A First Insight into Object-Aware Hardware Transactional Memory

Behram Khan, Matthew Horsnell, Ian Rogers, Mikel Luján, Andrew Dinn & Ian Watson
Advanced Processor Technologies Group
The University of Manchester, United Kingdom

ABSTRACT

The contribution of this paper is the first Hardware Transactional Memory (HTM) where the object structure is recognized and harnessed. Our approach is similar to hardware support of paged virtual memory using a virtually addressed cache and a TLB, and is based on a cache hierarchy that allows the addressing of objects by unique object identifiers. The object-aware HTM allows cache overflows of uncommitted data. It also enables a novel commit and conflict detection mechanism. In this preliminary evaluation, the Lee-TM application exhibits overflows that in most previous HTMs would have had to be handled by software, impacting on performance. The simulation provides an insight into the scalability characteristics of the proposed HTM, which uses object and field granularity, lazy versioning and lazy conflict detection. For example, with 32 cores the broadcast of write sets is at under 5% of the bus bandwidth, showing the potential of object-aware HTM systems.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures

General Terms

Design, Performance

Keywords

Object-oriented programming, Transactional Memory

1. INTRODUCTION

The majority of new applications are developed using Object-Oriented (OO) languages, with more than 60% relying on managed runtime systems. At the same time, state-of-the-art computer architectures are integrating more cores on a single chip. In foreseeable generations this will reach up to hundreds of cores in many-core architectures. The challenge is to provide a parallel programming model that is accessible for software developers, and enables the development of computer architectures that scale in the many-core era. To address this challenge we are interested in Transactional Memory (TM) [3] as a programming model, rather than simply as a replacement for locks. The difference is that a truly transactional program may manipulate large amounts of transactional data, have transactions which run for long periods and create significant amounts of conflict. These characteristics can invalidate assumptions made by previous HTMs [3]: (a) rare overflows of hardware resources dedicated to store uncommitted transactional data, e.g. [2], or (b) aborts are infrequent, e.g. [4]. Therefore we investigate whether an object-aware HTM can overcome these limitations and also reduce the demands on the network-on-chip (information about changes to fields within an object can potentially be communicated in a more concise form). This paper presents the first object-aware HTM and constitutes our first step on studying the HTM for a bus-based system. note that it does not preclude execution of non-OO TM languages. Further information about our work on TM can be found in [1].

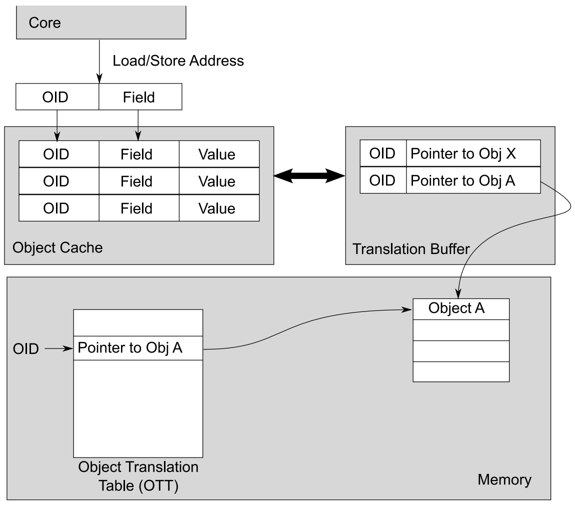

Figure 1: Hardware support for object addressing.

The motivation of previous schemes with direct hardware support for OO languages was mainly to ease the problems of memory management [6, 7]. Objects can be addressed via an Object IDentifier (OID) and field offset. The OID is translated using an Object Translation Table (OTT), a map of OIDs to memory addresses. This indirect object representation, although used in early implementations of OO languages, has generally been abandoned to avoid the inefficiency of an extra load per memory access. This inefficiency can be reduced by the provision of an *object cache*. The addresses issued by the core and the tags stored in the cache are viewed as an OID and field offset (although the cache tag

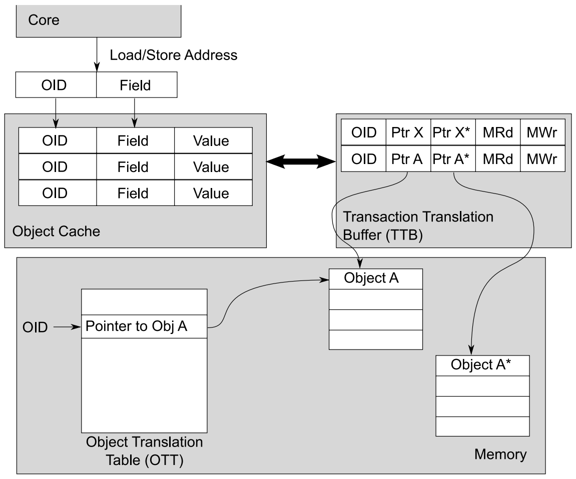

Figure 2: Object-aware HTM.

may contain a subset of the offset if multiple fields are stored per cache line). Cache hits on OIDs can clearly access the field data directly. However, on a cache miss, it is necessary to access memory via the OTT. This latter inefficiency can largely be removed by the provision of a Translation Buffer that caches recently used OID to memory address mappings (see Figure 1).

