Implementing AMD's Advanced Synchronization Facility in an out-of-order x86 core

Stephan Diestelhorst Martin Pohlack Michael Hohmuth Dave Christie Jae-Woong Chung Luke Yen

Advanced Micro Devices, Inc.

ASF_Feedback@amd.com

Abstract

AMD's Advanced Synchronization Facility (ASF) is an experimental architecture extension proposal aiming at making lock-free programming easier and at accelerating transactional memory systems.

We report our experiences implementing ASF in an out-of-order (OoO) CPU core simulator and our lessons learned for a future a real (silicon) implementation of ASF. Specifically, we describe how we integrated ASF into the pipeline of the simulated OoO core and how we handle the intricacies caused by the the inherently asynchronous multiprocessor memory-coherence protocol that can cause transaction aborts in any CPU state.

We present our ASF implementation's answers for four of ASF's key requirements: providing an architectural interface, rather than exposing microarchitecture directly; providing sequential memory access semantics; early abort semantics; and, capacity guarantees. We find relatively lightweight solutions for all of these requirements, but the OoO nature of the core necessitates many small changes to several CPU data structures to provide complete tracking of protected memory locations and timely reactions to conflicting memory access.

1. Introduction

Transactional programming is a promising paradigm for parallel programming that is based on a simplified programming model, but requires runtime support in the form of a software or hardware transactional memory (TM) system. Because software-only solutions come with a high overhead, hardware support for TM has been the subject of intense research in the past few years [5, 11].

Most hardware extensions (including the two implemented in real silicon [7, 10]) have been evaluated for (more or less) simple in-order processors. However, many modern commercial microprocessors employ out-of-order (OoO) cores. It is unclear whether results presented for in-order cores translate to OoO cores, for the following two reasons:

- First, OoO cores are significantly more complex than in-order cores, which also complicates the implementation of any hardware support for TM. For example, an OoO core could have multiple transactions in flight, competing for hardware resources and interfering with each other, or memory references could be reordered across the boundary of a transaction. Therefore, to assess the feasibility of TM support for an OoO core, it is important to evaluate it for an OoO model.
- Second, OoO cores exhibit a different performance profile than in-order cores. To accurately predict TM performance for modern microprocessors, it is crucial to use a simulator that closely tracks native performance for conventional workloads.

In this work we set out to implement a recent hardware-extension proposal, AMD's Advanced Synchronization Facility (ASF), in PTLsim, an instruction-driven near-cycle-accurate full-system OoO-core AMD64 simulator [24]. ASF is an experimental AMD64 architecture extension proposal developed by AMD [1]. It aims at making lock-free programming significantly easier and faster as well as accelerating TM systems [6].

We report our experiences integrating ASF with an existing OoO core simulator and our lessons learned for a future real (silicon) implementation of ASF. Specifically, we describe how we integrated ASF into the pipeline of the simulated OoO core and how we handle the intricacies caused by the the inherently asynchronous multiprocessor memory-coherence protocol that can cause transaction aborts in any CPU state.

We present our ASF implementation's answers for four of ASF's key requirements:

- Providing an architectural interface, rather than exposing microarchitecture directly
- Providing sequential memory access semantics
- Early abort semantics
- Capacity guarantees

We find that there are relatively lightweight solutions for all of these requirements, but that the OoO nature of the core necessitates many small changes to several CPU data structures to provide complete tracking of protected memory locations and timely reactions to conflicting memory access.

In this paper, we do not digress into the rationale that led to ASF's design, or any workloads we have used to validate our simulator or to evaluate ASF's performance. We refer interested readers to [6].

In the rest of this paper, Section 2 revisits the fundamentals of our work: it introduces ASF's programming interface and basic implementation options, provides background on OoO cores, and contrasts OoO speculation with ASF speculation. In Section 3 we present the implementation of ASF in the OoO core simulated by PTLsim. We discuss related work in Section 4. Section 5 summarizes our lessons learned and concludes the paper.

2. Fundamentals

2.1 ASF specification

ASF is purely experimental and has not been announced for any future product. However, it has been developed in the framework of constraints that apply to the development of a high-volume microprocessor.

```
; DCAS Operation:
; IF ((mem1 = RAX) && (mem2 = RBX)) {
;   mem1 = RDI;      mem2 = RSI;     RCX = 0;
; } ELSE {
;   RAX = mem1;      RBX = mem2;     RCX = 1;
; }  // (R8, R9, R10 modified)
DCAS:
  MOV       R8, RAX
  MOV       R9, RBX
retry:
  SPECULATE             ; Speculative region begins
  JNZ       retry       ; Page fault, interrupt, or contention
  MOV       RCX, 1      ; Default result, overwritten on success
  LOCK MOV R10, [mem1]  ; Specification begins
  LOCK MOV RBX, [mem2]
  CMP       R8, R10     ; DCAS semantics
  JNZ       out
  CMP       R9, RBX
  JNZ       out
  LOCK MOV [mem1], RDI  ; Update protected memory
  LOCK MOV [mem2], RSI
  XOR       RCX, RCX    ; Success indication
out:
  COMMIT
  MOV       RAX, R10
```

Figure 1: A DCAS primitive using ASF.

