Hardware acceleration for lock-free data structures and software-transactional memory

Stephan Diestelhorst*
Technische Universität Dresden
Systems Engineering Group
Dresden, Germany
stephan.diestelhorst@inf.tu-dresden.de

Michael Hohmuth
Advanced Micro Devices
Operating System Research Center
Dresden, Germany
michael.hohmuth@amd.com

ABSTRACT

In this paper, we report on a new CPU-architecture extension proposal, named Advanced Synchronization Facility (ASF), which is geared toward accelerating and easing lock-free programming and software transactional memory (STM). We present an initial performance simulation and usability study of ASF's application to a lock-free data structure (a singly linked list) and to accelerating a state-of-the-art STM system, TinySTM. Our results indicate that ASF can significantly increase the throughput and scaling behavior of these workloads: Single-thread performance increased by up to 15 %, and the factor of scaling to eight CPUs increased by up to 20 %.

1. INTRODUCTION

Future CPU generations will no longer be able to increase their single-thread performance exponentially. Instead, CPUs will scale the number of processing cores. In consequence, software will no longer get faster execution speeds automatically with each hardware upgrade, but will have to be adapted to the higher level of parallelism exposed by the CPU. Existing parallelization techniques get more and more complex with an increasing number of execution threads, which is why the software industry is looking for new, less complex parallel programming paradigms.

Transactional memory is a promising programming model that provides transactions (known from database technology) that take the burden for synchronizing concurrent data access off programmers' backs. However, today's software implementations of transactional memory, known as *Software Transactional Memory* (*STM*), still inflict too much overhead for synchronization and bookkeeping, making STMs impractical for the CPU count to be expected in the near future. One way to reduce this overhead is to accelerate STMs with new hardware mechanisms.

Another promising programming paradigm is that of lock-free data structures. Many authors have shown that lock-free algorithms perform and scale well and are robust against deadlocks, but to date these algorithms have been limited by incomplete hardware support: Lock-free programming relies on atomically modifying a set of memory locations using instructions like test-and-set and compare-and-swap (CAS). However, these instructions typically operate on only one or two words of memory and have a high latency, making lock-free programming impractical for more complex data structures or when low latency is required.

In this paper, we introduce a new hardware acceleration mechanism, *Advanced Synchronization Facility* (*ASF*). ASF is an experimental AMD64 architecture extension originally intended for the acceleration of lock-free algorithms. We evaluate ASF in the contexts of lock-free data structures and STMs.

Our evaluation results indicate that ASF has excellent potential for lock-free data structure acceleration. For an integer set implemented as a singly linked list, the ASF-based implementation has both better single-thread performance and better scalability than a lock-based implementation and a conventional lock-free version based on CAS.

We also applied ASF to the implementation of an STM system, TinySTM [4]. In two STM workloads we examined, a red-black-tree-based integer set and a singly-linked-list-based one, we observed a significant increase in multiprocessor throughput. Single-thread performance was comparable to or better than the baseline STM system in each case.

This paper is organized as follows. Section 2 presents related work. In Section 3, we introduce ASF and demonstrate how it simplifies lock-free programming. Section 4 applies ASF to the example applications used in our performance study: a lock-free linked-list implementation and TinySTM. In Section 5, we describe our simulation and performance measurement environment and compare the ASF-accelerated applications developed in Section 4 to their conventional counterparts. We conclude the paper in Section 6 with an outlook on future research directions.

2. BACKGROUND AND RELATED WORK

2.1 Lock-free data structures

Lock-free data structures do not use locks to coordinate concurrent accesses, avoiding most drawbacks of traditional locks, such as deadlock and priority inversion. Herlihy [7] showed that, given an atomic CAS primitive, all concurrent data structures can be implemented in a lock-free manner. Despite this general proof, only few lock-free implementations exist, such as the singly linked list, introduced by Valois [16]. Harris' later attempt [5] fixes bugs and is conceptually simpler, suggesting that correct and well-performing lock-free implementations are not trivial to find.

ASF aims at making lock-free programming significantly easier by

*Stephan Diestelhorst contributed to this work while interning at Advanced Micro Devices.

Revision 1.1 (Aug 14, 2008)

providing a mechanism that is both more powerful and more flexible than traditional primitives such as CAS.

