The Adaptive Transactional Memory Test Platform: A Tool for Experimenting with Transactional Code for Rock *

Mark Moir Kevin Moore Dan Nussbaum
Sun Microsystems Laboratories
{mark.moir,kevin.moore,daniel.nussbaum}@sun.com

Abstract

Sun has recently announced that its forthcoming multicore processor, code-named *Rock*, will support a form of hardware transactional memory. In this paper, we describe our *Adaptive Transactional Memory Test Platform* (*ATMTP*), a simulator that supports this feature and provides a first-order approximation of the success and failure characteristics of transactions on Rock. ATMTP is *not* an accurate model of Rock's implementation or performance, but nonetheless will allow developers to test and tune their code for Rock, before Rock-based systems are available. ATMTP will soon be available as open source so others can use it for this purpose and/or experiment with extending its functionality or adapting it to model variations on Rock's HTM support to guide research on future HTM implementations.

1. Introduction

Sun has recently announced [19, 20] that its forthcoming multicore processor code-named *Rock* will support a form of hardware transactional memory (HTM). In this paper, we describe our *Adaptive Transactional Memory Test Platform* (*ATMTP*), which we have developed to provide a platform for developing and testing code that will exploit Rock's HTM features.

Transactional memory (TM) [7, 16] has received significant attention from academic and industry researchers in recent years as a promising way to ease the development of correct, efficient and scalable concurrent programs. Transactional memory can be used to support explicit transactional programming styles, as well as to improve the performance and scalability of traditional lock-based programs and other synchronization mechanisms.

Transactional memory may be implemented entirely in software, or using special hardware support. Although software transactional memory (STM) can be implemented in existing systems, there is a growing consensus that some form of hardware support is desirable to improve performance. Most proposals for hardware transactional memory are very complex due to their goals of accommodating large transactions and ensuring correct interaction with various difficult events such as exceptions, interrupts, and context switches.

Rock's designers instead chose to support the most common and easy to implement transactions, but to not guarantee to support all transactions regardless of their size and duration. They reserved the right to simply fail transactions that exceed on-chip resources for HTM, or that encounter difficult instructions or situations. This decision to support so-called *best effort* HTM gave the Rock designers sufficient latitude that they could include plans for HTM into their design.

Although choosing to implement best-effort HTM significantly simplifies the task of hardware designers, it also introduces a burden for programmers, due to the difficulty of accurately predicting which transactions will fail and under what circumstances. However, a number of software techniques [4, 8, 9, 14, 18] have been proposed that combine transactions executed using HTM with software mechanisms in such a way that the overall mechanism works correctly regardless of which HTM transactions succeed. Such mechanisms can benefit from the performance and scalability afforded by the HTM support when it succeeds, while transparently hiding the limitations of the best-effort HTM when it does not. This adaptive approach allows improvements in HTM support over time to automatically yield improvements in performance and scalability, and gives hardware designers flexibility in achieving these improvements.

Results of early research suggest that software techniques that use best-effort HTM to boost performance are promising, but work in this area is in its infancy; there are many more opportunities to explore and challenges to face. We want to experiment with such techniques while exploiting features Rock will provide beyond "generic" best-effort HTM, particularly feedback about reasons for transaction failure. At the same time, we want to gain experience with the demands imposed by the need to take into account the specific limitations of Rock's best-effort HTM support. This has motivated us to develop the Adaptive Transactional Memory Test Platform.

ATMTP is *not* intended to accurately model Rock's implementation or performance; see Section 3.2 for a discussion of differences. Furthermore, we do not have final silicon for Rock at this time, and some details may change. Nonetheless, ATMTP correctly models Rock's HTM-related instructions, and fairly accurately reflects most of the circumstances that cause Rock transactions to fail. It thus provides a good platform for experimenting with HTM-based code that will behave similarly on Rock.

We have recently begun using ATMTP to experiment with a number of software techniques for exploiting Rock's HTM features. A companion paper [5], describes our work to date. While only in its early stages, this work has already yielded some encouraging results, and identified some challenges that we will likely face in exploiting Rock's HTM features. Techniques explored in that paper include implementing concurrent data structures with a transactional programming style, optimizing a dynamic software transactional memory system, using multiword synchronization primitives such as double-compare-and-swap (DCAS) in Java concurrency libraries, and using HTM to improve the scalability of lock-based programs, both implicitly and explicitly, and in a variety of contexts.

An important contribution of ATMTP is that it allows us and others to experiment with different restrictions, different resource levels, etc., to help guide future improvements to best-effort HTM implementations. For example, in some cases we have already made the modeling of certain restrictions on transactions optional, so we can explore the impact of such restrictions on the effectiveness of techniques that use the best-effort HTM.

The simulator will be open sourced soon so that others may do so too; see [17] and [21] for details and updates. By making ATMTP available as open source, we hope to enable other researchers to experiment with code that uses best-effort HTM, and to explore variations and extensions to the HTM support itself. Others might create useful extensions that help with debugging or performance tuning, etc.