ASF provides seven new instructions for entering and leaving speculative code regions (or *speculative regions*, for short), and for accessing protected memory locations (i.e., memory locations that can be read and written speculatively and which abort the speculative region if accessed by another thread): `SPECULATE`, `COMMIT`, `ABORT`, `LOCK MOV`, `WATCHR`, `WATCHW`, and `RELEASE` (the last three are not further discussed in this paper). Figure 1 shows an example of a double compare-and-swap (DCAS) primitive implemented using ASF.

Speculative-region structure. Speculative regions have the following structure. The `SPECULATE` instruction signifies the start of such a region. It also defines the point to which control is passed if the speculative region aborts: in this case, execution continues at the instruction following the `SPECULATE` instruction (with an error code in the rAX register and the zero flag cleared, allowing subsequent code to branch to an abort handler).

The code in the speculative region indicates protected memory locations using the `LOCK MOV` instruction, which is also used to load and store protected data.

`COMMIT` and `ABORT` signify the end of a speculative region. `COMMIT` makes all speculative modifications instantly visible to all other CPUs, whereas `ABORT` discards these modifications.

ASF supports a limited form of nesting that allows simple composition of multiple speculative regions into an overarching speculative region. Nesting is implemented by flattening the hierarchy of speculative regions: memory locations protected by a nested speculative region remain protected until the outermost speculative region ends.

Aborts. Besides the `ABORT` instruction, there are several conditions that can lead to the abort of a speculative region: contention for protected memory; system calls, exceptions, and interrupts; the use of certain disallowed instructions; and, implementation-specific transient conditions. Unlike in Sun's hardware TM (HTM) design [10], TLB misses do not cause an abort.

In case of an abort, all modifications to protected memory locations are undone, and execution flow is rolled back to the beginning of the speculative region in a setjmp/longjmp-like fashion by resetting the instruction and stack pointers to the values they had directly after the `SPECULATE` instruction. No other register is rolled back; software is responsible for saving and restoring any context that is needed in the abort handler. Additionally, the reason for the abort is passed in the rAX register.

Page faults (as well as other exceptions and interrupts) abort the speculative region before they are reported to the OS. This allows the OS to resolve any faults before the speculative region is retried.

Strong isolation guarantees. ASF provides strong isolation: it protects speculative regions against conflicting memory accesses to protected memory locations from other speculative regions and regular code concurrently running on other CPUs.

In addition, all aborts caused by contention appear to be instantaneous: ASF never allows any spurious side effects caused by ASF misspeculation in a speculative region to become visible. These side effects include nonspeculative memory modifications and page faults caused by dependencies on stale data.

Eventual forward progress. ASF architecturally guarantees eventual forward progress in the absence of contention and exceptions when a speculative region protects not more than four memory lines.[1] This guarantee enables easy lock-free programming without requiring software to provide a second code path that does not use ASF. Because the guarantee only holds in the absence of contention, software still has to control contention to avoid livelock, but that can be accomplished easily (for example, by employing an exponential-backoff scheme).

2.2 Basic ASF implementation variants

We designed ASF such that a CPU design can implement ASF in various ways. The minimal capacity requirements for an ASF implementation (four transactional cache lines) are deliberately low so existing CPU designs can support simple ASF applications, such as lock-free algorithms or small transactions, with very low additional cost. On the other side of the implementation spectrum, an ASF implementation can support even large transactions efficiently.

In this section, we present two basic implementation variants. We implemented these two variants in the simulator described in later sections of this paper.

Cache-based implementation. A first variant is to keep the transactional data in each CPU core's L1 cache and use the regular cache-coherence protocol for monitoring the transactional data set.

Each cache line needs two additional bits, a speculative-read and a speculative-write bit, which are used to mark protected cache lines that have been read or written by a speculative region, respectively. These bits are cleared when the speculative region ends. In case the speculative region is aborted, the cache also invalidates all cache lines that have the speculative-write bit set.

This implementation has the advantage that potentially the complete L1 cache capacity is at disposal for transactional data. However, the capacity is limited by the cache's associativity. Additionally, an implementation that wants to provide the (associativity-independent) minimum capacity guarantee of four memory lines using the L1 needs to ensure that each cache index can hold at least four cache transactional lines that cannot be evicted by nontransactional data refills.

LLB-based implementation. An alternative ASF implementation variant is to introduce a new CPU data structure called the *locked line buffer* (LLB). The LLB holds the addresses of protected memory locations as well as backup copies of speculatively modified

[1] *Eventual* means there may be transient conditions that lead to spurious aborts, but eventually the speculative region will succeed when retried continuously. The expectation is that spurious aborts rarely occur and speculative regions succeed the first time in common cases.

memory lines. It snoops remote memory requests, and if an incompatible probe request is received, it aborts the speculative region and writes back the backup copies before the probe is answered.