2.2 Transactional memory

Herlihy and Moss proposed transactional memory in [9], implemented in hardware. Recently, a large number of software implementations (STM) have been developed [8, 3, 6, 4], but despite steady improvements, they are still about an order of magnitude worse than native hardware in single-threaded performance. Hardware support to reduce this penalty has been proposed earlier. To keep architectural extensions modest, proposals primarily either (1) restrain the size of supported hardware transactions (e. g., HyTm [2, 10], PhTM [11]), or (2) limit the offered expressiveness (e. g., LogTM-SE [17], SigTM [14]), or both (HASTM [15]).

Each of these hardware approaches is accompanied by software that works around the limitations and provides the interface and features of STM: flexibility, expressiveness, and large transaction sizes.

ASF in contrast has a broader scope than only the acceleration of transactional memory and can be implemented with moderate hardware extensions. The result is a mechanism that has relatively small capacity (compared to those listed under (1)) and richer expressiveness (than those listed under (2)), but requires a more static setup than hardware proposals under both (1) and (2) and STMs.

2.3 Simulation

Evaluation of new hardware-extension proposals requires simulators that can provide accurate timing information. Besides the internal tools employed by CPU vendors, various tools model the microarchitecture of a modern out-of-order processor [1, 12, 13]. This paper uses PTLsim [18], because unlike other solutions it is freely available and largely supports the AMD64 instruction set. In addition, it supports *co-simulation*, transparent switching between simulation and native hardware to quickly execute uninteresting parts of the application under test. PTLsim also supports full-system simulation and features a rich CPU model, which provides detailed architectural statistics.

3. ADVANCED SYNCHRONIZATION FACILITY (ASF)

3.1 Overview

ASF is an experimental AMD64 extension that allows user- and system-level code to modify a set of memory objects atomically without requiring expensive synchronization mechanisms.

The ASF extension provides an inexpensive primitive from which higher-level synchronization mechanisms can be synthesized: for example, multi-word compare-and-exchange, load-locked-store-conditional, lock-free data structures, and primitives for software-transactional memory.

ASF is both more flexible and faster than existing lock-free atomic memory-modification approaches. Instead of offering new instructions with hardwired semantics (such as compare-and-exchange for two independent memory locations), ASF only exposes a mechanism for atomically updating multiple independent memory locations and allows software to implement the intended synchronization semantics.

ASF works by allowing software to declare critical sections that modify a specified set of protected memory locations. Protected memory that critical sections modify will become visible to other CPUs[1] either all at once (when the critical section finishes successfully) or never (if the critical section is aborted). CPUs can protect and speculatively modify up to 8 memory objects that can each be at most cache-line sized and need to be size-aligned. When ASF detects conflicting accesses to one of these objects, it aborts the critical section.

Unlike traditional critical sections, ASF critical sections do not require mutual exclusion. Multiple ASF critical sections on different CPUs can be active at the same time, allowing greater parallelism.

3.2 Critical section structure

ASF critical sections consist of two phases. In the first phase, the specification phase, software declares which memory objects should be protected. The second phase, the atomic phase, can modify these memory objects speculatively. If the atomic phase completes successfully, all such modifications become visible to all CPUs simultaneously and atomically. Otherwise, modifications to protected memory objects are discarded.

ASF introduces a set of new instructions that denote the beginning and end of ASF phases. An ASF critical section has the following structure:

- The specification phase is entered when the first declarator instruction, or *declarator*, (LOCK MOV, LOCK PREFETCH, and LOCK PREFETCHW instructions) occurs. Declarators are used to declare memory that ASF should protect.
- A VALIDATE instruction can be used in the specification phase to check whether any of the previously declared memory locations has been invalidated by a concurrent write operation.
- The ACQUIRE instruction denotes the end of the specification phase and the beginning of the atomic phase. ACQUIRE has a return code that signals whether the atomic phase has been entered successfully, and also sets the rFLAGS register accordingly. A return code of 0 signals success.
- ACQUIRE is followed by instructions that check the return code and jump to an error handler if it is not zero (typically just a JNZ).
- The atomic-phase instructions (standard x86 instructions, including standard load and store instructions) are executed.
- The COMMIT instruction denotes the end of the atomic phase.

Figure 1 shows example code that uses ASF to implement compare-and-exchange for two independent memory locations, dubbed *DCAS* for "double compare-and-swap." (This code uses immediate retry as the recovery strategy. A real implementation might have a more elaborate recovery strategy, for example exponential backoff.)

