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 an alternative to locks for implementing parallel algorithms. 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 transactional memory, it is only in the not-yet-released Haswell architecture from Intel that it has made an appearance. Palacios has presented us with a unique opportunity to emulate this Intel implementation in software.

In this paper, we present a basic implementation of transactional memory, which emulates the Haswell instructions, successfully performing memory transactions.

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 complexity that arises from attempting to use locks in real world implementations of parallel algorithms. We now discuss the basic model of a transaction.

2.1 Transactions

Transactions are a sequence of memory operations that are performed by a process such that the operations appear to happen serially and atomically [1]. When a transaction is finished, it either commits its changes to main memory, or it aborts the transaction and throws away its changes. In particular, when a transaction begins, it keeps track of all memory which is read and written during the transaction. If another process reads or writes this memory, the transaction is aborted. Upon an abort, a transaction may decide to attempt to run again, or revert to traditional locking mechanisms.

In addition to memory conflicts, which violate the atomic requirement, 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 otherhand, 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 and other such situations.

2.2 Intel Implementation

The Intel implementation generally follows the design of [1], which a few exceptions. Most notably, each transaction is only attempted a single time: in the event of a failure or abort, the code immediatly runs the failsafe. Additionally, the transactions are checked at a granularity of cache lines, rather than actual memory address.

Other changes?

The Palacios virtual machine manager offers us a unique opportunity to implement transactional memory without explicit hardware support.

3. PALACIOS IMPLEMENTATION

In order to implement transactional memory in Palacios, we 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.

4. STATE MACHINE

In order to keep track of all memory reads and writes during a transaction, and to respond accordingly, we implement a state machine.

- Diagram of the state machine?

- 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

- Keeping track of other cores

- Various aborts

5. DESIGN CHALLENGES

- Shadow paging subtlties

- Code injection

- Register clobbering

5.1 Current Assumptions

- AMD

- Decoder limitations

-

6. FUTURE WORK

7. CONCLUSIONS

8. 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.