VMM Emulation of Intel Hardware Transactional Memory

Maciej Swiech   Kyle C. Hale   Peter Dinda
Department of Electrical Engineering and Computer Science
Northwestern University
{m-swiech,k-hale,pdinda}@northwestern.edu

ABSTRACT

We describe the design, implementation, and evaluation of emulated hardware transactional memory, specifically the Intel Haswell Restricted Transactional Memory (RTM) architectural extensions for x86/64, within a virtual machine monitor (VMM). Our system allows users to investigate RTM on hardware that does not provide it, debug their RTM-based transactional software, and stress test it on diverse emulated hardware configurations, including potential future configurations that might support arbitrary length transactions. Initial performance results suggest that we are able to accomplish this approximately 60 times faster than under a full emulator. A noteworthy aspect of our system is a novel page-flipping technique that allows us to completely avoid instruction emulation, and to limit instruction decoding to only that necessary to determine instruction length. This makes it possible to implement RTM emulation, and potentially other techniques, far more compactly than would otherwise be possible. We have implemented our system in the context of the Palacios VMM.Our techniques are not specific to Palacios, and could be implemented in other VMMs.

1. INTRODUCTION

Hardware transactional memory (HTM) [4] is an enduring concept that holds considerable promise for improving the correctness and performance of concurrent programs on hardware multiprocessors. Today's typical server platforms are already small scale NUMA machines. A mid-range server may have as many as 64 hardware threads spread over 4 sockets. Further, it is widely accepted that the growth of single node performance depends on increased concurrency within the node. For example, the U.S. national exascale efforts are crystallizing around a model of billion-way parallelism [1], of which a factor of 1000 or more is anticipated to be within a single node [12]. Given these trends, correct and efficient concurrency within a single node or server is of overarching importance, not just to systems software, but also to libraries and applications.

ROSS '14, June 10, 2014, Munich, Germany
Copyright 2014 ACM 978-1-4503-2950-7/14/06 ...$15.00.
http://dx.doi.org/10.1145/2612262.2612265

HTM promises better correctness in concurrent code by replacing locking with transactions over instructions. Unlike locks, such transactions are composable, meaning that it is less likely to introduce deadlock and livelock bugs as a codebase expands. Furthermore, transactions have the potential for running faster than locks because the hardware is able to detect violations of transaction independence alongside of maintaining coherence.

Intel has made HTM a component of its Haswell platform, and chips with the first implementation of this feature are now widely available. This paper focuses on the restricted transactional memory (RTM) component of Intel's specification. RTM is a bit of a misnomer—it might better be called *explicit* transactional memory. With RTM, the programmer starts, aborts, and completes transactions using new instructions added to the ISA. Our work does not address the other component of Intel's specification, hardware lock elision (HLE), which is a mechanism for promoting some forms of existing lock-based code to transactions automatically—i.e., it is *implicit* transactional memory.

Our paper focuses on how to extend a virtual machine monitor (VMM) so that it can provide the guest with Intel Haswell RTM capability even if the underlying hardware does not support it. Furthermore, the limitations of this emulated capability can differ from that of the underlying hardware.

There are three primary use cases. The first is in testing RTM code against different hardware models, to attempt to make the code resilient to different and changing hardware. As we describe in more detail in Section 2, transaction aborts are caused not only by the detection of failures of transaction independence, but also by other events that are strongly dependent on specific hardware configurations and implementations. Hence, a transaction that may succeed on one processor model might abort on another model.

The second use case is to consider potential future RTM implementations, including those that might allow arbitrary length transactions. Current RTM hardware implementations limit transaction length due to cache size and write buffer size limitations. Our system is free of such limitations unless they are explicitly configured. Conceivably, this functionality could also be used to bridge RTM and software transactional memory.

The third use case is in debugging RTM code via a controlled environment. Through emulation, it is possible to introduce abort-generating events at specific points and observe the effects. It is also possible to collect detailed trace data from running code.

We have designed, implemented, and evaluated a system for Intel RTM emulation within the Palacios VMM. Our techniques are not specific to Palacios, and could be implemented in other VMMs as well. Our implementation will be available as part of the open-source Palacios codebase when this paper is published. Our contributions are as follows:

- We have designed a page-flipping technique that allows instruction execution while capturing instruction fetches, and data reads and writes. This technique avoids the need for any instruction emulation or complex instruction decoding other than determining instruction length. This greatly simplifies RTM emulation and could be applied to other services.
- We have designed an emulation technique for RTM based around the page-flipping technique, redo-logging, undefined opcode exceptions, and hypercalls. The technique is extensible, allowing for the inclusion of different hardware models, for example different cache sizes and structures.
- We have implemented the RTM emulation technique in Palacios. The entire technique comprises about 1300 lines of C code.
- We have evaluated our VMM-based RTM emulation technique and compared it with Intel's emulator-based implementation of Haswell RTM in the Software Development Emulator [6]. Our implementation is approximately 60 times faster when a transaction is executing, and has full performance when none are.