[1] In this paper, the term "CPU" refers to one logical CPU (one hardware thread executing x86 instructions), irrespective of how these logical CPUs are packaged. (Its use is synonymous to terms like "CPU core" and "x86 thread," which are not used in this paper.)

```
; DCAS Operation:
; IF ((mem1 = RAX) && (mem2 = RBX)) {
;   swap (mem1, RDI)
;   swap (mem2, RSI)
;   RCX = 1
; } ELSE {
;   RCX = 0
; }

DCAS:
retry:
    LOCK MOV R8, [mem1] ; Specification phase begins
    LOCK MOV R9, [mem2]
    ACQUIRE  RCX, 2     ; Try to enter atomic phase
    JNZ      retry      ; Retry if unsuccessful
    CMP      R8, RAX    ; Atomic-phase code
    JNZ      out
    CMP      R9, RBX
    JNZ      out
    MOV      [mem1], RDI
    MOV      RDI, RAX
    MOV      [mem2], RSI
    MOV      RSI, RBX
    MOV      RCX, 1
out:
    COMMIT              ; End of atomic phase
```

Figure 1: DCAS implemented using ASF

3.3 Critical section aborts

Critical sections can be aborted at any point because of contention, far control transfers (including those caused by interrupts and faults), or software aborts.

Specification phase aborts are signaled by an ACQUIRE return code. ACQUIRE has a count argument that must match the number of declarators, allowing it to detect whether an interrupt occurred in the specification phase.

ASF is an unusual x86 architecture feature in that ACQUIRE has setjmp-like semantics: Atomic-phase aborts not only discard all modifications to all modified protected memory objects, but also reset the instruction and stack pointers to the values they had when ACQUIRE was executed. This results in a reexecution of ACQUIRE, which now returns an error code and directs the control flow (via the following conditional jump) to the error handler.

When an interrupt occurs during a critical section, that critical section will be aborted. Note that before an interrupt or exception handler returns, operating-system code or other processes may have executed in the interim. This is of no consequence as no ASF-related state is maintained across context switches. Other processes may even have executed ASF critical sections that inspected or modified any of the locations targeted by the interrupted critical section. The interrupted software will simply reinspect the state of the shared data structure and attempt its critical section again.

3.4 Implementation and performance

There are several conceivable ways in which processors can implement ASF. Our architecture simulator currently implements ASF as follows:

- We implemented ASF on top of the cache-coherency protocol. Any contention for a protected cache line will abort the critical section in question.
- The back-up copies of protected memory locations (to be written back in case of an abort) are held in a separate per-CPU buffer, called the *locked-line buffer* (LLB).
- Our pipeline allows only one ASF critical section in flight, thereby serializing all of a CPU's critical sections. Declarator instructions starting a new critical section are prevented from issuing until the previous critical section's COMMIT instruction has been retired. In consequence, the processor does not have to track independent lock sets.

The LLB allows CPUs to evict protected, speculatively modified memory out of the caches if necessary. Despite not being part of the memory hierarchy—the LLB only holds backup data—it participates in the cache-coherency protocol and monitors for contention for protected memory regions. If the LLB detects a contending probe, it holds off the probe response until the backup copies have been written back to the memory hierarchy.

Our design is easy to implement in an existing architecture, but limits the instruction-level parallelism (ILP) that can be exploited because the CPU cannot speculate across multiple critical sections. Other than that, all ASF instructions can be fully pipelined and have little latency (about one clock cycle).

The performance can be enhanced further in several ways:

- An implementation can track multiple ASF critical-section instances in parallel to come close to the ILP exposed by an unprotected version of the code.
- An implementation can prevent instructions in a critical section from committing before COMMIT, keeping all modifications in the internal store buffer. While such a design would limit the number of instructions in a critical section to the size of the reorder buffer, it works without an LLB (thereby removing the LLB as a potential bottleneck) because all outstanding writes remain in the store buffer until COMMIT.

4. APPLYING ASF

4.1 An ASF-based linked list

In this section, we introduce an implementation of a singly linked list based on ASF. It serves as an example of how to integrate ASF into a programming language and will also be the target of our evaluation in Section 5.

Before we present the ASF-based implementation, we show a lock-based version for comparison (Figure 2). We will use this version in our performance comparison in Section 5, along with the CAS-based lock-free list implementation proposed by Harris [5].

