ASF: AMD64 Extension for Lock-free Data Structures and Transactional Memory

Jaewoong Chung*, Luke Yen*, Stephan Diestelhorst†, Martin Pohlack†, Michael Hohmuth†, David Christie*, Dan Grossman‡
*AMD Corporation, USA, {jaewoong.chung,luke.yen,david.christie}@amd.com
†AMD Corporation, Germany, {stephan.diestelhorst,martin.pohlack,michael.hohmuth}@amd.com
‡University of Washington, USA, djg@cs.washington.edu

***Abstract*—Advanced Synchronization Facility (ASF) is an AMD64 hardware extension for lock-free data structures and transactional memory. It provides a *speculative region* that atomically executes speculative accesses in the region. Five new instructions are added to demarcate the region, use speculative accesses selectively, and control the speculative hardware context. Programmers can use speculative regions to build flexible multi-word atomic primitives with no additional software support by relying on the minimum guarantee of available ASF hardware resources for lock-free programming. Transactional programs with high-level TM language constructs can either be compiled directly to the ASF code or be linked to software TM systems that use ASF to accelerate transactional execution. In this paper we develop an out-of-order hardware design to implement ASF on a future AMD processor and evaluate it with an in-house simulator. The experimental results show that the combined use of the L1 cache and the LS unit is very helpful for the performance robustness of ASF-based lock-free data structures, and that the selective use of speculative accesses enables transactional programs to scale with limited ASF hardware resources.**

***Keywords*-transactional memory, lock-free programming, x86 architecture**

I. Introduction

Multi-core processors are now prevalent in server and client systems. However, the complexity of parallel programming prevents programmers from taking advantage of abundant hardware parallelism. While scalable lock-free data structures avoid the drawbacks of the traditional lock-based programming such as deadlock and priority inversion, it remains difficult to develop and express their complex algorithms with basic atomic primitives such as single-word CAS (compare-and-swap) [1]–[3]. Transactional memory (TM) makes parallel programming easier by allowing programmers to execute a group of instructions atomically with transactions but needs hardware acceleration for performance [4]–[8].

Advanced Synchronization Facility (ASF) is an AMD64 hardware extension for lock-free data structures and transactional memory [9], [10]. It provides a *speculative region* that atomically executes speculative accesses in the region. Speculative regions are used to build flexible multi-word atomic primitives for lock-free programming and to provide first-generation hardware acceleration for transactional programming. The ASF ISA consists of five instructions to demarcate speculative regions, roll back a speculative region voluntarily, selectively annotate the memory accesses to be processed speculatively, and semantically drop a previous speculative load. ASF supports a minimum capacity guarantee that allows speculative regions to execute successfully on ASF hardware implemented with certain resource constraints. With the guarantee, programmers can avoid the burden of developing complex software backup mechanisms reported to make it often difficult to use purely best-effort hardware support [11].

We developed an out-of-order hardware design to implement ASF on a future AMD processor. The design 1) supports a minimum capacity guarantee, 2) allows a speculative region to survive TLB misses and branch mispredictions, 3) supports near function calls that do not change segment registers in a speculative region, 4) handles ASF instructions along a mispredicted execution path due to branch misprediction, 5) addresses the commit atomicity issue incurred when both the load-store (LS) unit and the L1 cache are used as speculative buffers under the AMD64 memory consistency model. The evaluation results with an in-house simulator show 1) that using the LS unit as speculative buffer in addition to the L1 cache is very helpful for performance robustness of ASF-based lock-free data structures against random data-sets and 2) that the selective use of speculative accesses with LOCK MOVs significantly reduces the speculative memory footprints of transactional programs and improves application scalability with limited ASF hardware resources.

The paper is organized as follows. Section II provides background on lock-free data structures and transactional memory. Section III explains the ASF ISA and the summary of the early reviews from a group of TM experts and potential custommers. Section IV describes the out-of-order hardware design for the ASF implementation. Section V presents the evaluation results. Section VI discusses related work, and Section VII concludes the paper.

II. Background

There have been myriad proposals for lock-free data structures [1]–[3], [12]. Lock-free synchronization is attributed to early work by Lamport [12]. Herlihy showed that, given an atomic compare-and-swap (CAS) primitive,

Table I. ASF instruction set architecture.

Category	Instruction	Function
Region Boundary	SPECULATE	Start a speculative region
	COMMIT	End a speculative region
Speculative Access	LOCK MOV Reg, [Addr]	Load from [Addr] to Reg speculatively
	LOCK MOV [Addr], Reg	Store from Reg to [Addr] speculatively
ASF Context Control	ABORT	Abort the current speculative region
	RELEASE [Addr]	Hint to semantically drop a previous speculative load from [Addr]

arbitrary concurrent data structures can be implemented in a lock-free manner [3]. More complex use cases of atomic primitives were shown for lock-free memory management [1]. Managed languages such as Java and C# provide programmers with concurrent data structure libraries such as the java.util.concurrent package [13].

Transactional Memory (TM) makes it easier for programmers to develop parallel programs. With TM, programmers enclose a group of instructions within a transaction to execute the instructions in an atomic and isolated way. The underlying TM system runs transactions in parallel as long as they do not conflict. Two transactions conflict when they access the same address and one of them writes to it. A hardware TM (HTM) system uses dedicated hardware to accelerate transactional execution [5]–[8]. It starts a transaction by taking a register checkpoint with shadow register files. Whenever the transaction writes to memory, the transactional data produced by the write are maintained separately from the old data by either buffering the transactional data in hardware buffers such as the cache or logging the old value (i.e., data versioning). It augments the cache with additional bits [6], [7] or uses separate hardware structures such as Bloom filters [8] to record the memory addresses read by the transaction in its read-set and those written in its write-set. A conflict between two transactions is detected by comparing the read-sets and the write-sets of both transactions (i.e., conflict detection). If a conflict is detected, one of the transactions is rolled back by undoing transactional writes, restoring the register checkpoint, and discarding transactional metadata (i.e., the read-/write-sets). Absent a conflict, the transaction ends by committing transactional data and discarding transactional metadata and the register checkpoint.

III. ASF ISA

A. ISA

Table I shows the five instructions ASF adds to the AMD64 architecture. The SPECULATE instruction starts a speculative region. 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 speculative regions are supported through flat nesting — parent speculative regions subsume child speculative regions [14].

Table II. Access rules on mixing speculative accesses and non-speculative accesses in the same speculative region.