Related work: Herlihy and Moss introduced HTM [4]. Recent work by Rajwar, Herlihy, and Lai showed how HTM could be extended such that hardware resource limitations would not be programmer visible [11]. Unbounded transactional memory [2] shows that hardware designs that allow arbitrarily long transactions are feasible, and this work also demonstrated that using such transactions would allow for significant speedups in Java and Linux kernel code. Hammond et al. [3] have argued for using such powerful transactions as the basic unit of parallelism, coherence, and consistency. In contrast to such work, our goal is simply to efficiently emulate a specific commercially available HTM system that will have model-specific hardware resource limitations. By using our system, programmers will be able to test how different hardware limits might affect their programs. However, because the conditions under which our system aborts a transaction are software defined, and the core conflict detection process will work for a transaction of any size, provided sufficient memory is available, our system could also be employed to test models such as the ones described above. IBM has produced their own implementation of HTM in the BlueGene/Q architecture [13].

Our system leverages common software transactional data structures, such as hashes and redo logs. Moore et al. developed an undo log-based hardware transactional memory system [10] which lets all writes go through to memory and rolls them back upon conflict detection. Our emulator rolls a redo log forward on a commit.

2. HASWELL TRANSACTIONAL MEMORY

The Haswell generation of Intel processors include an implementation of hardware transactional memory. The specification for *transactional synchronization extensions* (TSX) has the goals of providing support for new code that explicitly uses transactions, backward compatibility of some such new code to older processors, and allowing for hardware differences and innovation under the ISA-visible model [5]. There are two models supported by TSX, *hardware lock elision* (HLE) and *restricted transactional memory* (RTM). Our focus in this paper is on RTM.

In the RTM model, four additional instructions have been added to the ISA: XBEGIN, XEND, XABORT, and XTEST. If they are executed on hardware which does not support them, a #UD (undefined opcode) exception is generated. Code can use the XTEST instruction to determine if it is executing within a transaction. An RTM transaction is typically written in a form like this:

```
start_label:
     XBEGIN abort_label
     <body of transaction, may use XABORT>
     XEND
success_label:
     <handle transaction commited>
abort_label:
     <handle transaction aborted>
```

The XBEGIN instruction signifies the start of a transaction, and provides the hardware with the address to jump to if the transaction is aborted. The body of the transaction then executes. If no abort conditions arise, the transaction is committed by the XEND.

Conceptually, the core on which the body of the transaction, from XBEGIN to XEND, executes does reads and writes that are independent of those from other cores, and its own writes are not seen by other cores until after the XEND completes successfully. If another core executes a conflicting write or read, breaking the promise of independence, the hardware will abort the transaction, discard all of the completed writes, and jump to the abort label. The code in the body of the transaction may also explicitly abort the transaction using the XABORT instruction. The specific reason is written into RAX so the abort handling code can decide what to do.

Beyond conflicting memory reads and writes on other cores, and the execution of the XABORT instruction, there are numerous reasons why a transaction may abort. These form three categories: instructions, exceptions, and resource limits. Within each category, there are both implementation-independent and implementation-dependent items. One of the benefits of our emulated RTM system is to allow the testing of RTM code under different implementations.

Instructions: The XABORT, CPUID, and PAUSE instructions are guaranteed to abort in all implementations. In addition, the specification indicates a very diverse set of other instruction classes may also cause aborts, depending on the specific RTM implementation.

An important point is that our system does not require decoding and emulating general instructions, and to abort on one of these classes of instructions we need only decode an instruction sufficiently to identify its class. Any such decoding need happen only for the instructions within the transaction. Furthermore, many of these instructions are already detected in the VMM out of necessity (e.g., control and segment register updates, I/O instructions, virtualization instructions, most privileged instructions), and others can be readily intercepted without decoding (e.g. RFLAGS updates, debug registers, ring transitions, TLB instructions, software interrupts).

Exceptions: An exception on the core executing a transaction generally causes the transaction to abort, although the specification has variable clarity about which exception aborts are implementation-dependent and which are guaranteed. We assume that all exceptions that the Intel specification says "may" cause aborts "will" cause aborts. This behavior can easily be changed within our hardware model. As the VMM is already responsible for injection of general and special interrupts, it can easily detect aborts due to asynchronous exceptions. Detecting synchronous exceptions is slightly more challenging, as we discuss later.

Exception delivery within the context of a transaction abort has unusual, although sensible semantics. For synchronous exceptions, the abort causes the exception to be suppressed. For example, if a transaction causes a divide-by-zero exception, the hardware will abort the transaction, but eat the exception. For interrupts, the abort causes the interrupt to be held pending until the abort has been processed. For example, if a device interrupt happens during the execution of a transaction, the hardware will abort the transaction, and begin its fetch at `abort_label` before the interrupt vectors.

Resource limits: The specification indicates that transactions may only involve memory whose type is writeback-cacheable. Use of other memory types will cause an abort. Additionally, pages involved in the transaction may need to be accessed and dirty prior to the start of the transaction on some implementations. Finally, the specification warns against excessive transaction sizes and indicates that "the architecture provides no guarantee of the amount of resources available to do transactional execution and does not guarantee that a transactional execution will ever succeed."