The advantage of an LLB-based implementation is that the cache hierarchy does not have to be modified. Speculatively modified cache lines can even be evicted to another cache level or to main memory. (We assume the LLB can snoop probes independently from the caches and is not affected by cache-line evictions.)

Because the LLB is a fully associative structure, it is not bound by the L1 cache's associativity and can guarantee a larger number of protected memory locations. However, since fully associative structures are more costly, the total capacity typically would be much smaller than the L1 size.

2.3 Out-of-order core fundamentals

This section will briefly introduce some of the key concepts employed in current OoO microprocessors. A large number of publications exist on the matter; Hennessy and Patterson provide an extensive overview [14].

Many high-performance microprocessor cores do not process instructions in order (that is, not in the order demanded by the program executed), but rather reorder instructions to interleave long latency operations (such as complex computations and cache misses) with independent instructions, and to exploit instruction-level parallelism (ILP). With such OoO execution, book-keeping mechanisms need to be employed to maintain the sequential program semantics.

A central data structure called the reorder buffer (ROB) keeps track of in-flight instructions, their states, and required input operands. Dependencies among instructions are formed through producer–consumer relationships between instructions—operands required by one instruction are produced as results by an earlier one and are usually conveyed through registers. Because architecturally visible registers may be used by multiple independent in-flight instruction pairs, register renaming is used to separate these aliases.

Once an instruction has all input dependencies fulfilled, it is considered for execution and is eventually issued on one of the functional units of the core. Executing these instructions is not dependent on program order at this point anymore, but can proceed out of order: later instructions with fulfilled dependencies may execute before earlier instructions with unmet dependencies. Once the instructions complete execution, they forward the results to dependent in-flight instructions. The final pipeline step retires the instruction from the core. However, it processes completed in-flight instructions strictly in program order and thus maintains the sequential semantics of the code.

One source for long-latency operations is load instructions that miss in the cache. OoO execution helps here because the core can issue multiple cache-missing loads at once, thereby effectively overlapping the latencies for them. Several data structures keep track of in-flight memory operations: The load and store queue(s) of the core handle single load and store instructions, while a miss buffer keeps track of the pending cache-lines, which may be referenced by multiple in-flight memory operations.

Executing memory instructions out of order interferes with the global order of memory accesses in multiprocessor systems, impacting memory consistency guarantees [2, 21]. To free the application programmer from reasoning over the actual complex interactions, the core maintains stronger (simpler to reason about) guarantees by locally checking for consistency violations and selectively replaying memory instructions [13].

The core fetches instructions in the native AMD64 instruction-set architecture (ISA) from memory (the instruction cache) and decodes the instructions and operand information. A number of the instructions are not executed directly in the core, but are split up into multiple smaller instructions, so called microoperations (μops), instead. These flow through the pipeline independently and also retire in sequence.

2.4 Levels of speculation

Conditional branches make the code sequence dependent on data, which usually is produced only near the branch instruction and may be subject to long-latency operations, such as complex arithmetic and cache misses. To maintain a sufficiently large look-ahead instruction window, modern microprocessors employ branch prediction to forecast the instruction stream. If a conditional branch is predicted the wrong way (predicted taken vs. resolved not taken, and vice versa), instructions on the wrong branch have been executed. The instructions have to be removed from the core and their effects have to be undone, or annulled, and architectural state needs to be restored to a previous, known-good configuration.

Other predictions, such as predicting intrathread data dependencies (or their absence) for pairs of stores and loads with unresolved addresses (store-load aliases), or optimistic assumptions for scheduling conflicts and late resource shortages, may also cause re-execution of instructions.

In this paper, we will refer to this collection of speculation as employed by current OoO microprocessors as *OoO speculation* (OoO-spec). In contrast, we will refer to speculation caused by entering an ASF speculative region as *ASF speculation* (ASF-spec).

3. Pipeline and core integration

In this section we present how we integrated ASF with an OoO core simulated in an instruction-driven near-cycle-accurate AMD64 simulator, PTLsim [24]. We discuss the integration of both basic ASF implementation variants (introduced in Section 2.2) throughout this section, which we organize by high-level goals:

- In Section 3.1, we discuss the danger of using implementation artifacts as architecture, which motivates our choice of not reusing the existing OoO speculation mechanisms of the core for ASF speculation.
- Section 3.2 explains how we provide sequential ASF semantics on an OoO core.
- Section 3.3 describes our implementation of ASF's early-abort semantics.
- Section 3.4 discusses ASF's minimum capacity guarantee.

3.1 Avoiding implementation as architecture

Given that many cores already have mechanisms for keeping program state private, such as the store queue and an OoO-speculation mechanism, it is tempting to reuse these mechanisms for ASF speculation.

To illustrate, we consider the Rock processor [10], which relies on existing microarchitectural features to implement transactions: Rock uses the hardware's register checkpointing mechanism for keeping and restoring the register-file contents before starting speculation, and it keeps speculative memory updates in the core's store queue. Consequently, Rock needs to abort transactions when the capacity of either hardware resource is exhausted. Furthermore, Rock employs only a single level of speculation; the resolution of branch mispredictions, TLB misses, and other exceptional conditions abort an ongoing transaction. For these and other reasons, Rock does not give any guarantees on transaction success even in the absence of contention and interrupts.

