms that exploit hardware facilities for decreasing the overhead of transactions in good cases while composing well with state-of-the-art software transactional memory algorithms. We assume that the TM runtime system is implemented as part of a library with a well-specified interface for starting, committing, and aborting transactions, as well as performing transactional memory accesses (loads and stores).

We focus on C/C++ and use a full TM stack [5] consisting of a TM library and a transactional C/C++ compiler. This implementation complies with the specification of C++ support for TM constructs [15], which includes ensuring privatization safety for transactions, whereas publication safety [18] as required by the specification [15] is basically the responsibility of the programmer (i.e., C/C++ source code must be race-free) and the compiler (i.e., it must not introduce race conditions).

Informally, C++ transactions are guaranteed to execute virtually sequentially and in isolation as long as the program is race-free in terms of the upcoming C++ memory model [16] extended with specific rules for transactions.

In this paper, we present a family of novel HyTM algorithms that use AMD's ASF as HTM. We make heavy use of nonspeculative operations in transactions to construct efficient HyTM algorithms that improve on previous HyTMs. In particular, they decrease the runtime overhead, abort rates, and HTM capacity requirements of hardware transactions, while at the same time allowing hardware and software transactions to run and commit concurrently (this is further discussed in Section 2.2 and Table 3).

Our HyTM algorithms use two state-of-the-art STM algorithms from different research groups, LSA [25] and N Ore [8], for software transactions. LSA and NO rec focus on different workloads in their optimizations, e.g., a higher level of concurrency vs. lower single-thread overheads. The resulting HyTM compositions provide the same guarantees as the respective STMs.

We evaluate the performance of our algorithms on a near-cycle-accurate x86 simulator with support for several implementations of ASF [5] that differ notably in their capacity limits. Our HyTMs are embedded into a full TM software stack for C/C++.

Non-speculative operations are useful beyond HyTM optimizations. Table 1 shows three general-purpose synchronization techniques that we present in this paper, which are all combinations of both transaction-based synchronization and classic nontransactional synchronization using standard atomic instructions. The first technique can reduce HTM capacity requirements and has similarities to lock elision [22], whereas the other two are about composability with nontransactional synchronization. We will explain the techniques further in Sections 3 and 4. To make them applicable, the HTM does not only have to allow nonspeculative operations but it must also provide certain ordering guarantees (see Section 2.1). The rest of the paper is organized as follows. In Section 2, we present background information about ASF and TM in general, and we discuss related work on HyTM designs. We present our new HyTM algorithms in Sections 3 and 4, evaluate them in Section 5, and conclude in Section 6.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Explained in</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Monitor metadata but read data nonspeculatively.</td>
<td>Sec. 3</td>
</tr>
<tr>
<td>2. Use nonspeculative atomic read-modify-write operations to send synchronization messages.</td>
<td>Sec. 3 &amp; 4</td>
</tr>
<tr>
<td>3. Validate hardware transactions against software synchronization messages.</td>
<td>Sec. 4</td>
</tr>
</tbody>
</table>

Table 1: General-purpose synchronization techniques enabled by the availability of nonspeculative operations in hardware transactions.

Contributions.

In this paper, we present a family of novel HyTM algorithms that use AMD's ASF as HTM. We make heavy use of nonspeculative operations in transactions to construct efficient HyTM algorithms that improve on previous HyTMs. In particular, they decrease the runtime overhead, abort rates, and HTM capacity requirements of hardware transactions, while at the same time allowing hardware and software transactions to run and commit concurrently (this is further discussed in Section 2.2 and Table 3).

Our HyTM algorithms use two state-of-the-art STM algorithms from different research groups, LSA [25] and N Ore [8], for software transactions. LSA and NO rec focus on different workloads in their optimizations, e.g., a higher level of concurrency vs. lower single-thread overheads. The resulting HyTM compositions provide the same guarantees as the respective STMs.

We evaluate the performance of our algorithms on a near-cycle-accurate x86 simulator with support for several implementations of ASF [5] that differ notably in their capacity limits. Our HyTMs are embedded into a full TM software stack for C/C++.

Non-speculative operations are useful beyond HyTM optimizations. Table 1 shows three general-purpose synchronization techniques that we present in this paper, which are all combinations of both transaction-based synchronization and classic nontransactional synchronization using standard atomic instructions. The first technique can reduce HTM capacity requirements and has similarities to lock elision [22], whereas the other two are about composability with nontransactional synchronization. We will explain the techniques further in Sections 3 and 4.

To make them applicable, the HTM does not only have to allow nonspeculative operations but it must also provide certain ordering guarantees (see Section 2.1).

The rest of the paper is organized as follows. In Section 2, we present background information about ASF and TM in general, and we discuss related work on HyTM designs. We present our new HyTM algorithms in Sections 3 and 4, evaluate them in Section 5, and conclude in Section 6.

2. BACKGROUND AND RELATED WORK

Our objective is to investigate the design of hybrid transactional memory algorithms that exploit hardware facilities for decreasing the overhead of transactions in good cases while composing well with state-of-the-art software transactional memory algorithms. We assume that the TM runtime system is implemented as part of a library with a well-specified interface for starting, committing, and aborting transactions, as well as performing transactional memory accesses (loads and stores).

We focus on C/C++ and use a full TM stack [5] consisting of a TM library and a transactional C/C++ compiler. This implementation complies with the specification of C++ support for TM constructs [15], which includes ensuring privatization safety for transactions, whereas publication safety [18] as required by the specification [15] is basically the responsibility of the programmer (i.e., C/C++ source code must be race-free) and the compiler (i.e., it must not introduce race conditions).

Informally, C++ transactions are guaranteed to execute virtually sequentially and in isolation as long as the program is race-free in terms of the upcoming C++ memory model [16] extended with specific rules for transactions.

The compiler generates separate STM and HTM code paths for each transaction. A common transaction start function (see Algorithm 1) takes care of selecting STM or HTM code at runtime. A transaction first tries to run in hardware mode using a special ASF SPECULATE instruction (line 4). This instruction returns a non-zero value when jumping back after an abort, similarly to setjmp/longjmp in the standard C library. If the transaction aborts and a retry is unlikely to succeed (as determined on line 6, for example, because of capacity limitations or after multiple aborts due to contention), it switches to software mode. After this has been decided, only STM or HTM code will be executed (functions starting with atm- or htm-, respectively) during this attempt to execute the transaction.

In the rest of this section, we give an overview of the hardware TM support used for our hybrid algorithms and we discuss related work.

2.1 Advanced Synchronization Facility

AMD’s Advanced Synchronization Facility (ASF) is a proposal [1] of hardware extensions for x86_64 CPUs. It essentially provides hardware support for the speculative execution of regions of code. These speculative regions are similar to transactions in that they take effect atomically. We have shown in a previous study [5] that ASF can be used as an efficient pure HTM in a realistic TM software stack. The HyTM algorithms that we present in this paper are based on ASF and rely on a similar software stack.

AMD has designed ASF in such a way that it would be feasible to implement ASF in high-volume microprocessors. Hence, ASF comes with a number of limitations [1, 11, 6]. First, the number of disjoint locations that can be accessed in a transaction is limited either by the size of speculation buffers (which are expensive and thus have been designed with small capacity) or by the associativity of caches (when tracking speculative state in caches). Second, ASF transactions are not virtualized and therefore, abort on events such as context switches or page faults. These limitations illustrate that HyTM will be required to build a feature-rich TM for programmers.