Our interpretation of these parts of the specification is that a typical implementation is expected to be built on top of cache coherence logic. The implication is that transactions will behave differently on different hardware just to cache differences. The line size will likely define the conflict granularity for transactions. Two writes to the same cache line, but to different words, will likely conflict. Hence, the larger cache line, the more likely a transaction is to fall victim to an abort caused by a false conflict.

Our system allows the inclusion of a hardware model that can capture these effects, allowing the bullet-proofing of code that uses transactions, and the evaluation of the effects of different prospective hardware models on the code. Interestingly, because it is a software system, it creates the effect of hardware without resource limits.

3. DESIGN AND IMPLEMENTATION

The implementation of our RTM emulation system is in the context of our Palacios VMM [9], but its overall design could be used within other VMMs. We now describe our system, starting with the assumptions we make and the context of our implementation, followed by an explanation of the page-flipping approach the system is based on, and finally the architecture and operation of the system itself.

3.1 Assumptions

We assume that our system is implemented in the context of a VMM for x86/x64 that implements full system virtualization. Such VMMs can control privileged processor and machine state that is used when the guest OS is running, and can intercept guest manipulations of machine state. We assume the VMM infrastructure provides the following functionality for control and interception:

1 Shadow paging. We assume shadow paging in the VTLB model [7, Chapter 31] is available. It is not essential that the guest run with shadow paging at all times, merely that it is possible to switch to shadow paging during transaction execution.

2 Explicit VTLB invalidation. We assume that the VMM allows us to explicitly invalidate some or all entries in the shadow paging VTLB, independent of normal shadow page fault processing.

3 Shadow page fault hooking. We assume that the VMM allows us to participate in shadow page fault handling. More specifically, we assume it is possible for us to install a shadow page fault handler that is invoked after a shadow page fault has been determined to be valid with respect to guest state. Our handler can then choose whether to fix the relevant shadow page table entry itself, or can defer to the normal shadow page table fixup processing.

4 Undefined opcode exception interception. We assume the VMM allows us to intercept the x86/x64 undefined opcode exception when it occurs in the guest.

5 CPUID interception. We assume the VMM allows us to intercept the CPUID instruction and/or set particular components of the result of CPUID requests.

6 Exception interception. We assume the VMM allows us to selectively enable interception of exceptions and install exit handlers for them.

7 Exception/interrupt injection cognizance. We assume the VMM can tell us when a VM entry will involve the injection of exceptions or interrupts into the guest. If the VMM uses guest-first interrupt delivery in which an interrupt can vector to guest code without VMM involvement, then it must be possible to disable this for the duration of the transaction so that the VMM can see all interrupt and exception injection activity.

The hardware virtualization extensions provided by Intel and AMD are sufficient for meeting the above assumptions. The same capabilities that the hardware provides could also be implemented in translating VMMs or paravirtualized VMMs. Common VMMs already meet 1, 2, 3, 5, and 7 as a matter of course. Items 4 (undefined opcode interception) and 6 (exception interception) are straightforward to implement. In AMD SVM, for example, there is simply a bit vector in the VMCB where one indicates which exceptions to intercept. On VM exit due to such an interception, the hardware provides the specific exception number.

Palacios already met most of the assumptions given in Section 3.1. Shadow paging capabilities in Palacios reflect efforts to allow dynamic changes for adaptation. Palacios did not include support for assumptions 4 and 6 (exception interception). Perhaps ironically, our initial implementation of these two is for AMD SVM. However, Intel's VT also provides an exception bitmap to select which exceptions in the guest require a VM exit, so these changes could be readily made for VT. In Palacios, exception/interrupt injection cognizance (assumption 6) is implemented with a check immediately before VM entry, in SVM or VT-specific code.

For the sake of initial implementation simplicity, we focused here again on the SVM version.

3.2 Architecture

Figure 1 illustrates the architecture of our system. It shows a guest with two virtual cores, one executing within a transaction, the other not. The figure illustrates two core elements, the per-core MIME (Section 3.3), which extracts fine-grain access information during execution, and the global RTME (Section 3.4), which implements the Intel RTM model. The RTME configures the MIMEs to feed the memory reference information into the conflict hash data structures (for all cores), and the per-core redo log data structure (for each core executing in a transaction). The conflict hash data structures are used by the RTME to detect inherent memory access conflicts that should cause transaction aborts regardless of the hardware resource limitations. Additionally, the memory references feed a pluggable cache model, which detects hardware-limitation-specific conflicts that should cause transaction aborts. The RTME is also fed by the instruction sequences from cores operating in transactions, and by intercepted exceptions from the guest and injected exceptions or interrupts from the VMM, which also are needed to assess whether an abort should occur.

When no virtual core is executing in a transaction, we revert to normal execution of instructions by the hardware. The switch to the illustrated mode of operation occurs when an XBEGIN instruction is detected via an undefined opcode exception. Only this particular exception needs to be intercepted during normal (non-transactional) execution.