In contrast, ASF does give an architectural forward-progress guarantee (in the absence of contention) and a minimum-capacity guarantee for speculative regions. Microarchitectural conditions such as a TLB miss or a store-queue overflow must not prevent a

```
retry1:                                  retry1:
  SPECULATE                                asf.spec1
  ; First speculative region               asf.mfence1
  JNZ      retry1                          br.nz    retry1
  LOCK MOV [mem1], RBX                     asf.ld1  [mem1], RBX
  ...                                      ...
  COMMIT                                   asf.commit1
retry2:                                  retry2:
  SPECULATE                                asf.spec2
  ; Second speculative region               asf.mfence2
  JNZ      retry2                          br.nz    retry2
  LOCK MOV [mem2], RDX                     asf.ld2  [mem2], RDX
  ...                                      ...
  COMMIT                                   asf.commit2
```

Figure 2: Two speculative regions close to each other, with original assembly (left) and decoded µops with added fences (right).

speculative region from ever succeeding. Because it would be impossible to provide these guarantees (including the weaker guarantee of eventual forward progress) based on the OoO microarchitecture, we chose to implement the ASF mechanisms separately from (and complementary to) the OoO mechanisms.

3.2 Sequential ASF semantics

ASF has sequential programming semantics, which a core must preserve whether it employs OoO-speculative execution or not. For example, a protected memory access occurring inside a speculative region must not be reordered to occur before the beginning of a speculative region. In this section, we discuss two aspects of executing ASF-speculative memory accesses on an OoO core. We begin with issues raised by executing protected memory accesses out of order in Section 3.2.1. Section 3.2.2 discusses how ASF resources reserved for ASF-speculative data can be managed when the instructions referencing this data are OoO-speculative and later annulled.

3.2.1 Speculative-region flow

The execution order of instructions in an OoO core is only determined through data dependencies. Instructions without dependencies can execute in arbitrary order.

For ASF speculative regions, we need to decide whether particular memory accesses (`LOCK MOV`s) are executed inside or outside a speculative region. Hence, these instructions have to be ordered with respect to the marker instructions that begin / end such a containing region (`SPECULATE` and `COMMIT`).

We add special ASF memory-fence µops during decode of the `SPECULATE` instruction to attain this goal, as shown in Figure 2. These fences operate mostly like normal fences in that they create an artificial dependency on any later memory instruction that is only resolved once the fence has retired from the core. Later memory instructions can therefore only issue after the fence has retired.

The fences are ASF-specific in that they only affect ASF-spec memory instructions and not regular memory references.

Memory instructions are then ordered by the following ordering rules (with $\rightarrow$ being similar to Lamport's *happened-before* relation [16] and $S(X)$ denoting the event of instruction X being in pipeline stage S):

$$\mathit{issue}(\texttt{asf.spec}) \rightarrow \mathit{retire}(\texttt{asf.spec}) \quad (1)$$

$$\mathit{retire}(\texttt{asf.spec}) \rightarrow \mathit{retire}(\texttt{asf.mfence}_1) \quad (2)$$

$$\mathit{retire}(\texttt{asf.mfence}_1) \rightarrow \mathit{issue}(\texttt{asf.memop}) \quad (3)$$

$$\mathit{issue}(\texttt{asf.memop}) \rightarrow \mathit{retire}(\texttt{asf.memop}) \quad (4)$$

$$\mathit{retire}(\texttt{asf.memop}) \rightarrow \mathit{retire}(\texttt{asf.commit}) \quad (5)$$

Ensuring that

$$\mathit{issue}(\texttt{asf.spec}) \rightarrow \mathit{issue}(\texttt{asf.memop}) \rightarrow \mathit{retire}(\texttt{asf.memop}) \rightarrow \mathit{retire}(\texttt{asf.commit})$$

Rules 1 and 4 trivially follow from the regular pipeline flow; Rule 5 is ensured by retiring all instructions in order. We implemented the other rules by introducing artificial dependencies in the instructions' ROB entries.

Speculative-region overlap. Short speculative regions that execute neck-to-neck, such as in Figure 2, can be in flight in the core simultaneously because of the reorder window.

Keeping track of the state of multiple simultaneous speculative regions is complicated. Whereas conventional state containers—registers—are renamed to track simultaneous usage of shared resources, making all of ASF's state renameable is complex, because it contains not only the information of the speculative region state, the abort instruction and stack pointer, but also the entirety of the bits used to track the read/write sets.

While there may be safe approximations to full renaming (such as merging read/write sets), we have chosen a more straightforward approach by serializing the execution of consecutive speculative regions in the core. The heavy pipeline-serialization mechanisms, such as flushes and stalls, have a large performance impact because of the time needed to drain and fill the pipeline, decreasing performance especially for frequent, small speculative regions that would execute in few cycles.