In this paper, we describe ATMTP. We hope our paper will serve as a starting point for researchers who wish to use ATMTP either to develop new applications for best-effort TM or to evaluate the effectiveness of varying levels of TM support. We also briefly describe the HTM capabilities of Rock and how those capabilities are modeled in ATMTP. Finally, this paper provides an introduction to the implementation of ATMTP and outlines its differences from the Wisconsin GEMS 2.0 simulator on which it is based.

The rest of the paper is organized as follows. Section 2 describes Rock's HTM feature and gives some background and history on Wisconsin GEMS and ATMTP. Section 3 describes the behavior and implementation of ATMTP. Section 4 discusses how ATMTP is intended to be used. We describe our future plans for ATMTP in Section 5, and conclude in Section 6.

Name	Syntax
Checkpoint	`chkpt <fail_pc>`
Commit	`commit`
Read Checkpoint Status	`rd %cps, <dest_reg>`

Table 1. HTM instructions in ATMTP.

2. Background

ATMTP leverages the infrastructure of the Wisconsin GEMS simulator [11, 21], adapting its behavior to more closely model Rock's HTM features [19, 20]. This section presents an overview of Rock's HTM and describes the history of the GEMS and ATMTP simulators.

2.1 Rock

Sun's forthcoming multi-threaded multi-core Rock processor [20] uses aggressive speculation to provide high single-thread performance in addition to high system throughput. For example, on load misses, Rock runs ahead speculatively to increase memory-level parallelism by issuing subsequent memory requests early. The aggressive speculation is enabled by a checkpoint architecture: Before speculating, Rock checkpoints the architectural state of the processor. If the speculation turns out to have taken a wrong path, or if exceptions or other uncommon events occur during speculation, the hardware reverts back to the checkpoint and re-executes from there, perhaps in a more conservative mode.

One novel feature of Rock is the inclusion of a form of *best-effort* HTM. Rock's HTM allows software to define critical sections of code that behave as transactions: each either completes as an atomic group of operations such that all memory instructions (loads and stores) can be serialized consecutively in some global memory order, or fails, in which case it is as if none of the operations were even attempted. Rock supports HTM with two new instructions, `chkpt` and `commit`, and a new *checkpoint status* (`cps`) register (see Table 1). A transaction is started by a `chkpt` instruction, and is terminated by either a `commit` instruction or the failure of the transaction. If a transaction fails, some indication of the cause of failure is stored in the `cps` register, and control is transferred to the PC-relative offset (`fail_pc`) specified by the `chkpt` instruction.

A transaction may fail because of conflict with another transaction, an exception or asynchronous interrupt, or an attempt to execute an instruction that is not permitted within transactions, among other reasons. Because Rock promises only *best-effort* HTM, it is also free to fail transactions that exceed some limit on an on-chip resource such as space in a cache set or in the store buffer, or when some difficult-to-handle event occurs, such as a TLB miss. Furthermore, unlike some HTM systems proposed in the research literature [1, 6, 12, 13, 15], Rock leaves contention management almost entirely to software, implementing only a simple "re-

quester wins" policy [2]: an incoming memory request that conflicts with local transactional state is serviced immediately and the local transaction fails.

The code at `fail_pc`, executed upon transaction failure, may be arbitrary. It might, for example, simply retry the transaction, or it may read the `cps` register (using the `rd %cps` instruction) to decide whether to retry, to coordinate between competing threads using a software-implemented contention management policy, to fall back to a software transaction (as in HyTM [4]), or perhaps to abandon the operation entirely. Because a transaction in Rock may fail repeatedly for variety of reasons, in general it is important for software to provide some alternative to retrying forever.

Rock's HTM exploits the same mechanisms used to implement Rock's other forms of speculative execution. When a `chkpt` instruction is executed, the processor saves a checkpoint of the hardware register state to a set of shadow registers and enters speculative execution. In this mode, only the working register file is updated. Rock maintains the isolation and atomicity of transactions by buffering all stores in the store queue, keeping them invisible from the outside until the transaction commits, and by using the per-thread, per-L1-cache-line S-bits (speculative bits) to track locations read by transactions. These bits are cleared and the transactional stores buffered in the store queue are discarded if the transaction fails.

The choice to provide best-effort HTM functionality greatly simplified the task of providing TM support for Rock. Having the option to simplify the hardware in this way was very important; in fact, it seems unlikely that Rock would have even had any HTM support at all if this a simplification had not been an option. The downside of handling difficult situations in software, however, is that it significantly increases the difficulty of the task facing Rock's programmers. Most programmers prefer not to have to deal with the details of hardware limitations. Library and compiler support can be useful in this regard [4]. ATMTP provides a sorely needed platform upon which we can develop appropriate compiler and library technology to assist with this task, before actual Rock hardware becomes available.

2.2 GEMS and ATMTP history

ATMTP is based on the Wisconsin GEMS simulator [21], which in turn is based on the Simics [10] functional simulator for SPARC®. GEMS consists of two primary components, Ruby, a configurable memory system simulator, and Opal, a configurable out-of-order multithreaded processor model.

GEMS runs primarily in one of two modes. The first mode uses Ruby only, in which case, processors execute one non-memory instruction per (simulated) cycle. In the second mode, Opal simulates an out-of-order processor, using Ruby to determine memory access latencies. ATMTP is compatible with GEMS running Ruby *only*. Although Rock has a more sophisticated execution pipeline than the simple blocking model supplied by Simics, it is sufficiently different from the one modeled by Opal that the additional effort of integrating with Opal would not be worthwhile, given that an accurate performance model is not our goal.

