Compilation of Thoughts about AMD Advanced Synchronization Facility and First-Generation Hardware Transactional Memory Support

Jaewoong Chung, Luke Yen, Martin Pohlack, Stephan Diestelhorst, Michael Hohmuth, David Christie
Advanced Micro Devices, Inc.
jaewoong.chung,luke.yen,martin.pohlack,stephan.diestelhorst,michael.hohmuth,david.christie@amd.com

Abstract

AMD's Advanced Synchronization Facility (ASF) is an AMD64 extension for transactional programming and lock-free data structures. After we had released the ASF specification to the public, we contacted various transactional memory (TM) experts in academia and industry to get their opinions on ASF and suggestions for improvements. We found their feedback invaluable in understanding what the first-generation TM hardware support should look like and how to improve ASF. In this paper, we present the summary of their likes, dislikes, and concerns about ASF and explain our opinions on their suggestions. By sharing the reviews, we hope to encourage further involvement of TM experts in defining a desirable set of requirements for the first-generation TM hardware support. We believe that this will greatly help to bring out a better TM support sooner in commercial processors.

1. Introduction

Transactional memory (TM) is a promising solution to help programmers develop parallel programs [4, 7, 11, 13, 14]. With TM, programmers enclose a group of instructions within a transaction to execute them in an atomic and isolated way. The underlying TM system runs transactions in parallel as long as they have no inter-transaction data dependencies.

Advanced Synchronization Facility (ASF) is an AMD64 hardware extension for transactional programming and lock-free data structures [1, 8]. ASF consists of seven instructions. SPECULATE and COMMIT are for demarcating transaction boundaries. ABORT is for rolling back a transaction voluntarily. LOCK MOV is for selectively annotating the memory accesses to be processed transactionally. RELEASE is to semantically drop a transactional read access performed previously by LOCK MOV before COMMIT. Advanced programmers can use the ISA directly to implement lock-free data structures. They use LOCK MOV and RELEASE to control the number of words accessed transactionally between SPECULATE and COMMIT. By keeping the number of cache lines accessed in a transaction under what ASF guarantees to support in hardware, they do not need to have a software backup mechanism for transactional execution and can use ASF as a flexible extension of the existing single-word atomic primitives such as CMPXCHG [3]. Average programmers can rely on compilers that accept high-level language constructs such as atomic blocks [4] and that generate ASF-based code. WATCHR and WATCHW are used to set a system-wide access monitor on an address in a transaction and to detect memory accesses to the address originating from other cores in the system.

After the ASF specification [1] had been released, we contacted many experts ranging from professors to OS developers and game programmers to get various reviews on ASF. In this paper, we present the summary of the reviews that we find invaluable in understanding what the first-generation TM hardware support should look like and how to improve the current ASF specification. This summary paper presents what the reviewers liked and disliked about ASF. It also includes their concerns and our opinions on their suggestions. By sharing the reviews, we hope that readers formulate their own opinions on the issues discussed in the paper, make suggestions to the TM community, and identify more issues. All of this will greatly help to bring out a better first-generation TM hardware support in future commercial processors.

The paper is organized as follows. Section 2 provides an overview of ASF and programming examples. Section 3 presents the summary of the reviews and our opinions on them. Section 4 discusses related work and Section 5 concludes the paper.

2. ASF Overview

2.1 ISA

Table 1 shows the seven instructions ASF adds to the AMD64 architecture. SPECULATE starts a transaction. It takes a register checkpoint that consists of the program counter (rIP) and the stack pointer (rSP). The rest of the registers are selectively checkpointed by software in the interest of saving hardware cost. Nested transactions are supported through flat nesting — parent transactions subsume child transaction [13].