In contrast to several other HTM proposals, ASF provides selective annotation for speculative memory accesses. Speculative regions (SRs, the equivalent of transactions) are demarcated with new SPECULATE and COMMIT CPU instructions. In an SR, speculative/protected memory accesses, in the form of ASF-specific L00K MV CPU instructions, can be mixed with nonspeculative/unprotected accesses, i.e., ordinary load/store instructions (MV) as well as atomic instructions such as compare-and-set (CAS). Selective an-
notation requires more work on the compiler side, but allows the TM to use speculative accesses sparingly and thus preserve precious ASF capacity. Second, the availability of nonspeculative atomic instructions allows us to use common concurrent programming techniques during a transaction, which can reduce the number of transaction aborts due to benign contention (e.g., when updating a shared counter). In an SR, nonspeculative loads are allowed to read state that is speculatively updated in the same SR, but nonspeculative stores must not overlap with previous speculative accesses. Conflict detection for speculative accesses is handled at the granularity of a cache line.

ASF also provides CPU instructions for monitoring a cache line for concurrent stores (LOCK PREFETCH) or loads and stores (LOCK PREFETCHW), for stopping monitoring a cache line (RELEASE), and for aborting a SR and discarding all speculative modifications (ABORT).

Conflict resolution in ASF follows the “requester wins” policy (i.e., existing SRs will be aborted by incoming conflicting memory accesses). Table 2 summarizes how ASF handles contention when CPU A performs an operation while CPU B is in a SR with the cache line protected by ASF [1]. These conflict resolution rules are important for understanding how our HyTM algorithms work and why they perform well.

The ordering guarantees that ASF provides for mixed speculative and nonspeculative accesses are important for the correctness of our algorithms, and are required for the general-purpose synchronization techniques listed in Table 1 to be applicable or practical. In short, aborts are instantaneous with respect to the program order of instructions in SRs. For example, aborts are supposed to happen before externally visible effects such as page faults or nonspeculative stores appear. A consequence is that memory lines are monitored early for conflicting accesses (i.e., once the respective instructions are issued in the CPU, which is always before they retire). After an abort, execution is resumed at the SPECULATE instruction. Further, atomic instructions such as compare-and-set or fetch-and-increment retain their ordering guarantees (e.g., a CAS ordered before a COMMIT in a program will become visible before the transaction’s commitment). This behavior illustrates why speculative accesses are also referred to as “protected” accesses.

### 2.2 Previous HyTM Designs

Table 3 shows a comparison of our HyTM algorithms (second and third row) with previous HyTM designs. The columns list HyTM properties that have a major influence on performance.

- First, at least first-generation HTM will not be able to run all transactions in hardware. Thus there likely will be software transactions, which should be able to run concurrently with hardware transactions (see column two). Second, HyTMs should not introduce additional runtime overhead for hardware transactions, which would decrease HTM’s performance advantage compared to STM.
- Third, HTM capacity for transactional memory accesses is scarce, so HyTM should require as little capacity as possible.
- Furthermore, HyTM algorithms that do not guarantee privatization safety for software transactions have to ensure this using additional implementation methods (see Section 5), resulting in additional runtime overhead. Visible reads are often more costly for STMs than invisible reads and can introduce artificial conflicts with transactional HTM reads (e.g., if the STM updates an ore). In phased TM [17], the implementation mode for transactions is switched globally and (i.e., only software or hardware transactions are running at a time). This leads to no HTM overhead when in hardware mode, but even a single transaction that has to run in software reduces overall performance to the level of STM. The phased TM approach is orthogonal to hybrid TM.

Similarly, the HyTM [14] presented by Hofmann et al. uses a simple global lock as software fallback mechanism instead of an STM that can run several software transactions concurrently. Hardware transactions wait for a software transaction to finish before committing, but are not protected from reading uncommitted and thus potentially inconsistent updates of software transactions (“dirty reads”). Note that with ASF, hardware transactions are not completely sandboxed. For example, page faults due to inconsistent snapshots will abort speculative regions but will also be visible to the operating system.

Kumar et al. describe a HyTM [24] based on an object-based STM design with indirect via locator objects, which uses visible reads and requires small hardware transactions even for software transactions. Recent research has shown that STM algorithms with invisible reads and no indirect have significantly lower overhead (e.g., [9, 20, 8]).

Damron et al. present a HyTM [21] that combines a best-effort HTM with a word-based STM algorithm that uses visible reads and performs conflict detection based on ownership records. The HTM does not use selective annotation and thus hardware transactions have to monitor application data and TM metadata (i.e., ownership records) for each access, which significantly increases the HTM capacity required to successfully run transactions in hardware. Likewise, visible reads result in significant overheads for STMs. This HyTM is also used in a study about the HTM support in Rock [10].

The hardware-accelerated STM algorithms (HASTM) by Saha et al. [3] are based on ownership records (like LSA but unlike NOrec). HASTM in cautious mode monitors application data and does read logging, whereas our hybrid LSA algorithms (see Section 3 and row three) monitor ownership records and do not log reads. HASTM in aggressive mode monitors both application data and ownership records, thus suffering from higher HTM capacity requirements (evaluated in Section 5). Thus, only our hybrid LSA algorithms can change the ownership-record-to-memory mapping to achieve a larger effective read capacity. Transactional stores in HASTM are not accelerated but executed in software only. Furthermore, HASTM in cautious mode as presented in the paper does not allow more work on the compiler side, but allows the TM to use speculative accesses sparingly and thus preserve precious ASF capacity. Second, the availability of nonspeculative atomic instructions allows us to use common concurrent programming techniques during a transaction, which can reduce the number of transaction aborts due to benign contention (e.g., when updating a shared counter). In an SR, nonspeculative loads are allowed to read state that is speculatively updated in the same SR, but nonspeculative stores must not overlap with previous speculative accesses. Conflict detection for speculative accesses is handled at the granularity of a cache line.

ASF also provides CPU instructions for monitoring a cache line for concurrent stores (LOCK PREFETCH) or loads and stores (LOCK PREFETCHW), for stopping monitoring a cache line (RELEASE), and for aborting a SR and discarding all speculative modifications (ABORT).

Conflict resolution in ASF follows the “requester wins” policy (i.e., existing SRs will be aborted by incoming conflicting memory accesses). Table 2 summarizes how ASF handles contention when CPU A performs an operation while CPU B is in a SR with the cache line protected by ASF [1]. These conflict resolution rules are important for understanding how our HyTM algorithms work and why they perform well.

The ordering guarantees that ASF provides for mixed speculative and nonspeculative accesses are important for the correctness of our algorithms, and are required for the general-purpose synchronization techniques listed in Table 1 to be applicable or practical. In short, aborts are instantaneous with respect to the program order of instructions in SRs. For example, aborts are supposed to happen before externally visible effects such as page faults or nonspeculative stores appear. A consequence is that memory lines are monitored early for conflicting accesses (i.e., once the respective instructions are issued in the CPU, which is always before they retire). After an abort, execution is resumed at the SPECULATE instruction. Further, atomic instructions such as compare-and-set or fetch-and-increment retain their ordering guarantees (e.g., a CAS ordered before a COMMIT in a program will become visible before the transaction’s commitment). This behavior illustrates why speculative accesses are also referred to as “protected” accesses.

### 2.2 Previous HyTM Designs

Table 3 shows a comparison of our HyTM algorithms (second and third row) with previous HyTM designs. The columns list HyTM properties that have a major influence on performance.