3.3 Memory and Instruction Meta-Engine

A core requirement of transactional memory emulation is being able to determine the memory addresses and data used by the reads and writes of individual instructions. When a transaction is active on any core, all cores must log their activities, producing tuples of the form `{vcore, sequencenum, rip, address, size, value, type}` where *sequencenum* orders the tuples of a given *vcore*, *rip* is the address of the instruction being executed, *address* is the address being read or written, *size* is the size of the read or write, *value* is the value read or written, and *type* indicates whether the reference is a read, write, or instruction fetch.

Our design accomplishes this fine-grain capture of the memory operations and data of instruction execution via the Memory and Instruction Meta-Engine, or the MIME. One of the major contributions of this work is the novel page-flipping technique on which the MIME is based. This technique allows us to avoid instruction emulation and most aspects of instruction decoding. The MIME's page-flipping technique is based on the indirection and forced page faults made possible through shadow paging, and breakpoints to the VMM made possible through the hypercall mechanism.

Shadow paging and shadow page faults: It is necessary for the VMM to control the pages of physical memory that the guest has access to. Conceptually, with the VMM, there are two levels of indirection. Guest virtual addresses (GVAs) are mapped to guest physical addresses (GPAs) by the guest OS's page tables (the gPT), and GPAs are in turn mapped to host physical addresses (HPAs) by the VMM.

In shadow paging, the GVA$\rightarrow$GPA and GPA$\rightarrow$HPA mappings are integrated by the VMM into a single set of shadow page tables (the sPT) that express GVA$\rightarrow$HPA mappings that combine guest and VMM intent. The VMM makes the hardware use this integrated set of page tables when the guest is running. Any architecturally visible change to guest paging state needs to invoke the VMM so that the VMM can adjust the integrated page tables to incorporate it. In order to do so, the VMM intercepts TLB-related instructions and control register reads and writes. Hence, any operation the guest performs to alert the hardware TLB of a change instead alerts the VMM of the change. The VMM's shadow paging implementation thus acts as a "virtual TLB" (VTLB) and the shadow page tables are the VTLB state.

Suppose the guest creates a mapping (a page table entry) for the GVA `0xdeadb000`, which it does by writing this mapping to the gPT. The new mapping is not guaranteed to be architecturally visible until the TLB is informed. The guest does this by using an INVLPG instruction to flush any matching entry from the TLB. The VMM intercepts this instruction, where it informs the VMM that any entry that has the VTLB (the sPT) must be removed. When the guest later accesses some address on the newly mapped page, for example `0xdeadbeef`, the hardware walks the sPT, and on finding no entry, raises a page fault. The page fault is also intercepted by the VMM, which starts a walk of the gPT, looking for `0xdeadb000`. If no such entry existed in the gPT, the VMM would then inject a page fault into the guest. In this case, however, the gPT has a corresponding entry, and so the sPT is updated to include this entry, as well as a mapping to the appropriate HPA. Since the page fault occurred as a result of inconsistency between the sPT and gPT, it is referred to as a shadow page fault, and the guest OS is unaware that it ever happened. The next time the guest tries to access any address on the page `0xdeadb000` the sPT will have the correct mapping.

Breakpoint hypercalls: In addition to forced shadow page faults, the MIME also relies on being able to introduce breakpoints that cause an exit back to the VMM, which we accomplish with a hypercall. Both AMD and Intel support special instructions, `vmmcall` in the case of AMD, that force an exit to the VMM. To set a breakpoint at a given instruction, we overwrite it with a `vmmcall`, after first copying out the original instruction To resume execution, we simply copy back in the original instruction content and set the instruction pointer to it.

Process: We now describe the MIME process for executing an instruction using the following example:

```
prev:  addq %rbx, %rax
cur:   INSTRUCTION
next:  movq %rdx, %rbx
...
target:
...
```

Here, `cur` is the address of the instruction we intend to execute, while `next` is the address of the subsequent instruction, and `target` is a branch target if the current instruction is a control-flow instruction.

Write-only data flow instruction: Let us make the current instruction more specific, for example, suppose it is

```
cur:   movq %rax, (%rcx)
```

This instruction writes the memory location in the register `%rcx` with the 8 byte quantity in the register `%rax`. MIME executes this instruction, and other instructions in the following way. We begin this process with the requirement that

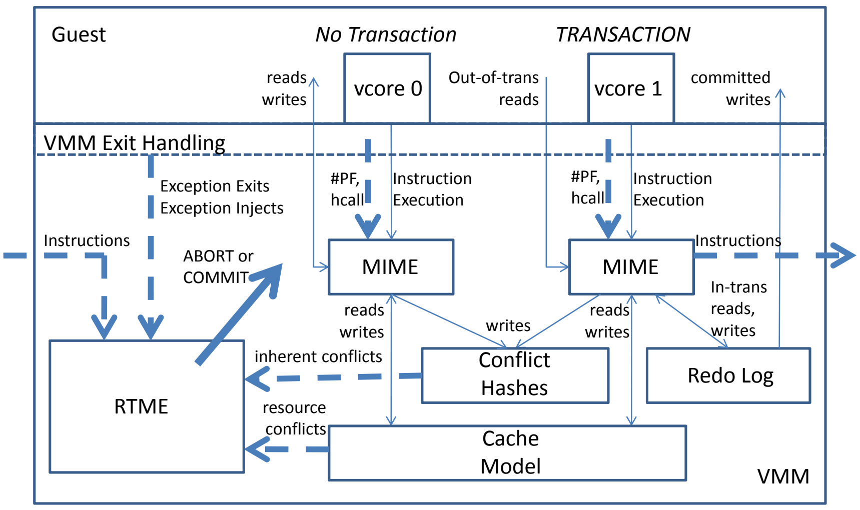

Figure 1: Overall architecture of the system

the sPT is completely empty.Note that the last step in the following reestablishes this for the next instruction.