LOCK MOV moves data between registers and memory like MOV, but with two differences. First, it should only be used within transaction boundaries; otherwise, a general protection exception (#GP) is triggered. Second, the underlying ASF implementation processes the memory access by LOCK MOV transactionally (i.e., data versioning and conflict detection for the access). A conflict against the access is

Category	Instruction	Function
Transaction Boundary	SPECULATE	Start a transaction
	COMMIT	End a transaction
Transactional Memory Access	LOCK MOV [Reg], [Addr]	Load from [Addr] to [Reg] transactionally
	LOCK MOV [Addr], [Reg]	Store from [Reg] to [Addr] transactionally
ASF Context Control	ABORT	Abort the current transaction
	RELEASE [Addr]	Undo a transactional load to [Addr] done by a previous LOCK MOV
Access Monitor	WATCHR [Addr]	Detect a store from [Addr] by the other cores
	WATCHW [Addr]	Detect a load or store to [Addr] by the other cores

Table 1. ASF instruction set architecture.

Status Code	Aborted by
ASF_CONTENTION	Transaction conflict
ASF_ABORT	ABORT instruction
ASF_CAPACITY	Transaction overflow
ASF_DISALLOWED_OP	Prohibited instructions
ASF_FAR	Exception, Interrupt

Table 2. ASF abort status codes set in rAX.

detected when either a transactional access from another transaction or a non-transactional access also touches the same cache line, and at least one of the accesses is a write. This ensures strong isolation of the memory accesses by LOCK MOV [7]. Since the detection is done at cache-line granularity, there can be false conflicts due to false data sharing in a cache line. To reduce design complexity, LOCK MOV is allowed only for the WB (writeback) memory access type [3]. We provide the minimum capacity guarantee as part of ISA so that transactions that access up to four distinctive memory words with LOCK MOV are guaranteed not to suffer from capacity overflows.

Since ASF allows transactional accesses and non-transactional accesses to be mixed within transaction boundaries, it is possible that the same cache line is accessed by both access types. ASF disallows only one case where a cache line modified by a transactional access is modified by a non-transactional access later in the same transaction. This rule aims to separate the previous transactional data that will be committed at the end of the transaction from the current non-transactional data that must be committed immediately. If this rule is violated, a #GP exception is triggered.

All the other cases are allowed. A transactional access following a non-transactional access to the same address is allowed since the non-transactional access is committed when the instruction triggering the access retires. A non-transactional load following a transactional load is allowed since loads do not conflict. A non-transactional load following a transactional store is allowed since the load just reads the result of the store in program order. A non-transactional store following a transactional load is allowed simply because it does not break the memory consistency maintained by the underlying ASF system. There are two sub-cases here with regard to another thread accessing the cache line. If another thread reads from the cache line, the value written by the non-transactional store is returned since the previous transactional load does not conflict with it and the non-transactional store has been already committed. If another thread writes to the cache line, a conflict is detected against the previous transactional load regardless of the non-transactional store.

RELEASE drops isolation on a transactional load access performed to an address by LOCK MOV. The underlying ASF implementation may stop detecting conflicts to the address with the semantics that the load access never happened. It is ignored if used on an address previously modified by LOCK MOV to prohibit discarding transactional data before committing a transaction.

COMMIT concludes a transaction. The register checkpoint is discarded and the transactional data are committed. A nested COMMIT does not finish a transaction for flat nesting. The underlying ASF implementation checks if there is a matching SPECULATE. If not, a #GP exception is triggered.

ABORT is for rolling back a transaction voluntarily. Transactional data are discarded, and the register checkpoint is restored. This brings the execution flow back to the instruction following the outermost SPECULATE and terminates transactional operation. ASF supports jumping to an alternative rIP at a transaction abort by manipulating the zero flag (ZF). ZF is cleared by SPECULATE and set when a transaction is aborted. JNZ (jump when not zero) with an alternative rIP can be placed right below SPECULATE. JNZ falls through at first since ZF is cleared by SPECULATE but jumps to the alternative rIP at transaction abort since ZF is set for an aborted transaction. Since the execution flow is out of transactional context after the transaction abort, JNZ needs to jump back to SPECULATE if the transaction is to be retried. The combination of SPECULATE and JNZ is essentially identical to an alternative design in which SPECULATE takes an alternative rIP as an operand since AMD64 processors translate this kind of complex instructions into multiple micro-operations (e.g., the micro-operation versions of SPECULATE and JNZ in this case). On detecting a transaction conflict, ASF performs the same abort procedure to roll back the conflicted transaction.