Previous Access	Current Access			
	Speculative		Non-speculative	
	Load	Store	Load	Store
Speculative Load/Store	Allowed	Allowed	Allowed	Not Allowed
Non-speculative Load/Store	Allowed	Allowed	Allowed	Allowed

The LOCK MOV instruction moves data between registers and memory like a MOV, but with two differences. First, it should be used only within speculative region; otherwise, a general protection exception (#GP) is triggered. Second, the underlying ASF implementation processes the memory access by the LOCK MOV speculatively (i.e., data versioning and conflict detection for the access). A conflict against the access is detected when either a speculative access from another speculative region or a non-speculative 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 done with LOCK MOVs [4]. 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 MOVs are allowed only for the WB (writeback) memory access type [15]. ASF provides the minimum capacity guarantee so that speculative regions speculatively accessing distinct and naturally-aligned memory words less than the minimum guarantee are guaranteed not to suffer from capacity overflows that abort a speculative region when it requires more hardware resources than the underlying ASF implementation can support.

Since ASF allows speculative accesses and non-speculative accesses to be mixed within a speculative region, it is possible that the same cache line is accessed by both access types in the same speculative region. ASF disallows the case where a cache line accessed speculatively in the past is modified by a non-speculative store later as shown in Table II. This rule aims to separate the previous speculative data that will be committed at the end of the speculative region from the current non-speculative data that must be committed immediately. If this rule is violated, a #GP exception is triggered. The other cases are allowed. A cache line accessed non-speculatively in the past is allowed to be accessed speculatively or non-speculatively later since the past non-speculative access was committed as soon as it retired. A cache line accessed speculatively in the past is allowed to be accessed speculatively or loaded non-speculatively since the load just reads the up-to-date values

in program order.

The RELEASE instruction is a hint to drop isolation on a speculative load from a memory address. The underlying ASF implementation may stop detecting conflicts against the address with the semantics that the load never happened. The RELEASE is ignored if the address was modified speculatively. This is to prohibit discarding speculative data before committing a speculative region.

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

The ABORT instruction rolls back a speculative region voluntarily. Speculative data are discarded, and the register checkpoint is restored. This brings the execution flow back to the instruction following the outermost SPECULATE and terminates speculative operation. ASF supports jumping to an alternative rIP at an abort by manipulating the Zero flag (ZF). ZF is set by a SPECULATE and cleared when a speculative region is aborted. A JNZ (jump when not zero) with an alternative rIP can be placed right below the SPECULATE. The JNZ falls through at first since ZF is set by the SPECULATE but jumps to the alternative rIP at an abort since ZF is cleared for the aborted speculative region. Since the execution flow is out of the ASF hardware context after the abort, the JNZ needs to jump back to the SPECULATE if the speculative region is to be retried. On detecting a conflict, ASF performs the same abort procedure to roll back the conflicted speculative region.

There are multiple reasons to abort a speculative region besides the ABORT instruction and a conflict. Since it is important for software to understand why a speculative region has failed and respond appropriately, ASF uses rAX to pass an abort status code to software. Since rAX is updated with the status code at an abort, compilers must not use rAX to retain a temporary variable over a SPECULATE. rAX is used for the status code since a new dedicated register for the status code would require additional OS support to handle context switches.

There are five abort status codes. ASF_CONTENTION is set when a speculative region is aborted by a conflict. ASF_ABORT is set by the ABORT instruction. ASF_CAPACITY is set when a speculative region is aborted due to capacity overflows. ASF_DISALLOWED_OP is set when a prohibited instruction is attempted within speculative regions. The 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 CALLs, FAR JUMPs, and SYSCALLs. The second group includes the instructions that trigger interrupts such as INTs and INT3s. The third group includes instructions that can be intercepted by the AMD-V (Virtualization) hypervisor [16]. ASF_FAR is set when a speculative region is aborted due to an exception (e.g., page fault) or an interrupt (e.g., timer interrupt). For design simplicity, ASF rolls back speculative regions 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 speculative region is aborted, and the page fault's linear address is stored in CR2 (Control Register 2) as usual before the OS page fault handler is invoked [15]. A speculative region is not aborted by TLB misses, branch misprediction, and near function calls that do not change segment registers.

B. Programming with ASF

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

Transactional Programming: It is straightforward to write transactional programs with ASF. A transaction is mapped to a speculative region whose memory accesses are done with LOCK MOVs between a SPECULATE and a COMMIT. However, it is more likely that average programmers use high-level TM language constructs and rely on TM compilers to optimize the use of LOCK MOVs to run transactions efficiently with limited ASF hardware resources. In [10], a TM compiler is developed to generate the ASF code from C/C++ programs written with high-level TM language constructs. It optimizes the speculative memory footprints of the ASF code by analyzing speculative regions, identifying memory accesses that do not require speculative accesses such as certain stack accesses, and not using LOCK MOVs for them.