ATMTP has evolved in parallel with GEMS. For previous work on HyTM [4] and PhTM [8], we modified GEMS's LogTM subsystem [12] to create a "generic" best-effort HTM simulator that supports Rock's `chkpt` and `commit` instructions. We first started this work on GEMS (/Ruby/LogTM) version 1.2, which was the latest version at the time. We modified LogTM to impose size and geometry limitations on transactions, and to transfer control to a fail address upon abort, rather than retrying as LogTM does.

In modifying LogTM to create our best-effort HTM simulator, we were careful to keep our changes as simple as possible. For example, when Simics encounters a `chkpt` instruction, we use the Simics instruction handling facility to get control, and then pass control directly to LogTM's begin-transaction routine, thus often treating LogTM as a "black box" in some sense. We also tried to make it obvious when looking at the code what those changes were, our intent being to ease porting as GEMS/Ruby/LogTM evolved. This software engineering strategy indeed greatly facilitated several upgrades to GEMS that we have imported since we first started this work, and ultimately the upgrade to GEMS 2.0 described below.

Our generic best-effort simulator was useful for developing and evaluating our ideas for using best-effort HTM, but it did not closely model various limitations that Rock imposes, and also did not model its `cps` register, a crucial feature for allowing software to make intelligent decisions about whether and when to retry an aborted transaction.

In recent months we have turned our attention to developing and evaluating code that will run on Rock, and thus we needed a simulator that more closely models Rock's limitations and features beyond those of the generic best-effort HTM supported by the previous simulator. This led us to further modify our simulator, resulting in ATMTP. Until recently, ATMTP was based on GEMS/Ruby/LogTM version 1.4, and we had to modify the simulator to make it more accurately reflect Rock's behavior. For example, LogTM uses *eager version management* [2]: transactional stores directly modify the affected memory locations, and these are rolled back in software upon abort, while Rock can abort transactions quickly in hardware. For another example, we had to modify LogTM's coherence protocol to achieve a "requester wins" conflict resolution policy like the one Rock uses.

Meanwhile, the Multifacet group at the University of Wisconsin released GEMS 2.0 [21], which includes configuration options to model a variety of HTM implementations, according to their version management and conflict resolution policies. We have recently ported ATMTP to GEMS 2.0, allowing us to choose a configuration that models lazy version management and eager conflict detection

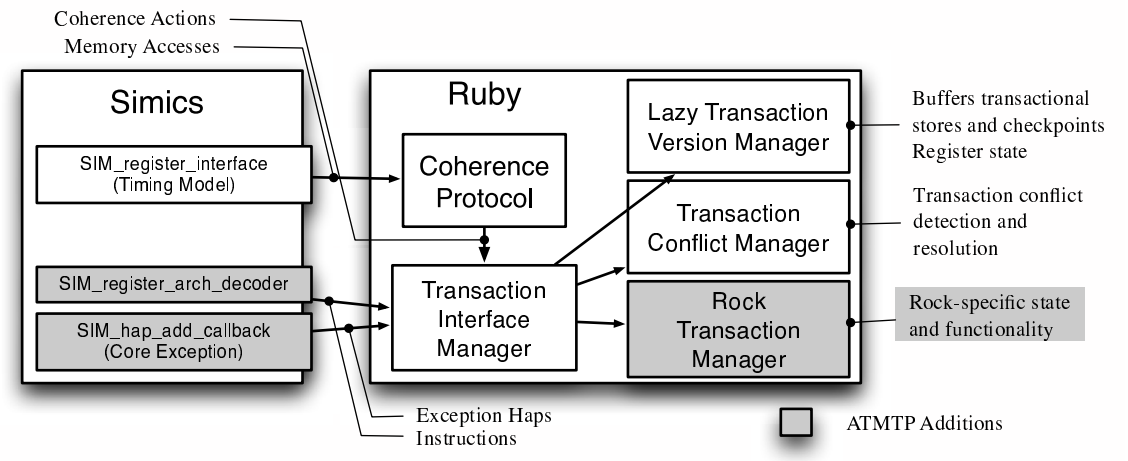

Figure 1. ATMTP organization.

(requester wins). Thus we were able to throw away some of our more complicated modifications that were needed to make the version based on GEMS/Ruby/LogTM 1.4 behave more like Rock. Furthermore, GEMS 2.0 introduced CMP support, whereas GEMS 1.4 only modeled an SMP system. This change again makes ATMTP closer to Rock, though it does not model multiple threads per core.

With this history in mind, in the next section we describe the version of ATMTP that is based on GEMS 2.0; we plan to open source this version soon and give it to the Multifacet group at Wisconsin so that they can integrate it into the GEMS 2.1 release.