There are multiple conditions for a transaction abort besides ABORT and a transaction conflict. Since it is important for software to understand why a transaction has failed and respond appropriately, ASF uses rAX to pass an abort status code to software, as shown in Table 2. Since rAX is updated with the status code at a transaction abort, compilers must not use rAX to retain a temporary variable over SPECULATE. A general purpose register is used for the status code rather than a new dedicated register that would require additional OS support to handle context switches.

There are five abort status codes. ASF_CONTENTION is set when a transaction is aborted by a transaction conflict. ASF_ABORT is set by ABORT. ASF_CAPACITY is set when a transaction is aborted due to transactional hardware-resource constraints. ASF_DISALLOWED_OP is set when a prohibited instruction is attempted within transaction boundaries. Prohibited instructions are categorized into three groups. The first group includes the instructions that may change the code segments and the privilege levels such as FAR CALL, FAR JUMP, and SYSCALL. The second group includes the instructions that trigger interrupts such as INT and INT3. The third group includes instructions that can be intercepted by the AMD-V (Virtualization) hypervisor [2].

ASF_FAR is set when a transaction is aborted due to an exception (e.g., page fault) or an interrupt (e.g., timer interrupt). Due to design simplicity, ASF rolls back transactions at exceptions and interrupts. To report which instruction triggered the exception, ASF adds a new MSR (Model Specific Register), ASF_Exception_IP, which contains the program counter (rIP) of the instruction triggering the exception. At a page fault, a transaction is aborted and as usual the page fault's linear address is stored in CR2 (Control Register 2) [3].

WATCHR and WATCHW set an access monitor to track memory

```
Push:
  SPECULATE
  JNZ <Push>
  LOCK MOV RAX, [RBX + head]
  MOV [RDX+ next], RAX
  LOCK MOV [RBX + head], RDX
  COMMIT

Pop:
  SPECULATE
  JNZ <Pop>
  LOCK MOV RAX, [RBX + head]
  CMP RAX, 0
  JE <Out>
  MOV RDX, [RAX + next]
  LOCK MOV [RBX + head], RDX
Out:
  COMMIT
```

(a) Lock-free LIFO

```
Insert:
  SPECULATE
  JNZ <Insert>
  LOCK MOV RAX, [table_lock]
  CMP RAX, 0
  JE <ActualInsert>
  ABORT
ActualInsert:
  // insert an element
  COMMIT


Resize:
  LOCK BTS [table_lock], 0
  JC <Out>
  // resize the table
  MOV [table_lock], 0
Out:
```

(b) Resizable Hashtable

Figure 1. Lock-free LIFO and resizable hashtable with ASF ISA.

accesses to an address originating from other cores. WATCHR detects a store to the address. WATCHW detects a load or a store to the address. If such accesses are detected, the transaction enclosing the instructions is aborted.

2.2 Programming with ASF

ASF supports three programming styles: transactional programming, lock-free programming, and collaboration with traditional lock-based programming.

Transactional Programming: It is straight-forward to write transactional programs with ASF. A transaction is enclosed by SPECULATE and COMMIT, and all memory accesses in the transaction are performed with LOCK MOV. ABORT is used for rolling back the transaction voluntarily.

Lock-free Programming: ASF makes it easy to construct lock-free data structures for which simple primitives such as CAS are either insufficient or inconvenient. For example, a lock-free LIFO list is a concurrent linked list that pushes and pops elements like a stack without locking. It can be implemented with a single-word CAS (Compare-And-Swap) instruction such as CMPXCHG. A new element B is pushed by first reading the top element A, setting B's next pointer to point to A, and then writing B to the link head with CAS that updates the link head only when the head still points to A. While providing better concurrency than the lock-based LIFO, the CAS-based implementation has the *ABA* problem [12] caused by the time window between reading A and executing CAS. If another thread pops A, pushes a new element C, and pushes A back during the time window, CAS will update the list header with B since the header still points to A. This breaks the list since C is lost. This issue has traditionally been addressed by appending a version number to the list head pointer which is atomically read and updated with the pointer. However, this requires a wider CAS operation and extra space consumed for the list head pointer. ASF avoids these requirements by detecting data races not based on data values but based on the accesses themselves. In the *Push* function in Figure 1(a), the current value of the head pointer (RBX + head) is loaded transactionally to a temporary register (RAX), which initiates conflict detection against

```
SPECULATE
LOCK MOV RAX, [mem1]
LOCK MOV RBX, [mem2]

/* a random op with RAX and RBX */

LOCK MOV [mem1], RAX
LOCK MOV [mem2], RBX
COMMIT
```