1 We enter the guest with `%rip=cur`.
2 The instruction fetch causes a shadow page fault, which exits back to the VMM, which hands it to the MIME.
3 The MIME discovers this is an instruction fetch by comparing the faulting address and the current `%rip` and noting the fault error code is a read failure. In response, it creates an sPT entry for the page the instruction is on. While the page is fully readable and writable by the MIME, the sPT entry allows the guest only to read it. The MIME then overwrites `next` with a hypercall, saving the previous content.
4 We enter the guest with `%rip=cur`.
5 The instruction fetch now succeeds. Instruction execution now succeeds as well, up to the data write. The data write produces a shadow page fault, which exits back to the VMM, which hands it the MIME.
6 The MIME discovers this is a data write by noting that the fault code is a write failure.
7 In response to the data write, the MIME maps a temporary staging page in the sPT for the faulting address, and it stores the address of the write.
8 We enter the guest with `%rip=cur`.
9 The instruction fetch and the data write now succeed and the instruction finishes, writing its result in the temporary staging page.
10 `%rip` advances to `next`, resulting in the fetch and execution of the hypercall (note that the code page is now mapped in), which exits back to the VMM, which hands it to the MIME.
11 The MIME now reads the value that was written by the instruction on the temporary staging page. It can now make this write available for use by other systems. For example, the RTM system will place it into its own data structures if a transaction is occurring. If no other system is interested, it copies the write back to the actual data page.
12 At this point the MIME has generated two tuples for the record: the instruction fetch and the data write.
13 The MIME now restores the instruction at `next`
14 The MIME invalidates all pages in the VTLB. Strictly speaking, only two pages are unmapped from the sPT, the code page and the temporary staging page.
15 If MIME-based execution is to continue with `next`, goto 1, otherwise we are done.

Read-only data flow instruction: For an instruction like

```
cur:   movq (%rcx), %rax.
```

which reads 8 bytes from the memory location given in `%rcx` and writes that result into the register `%rax`, execution is quite similar. At stage 5, a shadow page fault due to the data read will occur. In stage 6, the MIME will detect it is a data read and sanity check it if needed. In stage 7, the MIME will map the staging page read-only, and copy the data to be read to it. This data can come from a different system. For example, the RTM system might supply the data if the read is for a memory location that was previously written during the current transaction. After the instruction finishes, it will then provide two tuples for the record: the instruction fetch and the data read.

Read/write data flow instruction: It is straightforward to execute an instruction such as

```
cur:   addq %rax, (%rcx)
```

which reads the 8 bytes from memory at `%rcx` adds them to the contents of `%rax` and then writes the 8 byte sum back to the memory at the address in `%rcx`. For all but the final write, the execution is identical to that of the read-only instruction given above. After completing stage 7, the staging page will be mapped read-only, and thus there will be an additional shadow page fault corresponding to the write. This fault will be handled in the same manner as with the write-only instruction. After the instruction finishes execution, it will then provide three tuples for the record: the instruction fetch, the data read, and the data write.

Control flow instruction: If a control flow instruction reads data (e.g., an indirect jump) or writes data (e.g., a stack write on a call), these reads and writes are handled in the same manner as the preceding data flow instructions. Since all the conditions to be checked (e.g., flags) are known at this point, we can "emulate" the instruction, placing the hypercall at the jump target.

Generalization: Although the above description uses simple two operand instructions and the simplest memory addressing mode as examples, it's important to note that the technique works identically for different numbers of operands and for arbitrary addressing modes. Indeed, even for implicit memory operands, the hardware will produce shadow page faults alerting us to their presence. The primary limitation is that an instruction with multiple reads and/or multiple writes to the same page may not have all of its reads and writes captured. We describe this in detail later. All addressing mode computations, as well as segmentation, are done well before a page fault on instruction or data references can result. The hardware does this heavy-lifting.

Instruction decoding and emulation: Step 3 of the processing described above requires basic instruction decoding. The issue is that x86/x64 instructions are of variable length (from 1 to 15 bytes). Hence, in order to determine what `next` is, we need to be able to determine the size of the current instruction. If the MIME does not need to trace control-flow instructions, this is the only requirement. If control-flow instructions are to be handled by the MIME, then we must further decode control-flow instructions to the point where we can also determine their target address. Our implementation uses the open source Quix86 decoder [8] to do this decoding for us. No emulation is done at all—we rely on the hardware to do instruction execution for us instead.