To avoid this performance decrease, we chose to implement the serialization through the existing dependency rules, ASF memory barriers, and by not changing the state of the speculative region until `SPECULATE` and `COMMIT` hit the pipeline's retire stage. The serialization of two consecutive speculative regions then is ensured through the following dependency chain:

$$\mathit{issue}(\texttt{asf.memop}_1) \rightarrow \mathit{retire}(\texttt{asf.memop}_1) \rightarrow \mathit{retire}(\texttt{asf.commit}_1) \rightarrow \mathit{retire}(\texttt{asf.spec}_2) \rightarrow \mathit{retire}(\texttt{asf.mfence}_2) \rightarrow \mathit{issue}(\texttt{asf.memop}_2)$$

3.2.2 Misspeculation

Section 2.3 introduced multiple instances for speculation in the OoO core and how they could fail. ASF-speculative load and store instructions are also subject to these mechanisms and this has caused several challenges for our implementation, because of the complex interactions imposed by release and redistribution of resources due to misspeculation.

Precise ASF working-set tracking. Because of OoO speculation, the core may overestimate ASF's working set: misspeculated memory instructions can add spurious ASF-spec entries to the LLB or cache before the misspeculation is detected and the corresponding memory instructions are annulled.

The overestimation does not impact correctness of the execution conceptually (all lines that need protection are protected), but has performance implications, since the additional lines artificially increase contention and also put additional pressure on the limited capacity.

It is thus desirable to detect and remove spurious entries in ASF's working sets. However, recomputing the actual ASF-spec state of a cache line when annulling an ASF-spec memory access is challenging. It depends not only on in-flight memory instructions, but also has to take into account retired ASF-spec memory instructions of the current speculative region that have referenced the cache line.

Our LLB-based ASF design supports reference counting for that particular purpose and thus can track read/write sets precisely. Adding reference-counting mechanisms to the existing L1 cache would be expensive; thus, the L1-based ASF implementation currently may overestimate the read set.

Orphan cache entries. Although not precisely tracking ASF's working set in an L1-based ASF implementation is safe in principle, under specific timing constraints it can lead to orphan ASF-spec entries in the cache even though the originating speculative region has already successfully committed or aborted.

To illustrate, consider the following sequence of events: an ASF-spec load misses in the cache and sets up an ASF-spec miss-buffer entry to track the cache miss. The load eventually is annulled because it is on a wrongly predicted branch. The cache-miss handling cannot be aborted at this time. Eventually, the speculative region commits by successfully retiring the `COMMIT` instruction (the original dependency on the cache-missing load is not present anymore, since that load has been annulled). The cache line is eventually filled into the cache and gets its spec-read bit enabled because the corresponding miss-buffer entry was tagged as ASF-spec, leading to an orphan spec-read cache line.

Note that simply resetting the cache line's spec-read bit on annulment of referencing ASF-spec loads would be incorrect, because multiple in-flight loads (ASF-spec and non-ASF-spec) may still reference the miss-buffer entry. Similarly, the miss-buffer entry's ASF-spec state cannot be simply reset because it may still be referenced by other in-flight ASF-spec loads.

A simplified version of the recomputation introduced previously solves this issue: we reuse the existing reference from a miss-buffer entry to its associated in-flight loads and count the ASF-spec-load references. We observe that no retired load can contribute to the ASF-spec state of the miss-buffer entry because loads can only retire once their cache misses have been resolved. Therefore, the number of ASF-spec loads referencing the miss-buffer entry can always be computed online by counting all nonretired (in-flight) loads with such a reference, allowing miss-buffer entries to precisely track their ASF-spec state and eliminating the need for dedicated reference counting in the L1 cache. In result, no modification to the L1 cache is necessary, and we readily implemented this mechanism to prevent orphan spec-read cache entries in our ASF prototype.

Flash clearing all ASF-spec bits (of miss-buffer and cache entries) at the end of a speculative region (retirement of the `COMMIT` instruction) would also work around the orphan-cache-entries issue. However, our recomputation approach tracks ASF's working set more closely and thus reduces the likelihood of contention.

3.3 Abort semantics

ASF has eager conflict detection and provides early-abort semantics: it defines that no side effects (e. g., memory modifications or page faults) ever become visible caused by ASF misspeculation (i. e., further execution of a speculative region after is has been aborted). The rationale is that no ASF-speculative state should be able to leak unintentionally from an aborted speculative region.

This section discusses how our ASF implementation realizes early abort semantics. Section 3.3.1 explains that, to receive timely abort information, cores need to track access conflicts with protected data in more CPU data structures than just the cache or LLB because of the asynchronous nature of memory accesses in OoO processors. In Section 3.3.2 we describe how a core recovers when it has received an abort signal.

3.3.1 Conflict detection handshake

The global linearizability of ASF speculative regions and consistency of the read and write sets is ensured through eager conflict detection. Conflict detection has to start when or before the value of the load is bound [12] or the load is performed [22]. Usually, some limited form of conflict detection and additional ordering is already employed in current multiprocessor systems to provide suitable memory-consistency semantics to the application. To keep changes to this very sensitive area of microarchitecture small, it is advisable to reuse the existing mechanisms and extend coverage of the conflict observation until the speculative region commits. However, extending the monitoring period of the legacy mechanisms is difficult, because it again involves touching sensitive hardware and furthermore may not be possible due to design decisions, such as reliance on bounded delay for certain operations, or serialization of monitoring requests.