Figure 2. A flexible fetch-and-op pattern.

the head pointer. Then, the current head pointer value is assigned to the next pointer of a new element (RDX + next) being pushed. Finally, the head pointer is updated with the new element (RDX). In this way, the Push function is free of the ABA problem since the head pointer is protected by ASF throughout the function and a transaction conflict is detected when C is pushed. The *Pop* function works similarly except that it has an additional check (i.e., CMP RAX, 0) to see if the LIFO is empty. Moreover, ASF allows multiple elements to be popped in one atomic operation, by allowing one to safely walk the list to the desired extraction point, then updating the head pointer.

Collaboration with Lock-based Programming: It is beneficial for ASF-based code to work with traditional lock-based code in order to use it as a simple software backup mechanism covering uncommon cases. For example, consider a concurrent hashtable. It is easy to develop the ASF-based code that inserts/removes an element to/from the hashtable. Occasionally, the hashtable may need to be resized, which requires accessing all elements in the hashtable. If the hashtable is large, the limited hardware resource in a first-generation ASF implementation will cause a transaction capacity overflow.

Our recommendation is to implement a lock-based resizing code with a 1-bit hashtable lock, as shown in Figure 1(b). The insertion code starts a transaction and reads the lock bit (table_lock) with LOCK MOV. If the lock bit is not set, it jumps to *ActualInsert* and inserts a new element. If the lock bit is set, it busy-waits by aborting and retrying the transaction. The resizing code grabs the lock non-transactionally with BTS (bit test and set) [3]. The BTS instruction reads the lock bit, copies it to CF (Carry Flag), and sets the lock bit. If the lock bit is set, someone else is resizing the hashtable, in which case, it escapes the function (JC). If the lock bit was not set, it resizes the hashtable and finishes the function by resetting the lock bit. By setting the lock bit with BTS, it aborts all active insert transactions through transaction conflicts and blocks future insert transactions until the resizing code resets the lock bit. This ensures that the resizing code accesses the hashtable exclusively, and the hashtable is race-free during resizing. While the resizing code is not running, the transactions inserting elements execute in parallel since they read-share the lock bit.

3. Likes, Dislikes, Concerns, and Our Opinions

In this section, we present the summary of the reviews on our ASF specification. We also present our opinions.

3.1 Overall Rating and Usage

As the first x86 TM hardware-support specification, ASF was highly welcome with very positive expressions such as "really cool", "drooling on it", and "I want it now". Some reviewers perceived it as more of a flexible x86 atomic primitive beyond CAS due to the limited TM

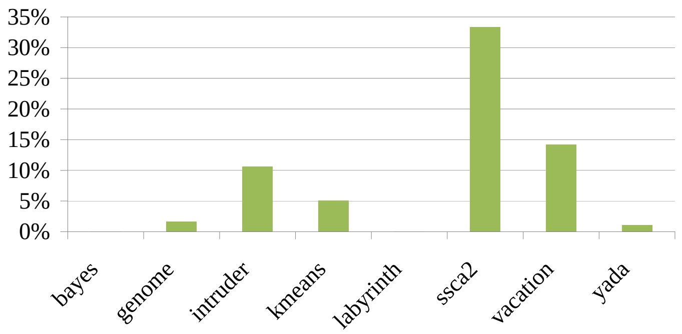

Figure 3. The ratios of the memory accesses instrumented with software barriers to all memory accesses within transaction boundaries in the STAMP benchmark suite [6]. The ratios of bayes and labyrinth are almost negligible.