2. AN OBJECT-AWARE HARDWARE TRANSACTIONAL MEMORY SYSTEM

There are two major functions required to implement TM. The first is the ability to handle both committed and uncommitted data while a transaction is executing, *memory versioning*. The second is the detection of interference among transactions, *conflict detection*. A third function required for lazy versioning TM system is commiting that data generated by a transaction, *transaction commit*. To simplify the descriptions of these three functions in our object-aware HTM, we will assume an 'all transactions all of the time' mode of operation as in [2].

Transactional Object Versioning — The support for indirection can be adapted to provide additional support for transactional objects. The basic mechanism involves keeping a reference, in the local translation, both to the currently committed version of the object and to a temporary version where transactional state can be buffered. An outline of this scheme is shown in Figure 2. Assume the object cache and the Transaction Translation Buffer (TTB) are empty at the start of a transaction. The TTB lines also keep the read set (MRd) and write set (MWr) of the copy. These can simply be single bits per field of the object. Reads of object fields occur via the normal translation to the committed object, copies are placed in the cache and the reads noted in the read set map. If a write occurs to the field of an object not yet written, an area of memory is allocated (Object A*) into which modifications of the original object (Object A) can be written. The address of this copy space is returned and is written into the TTB as the copy pointer (Ptr A*). On the first write a cache entry is made, any future reads and writes to that field will obtain the current local object cache value. An entry is also made in the write set.

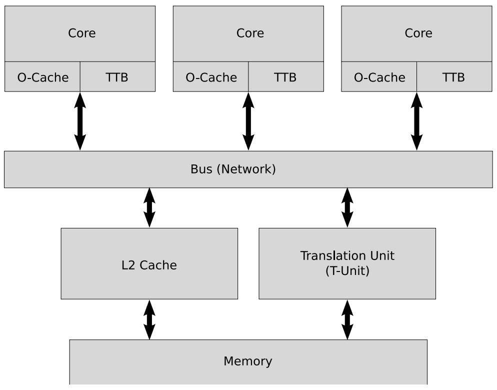

Figure 3: Simulated object-aware HTM.

If, during the execution of a transaction, it is necessary to displace a modified object field from the cache, that field is written to the object copy (Object A*). If there is now a subsequent read from that field, the decision whether to read the original object or the copy can be made from an observation of MWr. By this simple mechanism, we are able to provide a lazy versioning mechanism and direct hardware support for 'virtualization' of version information.

Observe that to commit an object, three steps are required: (1) any fields in the write set of the transaction must be written from the cache to the object copy (Object A*); (2) any un-modified fields should be copied into the object copy (Object A*) from the committed object (Object A); and (3) the pointer in the memory based OTT (Ptr A) must be replaced by the pointer to the copy (Ptr A*).

Figure 3 shows the basic structure of a multi-core system with a single Level 2 (L2) cache and memory unit which will be used in the evaluation. The object-aware cores are connected to a conventionally addressed L2 cache via a bus, and through that to memory. In an extended scheme this can be replaced by a more general on-chip network, as we do not assume support for bus based cache coherence. Associated with the memory is a translation unit (T-Unit) which provides a number of functions associated with the translation of object addresses, object cloning and committing. At its simplest when a core starts a transaction, it will flush its caches so that dirty objects are not present.

Transaction Commit — In the absence of conflicts, a transaction first flushes any modified lines from its object cache into the allocated copy space. The commit has not completed yet and other transactions can continue making progress. Once the modified data has been flushed from the cache, a request is made to lock the T-Unit. While the T-Unit remains locked a committing transaction copies any unmodified fields from the current committed objects into the transaction's copy objects, and then overwrites the pointers in the OTT. During this procedure the T-Unit broadcasts the OID and the write set of any modified objects. Although a locked T-Unit cannot be accessed by other cores, requests to the L2 cache or memory are satisfied for those cores using locally cached translations.

Conflict Detection — To detect overlaps between the write and read sets of transactions (i.e. conflicts) we have chosen lazy detection. This is more compatible with our aim to support a highly extensible communication structure.

Feature	Description
Object size	128B.
L1 object cache	32KB, private, 4-way assoc., 32B line, 1-cycle access.
TTB	24KB, private, 4-way assoc., 12B lines, 1-cycle access.
Network	256-bit bus, split-transactions, pipelined, no coherence.
L2 cache	4MB, shared, 32-way assoc., 32B line, 16-cycle access.
TU	4MB, shared, 32-way assoc., 12B lines, 16-cycle access.
Memory	100-cycle off-chip access.