Figure 3 shows part of the ASF-based implementation. It is essentially similar to Harris' CAS-based one, but can remove elements directly from the list instead of marking them first. The code sample highlights this difference. It also illustrates a programming-language interface to ASF, which we implemented using C macros.

```
void acquire(lock_t* lock) {
    do {
        while (lock->locked);
    } while ( CAS(&lock->locked,0,1) );
}
void release(lock_t* lock) {
    lock->locked = 0;
}

int set_remove(set_t *set, int val) {
    ...
    acquire(&set->lock);
    find(set, val, &prev, &next);

    if (next->val != val) {
        release(&set->lock);
        return 0;
    }
    prev->next = next->next;
    release(&set->lock);
    ...
    return 1;
}
```

Figure 2: Removal from a singly linked list protected by a single lock

```
int set_remove(set_t *set, int val) {
    ...
retry:
    /* Traverse the list to the element,
       without any locks / ASF protection. */
    find(set, val, &prev, &next);
    ...
    contained = 0;
    prev_next = asf_lock_load(&prev->next);
    next_next = asf_lock_load(&next->next);
    next_val  = asf_lock_load(&next->val);

    if (!asf_acquire(3)) {       /* atomic start */
         /* Could not acquire locations -> Retry */
         goto retry;
    }
    /* check for chaining errors */
    if (prev_next != next) {
        commit();
        goto retry;
    }
    if (next_val == val) {
        contained  = 1;
        prev->next = next_next;
        next->next = (node_t*)NULL;
    }
    asf_commit();                /* atomic end */
    ...
    return contained;
}
```

Figure 3: Lock-free removal from a singly linked list using ASF

Like Harris' CAS-based implementation, our ASF-based one does not support concurrent memory reclamation and requires that elements removed from the list do not change in type.[2]

4.2 STM acceleration

We now describe how we applied ASF to accelerate an STM system. We started from the idea that ASF could relieve the STM's metadata-bookkeeping tasks by monitoring memory locations for conflicting modifications in hardware instead of in software.

We used TinySTM as the baseline for our experiments [4]. TinySTM is a state-of-the-art lock-based STM system. It has comparatively low overhead and scales well to the number of CPUs found in today's shared-memory systems. TinySTM avoids some of the overheads of lock-free STMs, such as additional indirections.

TinySTM works by tracking the time interval in which the currently running transaction is valid. As long as no value newer than the end of the current interval is read, the overhead of revalidating the set of previously read memory locations can be skipped and deferred to one validation at commit time.

Read-set validation ensures that all previously read values are consistent for a given time interval. To this end, TinySTM keeps track of the read set (and the version of the previously read values) in an internal data structure that it updates on every read operation, implemented by TinySTM's `stm_load` routine. We applied ASF by monitoring and validating a part of the read set in hardware, saving some of the bookkeeping and validation overhead.

The `stm_load` routine is already quite small and well optimized, which is one of the reasons why TinySTM performs so well. The original version (Figure 4) executes the following steps:

1. Locate the metadata for the memory location.
2. Read the version number of the memory location.
3. Check whether the write lock is set. If so, abort the transaction.
4. Read the desired memory location.
5. Check again whether the write lock is set or whether the object's version number has changed. If so, abort the transaction.
6. Check whether the version of the memory location is still within or lower than the transaction's validity interval. If not, try to extend this interval—this requires revalidating the read set. If this fails, abort.
7. Append the memory location to the list of addresses to verify at the end of the transaction.

With ASF, it is possible to read the memory location with a LOCK MOV instruction to let the hardware monitor for concurrent alterations. This allows us to omit the second lock check and the

[2]This restriction can be lifted by using a doubly linked list and checking the back references during list traversal. With ASF, the changes to the presented singly linked list algorithm are small. In our experiments, this safer list was still slightly faster than Harris' singly linked list.

```
stm_word_t stm_load(stm_tx_t *tx,
                    volatile stm_word_t *addr)
{
    ...
    lock = GET_LOCK(addr);
    /* Read lock, value, lock */
    l = ATOMIC_LOAD_MB(lock);

restart:
    if (LOCK_GET_OWNED(l)) {
        /* Locked: Check if by us, if not abort. */
        ...
    }
    value = ATOMIC_LOAD_MB(addr);
    l2    = ATOMIC_LOAD_MB(lock);
    if (l != l2) { l = l2; goto restart;}
    /* Check timestamp */
    version = LOCK_GET_TIMESTAMP(l);
    /* Valid version? */
    if (version > tx->end) {
        /* No: Revalidate read-set
               if that fails abort. */
        ...
        /* Recheck lock, perhaps
           locked during validation. */
        l = ATOMIC_LOAD_MB(lock);
        if (l != l2) goto restart;
    }
    /* Good version: Add to read set */
    if (tx->r_set.nb_entries == tx->r_set.size) {
        /* Enlarge read set */
    }
    r = &tx->r_set.entries[tx->r_set.nb_entries++];
    r->version = version;
    r->lock    = lock;
    return value;
}
```

Figure 4: Simplified version of the original `stm_load` operation (TinySTM, write-through version)

version comparisons (Steps 5, 6), as well as recording the location in the read log (final step).