3. The Adaptive Transactional Memory Test Platform

ATMTP is based on the open-source LogTM-SE [22] simulator developed by the Multifacet group at the University of Wisconsin [21], which is included in version 2.0 of the GEMS simulation toolkit. We set the GEMS options `XACT_EAGER_CD=true` and `XACT_LAZY_VM=true`. Thus, the base system configuration provides an HTM implementation based on eager conflict detection and lazy version management; this is the system referred to as *EL* in [2]. By default, LogTM-SE uses perfect read and write signatures [3, 22] for conflict detection; we have not changed this option. Furthermore, the LogTM-SE simulator is based on the `MESI_CMP_filter_directory` coherence protocol, which models a single-chip CMP system.

Although the above-mentioned GEMS 2.0 options allow us to start with a base simulator that is closer to Rock's implementation than LogTM, on which previous versions of our simulator were based, we emphasize that ATMTP is *not* intended to make precise performance predictions about Rock. In particular, ATMTP makes no attempt to model Rock's pipeline, multithreaded cores, or network topology, and may not precisely model all conditions under which Rock aborts transactions. Furthermore, we have made no attempt to tune various configurable parameters of GEMS, such as memory latency, to accurately model Rock.

Despite these and other differences (see Section 3.2 for a more detailed comparison of Rock and ATMTP), to the extent reasonable we have tried hard to accurately reflect the instructions and events that will cause Rock transactions to abort. We therefore consider ATMTP to be a valuable platform for developing and tuning HTM-based code for Rock despite these differences.

The main pieces of the ATMTP system interact as follows (see Figure 1):

- ATMTP is integrated into the Ruby timing model. Ruby connects to Simics's timing model interface, via which Simics informs Ruby of all memory accesses. Ruby, which simulates caches and an interconnection network, determines the length of time the access would take and stalls Simics for that number of cycles.
- For instructions whose behavior ATMTP needs to specify or modify, we use the `SIM_register_arch_decoder` API that Simics provides to "wire in" an instruction handler to such instructions. Using this mechanism, we arrange that when the simulated processor attempts to execute a `chkpt` or `commit` instruction, ATMTP passes the request on to the underlying LogTM-SE functions that begin or commit transactions. We also use this mechanism to make a transaction fail if it attempts to execute certain instructions, as described in Section 3.1.2.
- Whenever a simulated instruction would generate an exception inside an active transaction, ATMTP instead fails the transaction. To achieve this behavior, ATMTP makes use of Simics's exception handling facilities by adding a callback (using `SIM_hap_add_callback`) that is activated whenever the simulated processor executes an ex-

Mask	Name	Description
0x002	COH	**Coherence** - Conflicting memory operation.
0x004	TCC	**Trap Instruction** - Executed a conditional (tcc) or non-conditional (ta) trap instruction.
0x008	INST	**Unsupported Instruction** - Executed an instruction not supported inside transactions.
0x010	PREC	**Precise Exception** - Execution generated a precise exception.
0x020	ASYNC	**Async** - Received an asynchronous interrupt.
0x040	SIZ	**Size** - Transaction write set exceeded the size of the store queue.
0x080	LD	**Load** - Line in read set evicted or Data TLB miss on a load.
0x100	ST	**Store** - Data TLB miss on a store.

Table 2. cps register: bit definitions.

ception. This enables ATMTP to intercept an exception and abort the transaction rather than processing the exception.

- The LogTM-SE simulator (configured as *EL*, described above) consists mainly of two components: the memory coherence protocol (MESI_CMP_filter_directory) and the TransactionInterfaceManager (Eager-Lazy TM in our case). The latter coordinates between the objects that implement the TM capabilities:
  - TransactionConflictManager,
  - TransactionIsolationManager (not shown), and
  - LazyTransactionVersionManager.

  ATMTP modifies the TransactionConflictManager and LazyTransactionVersionManager objects and introduces a new RockTransactionManager object. RockTransactionManager maintains Rock-specific transactional meta-state, including the fail_pc and the *checkpoint status* (cps) register, as described in more detail below.

3.1 ATMTP Components

In this section, we describe the changes we made to the GEMS 2.0 code base to make the simulated system more closely model Rock's behavior.

3.1.1 Rock's HTM Instructions

ATMTP's most important task is to model Rock's HTM instructions (see Table 1). The chkpt instruction, which starts a hardware transaction, takes a pc-relative fail_pc argument. This argument specifies a location to which execution jumps if the transaction fails. ATMTP implements this functionality by adding an instruction handler to Simics, which passes control directly to the LogTM-SE transaction-begin routine upon execution of the instruction.

Unlike LogTM-SE, upon transaction failure, Rock resumes execution at the fail_pc specified in the chkpt instruction. The chkpt instruction handler saves the fail_pc in the per-core RockTransactionManager object, which maintains transactional meta-state including the fail_pc. When a transaction fails, ATMTP sets the pc to the saved fail_pc, in addition to restoring the architectural state, as LogTM-SE does.

Transaction commit has the same functional behavior in ATMTP and in the LogTM-SE simulator. The commit instruction of ATMTP again uses a Simics instruction handler to wire into the commit functionality of LogTM-SE.

ATMTP adds support for the *checkpoint status* (cps) register, which is Rock's transaction-failure feedback mechanism. Table 2 specifies the meanings of the bits in ATMTP's cps register. ATMTP keeps the cps register as a field in the RockTransactionManager object. Whenever an event that will cause a transaction to fail occurs, ATMTP sets the appropriate bits in that field for later retrieval by the simulated program. A handler for the rd %cps instruction copies the value of the simulated cps register into the instruction's destination register.