Read optimization in RTM: In our earlier description of handling read-only instructions and read-write instructions, we describe the use of a staging page during the read—when a data read is detected, we copy the value to be read to a staging page and present this page to the guest. In RTM, this is required when a core executing a transaction reads a value it has previously written during the transaction. At this point, in order to maintain isolation, the written value exists only in a redo log and must be copied from it. For a core that is not executing in a transaction, or for an in-transaction read of a value that was not previously written in the transaction, the staging page can be avoided and the read allowed to use the actual data page. This optimization is included in our RTM system.

3.4 Restricted Transactional Memory Engine

As shown in Figure 1, the RTME uses per-virtual core MIMEs to capture instructions and their memory accesses in a step-by-step manner during execution. The only instructions it needs to emulate are the XBEGIN, XEND, XABORT, and XTEST instructions. Because these instructions are not available in the hardware, they cause an undefined opcode exception which is caught by the VMM and delivered to the RTME.

The initial XBEGIN is emulated by capturing its abort target, advancing RIP to the next instruction, and switching all virtual cores to MIME-based execution. Additionally, the RTME has the VMM enable exiting on all exceptions with callbacks to the RTME. A special check is enabled in the interrupt/exception injection code which is run before any VM entry. This check tests if an injection will occur on entry, and if so it invokes a callback to the RTME before the entry. Either callback is interpreted by the RTME as requiring a transaction abort for that virtual core.

From the next entry on, MIME-based execution occurs on all virtual cores. On all virtual cores, the writes seen by the MIME are written to the conflict hash data structures. For a virtual core that is not executing in a transaction, the writes are also reflected to guest memory, and all reads are serviced from guest memory. For a virtual core that is executing in a transaction, writes are sent to the redo log instead of to guest memory. Reads are serviced from guest memory, except if they refer to a previous write of the transaction, in which case they are serviced from the redo log. For all cores, reads and writes are also forwarded to the cache model by the RTME.

In addition to the callbacks described earlier, the RTME is also called back by the MIME as it executes its state machine. This allows the RTME to examine each instruction and its memory operations to see if an abort is merited. Instructions are checked against the list given earlier. For all memory operations, the RTME checks the conflict hash data structures and the cache model. The former indicates whether a conflict would have occurred assuming an ideal, infinite cache. For example, if this core is not in a transaction, and has just written a memory location that some other core that is in a transaction previously wrote, a conflict is detected and the other core needs to abort its transaction. The cache model determines if a conflict due to hardware limitations has occurred. For example, if the current write is coming from a core that is executing in a transaction, and that write would cause a cache eviction of a previous write from the transaction, the cache model would detect this conflict and indicate that the current core needs to abort its transaction. A final source of an abort is when the RTME detects the XABORT instruction during the MIME instruction fetch.

Handling a transaction abort is straightforward: the writes in the redo log are discarded, the relevant error code is recorded in a guest register, the guest RIP is set to the abort address, and the guest is re-entered. Transaction commits occur when the XEND instruction is detected, and are also straightforward: the RTME plays the redo log contents into the guest memory, advances the RIP, and re-enters the guest. For either an abort or a commit, we also check if it is the last active transaction. If so, we switch all cores back to regular execution (turning off MIME, callbacks, and exception interception, except for the illegal opcode exception, which is needed to detect the next XBEGIN).

XTEST instructions are identified by the RTME through a UD exception, if the instruction is run during an active RTM section, then the ZF flag is set, otherwise it is cleared.

Redo log considerations: Our redo log structure is not, strictly speaking, a log. Rather, it stores only the last write and read to any given location. However, during MIME execution, there exist short periods where the most recent write or read is actually stored on the MIME staging page. A versioning bit is used so that when the MIME-based execution of an instruction completes, it is possible to update the redo log with newer entries on the staging page. These aspects of the design allow us to compactly record all writes and internal reads of a transaction.

Conflict detection: In addition to the conflict hashes, conflict detection in the RTME uses a global transactional memory (TM) state, a global transaction context, a per-core TM state, and a per-core transaction number. The global TM state indicates whether any core is running a transaction, while the per-core TM state indicates whether the specific core is executing a transaction. Each core assigns sequence numbers to its transactions in strictly ascending order, and the per-core transaction number is the number of the current, or most recent transaction on that core. The global transaction context gives the number of the currently active transaction, or most recently completed (aborted or committed) transaction on each core.

When any core is running a transaction, all cores must record the memory accesses they make, we accomplish this through the use of two hash tables. The first, called the address context hash, is a chained hash mapping memory locations to timestamped accesses. Each entry in the hashed bucket represents the global transaction context at the time of a memory operation, which acts as a ordering, or timestamp. In this way we are able to both record all memory accesses done by a core, as well as keep track of when they occurred. Since all memory accesses are tagged with the global context, when a core is checking for conflicts it can simply look at accesses made with the same context as its current transaction number. Entries in the hash have the form `{addr : (global_ctxt)→(global_ctxt)→...}`

The second hash table, called the access type hash, keeps track of the type of memory operation that was run on an address in a given context (read, write, or both). When a memory operation is run by a core, it creates one entry for each core in its hash. Data is duplicated in this manner to facilitate quick lookup on conflict checking as well as garbage collection. Entries in this hash have the form `{addr|core_num|t_num : access_type}`

Suppose we are running on a guest with two virtual cores, and core 0 begins a transaction. Each core will begin running its MIME and recording its memory accesses. Now suppose core 1 runs an instruction which writes to memory address `0x53`. It will first note the global transactional context, and add a node to the bucket for address `0x53` in the address context hash with this context. It will then make two new entries for the access type hash, one for each core in the system. Each entry will map address `0x53`, a core number, and that core's transaction number to a structure indicating the access was a write.