The resulting new `stm_load_asf` routine (Fig. 5) uses ASF to monitor memory until ASF's capacity limit is reached (tracked with a thread-local counter variable), after which it transparently falls back to the original `stm_load` implementation when further extending the read set. Additionally, `stm_load_asf` prevents allocation of another protected memory location if the most recently allocated location and the current one share one cache line. This microoptimization is possible because ASF works on the granularity of the size of one cache line.

The `stm_load_asf` routine first protects the read value using ASF, then checks the lock. The subsequent VALIDATE ensures that the value has not been updated before reading the lock, thus allowing us to read the lock only once.

The ASF-based optimization works because it is easy and fast to check the validity of the ASF-protected memory locations along with those recorded in the read log: A simple VALIDATE instruction suffices. A final VALIDATE and COMMIT in the transaction-commit code completes the modification.

```
stm_word_t stm_load_asf(stm_tx_t *tx,
                        volatile stm_word_t *addr)
{
    stm_word_t res;
    ulong cache_addr = (ulong)addr & ASF_LINE_MASK;
    if (tx->asf_last & ASF_HINT_SOFTWARE)
        return stm_load(tx, addr);

    /* Aliasing on the last ASF line */
    if (tx->asf_last == cache_addr) {
      res = ATOMIC_LOAD_MB(addr);
      goto load_validate;
    }
    /* ASF capacity exceeded */
    if (tx->asf_entries >= ASF_ENTRIES) {
      tx->asf_last = ASF_HINT_SOFTWARE;
      return stm_load(tx, addr);
    }
    /* Check that the location is unlocked */
    res          = asf_lock_load(addr);
    tx->asf_last = cache_addr;
    tx->asf_entries++;
    stm_word_t l = ATOMIC_LOAD_MB(GET_LOCK(addr));
    if ( LOCK_GET_OWNED(l) )
        return stm_load(tx, addr);
    /* Validate recent ASF read-set */
load_validate:
    long asf_inv;
    asf_validate(asf_inv, tx->asf_entries);
    if (asf_inv) {
        stm_abort_self(tx);
        return 0;
    }
    return res;
}
```

Figure 5: Transactional load using ASF

5. EVALUATION

5.1 Evaluation setup

Given the high cost for developing a new processor core, instruction-set extensions are initially evaluated with processor simulators. We have chosen PTLsim [18] and have implemented ASF as described in Section 3. Additional modifications have been made to the simulator, partially bug fixing and architectural enhancements to bring its architecture more in line with our native hardware. We build on the work Yourst introduced in [18] making PTLsim behave similar to an AMD K8 core.

To show the significance of our simulation results, we will compare results from multi-threaded benchmarks on native hardware to our tuned simulator, thereby laying the foundation for the fidelity of the evaluation of the ASF extensions.

For the native measurements we have used a dual-socket system, equipped with two AMD Opteron™ processors (family 10h, Barcelona) running at 2.2 GHz. Each processor consists of four CPUs, each with private caches (L1D & L1I: 64 KByte, 2-way