- First, at least first-generation HTM will not be able to run all transactions in hardware. Thus there likely will be software transactions, which should be able to run concurrently with hardware transactions (see column two). Second, HyTMs should not introduce additional runtime overhead for hardware transactions, which would decrease HTM’s performance advantage compared to STM.
- Third, HTM capacity for transactional memory accesses is scarce, so HyTM should require as little capacity as possible.
- Furthermore, HyTM algorithms that do not guarantee privatization safety for software transactions have to ensure this using additional implementation methods (see Section 5), resulting in additional runtime overhead. Visible reads are often more costly for STMs than invisible reads and can introduce artificial conflicts with transactional HTM reads (e.g., if the STM updates an ore). In phased TM [17], the implementation mode for transactions is switched globally and (i.e., only software or hardware transactions are running at a time). This leads to no HTM overhead when in hardware mode, but even a single transaction that has to run in software reduces overall performance to the level of STM. The phased TM approach is orthogonal to hybrid TM.

Similarly, the HyTM [14] presented by Hofmann et al. uses a simple global lock as software fallback mechanism instead of an STM that can run several software transactions concurrently. Hardware transactions wait for a software transaction to finish before committing, but are not protected from reading uncommitted and thus potentially inconsistent updates of software transactions (“dirty reads”). Note that with ASF, hardware transactions are not completely sandboxed. For example, page faults due to inconsistent snapshots will abort speculative regions but will also be visible to the operating system.

Kumar et al. describe a HyTM [24] based on an object-based STM design with indirect via locator objects, which uses visible reads and requires small hardware transactions even for software transactions. Recent research has shown that STM algorithms with invisible reads and no indirect have significantly lower overhead (e.g., [9, 20, 8]).

Damron et al. present a HyTM [21] that combines a best-effort HTM with a word-based STM algorithm that uses visible reads and performs conflict detection based on ownership records. The HTM does not use selective annotation and thus hardware transactions have to monitor application data and TM metadata (i.e., ownership records) for each access, which significantly increases the HTM capacity required to successfully run transactions in hardware. Likewise, visible reads result in significant overheads for STMs. This HyTM is also used in a study about the HTM support in Rock [10].

The hardware-accelerated STM algorithms (HASTM) by Saha et al. [3] are based on ownership records (like LSA but unlike NOrec). HASTM in cautious mode monitors application data and does read logging, whereas our hybrid LSA algorithms (see Section 3 and row three) monitor ownership records and do not log reads. HASTM in aggressive mode monitors both application data and ownership records, thus suffering from higher HTM capacity requirements (evaluated in Section 5). Thus, only our hybrid LSA algorithms can change the ownership-record-to-memory mapping to achieve a larger effective read capacity. Transactional stores in HASTM are not accelerated but executed in software only. Further-

### Table 2: Conflict matrix for ASF operations ([1], §6.2.1).

<table>
<thead>
<tr>
<th>CPU A mode</th>
<th>CPU A operation</th>
<th>CPU B cache line state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speculative region</td>
<td>LOCK 32W (load)</td>
<td>OK B aborts</td>
</tr>
<tr>
<td>Speculative region</td>
<td>LOCK 32W (store)</td>
<td>B aborts B aborts</td>
</tr>
<tr>
<td>Speculative region</td>
<td>LOCK PREFETCH</td>
<td>OK B aborts</td>
</tr>
<tr>
<td>Speculative region</td>
<td>LOCK PREFETCHM</td>
<td>B aborts B aborts</td>
</tr>
<tr>
<td>Speculative region</td>
<td>COMMIT</td>
<td>OK OK</td>
</tr>
<tr>
<td>Any</td>
<td>Read operation</td>
<td>OK B aborts</td>
</tr>
<tr>
<td>Any</td>
<td>Write operation</td>
<td>B aborts B aborts</td>
</tr>
<tr>
<td>Any</td>
<td>PREFETCHW</td>
<td>OK B aborts</td>
</tr>
<tr>
<td>Any</td>
<td>PREFETCHM</td>
<td>B aborts B aborts</td>
</tr>
</tbody>
</table>

Notes:

1. “Yes” means that non-conflicting pairs of software/hardware transactions can run concurrently.

2. “Orecs” are ownership records (i.e., TM metadata with an M:N mapping from memory locations to orecs). “Data” refers to the application data accessed in a transaction.

3. The serial irrevocable mode that is present in most current STMs is a special case of the phased approach, as it can be used as a very simple software fallback for HTMs.

4. We consider its cacheline-based variants.
Table 3: Overview of HyTM designs.

<table>
<thead>
<tr>
<th>HyTM</th>
<th>HW/SW concurrency</th>
<th>HW txn load/store runtime overheads</th>
<th>HW capacity used for</th>
<th>Privatization safety (SW)</th>
<th>Invisible reads (SW)</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>HyNOrec-2</td>
<td>Yes, SW commits stall other HW/SW ops</td>
<td>Very small</td>
<td>Data</td>
<td>Yes</td>
<td>Yes</td>
<td>See Algorithm 7</td>
</tr>
<tr>
<td>HyLSA ( eager)</td>
<td>Yes</td>
<td>Small (load oreo)</td>
<td>Oreos and data updates</td>
<td>No</td>
<td>Yes</td>
<td>See Algorithm 3</td>
</tr>
<tr>
<td>Phased TM [17]</td>
<td>No</td>
<td>None</td>
<td>Data</td>
<td>N/A</td>
<td>N/A</td>
<td>Can use any STM</td>
</tr>
<tr>
<td>Hoffman et al. [14]</td>
<td>Little</td>
<td>None</td>
<td>Data</td>
<td>Yes</td>
<td>No</td>
<td>Dirty reads not prevented</td>
</tr>
<tr>
<td>Kumar et al. [24]</td>
<td>Yes</td>
<td>High (interlocked)</td>
<td>Data</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Damron et al. [21]</td>
<td>Yes</td>
<td>Small (load oreo)</td>
<td>Data and oreos</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>HASTM [20]</td>
<td>Yes</td>
<td>Medium (load+log oreo)</td>
<td>Read data</td>
<td>No</td>
<td>Yes</td>
<td>Stores in SW only</td>
</tr>
<tr>
<td>HASTM aggressive</td>
<td>Yes</td>
<td>Small (load oreo)</td>
<td>Read data and oreos</td>
<td>No</td>
<td>Yes</td>
<td>Stores in SW only</td>
</tr>
<tr>
<td>HyNOrec-DSS [8]</td>
<td>Partial, SW commits abort HW txns</td>
<td>None, but concurrent commits abort each other</td>
<td>Data and 2 locks</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>HyNOrec-DSS-2 [7]</td>
<td>Yes, SW commits stall other HW/SW ops</td>
<td>Very small, concurrent commits can still abort each other (but less likely)</td>
<td>Data and 3 locks/counters</td>
<td>Yes</td>
<td>Yes</td>
<td>This information is about their best-performing algorithms</td>
</tr>
</tbody>
</table>

Table 3: Overview of HyTM designs.

prevent dirty reads\(^3\), which can crash transactions in unmanaged environments such as C/C++.

Spear et al. propose to use Alert-On-Update (AOU) [19] to accelerate snapshots by reducing the number of necessary software snapshot validations in STMs based on ownership records. However, our LSA STM algorithm already has efficient time-based snapshots due to its use of a global time base, whereas AOU uses a commit counter heuristic, which can suffer from false positives that lead to costly re-validations. The details of the AOU algorithm are not presented, thus it is difficult to assess the remaining HyTM aspects and overheads (and we do not include it in Table 3).

Dalessandro et al. informally describe a HyTM [8] based on the NOrec STM ("HyNOrec-DSS"). It features low runtime overheads and capacity requirements but it shows less scalability because (1) commits of software transactions abort hardware transactions and (2) concurrent commit phases of hardware transactions can abort each other as well. We discuss and evaluate this in detail in Sections 4 and 5.