Lock-free Programming: ASF makes it easy to construct lock-free data structures for which simple atomic 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 instruction such as a CMPXCHG. The top element A is popped by first reading the pointer to A from the link head, reading the pointer to the next element B from A's next pointer, and then writing the pointer to B to the link head with a CAS that updates the link head only when the head still points to A. While providing better concurrency than a lock-based LIFO, the CAS-based implementation has the *ABA* problem [17] caused by the time window between reading the pointer to A and executing the CAS. If another thread pops A, pops B, and pushes A back during the time window, the CAS will update the list head with the pointer to B since the head still points to A. This breaks the list since B is not in the LIFO any more. This issue has traditionally been addressed by appending a version number to the list head which is atomically read and updated with the head. However, this

```
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 hash table with ASF ISA.

requires a wider CAS operation and extra space consumed for the list head.

ASF avoids these requirements by detecting data races not based on data values but based on the accesses themselves. In the *Pop* function in Figure 1(a), the pointer to A is loaded from the link head (RBX + head) to RAX speculatively, which initiates conflict detection against the link head. If the LIFO is not empty (i.e., CMP RAX, 0), the pointer to B is loaded from A's next pointer (RAX + next) to RDX. Finally, the link head is updated with the pointer to B (RDX). In this way, the Pop function is free of the ABA problem since the link head is protected by ASF throughout the function, and a conflict is detected if another core pops A concurrently. The *Push* function works similarly except that it does not have the CMP instruction 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 and then update the link head.

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

Our recommendation is to implement lock-based resizing code with a 1-bit hash table lock, as shown in Figure 1(b). The insertion code starts a speculative region and reads the lock bit (table_lock) with a LOCK MOV. If the lock bit is not set, the code jumps to *ActualInsert* and inserts a new element. If the lock bit is set, it busy-waits by aborting and retrying the speculative region. The resizing code grabs the lock non-speculatively with a BTS (bit test and set) [15]. 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 hash table, in which case, the code escapes the function (JC). Otherwise, the current invocation resizes the hash table and finishes the function by resetting the lock bit. By setting the lock bit with the BTS, the resize function aborts all active speculative regions inserting elements by conflicts and prevents future speculative regions from entering the insertion code until it resets the lock bit. This ensures that the resizing code accesses the hash table exclusively, and the hash table is race-free during resizing. While the resizing code is not running, the speculative regions inserting elements execute in parallel since they read-share the lock bit.

C. Discussion with Early Reviewers

One of the best ways to improve an ISA is to talk with and learn from potential customers and experts [18]. This section presents the summary of the early reviews about the ASF ISA from a group of TM experts and software developers.

Non-speculative Access in Speculative Region: There was concern about allowing non-speculative accesses in a speculative region for transactional programming since this feature could weaken isolation among speculative regions. Other reviewers liked the feature since it enables TM software tools to "punch through" a speculative region. This feature can, for example, be useful for debuggers to log information about outstanding speculative regions [19]. We advocate non-speculative accesses mainly for lock-free programming in favor of giving more programming freedom to software developers. In addition, the selective use of speculative accesses can be helpful in saving the ASF hardware resources for transactional programming as shown in Section V-C. However, in any case, programmers can always choose to use only speculative accesses to be on the safe side whenever they are concerned with weakening isolation.

Minimum Capacity Guarantee: Should ASF provide any guarantee or can it be purely best-effort (i.e., no guarantee at all)? Obviously, no guarantee is an easier choice for hardware designers and is advocated by some reviewers. However, other reviewers also pointed out that best-effort speculative regions would lack a good property of the existing atomic primitives (e.g., CAS) – that the primitives always commit in a certain way and make progress. They liked the minimum capacity guarantee (i.e., the largest memory footprint guaranteed not to cause capacity overflows) supported by ASF in two ways. First, it makes speculative regions look less like "black magic" for successful speculative execution.

Second, programmers can tell when they do not need to write software fallback code to deal with capacity overflows. We evaluate three design options to support the minimum capacity guarantee in Section V.

Nesting: Multiple reviewers suggested not to bother supporting nested speculative regions. They agreed that composability with nesting is important for transactional programming but argued that this may have to be supported by software at least for early ASF implementations. We agree with this argument and consider dropping the nesting support.

Contention Management: The baseline ASF contention-management policy is *attacker wins* where a speculative region issuing a conflicting memory access wins the conflict [20]. 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 speculative regions 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 we 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 the aborted speculative region after random back-off time [20].

Imprecise Exception: Since ASF aborts speculative regions 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. Some reviewers asked if ASF could provide more information about exceptions beyond ASF_Exception_IP to compensate imprecise exception, which surely is possible.

Another question was about stepping an outstanding speculative region through for debugging. We have an idea to enable the stepping by suspending a speculative region at a debug trap, running the debugger non-speculatively, and resuming the speculative region when returning from the trap. However, it incurs additional cost and will be considered only when there is a clear demand from software developers.