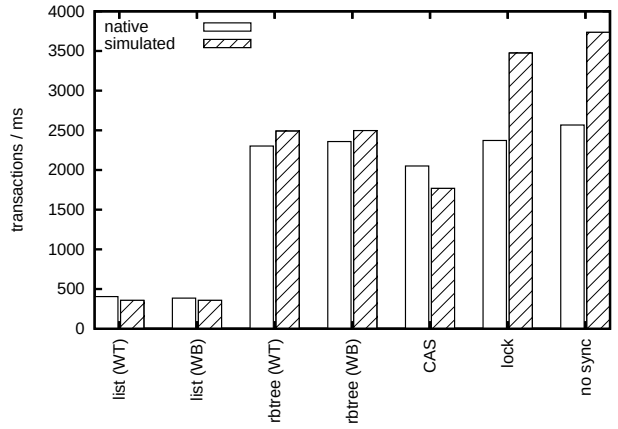

Figure 6: Single-thread performance for native and simulated execution

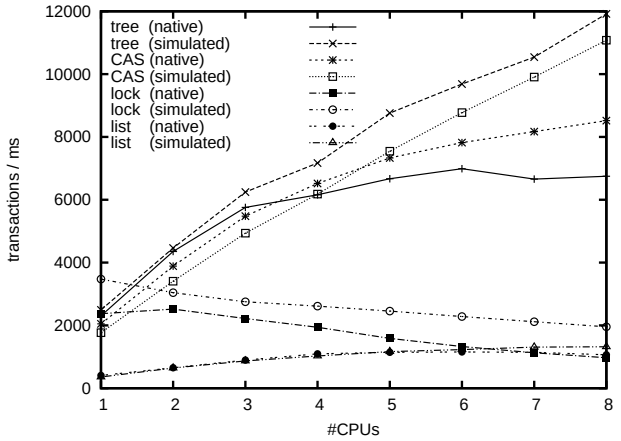

Figure 7: Multi-thread performance for native and simulated execution

set-associative; unified L2: 512 KByte, 16-way set-associative) in an exclusive hierarchy, and a shared (between the four CPUs) L3 cache (2 MB, 32-way) with a mostly exclusive (sharing-aware) configuration. Both sockets are connected with HyperTransport™ links, making main memory at each socket available in a ccNUMA fashion.

Benchmarking is done using the well-known "intset" workload, a data structure that provides methods for insertion, removal, and query of a set of integers. We use the test harness included in Felber's TinySTM distribution[3] [4] and add the different implementations of the set interface. TinySTM itself is also extended to use ASF primitives, as sketched in Section 4.2.

Because simulation is rather slow (depending on the number of simulated CPUs about 100,000 times slower), we have limited the number of operations on the intset to 5000 per thread. Variance is reduced by pinning worker threads statically to CPUs (avoiding the OS's balancer) and maintaining a fixed seed for reproducibility. Other parameters of the benchmark from TinySTM remain at their default values (set with 256 entries, entries range from 0 to 65535, 20 % rate of updates).

In addition to the implementations mentioned previously (sorted singly and doubly linked lists using ASF) and those contained in TinySTM (sorted singly linked list and red–black tree using STM), we have added Harris' implementation of a lock-free singly linked list, as described in [5]. Another singly linked list simply protected with a single spin-lock and a single-threaded implementation without any locks mark the limits of (poor) scalability and excellent single-thread performance.

5.2 Simulator precision

Figure 6 compares throughput for different implementations of the intset interface on native hardware as well as inside the simulator. Simulator precision varies, dependent on the actual implementation, but is within about 20 % of native performance, except for the implementation using just a single big lock and the one without any synchronization. The larger gap for the latter originates from the tight loop that traverses the list. We have tuned the simulator to schedule instructions in the simulated CPU as efficiently as possible, a behavior that the native hardware has as well if we increase the total number of intset operations per test. The difference is likely an artifact of the testing environment, which is not as controlled as within PTLsim, or an effect of the incomplete knowledge of the AMD Opteron processor's Barcelona core in PTLsim. With the present tight traversal loop (about 3 cycles per iteration), every additional stall has a large effect on total performance.

Multi-thread results are shown in Figure 7, and although PTLsim captures the general trend, simulation results scale better, especially at working-thread numbers larger than four. This deviation is caused by our simulation's interconnect model, which does not differentiate between local (between CPUs in the same socket) and cross-socket communication. On native hardware, these links differ in both latency and available bandwidth. Therefore, our simulation can be viewed as modeling a single-socket eight-core processor instead of two four-core processors.

Additionally, PTLsim and native hardware differ in the way they treat atomic read-modify-write (RMW) instructions: PTLsim simply grabs a simulator-internal lock for the affected memory location (without any delays), whereas native hardware drains execution until the instruction is not speculative, waits for buffered stores to complete, and then executes the instruction. This leads to highly different behavior for atomic RMW instructions on contended memory locations, such as spin locks (as used in the "big-lock" implementation).

5.3 Lock-free data structures

Well designed lock-free data structures usually offer performance superior to those implemented with coarse-grained locks and STM, because of increased parallelism and reduced overhead. Figure 8 shows the results for the various implementations of the intset interface: singly linked lock-free list using ASF (labeled ASF), Harris' CAS-based lock-free implementation (CAS), and the version that uses a single lock (lock).

It can clearly be seen that the ASF implementation outperforms both the CAS-based and the lock-based implementation.