support for general transactional programming with ASF. An interesting stereotype usage suggested for atomic operations was a flexible multi-word fetch-and-op as shown in Figure 2. Multiple data items are loaded transactionally (e.g., two data items in the figure), manipulated for a random op in private storage (e.g., registers or stack memory) non-transactionally, and stored back transactionally. If used, stack variables have to be alive only between SPECULATE and COMMIT. This usage encompasses many interesting cases such as multi-word compare-and-swap and multi-word fused-multiply-add (i.e., A = A + B x C). Some reviewers mentioned using ASF for speculative lock elision of small critical sections that works similar to the code in Figure 1(b).

3.2 Selective Annotation

Selective annotation of transactional memory accesses in a transaction enables *(a)* TM hardware resource saving for transactional programming and *(b)* flexible mixture of speculative accesses and non-speculative memory accesses for lock-free programming. Our observation from the existing transactional programs for STM systems is that only a small portion of memory accesses in a transaction has to be annotated with software barriers for transactional execution. Figure 3 shows the ratio of the memory accesses instrumented with software barriers to all memory accesses in transactions in the STAMP benchmark suite [6]. On average, the ratio is only 8%. There are various memory access patterns that contribute to the 92% of memory accesses that do not require software barriers. For example, as a CISC architecture, AMD64 has a small number of architectural registers and induces stack accesses to spill the registers. These stack accesses do not require conflict detection since they are to private data if the data do not escape the stack. If the stack variables are created after a transaction begins, the accesses to the variables do not need data versioning as well since the variables are effectively discarded by restoring the stack pointer when the transaction is aborted. As a result, these stack accesses do not require software barriers. Overall, the low ratio indicates that the majority of memory accesses in a transaction can be executed non-transactionally without compromising program correctness. Reviewers seem to easily acknowledge this opportunity of saving TM hardware resources.

On the other hand, as for the flexible mixture of transactional accesses and non-transactional accesses, some reviewers disliked allowing non-transactional accesses in a transaction since it could potentially

```
SPECULATE
LOCK MOV [mem1], RBX
MOV [mem2], RCX
COMMIT
```

Figure 4. A simple example that breaks the x86's memory consistency model.

weaken isolation among transactions. Other reviewers liked it since it enables TM software tools to "punch through" a transaction. This feature can, for example, be useful for debuggers to log information about outstanding transactions [10]. We advocate selective annotation in favor of giving more programming freedom to software developers. Programmers can always use transactional accesses to be on the safe side whenever they are concerned with weakening isolation.

Another concern with mixing transactional accesses and non-transactional accesses was about the exception triggered when a transactional access and a non-transactional access modify the same cache line. This can make ASF-based code less portable. For example, assume an object with two fields one of which is accessed transactionally and the other accessed non-transactionally. Depending on memory allocation schemes and runtime systems, the two fields may or may not be allocated in the same cache line, which means that the exception could be avoided in some systems but will be triggered in the other systems. This is troublesome and calls for open discussion.

3.3 Memory Access Ordering

There were questions about the cases where ASF breaks the x86's memory consistency model [3]. Figure 4 shows a very simple transaction with a transactional store and a following non-transactional store to two different cache lines. According to the x86 memory model, memory accesses should be observed in program order, which means that the transactional store should be exposed to the rest of the system first. However, in ASF, the transactional store is exposed after COMMIT is executed. The non-transactional store is exposed ahead in the reversed program order. We think that this deferred commit of the transactional store is essential for ASF to support atomicity. In our opinion, programmers should either use only transactional accesses for the code sensitive to the memory consistency model or be aware of this behavior and write the code accordingly. A COMMIT works as a memory barrier so that the memory accesses before the COMMIT are always exposed to the rest of the system ahead of the memory accesses after the COMMIT.