IV. Implementation

This section presents the current out-of-order hardware design for the ASF implementation on a future AMD processor.

A. Overview

While the main design challenges to the ASF implementation come from putting the long-lived ASF speculative hardware context and the short-lived out-of-order execution context together, at a high level, our design in Figure 2 is close to a cache-based HTM design [6], [7]. It buffers speculative data in the load/store queues and the L1 cache (i.e., lazy data versioning [6]). The L1 cache supports relatively large speculative regions for transactional programming. The load/store queues are used to provide a higher minimum capacity guarantee for lock-free programming since the minimum capacity guarantee with a 4-way set-associative L1 cache is limited to only four distinct memory words. The design adds one bit per load/store queue entry and two bits per cache line to mark speculative data. The AMD Coherent HyperTransport (cHT) protocol [21] is used to detect conflicting speculative/non-speculative memory accesses from other cores by checking incoming cache coherence messages against the additional bits. If a conflict is detected, the conflicted speculative region is aborted by discarding the speculatively modified data and resetting the bits. If COMMIT is reached without conflicts, the buffered speculative data are committed by gang-clearing the bits and sending the stores buffered in the store queue to the L1 cache. Following the reviewers' suggestion in Section III-C, this design does not support nesting for simplicity. Figure 2 shows the hardware structures for the ASF implementation. In the figure, the components changed or added for ASF are shown in grey. The rest of the section explains the ASF implementation details.

B. Out-of-Order Implementation Details

Beginning a Speculative Region: The SPECULATE instruction to start a speculative region is microcoded in the microcode ROM. On detecting a SPECULATE, the instruction decoder sets the InSP (in speculation) bit to remember the beginning of a speculative region. It signals the instruction dispatcher to read the SPECULATE microcode. The microcode 1) computes the next rIP so that rIP is restored to point to the instruction following the SPECULATE at an abort, 2) saves the next rIP and the current rSP in the shadow register file, and 3) executes a mfence (memory fence) micro-op. The mfence generates a dependency between SPECULATE and later LOCK MOVs, which prevents the LOCK MOVs from being executed ahead of the SPECULATE in the out-of-order execution stage [22]. The shadow register file is carved out of the existing micro-architectural register file used only by micro-ops.

Speculative Accesses: To track speculative accesses with LOCK MOVs, two bits are added per cache line: the SW (speculative write) bit for speculative stores and the SR (speculative read) bit for speculative loads as shown in Figure 2. The SW bit is also added per store queue entry, and the SR bit per load queue entry. A LOCK MOV is issued to the LS (load/store) unit and sets the SW bit of the store queue entry for a store operation and the SR bit of the load queue entry for a load operation. The data movement operation for the LOCK MOV is handled in the same way as a normal MOV. The AMD64 TLB refill hardware allows

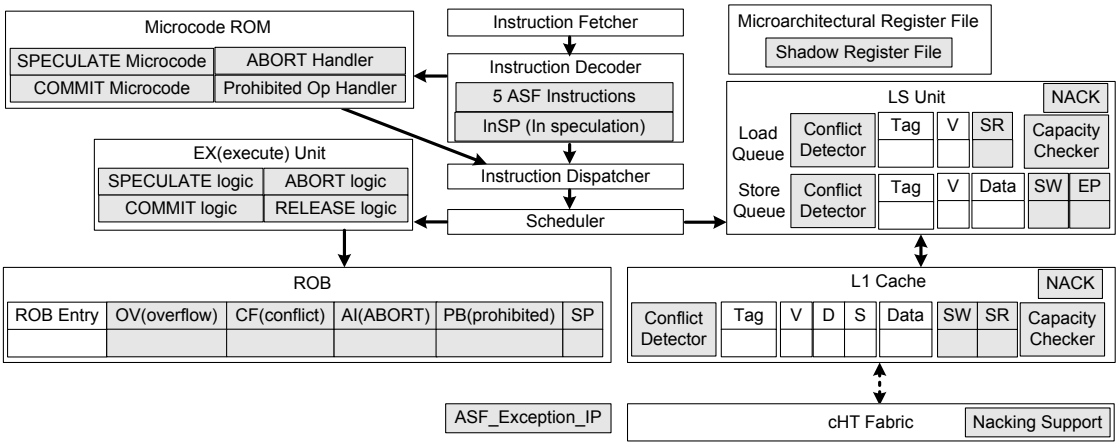

Figure 2. The hardware structures for ASF implementation. The components changed or added for ASF are shown in grey.

a speculative region to survive a possible TLB miss during address translation for the LOCK MOV by handling it in hardware. When the LOCK MOV retires, the SR bit of the load queue entry is cleared, and the corresponding SR bit in the L1 cache is set. The SW bit of the store queue entry is cleared when the speculative data are transferred from the store queue to the L1 cache along with setting the SW bit in the L1 cache. If speculative data are written to a cache line that contains non-speculative dirty data (i.e., the D (Dirty) bit is set, but the SW bit is not set), the cache line is written back first to make sure that the last committed data are preserved in the L2/L3 caches or the main memory.

The LS unit starts buffering speculative data only when the transfer of the SW/SR bits from the load/store queues to the L1 cache meets two conditions: 1) it causes a cache miss (i.e., no cache line to retain the bits) and 2) all cache lines of the indexed cache set have their SW and/or SR bits set (i.e., no cache line to evict without triggering a capacity overflow). If a non-speculative access meets the two conditions above, the L1 cache handles it as if the access is a UC (uncacheable) type to avoid a capacity overflow, and the L2 cache handles it directly. In order to hold as much speculative data as possible, the L1 cache eviction policy evicts the cache lines without the SW/SR bits set first. A cache line prefetched by a hardware prefetcher is inserted into the L1 cache for a speculative region only when it does not cause an overflow exception.

A speculative store buffered in the store queue is troublesome since the store queue is considered as a local write buffer in the AMD64 memory model [15]. Therefore, a store is visible to the rest of the system only after it is transferred to the L1 cache. To broadcast the existence of the buffered speculative store that cannot be transferred to the L1 cache without triggering a capacity overflow, an exclusive permission request for the store is sent to the cHT fabric when the store retires in the store queue. This enables the other cores to detect a conflict against the store. Once the exclusive permission is acquired, the EP (exclusive permission) bit per store queue entry is set to remember the acquisition. The COMMIT instruction later checks the EP bit to make sure that the store has been seen by the rest of the system for conflict detection before starting the commit procedure.

An overflow exception is triggered when the load/store queues do not have an available entry for an incoming LOCK MOV (i.e., the SW/SR bits of all entries are set in the queue the LOCK MOV needs to go to). A tricky problem is that non-speculative accesses should always make forward progress regardless of the number of the LOCK MOVs executed in a speculative region. Our design reserves one entry per queue for the non-speculative accesses to be able to execute them even when the rest of the load/store queue entries are full of speculative data. Another problem with the overflow exception is that LOCK MOVs along a mispredicted execution path due to branch misprediction can trigger a false overflow exception. To address this problem, the OV (overflow) bit is added per ROB entry. On detecting a capacity overflow from a LOCK MOV, instead of triggering an overflow exception immediately, the OV bit is set in the ROB entry of the LOCK MOV. If the LOCK MOV is on a mis-speculative path, the ROB entry and the hardware resources associated with the entry will be discarded by the existing branch misprediction recovery mechanism. A true capacity overflow will be serviced when the ROB entry reaches the bottom of the ROB (assuming no other abort conditions exist before the LOCK MOV) and triggers an overflow exception.