3.1.2 Transaction Failures

Consistent with its *best-effort* design philosophy, Rock fails transactions in various cases that the underlying simulator upon which ATMTP is based does not. ATMTP adapts the underlying simulator to reflect these differences. A list of conditions apart from actual transaction conflicts that cause transactions to fail follows.

Resource Limitations. ATMTP models Rock's read and write set limitations by failing a transaction when its footprint exceeds hardware capabilities. Specifically, ATMTP fails a transaction if a line in the transaction's read set is evicted from the L1 cache or if the transaction executes more than the maximum number of stores allowed in a transaction (see the description of ATMTP_XACT_MAX_STORES in Section 3.3). The first of these limitations is implemented by modifying the MESI_CMP_filter_directory protocol specification to notify the TransactionInterfaceManager when a line that has been accessed by the transaction is evicted from the L1 cache. Upon receiving such notification, the TransactionInterfaceManager aborts the transaction. The second is achieved by adding a store count to LazyTransactionVersionManager.

Synchronous Exceptions. Sensible meanings for interactions between transactions and the operating system are difficult to define, even for the purpose of writing papers. Rock therefore does not attempt to do so, instead failing any transaction that attempts such an interaction.

To model this choice, ATMTP fails a transaction any time the processor triggers a precise exception, caused by an event such as a software trap (e.g., for making a system call), an MMU miss or a divide-by-zero error. Making this happen is complicated by the fact that, by the time we discover that an instruction causes a precise exception, the simulator is already executing that instruction. Rolling back from that point would require complicated instruction-specific code in a number of places in the system.

To avoid this, we instead use the following trick: using Simics's *hap* mechanism, we register a handler that runs any time an exception occurs, and another handler that runs any time a return-from-exception occurs. The exception handler makes a note of the fact that we've taken an exception while running a transaction in the `RockTransactionManager`, and arranges to set the pc to point to a `retry` instruction at a known address, so that the `retry` will be the next instruction to execute.

That `retry` instruction therefore immediately follows the trapping instruction, performing all of the necessary trap-unwinding actions and triggering a return-from-exception event. The return-from-exception handler causes the transaction to abort, restoring all registers and setting the `pc/npc` to `failPC/failPC+4`, thus branching to the fail address.

The result is that the simulation behaves as if the exception never happened, and the transaction is aborted. This mechanism saves us from having to install complex and specialized instruction handlers for all instructions that might generate precise traps.

Asynchronous Interrupts. ATMTP detects when asynchronous interrupts occur within transactions. However, we have not yet implemented the mechanism to fail a transaction and then take the interrupt. Instead, ATMTP triggers a simulator assertion. We consider this preferable to the undefined behavior that a simulation would exhibit if we were to allow the interrupt to be taken. We currently avoid the situation entirely by turning interrupts off during transactional execution. However, because Rock would fail the transaction and then take the interrupt, we intend to make ATMTP do so as well. We plan to implement better modeling of asynchronous interrupts in the near future, possibly before releasing ATMTP.

Function Calls. Compilers for SPARC® processors often emit `save` and `restore` instructions in order to use *register windows* to avoid the overhead of saving and restoring registers to be used by the callee. In Rock, some combinations of `save` and `restore` instructions may fail a transaction. A consequence of this limitation is that many functions cannot be usefully called within a transaction.

In ATMTP, we approximate this behavior by failing a transaction when a `save` instruction is followed by a `restore` instruction in the same transaction. This is achieved by recording when a `save` instruction is executed in the `RockTransactionManager` object and modifying the Simics instruction handler for `restore` to fail the transaction if a `save` instruction has previously been executed in the transaction. This choice may not precisely model the conditions under which Rock will fail transactions due to `save` and `restore` instructions. However, we believe it captures two properties that are important for ATMTP to be useful for determining which techniques will and will not be successful on Rock:

- nested or repeated function calls within a transaction are likely to cause it to fail, but;
- transactions that return from a function at the beginning and call a function at the end—but otherwise call no functions—can succeed. We have found this to be a useful idiom for hiding details of using Rock's HTM feature in a library, for example as discussed in Section 4.1.

Other "Difficult" Instructions. To model Rock's limitations on the instructions that can be executed in transactions, ATMTP causes a transaction to fail if it attempts to execute any of the following instructions:

`membar`
`cas/casx/ldstub`
`wrpr/wrstr`
`flush`
`done/retry`
`udiv/sdiv` (and similar)
`chkpt`

ATMTP achieves this behavior using Simics instruction handlers.

3.2 Comparing Rock and ATMTP

ATMTP uses the simple single-issue in-order processor model provided by Simics. In contrast, Rock is multi-issue and highly chip-multithreaded, including both multiple hardware threads per core and multiple cores per chip. While Rock supports multi-chip systems, ATMTP only models single-chip systems. In addition, Rock uses novel speculation mechanisms to boost single-thread performance, which are not modeled by ATMTP.