3.4 Minimum Capacity Guarantee

The issue of minimum capacity guarantee (i.e., the largest transaction memory footprint guaranteed not to cause capacity overflows) is one of the hottest topics not only for ASF but also for any TM hardware support in general. Should processor vendors provide any guarantee about TM or can TM support be purely best-effort (i.e., no guarantee at all)? Obviously, no guarantee is an easier choice for processor vendors and is advocated by some reviewers. However, other reviewers also pointed out that best-effort hardware transactions lack a good property of the existing atomic primitives (e.g., compare-and-swap) — that the primitives always commit in a certain way and make progress. They liked the minimum capacity guarantee supported by ASF in two ways. First, it makes best-effort hardware transactions look less like "black

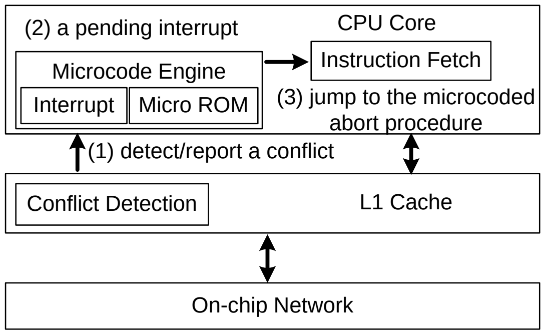

Figure 5. A time window for orphan transactions is from (1) to (3).

magic" for successful transactional execution. Second, programmers will know when they do not need to write software fallback code to deal with capacity overflows.

An obvious follow-up question was how we knew that the minimum capacity guarantee in the current ASF specification (i.e., four distinctive memory lines) was sufficient. The answer is that we did not know. As most readers can easily guess, the number four came from the likely set-associativity of four in the L1 cache. Since all AMD processors support out-of-order execution with the load/store queues, it should not be hard to increase the minimum capacity guarantee by leveraging the queues as transactional buffer as SUN Rock did [9]. However it is not easy to make a company-wide commitment on the minimum capacity guarantee for any future AMD processor with ASF support, as it may restrict the design freedom of future AMD micro-architectures.

3.5 Best-effort Maximum Capacity

Our discussion with AMD engineers brought up an interesting issue. The best-effort maximum capacity supported by ASF with no guarantee (i.e., the largest possible transaction memory footprint that may not cause capacity overflows) can also be problematic from the perspective of practical business. Assume a software product that has transactions bigger than the guaranteed minimum capacity but runs fine without capacity overflows due to additional best-effort transactional buffer provided by an AMD processor. The product does not have software fallback code to deal with capacity overflows simply because it just runs fine without the code. The problem happens with a potential next-generation AMD processor which provides a lower degree of best-effort support (e.g., a smaller best-effort transactional buffer). The software product would suffer from capacity overflows on this processor. According to the ASF specification that guarantees nothing for transactions bigger than the minimum capacity guarantee, it is clear that the software company has to add proper software fallback mechanisms. But what could happen in practice is that the company blames AMD for not being able to execute the code that used to run fine with older processors and demands AMD to fix it. One solution is to make the minimum capacity guarantee equal to the best-effort maximum capacity (i.e., no more best-effort approach). We present it as another open question for TM experts.

3.6 Abort

There was a question about the possibility of "orphan transactions" [11] in ASF. Orphan transactions are those that are marked to be aborted due to a transaction conflict by the underlying ASF system but not yet aborted. We noticed that there could be a time window for orphan transactions depending on ASF implementations. For example, one of the cost-effective ways we consider to implement the abort procedure is to deal with it as if it was a special interrupt. We refer to Figure 5 in our discussions. In step (1), the interrupt is triggered by the cache when a transaction conflict is detected with cache coherence protocol. In step (2), the interrupt is delivered to the CPU core by setting an interrupt bit. Finally, in step (3) the microcode engine checks the bit (typically at the end of issuing micro-ops of an x86 instruction) and starts the abort procedure. In this case, the window for orphan transactions starts at the time when the conflict is detected (i.e., (1)) and ends at the time when the microcode engine starts the abort procedure (i.e., (3)). With the out-of-order execution pipeline, many things can happen in this window. The potential problems with the window will have to be worked out with the designers of a specific baseline AMD processor. We intend not to expose any side effects of potential orphan transactions.