Conflict Detection: A conflict with another core is detected by checking the SW/SR bits in the LS unit and the L1 cache against incoming cache coherence messages. An invalidating message (for store) conflicts with the SW bit and the SR bit. A non-invalidating message (for load) conflicts with the SW bit. The baseline AMD processor already has the necessary CAM (content addressable memory) logic in

the LS unit and the L1 cache for conflict detection. The LS unit has the CAM logic to check the address tags of all loads and retired stores for different purposes. The L1 cache has the CAM logic for address tags. These CAM logics are extended to read out the SW/SR bits for conflict detection.

Unlike a store, a load is problematic for conflict detection since a conflict against the load has to be detected from the moment the loaded value is bound to a register. Since the value is bound before the load retires, there can be a false conflict due to in-flight speculative loads (i.e., those that have not retired yet) on a mispredicted execution path. To eliminate the false conflict, the CF (conflict) bit is added per ROB entry. A conflict with a SR bit in the load queue sets the CF bit of the ROB entry of the conflicted load if the load is in-flight. If the conflicted load is on a mispredicted execution path, its ROB entry with the false conflict information will be discarded before reaching the bottom of the ROB. If not, the ROB invokes the abort procedure when the ROB entry of the conflicted load reaches the bottom of the ROB. A conflict with the other speculative accesses (i.e., retired loads and stores either in the LS unit or in the L1 cache) is immediately reported to the ROB as a new interrupt, ASF_Conflict, since the retired accesses are free of branch misprediction. On detecting the ASF_Conflict interrupt, the ROB invokes the abort procedure for the conflict. The conflicted core replies to the conflicting core pretending that it could not find a matching cache line for the cHT protocol.

Aborting a Speculative Region: As explained in Section III-A, a speculative region can be aborted for various reasons. An abort is triggered when the ROB detects any of the OV (overflow) bit, the CF (conflict) bit, the AI (ABORT instruction) bit, and the PB (prohibited) bit set in the bottom ROB entry, or when the ROB receives an ASF_Conflict interrupt. The AI bit is set for the ABORT instruction, and the PB bit for the prohibited instructions. By checking the AI/PB bits at the bottom of the ROB, our design eliminates false aborts on a mispredicted execution path.

The abort procedure shown in Figure 3(a) is very similar to the normal interrupt handling procedure and is used in all abort cases. The ROB first ❶ initiates the pipeline flush that invalidates all ROB entries and load/store queue entries and ❷ signals the microcode ROM with one of the ASF abort status codes. Then, the uninterruptable abort handler in the microcode ROM conducts the following procedure. It ❸ invalidates the L1 cache lines with the SW bits, ❹ clears the SW/SR bits in the L1 cache, ❺ sets rAX with the signaled abort code, ❻ clears the ZF flag, and ❼ reads the saved rIP and rSP values from the shadow register file. At this point, the abort procedure bifurcates. If the abort code is ASF_FAR (i.e., the abort is due to exceptions or interrupts), the microcode ⑧ sets ASF_Exception_IP with the current rIP (i.e., the one that triggered the exception), ⑨ sets rIP and rSP with the saved rIP and rSP values, and ⑩ jumps to the existing exception handler in the microcode ROM. This makes the exception handler think that the exception is triggered by the instruction following a SPECULATE. If the abort status code is not ASF_FAR, the microcode ❽ sets rIP and rSP with the saved rIP and rSP values, and ❾ executes a jump micro-op to redirect the instruction fetcher to the saved rIP (typically JNZ as explained in Section III-A).

Committing a Speculative Region: The main challenge to committing a speculative region is that the simple gang-clear of the SW/SR bits in the LS unit and the L1 cache does not guarantee the atomicity of a committing speculative region under the AMD64 memory model [15]. This problem is shown in Figure 4(a). In the figure, Core1 ran a speculative region that wrote to X, Y, and other variables speculatively. The new X value (X=1) happened to be buffered in the L1 cache, and the new Y value (Y=1) in the store queue. Now, Core1 is about to commit the speculative region. If Core1 just gang-clears the SW/SR bits to commit the speculative region, the new Y value will stay in the store queue. The problem is that, if Core2 reads X and Y at this point, it will read the new X value (X=1) in the L1 cache and the old Y value (Y=0) in the main memory. This is because any value in the store queue of Core1 cannot be read by Core2 according to the AMD64 memory model (because the store queue is considered as the local write buffer of Core1). This breaks the atomicity of the speculative region because Core2 reads the mix of the new value (X=1) and the old value (Y=0). To eliminate this problem, our design uses a nacking mechanism that detects a conflicting cache coherence message from another core and nacks the message during the commit procedure. The core receiving the nack message retries the nacked cache coherence operation later. In Figure 4(b), the commit procedure starts the nacking mechanism first and then gang-clears the SW/SR bits in the load queue and the L1 cache. At this point, Core2 can read the new X value (X=1) in the L1 cache but cannot read any Y value since Core1 detects the conflicting read from Core2 by checking the SW bit set for Y in the store queue and nacks the read operation. On receiving the nack message, Core2 retries the read operation and obtains the new Y value (Y=1) after Core1 completes transferring the new Y value to the L1 cache. In this way, the nacking-based commit procedure guarantees the atomicity of the committing speculative region by preventing the other cores from reading the old values of memory addresses (e.g., Y=0 in this example) if the store queue of the committing core has new values to commit to the memory addresses (e.g., Y=1).