ATMTP's cache configurations are quite different from Rock's. In ATMTP, each processor has only a single strand and a private single-level cache. In Rock, the multiple strands on each core share the same L1 cache [20], which increases the likelihood that a transaction fails due to eviction of lines that hold a part of its read set. ATMTP does not model this effect. In our use of ATMTP so far [5], we

have made no attempt to more closely model Rock's cache organization or to tune these parameters to more accurately model Rock. Instead, we have used the GEMS 2.0 cache organization and kept the default GEMS parameters for their sizes and geometries, as well as memory latency (32KB, 4-way private L1 caches; 8MB, 8-way shared L2 cache; and 450-cycle memory latency).

The only GEMS option for which we have not used the default is the network topology, for which we have chosen `g_NETWORK_TOPOLOGY=PT_TO_PT`, modeling a point-to-point network. We made this choice out of convenience, because it saved us from having to create specification files for each simulated machine size.

It should be clear from the discussion above that the timing characteristics of Rock and ATMTP are substantially different; one should be circumspect about drawing conclusions about the performance of Rock based on behavior exhibited by runs under ATMTP.

Rock uses state associated with caches to detect conflicts between transactions, while ATMTP relies on the perfect signatures used by the underlying simulator [3, 22]. These apparently different mechanisms make the same decisions about which transactions to abort because i) perfect signatures do not result in false positives, and ii) transactions are aborted on eviction of L1 cache lines that have been accessed during the transaction, so the difference between the information stored by Rock's caches and the simulator's perfect signatures after such an eviction is not relevant.

Although precise exceptions due to TLB misses cause transactions to fail in both ATMTP and in Rock, their respective TLB configurations differ significantly. The frequency of transaction failures due to TLB misses will therefore likely be quite different on the simulator than on the real hardware. Despite this difference, ATMTP provides a valuable platform for studying the effect of traps due to TLB misses on transactions because it accurately reflects behavior due to cold misses, one of the primary concerns surrounding TLB misses and transactions. In both ATMTP and Rock, if a TLB miss causes a precise exception, thereby failing a transaction, the TLB fill needed to satisfy the faulting memory access does not occur because the transaction is aborted. Thus, the transaction will likely fail again if blindly retried.

To avoid this effect, the software could access data pages to be accessed in a transaction, in order to attempt to get appropriate mappings added to the DTLB, before retrying it. Somewhat less intuitively, TM software may similarly need to warm up the ITLB with mappings for all code pages to be accessed during a transaction. On ATMTP, this requires nontransactionally executing an instruction from each page. While low-level details of Rock and ATMTP are quite different, and we expect these issues to be somewhat easier to deal with on Rock than on ATMTP, ATMTP still provides valuable information as to the circumstances under which these issues arise.

3.3 Configuration Options

ATMTP adds five new configuration options to GEMS:

`ATMTP_ENABLED` (default: `false`) turns ATMTP on.

`ATMTP_ABORT_ON_NON_XACT_INST` (default: `false`) switches the behavior of ATMTP when an unsupported instruction is attempted inside a transaction. When that happens and this parameter is `true`, ATMTP fails the transaction and continues simulation. If it is `false`, ATMTP triggers a simulator assertion, halting the simulation and notifying the user immediately, which can help the user to identify accidental use of proscribed instructions.

This option's meaning is different for the `chkpt` instruction. If the option is `true`, then when ATMTP encounters a `chkpt` instruction within a transaction, the transaction fails, as it would on Rock. If it is `false`, however, ATMTPś behavior changes to support flat nesting. We have found this option useful for investigating the value of such a feature.

`ATMTP_ALLOW_SAVE_RESTORE_IN_XACT` (default: `false`) permits users to experiment with HTM support that does and does not include support for `save`/`restore` instruction pairs.

`ATMTP_XACT_MAX_STORES` (default: `32`) sets the maximum number of stores allowed before a transaction fails due to exhausting the capacity of hardware buffers.

`ATMTP_DEBUG_LEVEL` (default: `0`) allows the user to control which ATMTP-related events are reported in output sent to the simulator console. Like the `XACT_DEBUG_LEVEL` option in GEMS, this option controls reporting of events that are useful to users, as well as events that are useful to developers of the simulator.

The following lists the ATMTP events that are emitted at each level.

Level `0` disables reporting entirely.

Level `1` enables "failure" reporting, producing output for the events that can cause transaction aborts, such as store buffer overflow, and for `rd %cps` instructions.

Level `2` enables *transaction bracketing* by adding `chkpt` and `commit` instructions to the events reported at Level `1`.

Level `3` enables additional reporting for all transactional loads and stores.