If the memory accesses of a core executing a transaction conflict with those of any other core, the transaction must be aborted. To check for conflicts, we use the context hashes. Conflict checking could be done after each instruction, or when attempting to commit a transaction. In our implementation, conflict checking is cheap relative to instruction run time and so we generally do it after each instruction.

In Figure 2 we illustrate the process of conflict detection. At the end of an instruction in a transactional block, core 0 walks its redo log of writes, shown as step 1. In step 2, core 0 checks if any conflicting memory accesses have been made by looking at every other core's address context hash. In the figure, core 0 checks for conflicting accesses to memory address `0x53`. In step 3, core 0 finds an entry for `0x53` in core 1's address context hash, and walks the list of contexts during which core 1 accessed `0x53`. Core 0's current transaction number is 2, so the entry made during context `{2,3}` is a potential conflict. In step 4 core 0 checks the entry for address `0x53` in core 1's access type hash to identify the kind of access made. Having found that core 1 wrote to address `0x53` when core 0 was in transaction 2, a conflict is detected, and core 0 must abort its transaction, shown as step 5.

Garbage collection: Memory use expands during execution as the redo logs and conflict hashes grow in size. Redo logs are garbage collected at the end of each transaction.

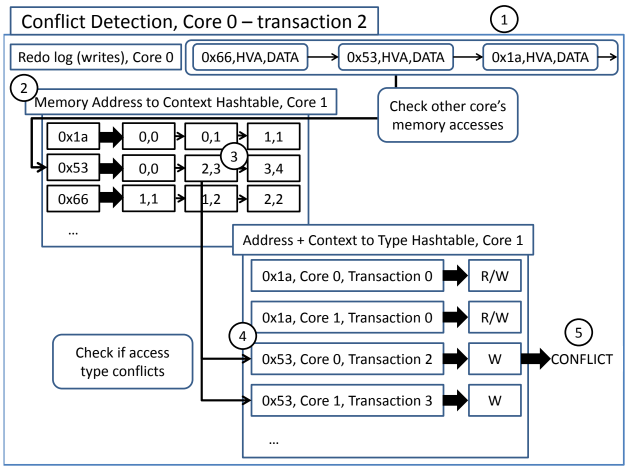
Conflict Detection, Core 0 – transaction 2
1
Redo log (writes), Core 0
0x66,HVA,DATA
0x53,HVA,DATA
0x1a,HVA,DATA
2
Memory Address to Context Hashtable, Core 1
Check other core's memory accesses
0x1a
0,0
0,1
1,1
0x53
0,0
2,3
3
3,4
0x66
1,1
1,2
2,2
...
Address + Context to Type Hashtable, Core 1
0x1a, Core 0, Transaction 0
R/W
0x1a, Core 1, Transaction 0
R/W
4
5
Check if access type conflicts
0x53, Core 0, Transaction 2
W
CONFLICT
0x53, Core 1, Transaction 3
W
...

Figure 2: An example of a write/write conflict detected by core 0 on a write from core 1

Since the conflict hashes contain information from multiple generations of transactions, we must answer the questions of how to determine which entries are garbage, and when to perform this garbage collection.

Garbage collection leverages the access type hash. We start by noting the current global transaction context, then we iterate over all the keys of the access type hash, and for each key (an address), we walk over its corresponding list of contexts. If we find a context that is strictly less than the global context, this means that there is no core left that may need to check that memory operation, as it happened during transactions that are no longer active. We can generate from this stale context the corresponding keys for the access type hash, and delete those keys from it. Finally, we delete the stale context from the list, and delete the key from the address context hash if the list is now of zero length. A locking strategy is employed to assure that a garbage collection and MIME accesses are mutually exclusive.

When to garbage collect is a more difficult question, as when we have an opportunity to do so, we cannot be certain about the state of other cores, or when the next opportunity to collect may occur. Currently in our implementation each core will garbage collect on every transaction completion.

4. EVALUATION

We considered three factors when evaluating our RTM implementation: its size, how it runs code with transactions, and the performance of the implementation relative to the native execution rate of the hardware and compared to a different emulator.

Test environment: All testing was done on a Dell PowerEdge R415. This is a dual socket machine with each socket having a quadcore AMD Opteron 4122 installed, giving a total of 8 physical cores. The machine has 16 GB of memory. It ran Fedora 15 with a 2.6.38 kernel. Our guest environment uses two virtual cores that run a BusyBox environment based on Linux kernel 2.6.38. The virtual cores are mapped one-to-one with unoccupied physical cores. This machine does not have an HTM implementation.