Figure 3(b) shows the overall commit procedure based on the nacking mechanism. The COMMIT instruction is microcoded. On detecting a COMMIT, the instruction decoder resets the InSP bit and signals the dispatcher to read the COMMIT microcode. The microcode ROM has a feature to stall dispatching micro-ops until a wait condition specified in the microcode is satisfied. A new wait condition is added for

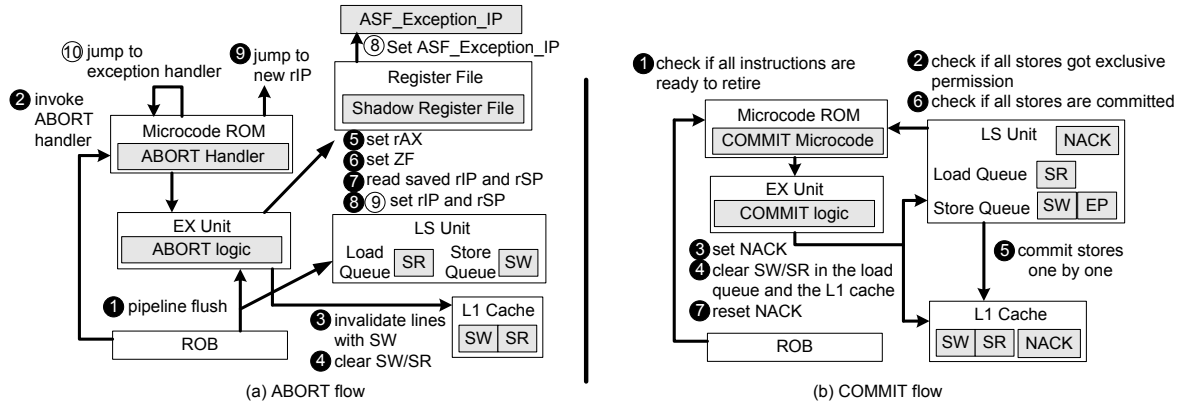

Figure 3. The execution procedures to abort and commit a speculative region.

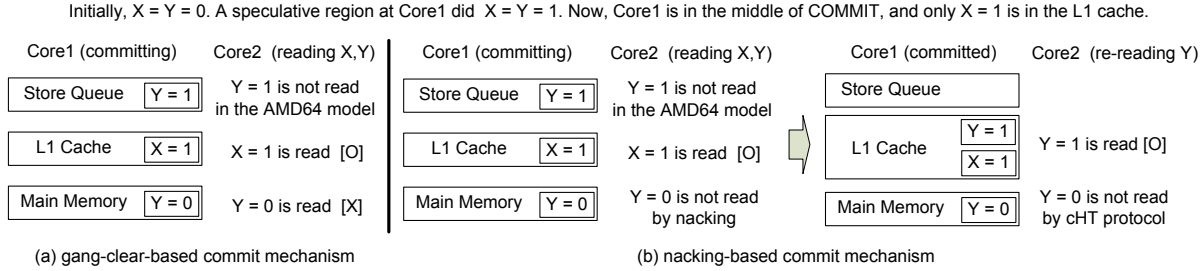

Figure 4. This example shows how the nacking mechanism guarantees the atomicity of a committing speculative region under the AMD64 memory model.

the COMMIT that checks ❶ if all instructions in the ROB are ready to retire without exceptions and ❷ if all retired stores in the store queue have obtained exclusive permissions (i.e., their EP bits are set). Once the condition is satisfied, the COMMIT microcode ❸ signals the L1 cache and the LS unit to set their NACK bits. Once the bits are set, conflicting cache coherence messages are nacked instead of aborting the current speculative region. There is no deadlock due to the nacking mechanism because the committing speculative region holds all necessary exclusive permissions to complete the COMMIT. Then, it ❹ clears the SW/SR bits in the L1 cache and the load queue to first commit the speculative data in them. This ❺ naturally enables the store queue to resume transferring the speculative data to the L1 cache since the SW/SR bits in the L1 cache are now cleared. The data transferred from the store queue do not set the SW bits in the L1 cache (done by checking the NACK bit) since the completion of the data transfer at this point means that the data are committed and made visible to the rest of the system. The microcode ROM stalls on another wait condition that ❻ checks if no store queue entry has the SW bit set. By the time this condition is met, all new values in the store queue are transferred to the L1 cache. Finally, it ❼ signals the L1 cache and the LS unit to reset their NACK bits, and the commit procedure completes here. Nothing is done to the shadow register file since it will be overwritten by the next SPECULATE. Since the ASF hardware context in this design can support a single speculative region, a speculative region should not enter the out-of-order execution stage before the previous speculative region commits. This case is naturally handled in our design since the microcode ROM prevents the instructions behind a COMMIT in the program order from being dispatched until the COMMIT completes.

Handling Branch Mis-prediction: Mis-predicted branches before or in the middle of a speculative region are troublesome since the speculative region can be fetched and executed along a mispredicted execution path. Our design uses the existing branch misprediction recovery mechanism to restore the ASF hardware resources occupied by the mispredicted ASF instructions of the speculative region. When the misprediction is detected, the recovery mechanism naturally discards the ROB entries and the load/store queue entries occupied by the mispredicted instructions. Nothing is to be done to the L1 cache since the SW/SR bits are transferred to the L1 cache only for retired instructions, and the mispredicted instructions never retire. The NACK bits in the L1 cache and the LS unit are not set at this point since a mispredicted COMMIT instruction will never start the commit procedure. This condition is true because the mispredicted branch will flush all younger instructions (including the COMMIT) when the branch resolves. The InSP bit is tricky to handle since it has to be restored to the value at the moment the

mispredicted branch was decoded. To restore the InSP bit properly, the instruction decoder tags the up-to-date InSP bit values along with the decoded instructions. The SP bit is added per ROB entry to record the tagged InSP bit value of each instruction. When a branch is found to be mispredicted, the SP bit of the branch's ROB entry is used to restore the InSP bit. If the mispredicted branch is before the speculative region, its SP bit is 0 and the InSP bit is reset. If it is in the middle of the speculative region, its SP bit is 1 and the InSP bit is set.

Handling Exceptions and Interrupts: Exceptions and interrupts in a speculative region are handled first by the ASF abort handler to abort the speculative region and then by the existing exception handler in the same way as they are handled without speculative regions. If an instruction before a SPECULATE in the program order triggers an exception after the SPECULATE has entered the execution stage, the ASF hardware context is restored in the same way as a mispredicted branch before a SPECULATE is handled.

Handling Prohibited Instructions: Since the prohibited instructions should not be allowed to enter the execution stage and modify non-speculative resources such as segment registers, our design detects them early at the decoding stage when the InSP bit is set. On detecting the instructions, the instruction decoder signals the microcode ROM to jump to the prohibited op handler. The handler 1) executes a micro-op that sets the PB (prohibited) bit of its own ROB entry, and 2) waits for the entry to reach the bottom of the ROB. Then, the ROB picks up the exception and initiates the abort procedure. This way, the prohibited instructions in a speculative region never enter the execution stage.

Supporting RELEASE: The RELEASE instruction is tricky to implement since it builds a dependency with LOCK MOVs around it. We use the bottom execution feature of the ROB to implement RELEASE. Once dispatched, a RELEASE does nothing until its ROB entry reaches the bottom of the ROB. When the entry reaches the bottom, the ROB signals the RELEASE execution logic to search for the SR bit to be reset by the RELEASE only in the L1 cache and in the portion of the load queue that contains retired loads. After resetting the matching SR bit, the RELEASE logic signals the ROB for its completion. This way, our design abides by the dependency among the RELEASE and the LOCK MOVs around it since the loads behind the RELEASE have not retired yet and their SR bits are not examined by the RELEASE logic.

V. Evaluation

A. Experimental Environment