The performance advantage over the CAS-based implementation comes from three facts: First, the ASF lists do not keep deleted el-

[3] Available from http://www.tinystm.org

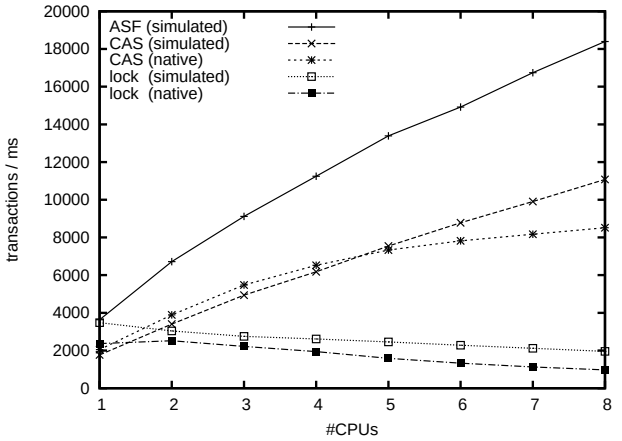

Figure 8: Lock-free data structures and single lock implementation

ements in the list for later clean-up. This keeps the list short and saves the overhead of marking and cleaning up later. Second, ASF does not guarantee progress and thus does not need any synchronization when contention on the same memory location occurs. Finally, ASF does not serialize the other memory accesses in the CPU, leaving more potential for parallel and out-of-order execution.

5.4 Acceleration of STM

In Figures 9, 10, and 11 we compare a standard TinySTM against the ASF-accelerated version from Section 4.2. The large tree and the linked list both benefit from the reduced revalidation overhead. The number of revalidations grows with the level of concurrency (frequent validity-interval extensions) and thus the accelerated version benefits more at higher CPU counts.

Surprisingly, the small tree (in Figure 9) does not profit from the acceleration although its entire read set should fit into ASF. We believe we observe this behavior because of the small read set, which makes the validation in the standard STM still reasonably fast. This reduces the advantage of ASF's fast VALIDATE and brings out some unknown overhead. We will investigate further into where this overhead of the accelerated STM comes from and how it can be avoided.

Figures 9, 10, and 11 also contain the performance of the native execution using the unmodified STM for reference. As we pointed out previously, the simulator does not yet model the limitations of the interconnect between the two sockets in the system, which obviously limits performance for the red–black tree on native hardware for CPU counts greater than four when cross-socket communication is necessary.

6. CONCLUSION AND FUTURE DIRECTIONS

In the lock-free programming and STM scenarios we have analyzed, ASF has provided substantial performance improvements—up to 15 %. Additionally, ASF significantly simplifies lock-free programming.

In the remainder of this section, we outline a few directions for future research.

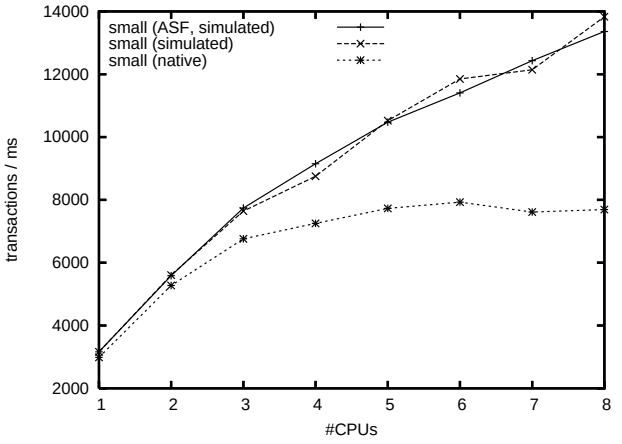

Figure 9: Comparison of ASF-accelerated and standard TinySTM with red–black tree containing 128 initial elements

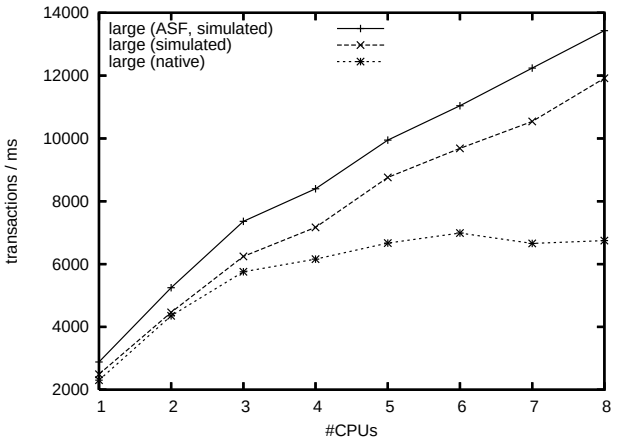

Figure 10: Comparison of ASF-accelerated and standard TinySTM with red–black tree containing 256 initial elements