Therefore, the responsibility for monitoring ASF-spec data eventually has to transition from the legacy mechanisms (such as the miss buffer) to ASF's monitoring facility (such as the LLB or the augmented L1 cache). During the transition, it has to be ensured that the data element is never without conflict observance, necessitating atomic transitions or overlapping intervals of conflict-detection responsibility.

For our prototype, we reuse the existing miss buffers and flag cache lines as soon as they are initially probed (for cache hits) or when they are delivered to L1 (for cache misses) with the according ASF-spec bits. Our LLB-based implementation similarly allocates entries as soon as possible, too. This design saves an additional cache lookup at a later point in time (to set the respective ASF-spec bits) and ensures overlapping contention monitoring.

The timing between the hand-over from one conflict detection mechanism to another (in particular to the enhanced L1 cache) has been a source of a lot of complexity. For example, one issue we encountered was caused by store-to-load forwarding, in which a load receives the data directly from an earlier store to the same address in the same thread. These loads effectively bypass the caches, circumventing any conflict-detection mechanism implemented in the cache. This issue was solved by creating additional entries in the L1 cache to ensure proper conflict monitoring.

3.3.2 Abort implications

Speculative regions abort whenever a conflicting concurrent data access is detected (requester-wins conflict resolution policy), which may happen asynchronously to other core timing. As outlined previously, we use the existing cache-coherence mechanisms to detect these conflicting memory accesses. Whenever an ASF-speculatively modified line is read by another core, it must be ensured that the requesting core receives the backup copy with the probe answer, and not the updated data.

Therefore, the timing between probes, replies, and the rollback operation is crucial for correct operation. To reduce the delay between the arrival of the conflicting probe and the final probe answer, we introduce partial rollbacks. These rollbacks undo modifications only for the requested line, deliver the probe answer, and then signal the core for further abort handling.

Aborting the core can then proceed independently of probe handling, at the core's discretion. The core checks for detected conflicts every cycle. If one is found, the core triggers the full rollback, encodes the abort reason into the rAX register, sets the flags register accordingly, and resets the instruction and stack pointer to the values right after `SPECULATE`. Finally, a pipeline flush and reset of the instruction fetcher (similar to the resolution of a mispredicted conditional branch) completes the abort.

Although checking for abort conditions every cycle seems sufficient on the surface, we had to address two subtleties of modern cores, which we describe in the remainder of this section.

Intra-cycle parallelism. It is possible for a specifically timed store operation to the line already rolled back to retire in the same cycle in which the abort condition was detected, but before the pipeline flush, essentially proceeding in parallel to the ongoing abort. Therefore, it is important to avoid disabling write-set tracking too early. Otherwise, the store would be able to make ASF-

```
  SPECULATE
  JNZ      abort_handler
traverse:
  LOCK MOV RDX, [RSI + val]     ; Load val
  CMP      RDX, RDI             ; Element found?
  JE       found
  LOCK MOV RSI, [RSI + next]    ; Load next pointer
  TEST     RSI, RSI             ; End of list?
  JNZ      traverse
  COMMIT                        ; Element not found
  ...
found:
  COMMIT                        ; Element found
```

Figure 3: A small linked-list traversal loop searching for a particular element, illustrating potential inflation of speculative working set because of mispredicted branches: The "Load next pointer" instruction may be mispredicted and use up ASF resources needed for maintaining ASF's capacity guarantee.

speculative modifications permanent (despite the abort of the enclosing speculative region).

µop splitting. As described in Section 2.3, native-ISA (AMD64) instructions do not have to proceed atomically through the core. Instead, they may be split up into smaller µops. These µops flow through the pipeline independently and also retire in sequence, which creates another subtlety with respect to the asynchronous nature of aborts: an abort may trigger when only a subset of the µops comprising an instruction have retired and updated the architectural state.

The most critical instructions regarding this behavior are `CALL` and `RET`, because they both access the stack pointer, the instruction pointer, and memory. Their partial retirement is, however, contained by ASF, because the abort resets both registers to a consistent value (and no guarantees for stack values below the stack pointer are given).

3.4 Capacity guarantees

The ASF specification mandates that implementations support a minimal number of read/write set entries (four cache lines), regardless of address layout and other aspects (such as TLB misses, branch misprediction, etc.).

Supporting such a guarantee under the OoO execution regime is complicated by several interactions. As described previously, ASF-speculative memory instructions may flag cache lines as speculative optimistically, artificially increasing the speculative region's working set and reducing the number of available entries that an application can really use. In particular, ASF loads behind unresolved and mispredicted branches, such as mispredicted pointer traversal loops (Figure 3), can cause this behavior.

Furthermore, loads may be issued out of order and may also fill missing cache lines in arbitrary order, depending on their residence in the underlying memory hierarchy (e. g., line present in L2 cache vs. line fetched from remote main memory). Determining precisely if and when the capacity limit is reached is therefore not clear-cut.