There were suggestions to clarify what happens with the registers other than rIP (program counter) and rSP (stack pointer) when a transaction aborts. The current specification guarantees the restoration of only rIP and rSP, leaving the other registers to be restored by software. The question was if those registers that were not modified in the aborted transaction are guaranteed to remain unchanged after the abort. The current specification does not guarantee it. However, we agree that this guarantee can enable interesting compiler optimizations to reduce the software–register-checkpoint overhead. We consider adding this guarantee to the next version of ASF.

3.7 Software Fallback

In comparison to traditional lock-based code, some reviewers did not like the programming pattern of combining hardware transactions and software-fallback code since it makes the best-case faster but the worst-case slower. This is not a clear win from the performance perspective unless there is a good proof showing that the best-case is the common case. We agree that it depends on application characteristics if the hardware TM support helps improve performance.

3.8 Nesting

Multiple reviewers suggested not to bother supporting nested transactions. They agreed that transaction composability with nesting is important but argued that this may have to be supported by software for first-generation TM hardware support. While it is quite easy for us to support flat nesting with a simple nesting depth counter [13], we agree that nested transactions will be rare at least with the limited TM hardware support of first-generation ASF implementations.

3.9 Contention Management

The baseline ASF contention-management policy is *attacker wins* where a transaction issuing a conflicting memory access wins a transaction conflict [5]. This can cause live-locks, and some reviewers expressed that "dead-lock is hard to deal with, but live-lock is harder". We chose the attacker wins policy for two reasons: 1) it is cheap to implement and 2) the complexity of modern processor designs tends to introduce random back-off latencies when transactions are re-executed, which can eliminate live-locks naturally in some cases. However, we agree that there still is a danger to suffer from live-locks and are developing cost-effective hardware schemes to eliminate live-locks. For now, we expect that software backup code either takes an alternative execution path or retries an aborted transaction after random backoff time.

3.10 RELEASE

In addition to the general difficulty of using early release [5], there was a concern about the case where RELEASE can unintentionally release transactionally accessed data. For example, assume two memory words are accessed transactionally but only one word is intended to be released. Depending on memory allocation mechanisms, the two memory words may or may not be located in the same cache line. The RELEASE instruction can only release whole cache lines. Consequently, if both words are located in the same cache line, both will be released together. Though we think that this problem has to be essentially dealt with by software developers, we also consider a hardware mechanism that detects this case with a set of counters incremented whenever distinctive locations of a cache line are transactionally accessed. An exception could be triggered when a RELEASE instruction is executed on a cache line with the counter value bigger than one.

3.11 Imprecise Exception

Since ASF aborts transactions at exceptions, the processor state observed by the OS exception handler is different from the processor state of the moment exceptions are triggered. In other words, ASF makes exceptions imprecise from the perspective of software. There were questions about what kinds of information the OS can get when a page fault happens. We thought that it to be enough for ASF to provide the accurate rIP and the faulting memory address. However, there were questions about other information such as the rSP value at a page fault. We were not told what the information is exactly for but certainly can consider providing more information if needed. Another question was about stepping an outstanding transaction through for debugging. We have an idea to allow for the stepping by suspending a transaction at a debug trap, running the debugger non-transactionally, and resuming the transaction when returning from the trap. However, it incurs additional cost to do that and will be considered only when there is a clear demand from software companies.

3.12 Cache-line Awareness

A reviewer disliked the fact that the current specification defines a transaction conflict in connection with cache lines instead of describing it more abstractly. We partly agree with him since this definition style may reduce the design freedom in choosing implementation schemes. However, it is highly likely that ASF implementations will leverage the existing cache coherence protocols for conflict detection. We think the concrete descriptions of the conditions for conflict detection with cache lines is better for programmers and compiler developers to help understand exactly when a transaction conflict happens.