We are using two in-house simulators to explore the design space for ASF. One simulator focuses on the internals of a future out-of-order core design, and the other simulator focuses on the overall multi-core processor design with a simplified core model for scalable tests. Unfortunately, we could not publish the results from the first simulator by the time of writing this paper. We present the results from the second simulator in this paper. We implemented the ASF design explained in Section IV and three variations by changing the speculative buffer structures as shown in Table III. ASF-LC is the baseline implementation that uses both the L1 cache and the LS unit as speculative buffer. ASF-C8 and ASF-C4 use only the L1 cache with different set-associativities as speculative buffer. ASF-AS does not support LOCK MOV and handles all memory accesses in a speculative region speculatively. All four designs use the system parameters in Table IV.

Table IV. System parameters for simulation.

Feature	Description
CPU	3 GHz, 8 x86 cores, 32 load queue entry, 24 store queue entry
L1 Cache	32 KB, 4-way, 64B line, MOESI, write-back, 3 cycle hit time, private
L2 Cache	8 MB, 8-way, 64B line, MOESI, write-back, 15 cycle hit time, shared
Memory	4 GB, DDR3-1600, tCAS,tRP,tWR,tRCD=12.5ns, tRAS=35ns, 2 memory channels
Interconnect	AMD Coherent HyperTransport

We used the STAMP benchmark suite [23] for transactional programming tests. We took out and modified the data structures in the library of the benchmark for lock-free programming tests. The benchmark has eight transactional applications. For each application, the STAMP benchmark suite provides two versions that have the same transactions: one for HTM and the other for STM. The HTM version has only markers for the beginning and the end of a transaction. The STM version additionally annotates transactional memory accesses with software barriers. We used the STM version to obtain the ASF code for ASF-LC, ASF-C8, and ASF-C4 by substituting the software barriers with LOCK MOVs and the transaction begin/end functions with SPECULATEs and COMMITs. This allows us to test the optimistic case of using a TM compiler to generate the ASF code [10]. We used the HTM version for ASF-AS. The application-level abort handler (i.e., the target function of the JNZ after a SPECULATE) restarts aborted speculative regions after random back-off delays. The handler falls back to a software mechanism to deal with an overflow exception. The basic idea of the mechanism is for an overflowed core 1) to wait for all the other cores to come out of speculative regions, 2) to prevent them from entering another speculative region, 3) to execute the non-speculative version of the overflowed speculative region, and 4) to allow the other cores to enter speculative regions again after the overflowed speculative region finishes.

Table III. Four ASF design options tested.

Scheme	Speculative Buffer	Minimum Guarantee	Maximum Capacity
ASF-LC(LS + Cache)	32KB 4-way L1 cache + 32 load queue + 24 store queue	27 Words (4 way + 24 store queue entries - 1 for non-speculative access)	33216 Bytes (32768 + (24 + 32) * 8)
ASF-C8(Cache 8-way)	32KB 8-way L1 cache	8 Words	32768 Bytes
ASF-C4(Cache 4-way)	32KB 4-way L1 cache	4 Words	32768 Bytes
ASF-AS(All Speculative)	32KB 4-way L1 cache	4 Words	32768 Bytes

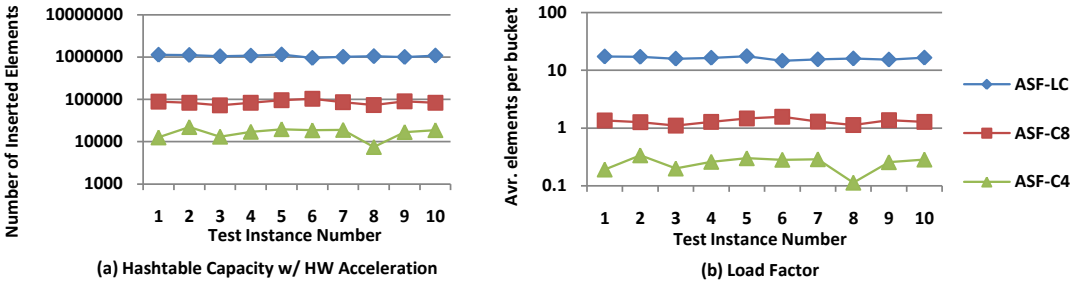

Figure 5. Comparison of ASF-LC, ASF-C8, and ASF-C4 for performance robustness of lock-free data structures.

B. Lock-free Programming

Since the scalability of ASF-based data structures such as hash tables and RB trees is evaluated as part of the STAMP applications in Section V-C, this section focuses on the performance robustness of the ASF design options for lock-free programming. As pointed out by reviewers in Section III-C, the best-effort nature of TM-like hardware support can make it hard for programmers to deal with performance issues from the pure software perspective. To test how ASF-LC, ASF-C8, and ASF-C4 deal with random data-sets robustly, we developed a micro-benchmark that uses the chained hash table modified from the STAMP benchmark for ASF. It allocates 100M 4-Byte memory blocks, assigns unique IDs to the blocks, and spawns multiple threads that pick the blocks randomly and insert them into the hash table with 65536 buckets after checking redundancy until the first overflow exception is triggered. We repeated the test ten times with different seeds for random number generation. Figure 5(a) shows that ASF-LC allows the hash table to contain 12.3x more blocks than ASF-C8 and 63.9x more blocks than ASF-C4 in full ASF hardware acceleration. This is because ASF-LC leverages the fully-associative load/store queues to avoid a capacity overflow when traversing the chains of outlier buckets that happen to have much more elements than the average number of elements per bucket (i.e., load factor). On the contrary, ASF-C4 does not gracefully deal with the outlier buckets even when the load factor is as low as 0.25 on average as shown in Figure 5(b) and significantly decreases the expected hash table capacity supported by ASF in full hardware acceleration.

C. Transactional Programming

Figure 6 shows the speedups of the four design options with up to eight cores. The execution time of each application with multiple cores is normalized to the sequential execution time of the application without transactions. The applications are categorized into three groups according to their scalability.

For the first group that consists of bayes, labyrinth, and yada, ASF-AS scales poorly in comparison to ASF-LC/C8/C4. This is because ASF-AS does not support LOCK MOVs that enable the selective use of speculative accesses to reduce the speculative memory footprints of speculative regions. Table V shows that the ratio of speculative accesses with LOCK MOVs in a speculative region is only 8.25% on average for the STAMP applications. This low ratio clearly indicates that the applications can consume much less ASF hardware resources with ASF-LC/C8/C4. We investigated the sources of the non-speculative accesses in speculative regions and found that many of them were stack accesses to spill the limited number of architectural registers in the AMD64 architecture [15]. The table also shows that 88.94% of transactions fit into the buffer with ASF-AS on average. While this confirms the observation from a previous study that the majority of transactions are likely to be short-lived even for realistic transactional programs [24], 11.06% of the transactions overflowing the cache are long-lived transactions whose memory footprints are much larger than short-lived ones [23]. These transactions hurt overall performance substantially and prevent ASF-AS from scaling. Bayes and yada perform better with ASF-LC and ASF-C8 than ASF-C4 since many capacity overflows due to set-associative conflict misses are eliminated by the higher cache set-associativity of ASF-C8 and the fully associative load/store queues of ASF-LC.

For the second group that consists of intruder, kmeans and SSCA2, all design options perform very similarly since these