In concurrent work [7] that has been published after our first results [12], Dalessandro et al. describe optimizations of HyNOrec-DSS ("HyNOrec-DSS-2", last row) and evaluate them on Rock [10] and on ASF. They try to reduce conflicts on metadata (NOrec’s global lock, see Section 4) by distributing commit notifications using speculative stores over several counters, which leads to additional runtime overhead for software transactions because they then have to validate all these counters (and at least two) after each transactional load. In contrast, our HyTMs use non-speculative read-modify-write operations for such notifications (the second technique in Table 1), which enables software transactions to validate using only a single counter. Their algorithms also use non-speculative loads to validate during a hardware transaction’s runtime (“lazy subscription”, the third technique in Table 1) but still use speculative reads for validation during commit, and thus require more HTM capacity than our algorithm. Furthermore, they propose an optimization similar in spirit to phased TM, but embedded into the HyTM algorithms (“SWEExist”), which avoids commit-time synchronization with software transactions if none is running. However, this requires speculative accesses to one further location (thus increasing HTM capacity requirements), and only helps in workloads in which software transactions are rare. SWEExists could be applied to our algorithms as well and could increase scalability if mostly hardware transactions execute. Their evaluation results on Rock cannot be easily compared with ours because Rock is fairly limited compared to ASF. On ASF, they only show results for one ASF implementation, LLB256 (see Section 2.1), which has sufficient capacity to run almost all transactions in hardware and represents the best case in terms of HTM capacity. Other ASF implementations with reduced capacity (e.g., because of cache associativity or a smaller LLB) might be more likely to appear in real hardware but in turn make extra speculative accesses for HyTM metadata much more costly. Furthermore, the choice of LLB256 makes it more difficult to compare their optimizations in detail to ours because, as we show in Section 5, the interesting behavior of HyTMs (and arguably, the target workload for best-effort HTM) appears with workloads in which software transactions are not rare.

As shown in Table 3, the new HyTM algorithms that we present in this paper improve on previous designs. In the class without oreos, HyNOrec-2 provides a high level of concurrency and good scalability while not wasting HTM capacity and requiring only a very small runtime overhead. For HyTMs with oreos, HyLSA features either lower HTM capacity requirements or a smaller runtime overhead.

3. THE HYBRID LAZY SNAPSHOT ALGORITHM

Our first algorithm extends the lazy snapshot algorithm (LSA) first presented in [25]. LSA is a time-based STM algorithm that uses on-demand validation and a global time base to build a consistent snapshot of the values accessed by a transaction. The basic version of the LSA algorithm is shown in Algorithm 2 and briefly described below (please refer to the original paper for further details [25]).\(^6\)

Transaction stores are buffered until commit. The consistency of the snapshot read by the transaction is checked based on versioned locks (ownership records, or oreos for short) and a global time base, which is typically implemented using a shared counter. The oreo protecting a given memory location is determined by hashing the address and looking up the associated entry in a global array of oreos. Note that, in this design, an oreo protects multiple memory locations.

To install its updates during commit, a transaction first acquires the locks that cover updated memory locations (line 38) and obtains a new commit time from the global time base by incrementing it atomically (line 43). The transaction subsequently validates that the values it has read have not changed (lines 45 and 53–57) and, if so, writes back its updates to shared memory (lines 48–49). Finally, when releasing the locks, the versions of the oreos are set to the commit time (lines 51–52). Reading transactions can thus see the value of a memory location at the commit time.

\(^{3}\)It first checks the version in an ownership record and then loads data speculatively. Executing these steps in reverse order fixes this problem.

\(^{6}\)Because of our efforts to present algorithms using a common notation, the presentation of LSA and NOrec [8] differ slightly from the versions found in the original papers. Notice that we use the notation cas(addr : expected-val \rightarrow new-val) for the compare-and-set operation.
virtual commit time of the updated memory locations and use it to check the consistency of their read set. If all loads did not virtually happen at the same time, the snapshot is inconsistent.

A snapshot can be extended by validating that values previously read are valid at extension time, which is guaranteed if the versions in the associated orecs have not changed. LSA tries to extend the snapshot when reading a value protected by an orec with a version number more recent than the snapshot’s upper bound (line 25), as well as when committing to extend the snapshot up to the commit time, which represents the linearization point of the transaction (line 45).

We now describe the hybrid extensions of LSA using eager conflict detection (shown in Algorithm 3). A variant with lazy conflict detection is presented in the companion technical report [23]. Note that the HyTM decides at runtime whether to execute in hardware or software mode, as explained in Section 2 and Algorithm 1.

Transitional loads first perform an ASF-protected load of the associated orec (line 6). This operation starts monitoring of the orec owned by (other) software transaction (see Section 2.1). If the transaction is not aborted before returning a value, this means that the orecs associated with this address and all previously read addresses have not changed and are not locked, thus creating an atomic snapshot.

This represents an application of the first of the synchronization techniques listed in Table 1: We only monitor metadata (i.e., the orec) but read application data nonspeculatively. This enables the HyTM to influence the HTM capacity required for transactions via its mapping from data to metadata, which in turn can make best-effort HTM usable even if transactions have to read more application data than provided by the HTM’s capacity. In turn, the HTM has to guarantee that the monitoring starts before the nonspeculative load.

### Algorithm 2 LSA STM algorithm (encounter-time locking/write-back variant) [13]

```plaintext
1. Global state:
2. clock ← 0
3. orecs: word-sized ownership records, each consisting of:
4. locked: bit indicating if orec is locked
5. owner: thread owning the orec (if locked)
6. version: version number (if locked)
7. State of thread p:
8. lb: lower bound of snapshot
9. ub: upper bound of snapshot
10. r-set: read set of tuples (addr, val, ver)
11. w-set: write set of tuples (addr, val)
12. stm-start(p):
13. lb ← ab ← clock
14. r-set ← w-set ← Ø
15. stm-load(orec addr, val):
16. (orec, val) ← (orecs[hash(addr)], +addr) ⊢ post-validated atomic read [13]
17. if orec locked then
18. if orec.owner ≠ p then
19. abort() ⊢ orec owned by other thread
20. if (addr, new-val, s) ∈ w-set then
21. val ← new-val ⊢ update write set entry
22. else
23. if orec version ≠ ub then
24. if ¬ validate() then
25. abort() ⊢ cannot extend snapshot
26. val ← addval
27. r-set ← r-set ∪ {(addr, val, orec.version)} ⊢ add to read set
28. return val
29. stm-store(addr, val, p):
30. orec ← orecs[hash(addr)]
31. if orec.locked then
32. if orec.owner ≠ p then
33. abort() ⊢ orec owned by other thread
34. else
35. if (addr, s, ver) ∈ r-set ∧ ver ≠ orec.version then
36. abort() ⊢ read different version earlier
37. if ¬ cas(orecs[hash(addr)] : orec ← (true, p)) then
38. abort() ⊢ cannot acquire orec
39. w-set ← w-set \ {(addr, s) ∪ {(addr, val)}} ⊢ add to write set
40. stm-commit(p):
41. if w-set ≠ Ø then
42. if is transaction read-only? then
43. commit timestamp
44. if ub ≠ lb + 1 then
45. if ¬ validate() then
46. abort() ⊢ cannot extend snapshot
47. o-set ← Ø ⊢ set of orecs updated by transaction
48. for all (addr, val) ∈ w-set do
49. add val to o-set ∪ {hash(addr)}
50. if all o ∈ o-set do
51. orec[ ] ← (false, ub) ⊢ release orecs
52. validate():
53. for all (addr, val, ver) ∈ r-set do
54. Are orecs free and version unchanged?
55. orec ← orecs[hash(addr)]
56. if (orec.locked ∧ orec.owner ≠ p) ∨ ¬ (orec.locked ∧ orec.version ≠ ver) then
57. abort() ⊢ inconsistent snapshot
```