4. Using ATMTP

4.1 Writing TM Programs

Rock's TM is controlled by two instructions, `chkpt` and `commit`. All code executed between those instructions is considered part of a transaction. The (inline) assembly func-

```
    !
    ! p is in %o0; value returned in %o0.
    !
    .inline try_HTM_increment
    .volatile
    sub    %g0, 1, %o1     ! %o1 <== 0xffffffff
    chkpt  1f
    ld     [%o0], %o1      ! %o1 <== *p
    add    1, %o1, %o2     ! %o2 <== *p + 1
    st     %o2, [%o0]      ! *p  <== *p + 1
    commit

1:  mov    %o1, %o0        ! Return value in %o0.
    .nonvolatile
    .end
```

Figure 2. Using a transaction directly to attempt to increment a counter.

```
!
!inline uint32_t begin_transaction();
!
    .inline begin_transaction
    .volatile
    chkpt  1f
    ba     2f              ! Return "success".
    mov    1, %o0          ! in delay slot.
1:  clr    %o0             ! Return "failure".
2:  .nonvolatile
    .end
!
!inline void commit_transaction();
!
    .inline commit_transaction
    .volatile
    commit
    .nonvolatile
    .end
```

Figure 3. Simple library routines to hide HTM details from programmer.

tion shown in Figure 2 uses a transaction to attempt to atomically increment a `uint32_t` counter pointed to by `p`.

`try_HTM_increment()` initializes `%o1` with the "transaction failed" value `0xffffffff`. It then starts a hardware transaction, which reads the counter into `%o1`, increments it into `%o2` and stores the result back into the counter. If the transaction succeeds, the value that the counter contained before the increment operation was performed is returned. Otherwise, the counter is not modified and the "transaction failed" value `0xffffffff` is returned.

Coding even simple examples such as the increment routine shown above can be tedious, so it is desirable to expose the HTM mechanism to users of higher-level languages via an easy-to-use idiom. The inline assembly functions in Figure 3 show how such an idiom might be implemented.

```
// Attempt to atomically increment a counter
// returning the counter's value if the attempt
// succeeds and 0xffffffff if it fails.
//
inline uint32_t
try_HTM_increment(uint32_t *p)
{
  uint32_t ret = 0xffffffff;
  if (begin_transaction()) {
    ret = (*p)++;
    commit_transaction();
  }
  // If the transaction fails, control gets here
  // without ret being modified.
  return ret;
}
```

Figure 4. Example using library routines for increment operation.

The `begin_transaction` inline-assembly function includes both the `chkpt` instruction and the transaction fail path code. When first run, `begin_transaction` immediately returns `1`. If the transaction fails, all of its speculative updates are discarded and execution resumes at the `fail_pc` (label `1`). In that case, `begin_transaction` returns a second time, with a return value of `0`, denoting that the transaction has failed. The `commit_transaction` function consists simply of a single `commit` instruction. These functions can be used as illustrated by the C-language function `try_HTM_increment()` shown in Figure 4, which works identically to the initial example given in Figure 2.

To help users to create code that uses Rock's instructions, we plan to include example inline assembly functions like those described above in our open source release.

Several other ways that programs might take advantage of Rock's HTM are described in a companion paper [5].

4.2 Running ATMTP

ATMTP is used in a manner similar to other GEMS-based simulators. GEMS, driven by Simics, is a full-system simulator that behaves to the first order like a real machine. The target (simulated machine) runs a complete operating system, such as Solaris™. The simulated machine can be controlled through a console window similar to a standard unix terminal.

Simulation can be started in a fast mode without any timing simulators loaded. In this mode (using Simics only), simulations run relatively quickly, proceeding somewhere between one and two orders of magnitude slower than native speed. Users can copy files onto the simulated machine and run benchmark warmup in this fast mode. When the benchmark reaches steady state (or whichever phase is most interesting to the user), the user can load the appropriate GEMS module (in our case, ATMTP) and begin to simulate the target machine in detail. Because the Rock HTM instructions

Name	Description
xact_log_overflows	store-buffer overflow
xact_cache_overflows	L1 cache spill
xact_unsup_inst_aborts	attempt to execute "difficult" instruction
xact_save_rest_aborts	attempt to execute restore after save

Table 3. New Statistics Collected by ATMTP.

are not implemented by Simics, ATMTP must be loaded before this feature can be used.

4.3 Using results from ATMTP

In [5], we have presented experimental results as the number of operations per second completed on a shared data structure during a fixed time interval. This is useful for broad qualitative comparisons of different techniques, but because ATMTP does not model Rock's implementation (see Section 3.2), such measurements do not provide accurate performance predictions for Rock.

However, ATMTP produces a host of statistics that are useful for understanding how effective a particular use of Rock's HTM feature is. For example, the LogTM-SE simulator summarizes the number of transaction aborts by program counter and by conflict address. Table 3 describes several statistics which count non-contention-related events that cause transactions to abort in ATMTP, and so are germane only to the best-effort model that ATMTP supports. These and other statistics provide valuable tools for understanding the behavior of a transactional program, and in particular identifying which uses of HTM are ineffective and determining how to modify them to make them effective.

Also, ATMTP adds a new ATMTP_DEBUG_LEVEL option. This option can be set to enable the production of various levels of reporting about events in a transactional program's execution. See Section 3.3 for details about this option.

Because ATMTP provides such visibility into transactional execution, we expect it to be a valuable tool even after Rock is available.

5. Future Work

ATMTP has already been useful to us as a development platform for applications that use Rock's HTM. Thus far we have demonstrated the effectiveness of HyTM and PhTM on Rock, tested transaction-based lock elision for Java and for a C++ STL class, and tested an NCAS (n-location compare-and-swap) mechanism based on Rock transactions [5]. We plan to build on this preliminary work by considering more applications and more techniques for exploiting HTM.