Non-ASF-spec memory instructions may also compete for space in the employed conflict detection device, in particular if an existing structure, such as the L1 cache, is reused for that purpose. It may be possible that non-ASF-spec entries displace ASF-spec entries, thwarting any possible capacity guarantee.

Finally, the organization of the speculative storage and tracking device heavily impacts the feasible minimal guarantee. Set-associative caches have a small worst-case minimal capacity—their associativity—because all requested addresses may alias into the same cache index. Other devices such as Bloom filters [4] may allow tracking of an arbitrary number of elements (with decreasing precision), but do not provide space for backup copies to support ASF-spec stores.

In summary, a naïve implementation does not even guarantee the worst-case capacity of the storage container (i. e., the associativity of the L1 cache for a cache-based implementation). Additional ordering and priority mechanisms are necessary to give such a guarantee, for example by carefully ordering accesses to capacity-critical parts of the storage device. However, strictly serializing all memory accesses would reduce overall performance and complicate core design.

For our LLB-based implementation, we have therefore crafted a staged buffer that has a (small) first stage where cache lines are held as long as they are only referenced by OoO-speculative in-flight memory instructions. Whenever one of these instructions retires, the line in the LLB transitions to the non-OoO-spec second buffer stage. The minimal guarantee is then enforced by the non-OoO-spec second buffer stage, while the first OoO-spec stage basically controls how much (OoO-)speculation can go on. This design allows us to carefully trade performance (through higher ILP) for additional buffer space (for the additional first stage buffer).

Memory instructions have to wait until a free entry in the first stage is available before they can issue. To avoid deadlocks through OoO fill-up of the speculative buffer stage, we carefully replay later memory instructions (further down in the program flow) that have already been granted an entry to make room for the earlier ones waiting for a free entry.

Our cache-based implementation currently lacks these features, because it aims at reusing most of the existing cache implementation. Hence, it does not yet meet ASF's required minimal capacity guarantee under certain circumstances.

4. Related work

Using simulation is the most common approach for evaluating hardware-extension proposals for accelerating TMs because simulation can be realized with much less effort and lower costs than a hardware prototype. In related work, simplified simulation approaches are employed often. For example, trace generation and timing simulation are separated or simple in-order core models are applied.

This trading of simulation accuracy and speed for effort is, of course, a valid approach for research proposals in which creating a new simulation infrastructure or vastly extending existing simulators is not possible. In this paper, we present a prototype proposal targeted at implementation in current high-volume OoO microprocessors and want to achieve a very high simulation accuracy to accurately predict behavior and potential pitfalls for a silicon implementation.

Herlihy and Moss [15] employ the Proteus simulator for evaluating their TM proposal. The target programs to be simulated have to be written a superset of C, and calls into the simulator are created, for example, for calls to shared memory. Memory timing is only simulated for shared memory areas. Proteus is execution-driven, and cycle counting is embedded by a preprocessor into the simulation target programs.

Ananian et al. [3] use cycle-accurate simulation of a simplified architecture to evaluate their unbounded TM (UTM) proposal. The authors utilize UVSIM and simulate OoO MIPS 10K processors. The simulator supports cycle-accurate simulation and was extended to support a simplified HTM model (named LTM). Also, a trace-driven simulator is used that evaluates memory references and transactional operations. No detailed discussion regarding the implementability of UTM or LTM in an OoO architecture is provided.

Moore et al. [20] present log-based TM (LogTM) and employ Simics [17] for the processor model (single-issue, in-order). They use a multilevel memory model and integrated LogTM with Wisconsin GEMS [18]. The authors report that HTM instructions are implemented via Simics "magic," which leads us to believe that it would be hard to draw any conclusions for an implementation of LogTM in a real microarchitecture.

Damron et al. [8] introduce hybrid TM. Wisconsin GEMS with LogTM is used for simulation. Instead of letting hardware (LogTM here) do retries for transactions, the authors modified GEMS to hand over control for retry to software after the first failure.

Yen et al.'s LogTM-SE [23] was implemented using OoO processor cores supporting two-way SMT. Their implementation was done using a modified version of Wisconsin GEMS 2.0 for the SPARC ISA. LogTM-SE needs cache-coherence protocol changes (and NACKs probe requests on conflicts), benefits from OS adaptations, and does not provide minimal guarantees (e.g., conflict detection may produce false positives due to signature implementations). In contrast, our PTLsim implementation prevents changes to cache coherence protocols and HyperTransport, does not usually require OS interactions, and provides minimal guarantees. In this paper, we focus on the integration of HTM with OoO cores, and we believe our discussions are applicable to previously proposed HTM systems.