### Algorithm 3 HyLSA — Eager variant (extends Algorithm 2)

```plaintext
1. State of thread p:
2. o-set: set of orecs updated by transaction
3. htm-start():
4. o-set ← Ø
5. htm-load(addr, p):
6. LOCK MV: orec ← orecs[hash(addr)] ⊢ protected load
7. if orec.locked then
8. ABORT ⊢ orec owned by (other) software transaction
9. val ← addr ⊢ nonspeculative load
10. return val
11. htm-store(addr, val, p):
12. LOCK MV: orec ← orecs[hash(addr)] ⊢ protected load
13. if orec.locked then
14. ABORT ⊢ orec owned by (other) software transaction
15. LOCK PREFETCHW orec ⊢ watch for concurrent loads/stores
16. LOCK MV: addt ← val ⊢ speculative write
17. o-set ← o-set ∪ {hash(addr)}
18. htm-commit():
19. if o ≠ 0 then
20. ABORT ⊢ is transaction read-only?
21. for all v ∈ o-set do
22. LOCK MV: orec[ ] ← (false, ct) ⊢ commit timestamp
23. COMMIT ⊢ commit hardware transaction
```

Transactional stores proceed as loads, first monitoring the orec and verifying that it is not locked (lines 12–14). The transaction then watches the orec for reads and writes by other transactions (PREFETCHW on line 15). The operation effectively ensures eager detection of conflicts with concurrent transactions. Finally, the updated memory location is speculatively written (line 16).

Upon commit, an update transaction first acquires a unique commit timestamp from the global time base (line 20). This will be ordered after the start of monitoring of previously accessed orecs, but will become visible to other threads before the transaction’s commit (see Section 2.1). Next, it speculatively writes all updated orecs (lines 21–22), and finally tries to commit the transaction (line 23). Note that these steps are thus ordered in the same way as the equivalent
alent steps in a software transaction (i.e., acquiring orecs or recording orec version numbers before incrementing clock, and validating orec version numbers or releasing orecs afterwards). If the transaction commits successfully, then we know that no other transaction performed conflicting accesses to the orecs (representing data conflicts). Thus, the hardware transaction could have equally been a software transaction that acquired write locks for its orecs and validated that their version numbers were not changed. If the hardware transaction aborts, then it only might have incremented clock, which is harmless because other transactions cannot distinguish this from a software update transaction that did not update any values that have they have read.

By non specularly incrementing clock (line 20), a hardware update transaction sends a synchronization message to software transactions, notifying them that they might have to validate due to pending hardware transaction commits. It is thus an application of the second general-purpose technique in Table 1. Because ASF provides non speculative atomic read-modify-write (RMW) operations, hardware transactions can very efficiently send such messages. In contrast, using speculative stores would lead to frequent aborts caused by consumers of those messages. If using just non speculative stores instead of RMW operations, concurrent transactions would have to write to separate locations to avoid lost updates, which in turn would require observers to check many different locations. In the case of HyLSA, this would also prevent the efficiency that is gained by using a single global time basis. The ordering guarantees that ASF provides for non speculative atomic RMW operations are essential because it allows hardware transactions to send messages after monitoring data and before commit or monitoring further data.

To ensure privatization safety for the hybrid LSA algorithm, we use a typical STM quiescence-based protocol (not shown in the pseudo-code). Basically, update transactions potentially privatize data, so they have to wait until concurrent transactions that might have accessed the updated locations have finished or have extended their snapshot far enough into the future (so that they would have observed the updates). Because hardware transactions will be aborted immediately by conflicting updates, their snapshot is always most recent and we do not need to wait for them.

4. THE HYBRID NOREC ALGORITHMS

In this section, we take another algorithm from the literature, NOREC [8], and discuss how to turn it into a scalable HyTM for ASF. Roughly speaking, NOrec uses a single orec (a global versioned lock) and relies on value-based validation (VBV) in addition to time-based validation. The basic version of the NOrec algorithm and its original hybrid variant (HyNOREC-DS), as informally described by Dalessandro et al. [8], are briefly summarized here and further described in a companion technical report [23]. To simplify the presentation, we will show these algorithms embodied in or as a modification of our initial HyTM algorithm. NOrec is basically the STM code in Algorithm 4 when discarding esl.

The NOrec algorithm is quite similar, when ignoring VBV, to timestamp-based TMs like TL2 [9] or LSA [25]. The main difference with the description given in Section 3 is that NOrec uses a single orec (gsl, line 2) and does not acquire the lock before attempting to commit a transaction. As a consequence, it yields a very simple implementation and allows for a few optimizations. In particular, it is not necessary to track which locks are covering loads or stores, and the lock itself can serve as time base (lines 30/36, 13, and 40). However, such a design would not scale well when update transactions commit frequently because timestamp-based validation would also fail frequently (e.g., in the checks on lines 21 and 30). Therefore, NOrec attempts value-based validation (lines 42–44) whenever timestamp-based validation is not successful (lines 22 and 31).

With VBV, the consistency of a transaction’s read set is verified on the basis of the values that have been loaded instead of the versions of the orecs. The disadvantage of using values is that one has to potentially track more data in the read set because several ad-

---

**Algorithm 4** HyNOrec-0: STM acquires locks separately

1. Global state:
   2. gsl: word-sized global sequence lock, consisting of:
   3. locked: most significant bit, true iff locked
   4. clock: clock (remaining bits)
   5. esl: extra sequence lock

6. State of thread ξ:
   7. sl: thread local sequence lock
   8. r-set: read set of tuples (addr, val)
   9. w-set: write set of tuples (addr, val)
10. update: are we in an update transaction?

11. stm-start():
12. repeat
13.     sl ← gsl  # get the transaction’s start time
14.     until ⌈sl.locked⌉  # wait until concurrent commits have finished
15.     r-set ← w-set ∪ {addr ← ∗}  # wait until concurrent commits have finished
16.     stm-commit():
17.     if (addr, new-val) ∈ w-set then  # read after write?
18.         val ← new-val
19.     else
20.         val ← addr
21.         while sl ≠ gl do  # timestamp-based validation
22.             sl ← validate()  # value-based validation
23.             val ← addr
24.         r-set ← r-set ∪ (addr, val)
25.         return val
26.     stm-store(addr, val):
27.     w-set ← w-set ∪ {(addr, val)}  # updates are buffered
28.     stm-commit():
29.     if w-set ≠ ∅ then  # is transaction read-only?
30.         while (cas(gsl, sl → (true, sl.clock)) do  # acquire commit lock
31.             sl ← validate()  # value-based validation
32.             esl ← (true, sl.clock)  # also acquire extra lock (no need for cas)
33.         for all (addr, val) ∈ w-set do  # write updates to memory
34.             val ← valu  # value-based validation
35.             esl ← (false, sl.clock + 1)  # release locks and increment clock
36.             gsl ← (false, sl.clock + 1)
37.     validate():
38.     repeat
39.         until sl ≠ gl  # get current time
40.         c ← gsl  # get current time
41.         until c.locked  # wait until concurrent commits have finished
42.         for all (addr, val) ∈ r-set do  # inconsistent snapshot
43.             if addr ≠ val then
44.                 abort()  # inconsistent snapshot
45.                 until sl ← gl  # get current time
46.         until sl ← gl  # return c
47.     htm-start():
48.         LOCK MOV: l ← esl  # protected load (monitor extra lock)
49.         if llocked then  # extra lock available?
50.             ABORT  # no: spin by explicit self-abort
51.         update ← false  # initially not an update transaction
52.     htm-load(addr, val):
53.         LOCK MOV: valu ← addr  # speculative write
54.         update ← true  # speculative write
55.         if update then  # we are in an update transaction
56.             if valu ≠ esl  # main lock available?
57.                 ABORT  # no: we will be aborted anyway
58.                 LOCK MOV: gsl ← (false, l.clock + 1)  # release lock, incr. clock
59.             COMMIT  # commit hardware transaction
Algorithm 5 HyNOrec-DSS: HyTM by Dalessandro et al. [8] (extends Algorithm 4)