As more details about Rock are released, we plan to continue to refine ATMTP to more closely model the actual hardware. In particular, we hope eventually to be able to model Rock's TLB architecture in ATMTP. Although these changes will still leave a significant gap betwen the hardware and the simulator in terms of performance modeling, they will give application developers a clearer picture of which transactions will succeed and fail on the hardware.

In addition to making ATMTP model Rock more closely, we may also extend the functionality of the simulator, for example to enhance its usefulness for diagnosing performance problems and the reasons for transaction failure. We also plan to extend the simulator to allow exploration of potential future HTM implementations.

We will soon release ATMTP as open source under a GPLv2 license as part of the Wisconsin GEMS 2.1 release; please check our website [17] and the GEMS website [21] for details and updates. We will thus enable researchers and developers to experiment with code that exploits Rock's HTM feature before Rock is available, and also to use it to better understand the behavior of transactional code even after Rock is available. And by making ATMTP available as open source, we enable other TM researchers to experiment with and share their ideas about possible future enhancements to Rock's HTM support, enhanced functionality in the simulator, etc.

6. Concluding Remarks

Sun's forthcoming multi-threaded multi-core processor, code-named Rock, will provide a form of best-effort hardware transactional memory (HTM) that promises to be a major step forward for transactional memory in research and in practice. This paper introduces the Adaptive Transactional Memory Test Platform (ATMTP), a simulator that supports Rock's HTM feature and provides a good approximation of the circumstances in which Rock transactions will succeed.

In the time before Rock is available, ATMTP will enable developers and researchers to develop and test code that uses Rock's HTM feature, and to gain insight into the benefits that can be achieved using Rock's transactional memory feature, as well as the challenges that must be overcome in doing so. We hope that ATMTP will also accelerate research into Hardware/Software hybrid TM systems and other applications of best-effort transactional memory.

ATMTP also provides greater visibility into the causes of transaction failure than Rock does. Thus, it will continue to be a valuable tool after Rock is available, both for diagnosing specific performance problems, as well as gaining insight into which causes are responsible for most transaction failures, thus helping software developers use Rock's HTM

more effectively. Finally, by open sourcing ATMTP, we hope to encourage and facilitate research into improved HTM implementations, as well as improved simulation tools.

Acknowledgments

We thank the Multifacet Group at the University of Wisconsin for their support using the GEMS and LogTM simulators and for agreeing to distribute the ATMTP with GEMS. We also thank Shailender Chaudhry, Bob Cypher, Martin Karlsson, and Marc Tremblay for many useful conversations about Rock's design and behavior. We would especially like to thank Victor Luchangco for his invaluable discussions and editing services. Finally, we are grateful to the anonymous referees for valuable feedback.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proc. 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Feb. 2005.

[2] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. *SIGARCH Comput. Archit. News*, 35(2):81–91, 2007.

[3] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[4] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proc. 12th Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[5] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum. Applications of the adaptive transactional memory test platform. Transact 2008 workshop. http://research.sun.com/scalable/pubs/TRANSACT2008-ATMTP-Apps.pdf.

[6] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proc. 31st Annual International Symposium on Computer Architecture*, June 2004.

[7] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[8] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *Workshop on Transactional Computing (Transact)*, 2007. http://research.sun.com/scalable/pubs/TRANSACT2007-PhTM.pdf.

[9] S. Lie. Hardware support for unbounded transactional memory. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, May 2004.

[10] P. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenstrom, and B. Werner. SimICS/sun4m: A virtual workstation. In *Proceedings of the USENIX 1998 Annual Technical Conference (USENIX '98)*, June 1998.

[11] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[12] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proc. 12th Annual International Symposium on High Performance Computer Architecture*, 2006.

[13] K. E. Moore, M. D. Hill, and D. A. Wood. Thread-level transactional memory. *Technical Report: CS-TR-2005-1524, Dept. of Computer Sciences, University of Wisconsin*, Mar. 2005.

[14] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware atomicity for reliable software speculation. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 174–185, New York, NY, USA, 2007. ACM.

[15] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proc. 32nd Annual International Symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, 2005.

[16] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, Special Issue(10):99–116, 1997.

[17] Sun Microsystems Laboratories. Scalable Synchronization Research Group. `http://research.sun.com/scalable`.

[18] F. Tabba, C. Wang, J. R. Goodman, and M. Moir. NZTM: Nonblocking zero-indirection transactional memory. In *Workshop on Transactional Computing (Transact)*, 2007. http://research.sun.com/scalable/pubs/TRANSACT2007-NZTM.pdf.

[19] M. Tremblay. Transactional memory for a modern microprocessor. Keynote speech at 26th Annual ACM Symposium on Principles of Distributed Computing, Aug. 2007.

[20] M. Tremblay and S. Chaudhry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread CMT SPARC® processor. In *IEEE International Solid-State Circuits Conference*, Feb. 2008.

[21] Wisconsin Multifacet Project. Multifacet GEMS: General Execution-driven Multiprocessor Simulator. `http://www.cs.wisc.edu/gems`.

[22] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 261–272, Washington, DC, USA, 2007. IEEE Computer Society.