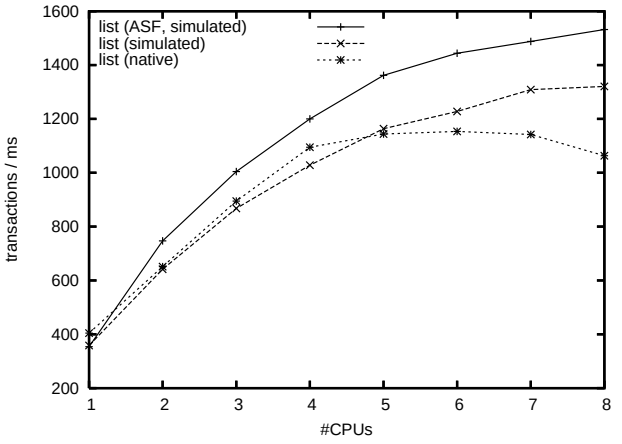

Figure 11: Comparison of ASF-accelerated and standard TinySTM with linked list

Accelerating lock-free STMs. The work presented in this paper attempted to accelerate TinySTM, one of the best performing STM systems available. TinySTM belongs to the class of STM systems that are based on locks. Lock-based STMs have largely replaced lock-free STM systems because they have less single-thread overhead while still scaling well to the number of CPUs found in today's shared-memory multiprocessor systems. However, lock-based STMs have two drawbacks to lock-free ones: susceptibility to lock-holder preemption, causing locks to be held longer than necessary; and lower scalability as the number of CPUs grows beyond what is found in today's systems. Therefore, one direction of future research is to use ASF-like hardware acceleration to reduce the overhead of lock-free STMs to the level of lock-based ones.

Compiler integration. In Section 4.1, we have sketched a C-preprocessor-macro-based interface to ASF. We acknowledge that a more robust and usable interface is needed to make use of ASF in programming languages. This requirement is reinforced by ASF being targeted not only to STM runtimes but also to lock-free application code.

The latter use case additionally raises the question of backward compatibility. The compiler interface should support application code that needs to work regardless of whether ASF is present or not.

Hardware changes. ASF can be used to protect both reads and writes against conflicting memory accesses, but the latter is bound to ASF's roll-back facility: It is currently not possible to discard memory modifications without ACQUIRE, which resets stack and instruction pointer to the values they had at the beginning of the atomic phase in case of contention. Accelerating STM-write operations would benefit from a more flexible mechanism.

Simulator precision. We outlined in Section 5.2 that our simulator lacks precision for tight loops and when modeling cross-socket communication. We plan to tackle especially the latter shortcoming to enable better prediction of highly parallel workloads.

Acknowledgments

ASF has been developed by an AMD team lead by Dave Christie and Mitch Alsup. We would like to thank Dave Christie (AMD), Andi Kleen (Novell), and Torvald Riegel (Technische Universität Dresden) for helpful discussions. We thank Matt Yourst for help with setting up PTLsim.

7. REFERENCES

[1] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[2] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel 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.

[3] 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.

[4] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2008.

[5] Tim Harris. A pragmatic implementation of non-blocking linked-lists. *Lecture Notes in Computer Science*, 2180:300, 2001.

[6] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402. October 2003.

[7] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 197–206, New York, NY, USA, 1990. ACM.

[8] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, pages 92–101, July 2003.

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

[10] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of Symposium on Principles and Practice of Parallel Programming*, March 2006.

[11] Yosef Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased transactional memory. In *Second ACM SIGPLAN Workshop on Transactional Computing*, 2007.

[12] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[13] Carl J. Mauer, Mark D. Hill, and David A. Wood. Full-system timing-first simulation. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems (SIGMETRICS)*, pages 108–116, New York, NY, USA, 2002. ACM.

[14] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. *SIGARCH Comput. Archit. News*, 35(2):69–80, 2007.

[15] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural support for software transactional memory. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[16] John D. Valois. Lock-free linked lists using compare-and-swap. In *Symposium on Principles of Distributed Computing*, pages 214–222, 1995.

[17] Luke Yen, Jayaram Bobba, Michael M. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LLogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture(HPCA)*. February 2007.

[18] M.T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 23–34, April 2007.