3.13 Far Call and Ring 0

There were questions about the reason to prevent far calls (i.e., function calls that change segment registers) in a transaction. The answer is that we want to avoid implementing additional hardware schemes to eliminate security issues with program control transfer. The program control is transferred to the OS at system calls, exceptions, and interrupts [3]. If a control transfer happens in an application transaction and the underlying ASF implementation is not equipped with additional hardware to deal with program control transfer, the OS code is executed as part of the transaction. The problem is that if the transaction fails to complete, there can be security problems. For example, like most modern processors supporting security features to separate the OS and applications, the x86 architecture allows the OS and applications to use different code segments and privilege levels by changing the code segment selectors at the boundary of system calls [3]. If an ASF implementation does not have additional hardware to manage the segment registers that contain the segment selectors, an application transaction aborted in the middle of executing a system call will be restarted with the OS privilege level since the segment registers will still hold the segment selectors for the OS. This results in a security breach. Malicious programs can take advantage of this security hole to get the OS privilege level with a contrived multi-threaded TM code that forces a transaction in the middle of a system call to conflict with another transaction intentionally. We have a general hardware design to prevent security problems like this but have not reflected it yet in the specification due to its additional hardware cost.

On the other hand, the current ASF specification supports transactions in Ring 0 [3] (i.e., transactions that stay in the kernel mode throughout their lifetime).

4. Related Work

SUN developed their TM hardware support in the Rock processor [9]. There are several differences between ASF and SUN's TM support. First, ASF supports selective annotation of transactional memory accesses for efficient TM resource use. Second, it offers a minimum capacity guarantee to help programmers develop sophisticated lock-free data structures without complex backup software. Third, near function calls (i.e., function calls that do not change segment registers) and TLB misses do not abort transactions with ASF. Fourth, ASF takes a register checkpoint of rIP and rSP at the beginning of a transaction, leaving the rest of the registers to be managed by software.

5. Conclusions

In order to stimulate discussions on what the first-generation TM hardware support in commercial processors should look like, we present the summary of the various reviews on ASF and our opinions on them. We believe that this will enable a better TM hardware support to come out earlier.

6. Acknowledgments

We appreciate all TM experts who provided invaluable reviews for ASF. Their suggestions are duly considered for improvement of the current ASF specification.

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

7. References

[1] Advanced Synchronization Facility. http://developer.amd.com/CPU/ASF/Pages/default.aspx.

[2] AMD Virtualization. http://www.amd.com/us/products/technologies/virtualization/Pages/virtualization.aspx.

[3] AMD64 Architecture Programmer's Manual. http://developer.amd.com/documentation/guides/Pages/default.aspx.

[4] Intel c++ stm compiler. http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition-20/.

[5] J. Bobba, K. E. Moore, et al. Performance pathologies in hardware transactional memory. In *ISCA '07: Proc. of the 34th Intl. Symp. on Computer architecture*, pages 81–91, 2007.

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

[7] C. Cao Minh, M. Trautmann, J. Chung, et al. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *the Proc. of the 34th Intl. Symp. on Computer Architecture*. June 2007.

[8] D. Christie, J. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Riviere. Evaluation of amd's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of Eurosys 2010 Conference*, Paris, France.

[9] D. Dice, Y. Lev, et al. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS'09: 14th Intl. Conf. on Architectural Support for Programming Languages and Operating System*, 2009.

[10] V. Gajinov, F. Zyulkyarov, et al. Quaketm: parallelizing a complex sequential application using transactional memory. In *ICS '09: Proc. of the 23rd intl. conf. on Supercomputing*, 2009.

[11] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *the Proc. of the 20th Intl. Symp. on Computer Architecture*, May 1993.

[12] IBM Corporation. *IBM System/370 Extended Architecture, Principles of Operation*. IBM Publication No. SA22-7085, 1983.

[13] A. McDonald, J. Chung, et al. Architectural Semantics for Practical Transactional Memory. In *the Proc. of the 33rd Intl. Symp. on Computer Architecture*, June 2006.

[14] K. E. Moore, J. Bobba, et al. LogTM: Log-Based Transactional Memory. In *the Proc. of the 12th Intl. Conf. on High-Performance Computer Architecture*, Feb. 2006.