Moir et al. [19] discuss an adaptive TM test platform (ATMTP) and demonstrate its use in [9]. They provide a simulation environment targeted at Sun's Rock processor, especially its HTM aspects. The authors explicitly state that the model is not aimed at accuracy but for gaining early experience. ATMTP is based on Wisconsin GEMS 2.0. The detailed memory model Ruby is used, but not the OoO processor model (Opal). Instead, Simics with a simple model with one instruction per cycle covers the processor. LogTM (now integrated in Ruby) provides similar semantics to Rock's speculative cache bits. Rock's limitations (e. g., overflow of limited register window) are approximated in ATMTP. Up to now, only single-chip systems are supported. In [10], early Rock prototypes are compared to ATMTP.

Sun [10] and Azul Systems [7] have developed actual multicore processors with HTM mechanisms. Their implementations are based on in-order architectures. Both HTMs have a few notable differences to ASF. We have introduced Sun's Rock processor in Section 3.1. Azul Systems' HTM does not abort transactions in case of interrupts and exceptions and does not support selective annotation.

In contrast to the often-employed combination of GEMS and Simics with its single-issue in-order processor model, we have implemented and simulated ASF using PTLsim's OoO processor model to obtain detailed predictions and experience with implementing ASF in a modern OoO processor.

5. Lessons learned and conclusion

In this paper we outlined an implementation of ASF for the OoO core simulated by PTLsim. We reviewed four requirements imposed by ASF and how we addressed them in our ASF implementation for the OoO core:

- An architectural interface, rather than exposing microarchitecture directly
- Providing sequential memory access semantics in an OoO core
- Early abort semantics despite asynchronous memory requests
- Handling capacity guarantees in light of cache contents arriving out of order

We found relatively lightweight solutions for all of these requirements, but the OoO nature of the core necessitates many small changes to several CPU data structures to provide complete tracking of protected memory locations and timely reactions to conflicting memory access.

We found that, somewhat counterintuitively, the existing microarchitecture mechanisms for OoO speculation do not ease the implementation of ASF speculation. The reason is that ASF guarantees eventual forward progress (in the absence of contention), and an OoO core can annul speculative instructions for many more reasons than allowed for ASF aborts.

We stress that implementability is of major importance for any hardware-extension proposal, and argue based on our findings that ASF has passed this test. We believe that most of our findings directly relate to a real OoO core implementation. However, we did identify functional areas of ASF—in particular, the minimum-capacity guarantee—in which the benefits may not outweigh the additional implementation complexity. Further research is required to motivate inclusion or exclusion of the feature.

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/ 2007-2013) under grant agreement N^{o} 216852.

References

[1] *Advanced Synchronization Facility - Proposed Architectural Specification*. Advanced Micro Devices, Inc., 2.1 edition, Mar. 2009.

[2] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *Computer*, 29(12):66–76, 1996. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.546611.

[3] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, D.C., USA, 2005. IEEE Computer Society. ISBN 0-7695-2275-0. doi: http://dx.doi.org/10.1109/HPCA.2005.41.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/362686.362692.

[5] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: why is it only a research toy? *Commun. ACM*, 51(11):40–46, 2008.

[6] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière. Evaluation of AMD's Advanced Synchronization Facility within a complete transactional memory stack. In *EuroSys '10: Proceedings of the 5th ACM European conference on Computer systems*. ACM, Apr. 2010.

[7] C. Click. Azul's experiences with hardware transactional memory. In *HP Labs - Bay Area Workshop on Transactional Memory*, 2009.

[8] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS*, 2006.

[9] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum. Applications of the Adaptive Transactional Memory Test Platform. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, Feb. 2008.

[10] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS*, 2009.

[11] U. Drepper. Parallel programming with transactional memory. *Communications of the ACM*, 52(2):38–43, Feb. 2009.

[12] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *ASPLOS-IV: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, New York, N.Y., USA, 1991. ACM. ISBN 0-89791-380-9. doi: http://doi.acm.org/10.1145/106972.106997.

[13] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *In Proceedings of the 1991 International Conference on Parallel Processing*, pages 355–364, 1991.

[14] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. ISBN 0-12-370490-1.

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

[16] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359545.359563.

[17] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner. SimICS/sun4m: a virtual workstation. In *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 10–10, Berkeley, Calif., USA, 1998. USENIX Association.

[18] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/1105734.1105747.

[19] M. Moir, K. Moore, and D. Nussbaum. The Adaptive Transactional Memory Test Platform: A Tool for Experimenting with Transactional Code for Rock. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, Feb. 2008.

[20] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: Log-based transactional memory. In *High-Performance Computer Architecture*, 2006.

[21] S. Owens, S. Sarkar, and P. Sewell. A Better x86 Memory Model: x86-TSO. In *TPHOLs '09: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, pages 391–407, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03358-2. doi: http://dx.doi.org/10.1007/978-3-642-03359-9_27.

[22] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *ISCA '87: Proceedings of the 14th annual international symposium on Computer architecture*, pages 234–243, New York, N.Y., USA, 1987. ACM. ISBN 0-8186-0776-9. doi: http://doi.acm.org/10.1145/30350.30377.

[23] L. Yen, J. Bobba, M. M. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LLogTM-SE: Decoupling hardware transactional memory from caches. In *HPCA*, 2007.

[24] M. T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *ISPASS*, 2007.