Implementation size: Our implementation of RTM emulation is an optional, compile-time selectable extension to

Palacios, and we made an effort to limit changes to the core of Palacios itself. There were two major areas where we had to modify the Palacios core, namely (1) handling of exceptions and interrupts, some of which are needed to drive the RTME, and (2) page fault handling, allowing some page faults to drive the MIME. These changes and the entirety of the extension code comprise 1300 lines of C. Given the size and very clear changes to the Palacios codebase, it should be possible to port our implementation to other VMMs.

Test cases: To test the correctness of our implementation, we needed a test suite which would present the implementation with various behaviors, and an ability to test the outcome. GCC 4.7 includes support for compiling the Haswell transactional instructions, but the test cases shipped with it only evaluate the behavior of software transactional memory. We found we had to write our own test cases, which test the following scenarios: (1) transaction calls XABORT after making no changes to memory, (2) transaction calls XABORT after having "written" to memory, (3) transaction writes memory with an immediate value, (4) transaction reads memory into a register, (5) transaction writes a register to memory, (6) transaction reads and writes the same memory location, (7) transaction thread writes to distinct, addresses, and (8) transaction and non-transactional thread write to overlapping addresses. The test cases are written using pthreads. After the threads set their affinity for distinct virtual cores, and synchronize, they then repeat their activity. Hence, over time, various possible orderings of execution are seen, as are aborts due to external causes (e.g., interrupts). These test cases form a "correctness test" of our implementation, which it passes.

Performance: The MIME-based execution model must obviously be slower than normal execution under the VMM or at native speeds. To quantify this slowdown over native, we ran an additional microbenchmark on our system and on a new, first-generation Haswell machine, an HP Proliant DL320e with a single-socket, quad-core Intel Xeon E3-1720v3 and 8GB RAM.

This benchmark consists of one thread pinned to a single core that enters a transaction, writes to a memory location, and then exits the transaction. The benchmark measures the time spent running 10 such transactions, and is intended to typify a common transactional code path. We averaged this runtime over 100 runs. To ensure accurate timing for RTM emulation, we used the TSC in passthrough mode to measure elapsed time. We found that on native, the average time spent running the 10 transactions was 2.57usec, while under MIME, the average time was 853.88usec.

Future work on transactional memory emulation will include comparing performance of multi-threaded applications, more complicated transational semantics including transaction restarts, and testing transactions that the hardware would not be able to support, such as those that exceed architectural limits.

We also ran our test cases on Intel's Software Development Emulator (SDE), where we found a slowdown on the order of 90,000×—our RTME runs approximately 60 times faster during a transaction. Moreover, the SDE's overhead occurs all the time, as one might expect for full emulation. There is a caveat in these numbers, however. The cache model we are using in our RTME is the null model (no aborts due to hardware resource limits), while Intel's is not. That said, we found that the average MIME "step" – the average time to process a read or write – took on the order of 7,000 cycles. This means that we would need to use a cache emulator that took >400,000 cycles per read or write in order for our system to slow down to speeds of emulation.

5. CONCLUSIONS

We developed an implementation of Intel's HTM extensions in the context of a VMM using MIME, a novel page-flipping technique. Our implementation allows the programmer to write code with TSX instructions, allows for bullet-proofing of code for various hardware architectures, as well as allowing tight control of the environment under which a transaction is occurring. We are able to achieve this with limited instruction decoding, and at speeds approximately 60 times faster than under emulation.

6. REFERENCES

[1] Advanced Scientific Computing Advisory Committee. The opportunities and challenges of exascale computing. Technical report, Department of Energy, Fall 2010.

[2] C. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA '05)*, February 2005.

[3] 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 *Proceedings of the 31st annual international symposium on Computer architecture (ISCA '04)*, June 2004.

[4] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture (ISCA '93)*, May 1993.

[5] Intel. Intel architecture instruction set extensions programming reference. `http://software.intel.com/sites/default/files/m/3/2/1/0/b/41417-319433-012.pdf`, 2012.

[6] Intel. Intel software development emulator, November 2012. v. 5.31.0.

[7] Intel. Intel 64 and ia-32 architectures software developer's manual volume 3c, chapter 32. `http://download.intel.com/products/processor/manual/325384.pdf`, 2013.

[8] A. Kudryavtsev, V. Koshelev, B. Pavlovic, and A. Avetisyan. Modern hpc cluster virtualization using kvm and palacios. In *Proceedings of the Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit (FederatedClouds '12)*, September 2012.

[9] J. R. Lange, P. Dinda, K. C. Hale, and L. Xia. An introduction to the palacios virtual machine monitor—version 1.3. Technical Report NWU-EECS-11-10, Department of Electrical Engineering and Computer Science, Northwestern University, November 2011.

[10] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. Logtm: log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA '06)*, February 2006.

[11] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd annual international symposium on Computer Architecture (ISCA '05)*, June 2005.

[12] S. Sachs and K. Yelick. Ascr programming challenges for exascale. Technical report, Department of Energy, 2011.

[13] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of blue gene/q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*, September 2012.