1. stm-acquire-locks();
2. SPECULATE  ▷ start hardware transaction (retire code omitted)
3. LOCK MOV : l ← gsl
4. if l = 0 then
5. LOCK MOV : gsl ← (true, sl, clock)  ▷ try to acquire commit lock
6. LOCK MOV : esl ← (true, sl, clock)  ▷ also acquire extra lock
7. COMMIT
8. return l = sl  ▷ true ⇔ locks were acquired atomically
9. stm-commit();  ▷ replaces function of Algorithm 4
10. if w-set # Φ then  ▷ in transaction read-only?
11. while ¬ stm-acquire-locks() do  ▷ acquire gsl and esl atomically
12. sl ← validate)
13. for all (addr, val) ∈ w-set do  ▷ write updates to memory
14. addr ← val
15. esl ← (false, sl, clock + 1)  ▷ may abort hardware transaction
16. gsl ← (false, sl, clock + 1)  ▷ release lock and increment clock
17. htm-store(addr, val);  ▷ replaces function of Algorithm 4
18. LOCK MOV : addr ← val  ▷ speculative write
19. htm-commit();  ▷ replaces function of Algorithm 4
20. LOCK MOV : l ← gsl
21. LOCK MOV : gsl ← (false, l, clock + 1)  ▷ release lock and increment clock
22. COMMIT  ▷ commit hardware transaction

Algorithm 6 HyNOrec-1: HyTM writes gsl non-speculatively (extends Algorithm 4)

1. htm-start();  ▷ replaces function of Algorithm 4
2. wait until ′esl locked  ▷ spin while extra lock unavailable
3. LOCK MOV : l ← esl
4. if locked then
5. ABORT  ▷ extra lock available?
6. update ← false  ▷ no explicit self-abort
7. htm-commit();  ▷ replaces function of Algorithm 4
8. if update then
9. l ← atomic-fetch-and-increment(gsl)  ▷ increment gsl clock (gsl locked is MSB)
10. if l locked then
11. ABORT  ▷ main lock unavailable, we will be aborted anyway
12. COMMIT  ▷ commit hardware transaction

Dresses often map to the same ore. VBV is typically paired with serialized commit phases. In NOrec, this is enforced on lines 14 and 41.

Our implementation of NOrec differs in a few points from the original implementation [8]. Notably, in our implementation, when writing back buffered updates upon commit, we only write to precisely those bytes that were modified by the application, whereas the original implementation always performs updates at the granularity of aligned machine words. This more complex bookkeeping introduces higher runtime overheads but is required for the STM to operate correctly according to the C/C++ TM specification [15].

The reason for creating a hybrid extension to NOrec is that this algorithm can potentially provide better performance for low thread counts because it does not have to pay the runtime overheads associated with accessing multiple ores. In turn, LSA is expected to provide better scalability with large thread counts or frequent but disjoint commits of software transactions. Therefore, both algorithms are of practical interest depending on the target architecture and workload.

The main approach of HyNOrec-DSS (Algorithm 5) is to use two global sequence locks, gsl and esl. Software transactions acquire both locks on commit and increment their version numbers after committing, whereas hardware transactions monitor esl for changes and increment only gsl’s version on commit. Thus, software transactions are notified about data updates using gsl, and will use esl to abort hardware transactions and prevent them from executing during software commits. From the perspective of software transactions, committed hardware transactions are thus equivalent to software transactions that committed atomically.

The major problem of HyNOrec-DSS is that it does not scale well in practice (see Section 5). For example, Dalessandro et al. assume [8] that the update of the contented gsl by every hardware transaction (line 21 in Algorithm 5) is not a performance problem because it would happen close to the end of a transaction. However, we observed in experimental evaluation a high rate of aborts and poor overall performance for this algorithm.

In what follows, we will construct a new algorithm, HyNOrec-2, which performs much better while being no more complex. gsl and esl are used by software and hardware transactions to synchronize with each other, so our key approach is to apply the last two techniques from Table 1 and use non-speculative operations to let hardware transactions synchronize more efficiently via these variables. To better explain and evaluate the different optimizations involved, we additionally show two intermediate algorithms.

Algorithm 4 shows our first (intermediate) NOrec-based HyTM, this time considering the addition of esl, which will serve as the basis for the other two variants. As a first straightforward optimization, a hardware transaction has to update gsl only if it will actually update shared state on commit (line 59).

Second, we do not need to use a small hardware transaction to update both gsl and esl in at-commit. This is not necessary because esl is purely used to notify hardware transactions about software commits and can only be modified by a software transaction that previously acquired gsl (line 30). In contrast to Algorithm 5, this allows hardware transactions to try to commit at a time where gsl has been acquired but esl has not yet been updated (which would have aborted the hardware transaction). However, this case can be handled by just letting the hardware transaction abort if gsl has been locked (line 62).

This second change is not about performance but it allows us to have a software fallback path in the HyTM that does not depend on HTM progress guarantees (e.g., no spurious aborts), which are surprisingly difficult to implement [11]. Also, programs can use the software path in the HyTM as is on hardware that does not support ASF.

Algorithm 4 can still suffer from conflicts on gsl if updating hardware transactions commit frequently. Algorithm 6 shows that we can replace the speculative update of gsl with a non-speculative atomic fetch-and-increment instruction (line 9),7 which allows hardware transactions that access disjoint data to not abort each other anymore and makes the algorithm scale better. This is an application of our second general-purpose technique and has similarities to acquiring a commit time non-speculatively in HyLSA (see Section 3 for a detailed discussion).

To understand why this is possible, consider possible orderings of the hardware transaction’s fetch-and-increment and a software transaction’s compare-and-set (CAS) on gsl. If the increment gets ordered first, the CAS will fail and will cause a software transaction validation. If the software transaction accesses during validation any updates of the hardware transaction before the former can commit, it will abort the hardware transaction, making the situation look like if some transaction committed without updating anything. If in contrast the CAS comes first, the hardware transaction will

7As a matter of fact, esl.clock can contain any value as long as the lock bit is updated properly because such an update will abort hardware transactions monitoring esl.

8Note that the fetch-and-incremented will be ordered before the commit of the transaction. Also, using a typical compare-and-set loop instead of the fetch-and-increment yields lower performance according to our experiments.
notice that gsl was locked before it incremented gsl and will abort. The hardware transaction’s update to gsl is harmless because no transaction interprets gsl.clock if gsl is locked. Additionally, hardware transactions spin nonspeculatively if esl is locked before accessing it speculatively to avoid unnecessary aborts (line 2).

