Virtualized Transactional Memory

Zachary Bischof
Northwestern University
Evanston, IL, USA
zbischof@u.northwestern.edu

Marcel Flores
Northwestern University
Evanston, IL, USA
marcel-flores@u.northwestern.edu

Maciej Swiech
Northwestern University
Evanston, IL, USA
maciejswiech2007@u.northwestern.edu

ABSTRACT

Transactional memory has long been considered as a useful alternative to locks for implementing parallel algorithms, due to its simpler programming interface. In particular, it offers an optimistic approach to parallelization, and attempts to perform an operation assuming no contention will occur. Despite the potential gains of hardware transactional memory, it is not yet widely supported in hardware. This will change soon with Intel's upcoming Haswell architecture, which provides support for certain transactions. However, there is not an easy way to measure how the added support for these instructions will help improve performance.

In this paper, we leverage Palacios, a virtual machine monitor, to add preliminary support for Intel's publicly available transactional memory specification. By switching between direct execution on hardware and simulated support of hardware transactional memory, we are able to test the impact of these instructions on application performance, providing developers with a unique chance to estimate how hardware support could affect their application's performance. The focus of this work is to explain how we were able to add support to Palacios to test these features.

1. INTRODUCTION

Transactional Memory is pretty cool stuff. We decided to implement it.

2. BACKGROUND

Transactional memory has long been proposed as a solution to the programmatic complexity that arises from attempting to use mutual exclusion locks in real world implementations of parallel algorithms. We will now consider a basic overview of the general idea of transactional memory. We will then go on to describe the version that Intel has included in the specifications for the new Haswell processors.

2.1 Transactions

We define a transaction to be a sequence of memory operations that are performed by a processes such that, to all other processes, the operations seem to have been performed atomically. Furthermore, the operations must appear to the process performing the transaction as having happened one after another, with no interference from other processes. When a transaction is finished, it either commits the results of its operations to main memory, or it aborts the transaction and throws away its changes, depending on the results of a memory validation.

In order to protect the appearance that the operations have executed serially without interference, the validation must detect if other processes have written to memory that has been used by the transaction. Likewise, in order to preserve the atomicity, the transaction must not allow other processes to interfere with the commit. Therefore, the validation must detect if other processes have written to memory that is read by the transaction (and thereby interfering with the appearance of serial execution), or other processes read memory written by the transaction (potentially interfering with atomicity). Upon an abort, a transaction may decide to attempt to run again, or revert to traditional mutex locking mechanisms. [1]

In addition to memory conflicts with other user processes, transactions may also be aborted in the event of interrupts, context switches, or other forced changes in the control flow, as they would violate the requirement that the transaction happen entirely serially.

Transactions of this form can be thought of as an optimistic approach to concurrency, contrasting with the more pessimistic approach of locks. Rather than explicitly preventing other processes from accessing memory, the transaction assumes that it will probably be able to complete unencumbered. If a conflict does occur, the transaction is able to deal with it accordingly. Locks, on the other hand, assume that conflicts are likely to occur, and directly block other processes from accessing the relevant portions of memory. Transactions therefor stand to offer a performance gain, as forced serialization can be avoided in many cases.

In addition, transactional memory stands to decrease the complexity of multi-process applications. Programmers would no longer be weighed down by keeping track of which process is holding which locks, and no longer needs to worry about deadlocks, starvation, and other side effects that can result from incorrect use of locks.

2.2 Intel Implementation

The starting with the Haswell, Intel has inluded the Intel Transactional Synchronization Extension, which is their implementation of transactional memory. The extension includes two interfaces: the first is Hardware Lock Elision, a system designed to work on legacy processors, and of no further interest to us. The second is Restricted Transactional Memory (RTM). This includes instructions for beggining a transactional region (and specifying the code to run in the event of a failure), aborting a transaction, and ending a transactional region. (TODO: Cite the intel manual somewhere near)

The Intel implementation generally follows the design of [1], with a few exceptions. Most notably, each transaction is only attempted a single time: in the event of a failure or abort, the code immediatly resorts to the failsafe. Additionally, the transactions are checked at a granularity of cache lines, rather than actual memory address. In order to validate each transaction at the time of completion, this implementation stores a read set, the locations of memory reads from the transaction, and the write set. An abort is triggered if another processor reads from an address that appears in the write set or writes to an address in either set.