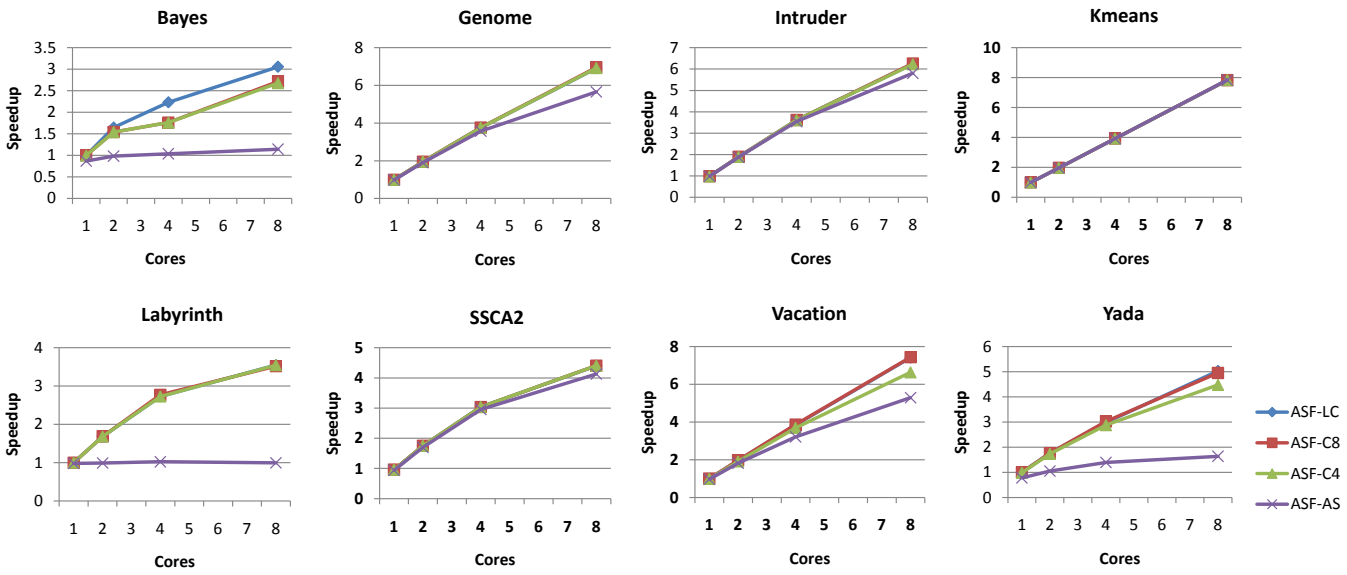

Figure 6. Scalability of the ASF design options. The execution time of each application is normalized to the sequential execution time of the application without transactions.

Table V. The number of transactions, the ratio of transactions running without overflow exceptions, and the ratio of speculative accesses in a speculative region for the STAMP applications.

		bayes	genome	intruder	kmeans	labyrinth	ssca2	vacation	yada	average
Number of Transactions		522	19488	54926	87412	144	93709	4096	13924	
Speculative Access Ratio		0.03%	1.65%	10.62%	5.07%	0.00%	33.33%	14.19%	1.12%	8.25%
Transactions w/o Overflow	ASF-LS	100%	100%	99.99%	100%	100%	100%	94.70%	99.35%	99.25%
	ASF-AS	87.16%	98.76%	99.88%	99.99%	55.56%	99.97%	87.57%	82.63%	88.94%

applications have only short-lived transactions and trigger few capacity overflows as shown in Table V. Therefore, they do not take advantage of the selective use of LOCK MOVs, a high cache set-associativity, or the load/store queues. Genome and vacation belong to the third group and show medium performance differences among the four design options. Vacation scales better with ASF-LC and ASF-C8 due to the reduction of set-associative conflict misses.

Figure 7 shows the execution time breakdown of the STAMP applications running with eight cores (normalized to the execution time of ASF-AS). The results from the second group are not presented since the design options behave all similarly. In each bar, *Halted* is the idle time due to single-threaded code for initialization. *Busy* is for active execution time, *Mem* for the stalled time due to memory accesses, *Virtualization* for the time related to the software overflow mechanism, *Aborted* for the time wasted due to abort, *Overflow* for the execution time of the speculative regions restarted due to capacity overflows, and *Barrier* for the time to synchronize at application barriers. The figure shows that most of the virtualization time and the overflow time are removed by the selective use of LOCK MOVs in ASF-LC/C8/C4 and that they suffer much less from capacity overflows. For bayes, vacation and yada, ASF-LC and ASF-C8 further eliminate these times by removing capacity overflows due to set-associative conflict misses.

VI. Related Work

In addition to the related work discussed in Section II, there are two closely-related proposals to provide HTM supports in commercial processors: SUN's Rock processor [25] and the Azul system [26]. In comparison to the Rock processor, ASF supports the selective use of speculative accesses for efficient ASF hardware resource utilization, offers a minimum capacity guarantee to help programmers use ASF as a standalone atomic primitive with no software support for lock-free programming, and supports TLB misses, near function calls, and branch misprediction recovery without aborting speculative regions. The Azul system uses additional bits per L1 cache line to manage transactional data and does not support the selective use of speculative accesses.

VII. Conclusions

This paper presents Advanced Synchronization Facility (ASF), an AMD64 hardware extension for lock-free data structures and transactional memory. We explain the ASF ISA and the out-of-order hardware design for ASF implementation. The evaluation results show that an ASF-based

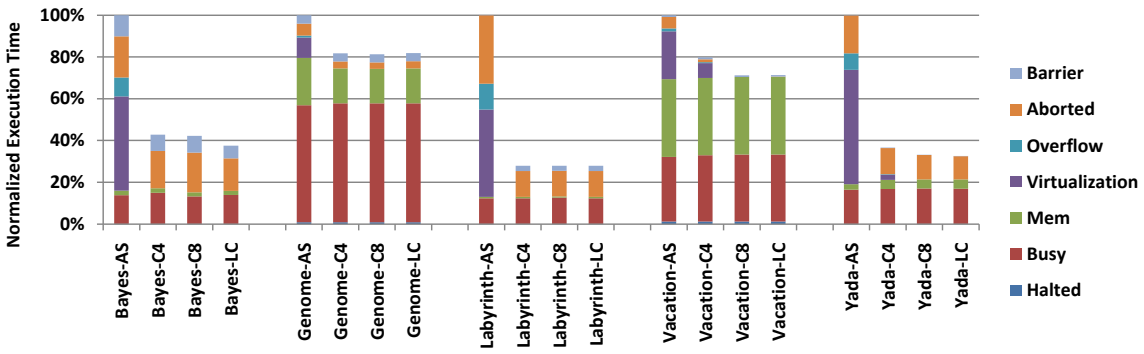

Figure 7. Execution time breakdown of the STAMP applications running with eight cores. It is normalized to the execution time of ASF-AS.

lock-free data structure can perform robustly against random data-sets by using both the LS unit and the L1 cache as speculative buffer, and that the selective use of speculative accesses improves application scalability with limited ASF hardware resources.

References

[1] M. M. Michael, "Scalable lock-free dynamic memory allocation," *SIGPLAN Notices*, 2004.

[2] E. Ladan-mozes and N. Shavit, "An optimistic approach to lock-free fifo queues," in *In Proc. of the 18th Intl. Symp. on Distributed Computing*, 2004.

[3] M. Herlihy, "A methodology for implementing highly concurrent data structures," in *PPOPP '90: Proc. of the 2nd ACM SIGPLAN symp. on Principles & practice of parallel programming*, 1990.

[4] 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*, Jun. 2007.

[5] 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.

[6] L. Hammond, V. Wong *et al.*, "Transactional Memory Coherence and Consistency," in *the Proc. of the 31st Intl. Symp. on Computer Architecture (ISCA)*, Munich, Germany, Jun. 2004.

[7] 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.

[8] L. Yen, J. Bobba *et al.*, "LogTM-SE: Decoupling Hardware Transactional Memory from Caches," in *the Proc. of the 13th Intl. Symp. on High Performance Computer Architecture*, Feb. 2007.

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

[10] 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.

[11] D. Dice, Y. Lev *et al.*, "Simplifying concurrent algorithms by exploiting hardware transactional memory," in *SPAA '10: Proc. of the 22nd ACM Symp. on Parallelism in Algorithms and Architectures*, 2010.

[12] L. Lamport, "Concurrent reading and writing," *Commun. ACM*, vol. 20, no. 11, 1977.

[13] "Package java.util.concurrent," http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/package-su%mmary.html.

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

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

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

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

[18] J. Chung, L. Yen, M. Pohlack, S. Diestelhorst, M. Hohmuth, and D. Christie, "Compilation of Thoughts about AMD Advanced Synchronization Facility and First-Generation Hardware Transactional Memory Support," in *Transact '10*, Paris, France.

[19] 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.

[20] 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*, 2007, pp. 81–91.

[21] "AMD HyperTransport Technology," http://www.amd.com/us/products/technologies/hypertransport-technology/P%ages/hypertransport-technology.aspx.

[22] S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, J. Chung, and L. Yen, "Implementing AMD's Advanced Synchronization Facility in an out-of-order x86 core," in *Transact '10*, Paris, France.

[23] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multiprocessing," in *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, 2008.

[24] J. Chung, H. Chafi *et al.*, "The Common Case Transactional Behavior of Multithreaded Programs," in *the Proc. of the 12th Intl. Conf. on High-Performance Computer Architecture*, Feb. 2006.

[25] 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.

[26] "Azul system," http://www.azulsystems.com.