The remaining problem of Algorithm 5 is that committing a software transaction aborts all hardware transactions that execute concurrently. One might see this as a minor issue assuming that, typically, software transactions are much longer than hardware transactions, but this is not necessarily the case. There are several reasons why a transaction cannot use ASF. For example because it contains instructions that are not allowed in ASF speculative regions (e.g., rdtsc), or because its access pattern quickly exceeds the associativity of the cache used to track the speculative loads, hence leading to capacity aborts after only few accesses.

Fortunately, software transactions can commit without having to abort nonconflicting hardware transactions. The key insight to understand this second extension is that the monitoring in hardware transactions is like an over-cautious form of continuous value-based validation (any conflicting access to a speculatively accessed cache line will abort a transaction). In NOrec, software transactions tolerate concurrent commits of other transactions by performing value-based validation when necessary.

Our final optimization is shown in Algorithm 7. Hardware transactions do not monitor esl using speculative accesses anymore. The purpose of esl is to prevent hardware transactions from reading inconsistent state such as partial updates by software transactions. To detect such cases and still obtain a consistent snapshot, hardware transactions first read the data speculatively (line 4) and then wait until they observe with nonspeculative loads that esl is not locked (line 5). If this succeeds and the transaction reaches line 6 without being aborted, it is guaranteed that it had a consistent snapshot at line 5 at a time when there were no concurrent commits by software transactions. Again, note that ASF will have started monitoring the data before performing the subsequent nonspeculative loads.

The reasoning for waiting until gsl is not locked on line 10 is similar and just applied to the commit optimization in HyNOrec-1. Waiting for gsl is as good as waiting for esl because esl will be locked iff gsl is locked (see Algorithm 4).

Thus, hardware transactions essentially validate against commit messages by software transactions (the third general-purpose technique in Table 1). This consists of the nonspeculative spinning on esl (reading commit messages by software transactions) combined with the implicit value-based validation performed by ASF monitoring the data accessed by the hardware transaction. The nonspeculative accesses allow hardware transactions to observe and tolerate software commits that create no data conflicts (i.e., pass value-based validation).

Note that esl could be removed and replaced by just gsl. A downside of this approach is that it would increase the number of cache misses on line 5 because both hardware and software commits would update the same lock. Therefore, we keep the separation between gsl and esl.

5. EVALUATION

To evaluate the performance of our HyTMs, we use a similar experimental setup as in a previous study [5]. We simulate a machine with sixteen x86 CPU cores on a single socket, each having a clock speed of 2.2 GHz. The simulator is near-cycle-accurate (e.g., it simulates details of out-of-order execution). We evaluate three ASF implementations. Two of them, “LLB8” and “LLB256” can track/ buffer speculative loads and store in a fully-associative buffer that holds up to 8 or 256 distinct cache lines. “LLBSL1” is a variant which uses only buffers for speculative stores but uses the L1 cache to track transactional loads. We show these ASF implementations because they have different costs when implemented in a microprocessor (e.g., required chip area). LLB8 will have to resort to the STM code path often because most transactions will exceed its capacity. LLB256 is sufficient to run almost all transactions in our benchmarks in the HTM code path, but is more expensive. LLBSL1 represents a middle ground. Its capacity for loads is limited by either the cache’s size (1024 lines) or its associativity (2-way).

In order to get the best performance, the compiler links in the TM library statically and optimizes the code by inlining TM functions. The STM implementations that we compare against are “LSA” (a version of TinySTM [20] using write-through mode, eager conflict detection, and ensuring privatization-safety, similar to Algorithm 2) and “NOrec” [8] (similar to the STM code in Algorithm 4). The baseline HTM (“HTM”) uses serial-irreversible mode as simple software fallback. The HyTM implementations have the same names as the respective algorithms (e.g., Algorithm 7 is denoted “HyNOrec-2”) and use the LSA and NOrec implementations for their software code paths.

As benchmarks, we use selected applications from the STAMP TM benchmark suite [4] and the typical integer set microbenchmarks (IntegerSet). The latter are implementations of a sorted set of integers based on a skip list, a red-black tree, a hash table, and a linked list. During runtime, several threads use transactions to repeatedly execute insert, remove, or contains operations on the set (operations and elements are chosen randomly). All set elements are within a certain key range, and the set is initially half full. Table 4 shows the configurations that we consider. In HashTable all transactions are update transactions (insert or remove operations), in all other benchmarks the update rate is 20%. However, these operations only insert (remove) an element if it is absent from (part of) the set, so the actual percentage of update transactions can be smaller. We use the Hoard memory allocator [2] in HashTable and glibc 2.10 standard malloc in the other benchmarks.

Because we do not have enough space to show all measurements in the same level of detail, we first present a few interesting cases. Table 4 shows which percentage of transaction commits happen on the hardware code path in comparison to the total number of commits. LLB256 provides sufficient capacity to execute all transactions in our IntegerSet configurations in hardware. In contrast, LLBB8’s capacity is most often too small. Note that in our implementations, only permanent ASF abort reasons like exceeding ASF’s capacity make the HyTM switch to the software code path. Contention will not result in such a switch unless a transaction suf-
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Range</th>
<th>Commits on hardware code path (%)</th>
<th>LLB8</th>
<th>LLB8L1</th>
<th>LLB256</th>
</tr>
</thead>
<tbody>
<tr>
<td>SkipList Large</td>
<td>LLB8</td>
<td>&lt;1%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SkipList Small</td>
<td>LLB8</td>
<td>&lt;1%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RBTree Large</td>
<td>LLB8</td>
<td>0–2%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RBTree Small</td>
<td>LLB8</td>
<td>2–10%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HashTable</td>
<td>LLB8</td>
<td>100%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LinkedList Large</td>
<td>LLB8</td>
<td>1–3%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LinkedList Small</td>
<td>LLB8</td>
<td>30–60%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: IntegerSet microbenchmarks and approximate ratio of HTM commits to total number of commits.

Figure 1: Ratio of HTM commits to total number of commits.

Figure 2: Comparison of HyNOrec algorithms. The hardware contention abort rate is the number of aborts due to ASF contention per transaction that commits in hardware or switches to the software codepath.

Figure 3: Comparison of HyNOrec algorithms (HashTable).

Figure 4: Ratio of HTM commits to total number of commits for 8 threads and read-only LinkedList of various sizes (range), when accessing data speculatively (SDL) or not.

will not be aborted by commits of nonconflicting software transactions, and will find out quickly that it should switch to software, then taking advantage of STM scalability.

When using LLB8L1 (right side), many transactions can execute in hardware (see Table 4 and Figure 1). HyNOrec-2 also scales much better in this case, showing that its ability to survive commits of nonconflicting software transactions (e.g., in contrast to HyNOrec-1) is beneficial even when the majority of transactions execute in hardware. Furthermore, HyNOrec-DSS suffers from many more aborts than the other TMs. To explain this, we show results for HyNOrec-DSSU, which is like HyNOrec-DSS but only updates gsl when update transactions commit, thus reducing the number of speculative updates to gsl (SkipList has 20% update transactions). HyNOrec-DSSU performs similar to HyNOrec-0, indicating that this part of the HyNOrec-0 optimizations is crucial. HyNOrec-DSS never performed better than HyNOrec-0 in any of our benchmarks and often performed significantly worse. It does not scale beyond 4 to 6 threads in IntegerSet unless transactions execute the software code path most of the time. Therefore, we discard HyNOrec-DSS from now on.

Figure 3 shows HashTable, which runs short and mostly update transactions. HyNOrec-1 performs and scales much better than HyNOrec-0 and suffers from very few aborts, whereas the rate of aborts due to contention is still significant for HyNOrec-0. This shows that updating gsl nonspeculatively is an important optimization, especially if commits of update transactions are frequent. Second, it highlights that updating gsl speculatively can indeed lead to contention.