Since this is a hardware implementation, transactions may also abort if the read and write sets exceed a certain, hardware dependent, capacity. The transaction will also be aborted for any numer of instructions that may interfere with control flow: such as the writing of control registers, any variety of interrupt, processor state changes, TLB control, and so on. The specification also suggests that aborts may occur in the case of self modifying code, or other unpredicable scenarios.

The Intel implementation also offers support for nested paging, however we save discussion of this topic for later work. (Maybe just don't mention this).

The Palacios virtual machine manager offers us a unique opportunity to implement transactional memory which emulates both the original design of transactional memory, as well as a number of the subtleties observed in Intel's implementation.

3. PALACIOS IMPLEMENTATION

In order to implement transactional memory in Palacios, we attempt to follow Intel's Haswell specifications, providing us with a consistent interface with which to use transactions, as well as making future comparisons to direct hardware implementations easier. The implementation relies on storing all information that would normally be kept by the processor in the datastructures availible to Palacios. As a result, a great deal of the complexity of our implementation arises from our need to exit the VM and perform actions in the VMM. We now consider the steps in our implementation.

3.1 State Machine

We implement the following states to keep track of a transaction:

1. Ifetch
2. Exec
3. Abort

In addition, there is a transaction mode, which indicates whether a transaction is currently running. When the VM is launched, the mode is set to OFF, and the state is ignored.

3.1.1 Beginning a Transaction

The transaction begins when the VM enounters an XBEGIN instruction. Since the VM is running on hardware which does not yet support this instruction, this causes the VM to exit. On an AMD machine, the SVM handler then indicates that the undefined exception was indeed the cuase for the exit. An undefined exception handler is then called, which performs a limited decoding to see if the exit was caused by one of the three Haswell transactional memory instructions.

If the instruction is an XBEGIN, the VMM begins the process of launching the transaction. First, the argument to XBEGIN is stored, as this indicates the location of the failsafe code. A vmmcall is then registered with the VMM (we further describe this later). Next the various read and write set data structures are initialised, the transaction state is set to "IFETCH", and the RIP is advanced to the next instruction.

Next, we invalidate the current shadow page table. This will cause the VM to exit to Palacios in the event of any memory access, allowing us to record it. Since we are only eliminating the shadow page table, we are not actually tampering with the state of the guest, which allows this manipulation to go undetected by the guest.

3.1.2 Handling an Ifetch

Since we have eliminated the shadow page tables, as soon as the VM resumes executing, a pagefault is caused when attempting to read the instruction. Since the transaction mode has been set to ON, we are able to catch this pagefault, and in particular know both the faulting address and the associated error code. We confirm that this fault is caused by an ifetch through both the error code and by explicitly checking that the RIP and faulting address match. We advance the transaction state to "EXEC", as we are now executing a transaction.

- Note what we keep track of

- Undefined exception handler

- Mechanism by which we capture reads and writes

-General shadow page manipulations

-Staging page

- Actual trans mem handlers

Figure 1: The state machine which controls how our implementation records transactions

- Keeping track of other cores

- Various aborts

4. DESIGN CHALLENGES

- Shadow paging subtlties

- Code injection

- Register clobbering

4.1 Current Assumptions

- AMD

- Decoder limitations

- Assume all activity comes from a single core (will be fixed)

- All memory activity from a transaction fits on a page (with no offset collisions)

- All instructions fit on a page, and overwriting the next instruction can be done on the same page.

- All interrupts (aside from the obvious ones) are ignorable

5. FUTURE WORK

- Fix major assumptions, in particular those related to pages and transaction size limitations.

6. CONCLUSIONS

- The virtual machine is a viable place to implement paradigms that do not yet exist in hardware

- While by no means fast, in our very limited transaction test space, performance was tolerable.

7. REFERENCES

[1] M. Herlihy and J. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.