ASF capacity requirements for execution under the HyNOrec TMs are similar to HTM. However, even though HyNOrec-2 does not access more data speculatively than an HTM, it can effectively reach capacity limits earlier. It has to always check esl, which keeps gsl in the cache and can thus reduce the capacity limit by one, which can matter if the effective limit is the cache associativity.
After looking at the HyNOrec algorithms, let us now focus on HyLSA. Its capacity requirements are different than those of HyNOrec. HyLSA buffers updates speculatively and, thus, for stores, needs ASF capacity for both data and orecs. However, for loads, only orecs are accessed speculatively, and the hash function that maps data to orecs influences capacity requirements. In our implementations, word-sized data are mapped to word-sized orecs (i.e., we discard the lower three bits of an address and select with the remaining bits a slot in an array with $2^{20}$ orecs). Orecs are not cache-line padded because padding would likely increase capacity requirements for HyLSA unless more than one adjacent cache line maps to the same orec. Without padding, hardware transactions detect conflicts on cacheline granularity, whereas STM transactions can detect conflicts on word-size granularity and can thus potentially scale better in high-contention workloads.

Table 4 and Figure 1 show that HyLSA is already more likely to hit capacity limitations than HyNOrec just because it needs twice the capacity for stores, so it is important for HyLSA to read data nonspeculatively. Figure 4 illustrates this point further, showing that when accessing data and orecs speculatively (HyLSA-eager-SDL), less transactions can execute the hardware code path. HASTM in aggressive mode and the HyTM by Damron et al. also suffer from this (see Table 3).

Figure 5 presents a concluding overview of TM performance with the integer set microbenchmarks. We show configurations that are representative or that highlight interesting properties. HashTable performs similar on all ASF implementations and scales very well, but ultimately suffers from external bottlenecks (e.g., the memory allocator). LLB8 is (on all other benchmarks) not sufficient to run many transactions in hardware, and STMs perform slightly better than HyTMs because the latter try to first execute in hardware (unsuccessfully).

HyNOrec-2 has very good overall performance, especially on scalable workloads. It is not aborted by nonconflicting concurrent commits of software transactions, which is one of the reasons why it performs better than HyNOrec-0 (e.g., in SkipList-Large on LLB8L1, see Figure 1 for HTM ratio). Pure HTM has the lowest overhead but its simple fallback mode (serial execution) can quickly decrease its performance. HyLSA has higher runtime overhead than HyNOrec but typically scales well.

In the small LinkedList on LLB256, all transactions can execute in hardware but HyTMs and HTM do not scale. The reason for this behavior is that ASF’s conflict detection is on the granularity of cache lines, whereas STMs can use smaller granularities (word-sized in LSA, value-based validation in NOrec), which can be beneficial in high-contention workloads with a high level of false sharing. As explained before, HyLSA could use the indirectness of the orecs and the memory-to-orec hash function to emulate a smaller granularity for conflict detection. However, this will waste ASF capacity and thus does not seem to be a generally useful strategy. Instead, a HyTM should perhaps switch proactively to software to try to employ a more contention-resistant STM algorithm.

To conclude the evaluation, we show performance results for selected applications from STAMP (see Table 5) in Figure 6. We chose benchmarks that are stable and have parallelism in their workloads, and executed them using STAMP’s standard parameter configurations for simulator environments. LLB256 is again sufficient to execute all transactions in hardware. SSCA2 and KMeans have small transactions, but interestingly HyNOrec-0 seems to require just a little too much capacity (in contrast to the other

![Figure 5: Overview of scalability of TMs with IntegerSet.](image)

Table 5: Approximate ratio of HTM commits to total number of commits in STAMP.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LLB8</th>
<th>LLB8-L1</th>
<th>LLB256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genome</td>
<td>HTM, HyNOrec-2: 65%</td>
<td>HTM: 90-95%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>HyNOrec-1: 60%</td>
<td>HyNOrec: 85-90%</td>
<td>75%</td>
</tr>
<tr>
<td></td>
<td>HyLSA: 38-42%</td>
<td>HyLSA: 75%</td>
<td>100%</td>
</tr>
<tr>
<td>KMeans-Hi</td>
<td>HTM, HyNOrec-[12]: 100%</td>
<td>HTM: 95-100%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>HyLSA, HyNOrec: 25%</td>
<td>95-100%</td>
<td>100%</td>
</tr>
<tr>
<td>Vacation-Hi</td>
<td>0%</td>
<td>HTM: 13-14%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>HyNOrec: 9-12%</td>
<td>HyNOrec: 9-12%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>HyLSA: 3-5%</td>
<td>HyLSA: 3-5%</td>
<td>100%</td>
</tr>
<tr>
<td>Vacation-Lo</td>
<td>0%</td>
<td>HTM: 8-11%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>HyNOrec: 6-9%</td>
<td>HyNOrec: 6-9%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>HyLSA: 1-2%</td>
<td>HyLSA: 1-2%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Unfortunately, the current version of the ASF simulator does not provide the ASF ordering guarantees for nonspeculative accesses in all cases. To be able to run the same HyLSA TMs in all benchmarks, we had to add memory barriers (i.e., an `fence` instruction) between the speculative load of an orec and the nonspeculative load of data (e.g., lines 6 and 9 in Algorithm 3).
HyNOrec TMs, it accesses \textit{ esl} and \textit{gsl} speculatively). Genome on LLB8 also exhibits this behavior. HyLSA’s larger capacity requirements for stores decrease the HTM ratio as well.

HyNOrec-2 performs best among the HyTMs most of the time and is often close to or better than HTM. Its performance suffers in Genome due to its software fallback (NOrec) performing worse than LSA. It often performs much better than HyNOrec-0 (and HyNOrec-DSS), thus demonstrating the benefits of our optimizations. HyLSA has higher runtime overhead than HyNOrec but scales well.

6. CONCLUSION

In this paper, we have proposed and evaluated novel hybrid software/hardware transactional memory algorithms. As shown in Table 3, they improve upon previous HyTM algorithms by either allowing for a larger level of concurrency between hardware and software transactions, by reducing runtime overhead of hardware transactions, or by requiring less HTM capacity and thus allowing more transactions to run with hardware acceleration. We confirmed this through experimental evaluation on a near-cycle-accurate x86 simulator with support for AMD’s ASF hardware extensions.

While previous HyTM designs have used nonspeculative memory accesses inside of hardware transactions, we show that this has a much larger potential and importance if algorithms also make use of nonspeculative atomic read-modify-write instructions. We also found it very useful that ASF monitors speculatively accessed locations eagerly for conflicting accesses by other threads. We believe that the general-purpose techniques that we used in our algorithms (Table 1) apply not just to HyTM but can be useful in general for concurrent algorithms based on new synchronization hardware like ASF.

Acknowledgements. We would like to thank Stephan Diestelhorst of AMD for his clarifications regarding the ASF specification, and Tim Harris for his suggestions in improving this paper. The research leading to these results has received funding from the European Community’s Seventh Framework Programme (FP7/2007-2013) under grant agreement No 216852.

7. REFERENCES


![Figure 6: Overview of scalability of TMs in selected STAMP benchmarks. SSCA2 performs similar on all ASF implementations. KMeans-Lo performs roughly similar to KMeans-Hi, and KMeans-Hi on LLB8L1 is similar to LLB256. Vacation-Hi performs similar to Vacation-Lo, and Vacation-Lo on LLB8 is similar to LLB8L1.](image)