Application	read set/Tx		write set/Tx		Inst/Tx		Overflow	
	Mean	CoV	Mean	CoV	Mean	CoV	Txs	Lines
Lee-TM-t	117.5	2.9	78.8	3.1	373810.6	4.1	287	407255
Lee-TM-ter	15.8	1.4	78.8	3.1	373813.9	4.1	291	407683

Table 1: Simulation parameters & Lee-TM profile.

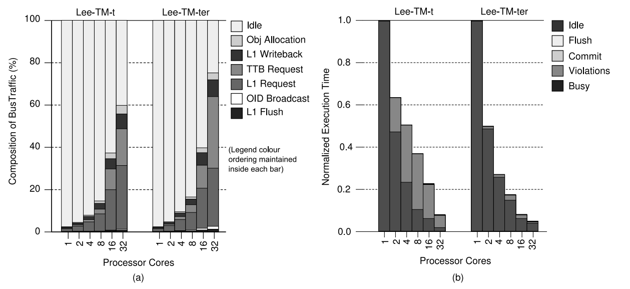

Figure 4: Bus traffic (a) and execution time (b).

When a transaction is ready to commit, it will write its changes to its copy object and then attempt to lock the T-Unit so that any unmodified fields can be copied from the currently committed objects and the pointers to the transaction's copy of objects can be installed in the OTT. At this point it will broadcast messages containing OIDs and the write set of all changed objects. Observing cores (i.e. cores executing a transaction that have received the broadcast message) must compare the read set of its OIDs in its local TTB with the write set of any matching OIDs in the broadcast message. Any overlap will cause an abort and restart in the observing core.

The broadcast messages need only contain the OID and the write set mask of the objects to be committed and thus the bandwidth required should be far less than a scheme which broadcasts all addresses which have been written to. In our current implementation, the broadcast takes place to all cores and bandwidth is therefore wasted if they do not require the information. An alternative under investigation is to attach a directory to the translation unit to keep track of object sharing.

3. EVALUATION

As with the first versions of other HTM systems [4, 2], this initial implementation of the object-aware HTM disallows transaction suspension, migration or context switches. To evaluate our object-aware HTM system a prototype platform has been developed, comprising an event-driven simulator with an IPC of 1 for all but memory operations (transactional performance is essentially memory bound), and an associated static Java compiler and runtime system. Latency, bandwidth and contention for shared resources is modelled at a cycle-level for all caches, network and memory models within the simulator. Timing assumptions and architectural configurations are listed in Table 1. The compiler generates Java bytecode that is then translated into machine code. It is linked alongside the runtime system code. The runtime system has been extended to support the allocation of transactional object copies, which in the current system are allocated in a reserved region of the heap space. We exercise the proposed computer architecture using a transactional version of Lee's routing algorithm [5]; Lee-TM-t (transactional) and Lee-TM-ter (early release).

Table 1 also presents transaction profile statistics for the application used. We include both the arithmetic mean and Coefficient of Variance (CoV) for read sets, write sets and instructions per transaction. There is a significant variation in the length of transactions and hence the size of the working sets (measured as number of objects). Lee-TM-t and Lee-TM-ter create large working sets that overflow the 32KB L1 object cache.

A breakdown of the total execution time and bus traffic are shown in Figure 4 with the number of cores doubling from 1 up to 32. The execution time is composed of *idle* cycles (due to work imbalance), *commit* cycles (the commit overhead), *violation* cycles (time spent executing within aborted transactions) and finally *busy* cycles (the useful committed work). Higher busy ratios are better.

Lee-TM-t shows a significant number of aborted transactions, accounting for 25 to 73% of the total execution time for 2 and 32 cores respectively. As reported in [5], Lee-TM-ter exploits application knowledge using early release, which reduces the conflicts and average read set to 15.8 objects, and increases the speedup at 32 cores from 12.6 to 20.9.

The general trend is that the amount of *idle* time on the bus decreases as more cores are added. The majority of traffic growth is associated with four request types: *L1 request*, *TTB request*, *Object allocation* and *L1 Flush*. *L1 request* and *L1 Flush* can be distributed in a more extensible system to avoid them becoming a bottleneck. *TTB request* and *Object allocation* traffic is an artifact of our current implementation, as any requests to the T-Unit while locked, triggers continual retries until the T-Unit is unlocked. The retries can be easily eliminated by a back-off or queueing policy.

The remaining traffic generated in the system is associated with broadcasting of OIDs and write sets. The *OID broadcast* traffic on the bus, which includes broadcast of the OID and the write set map, is the minimum amount of information that needs to be broadcast to other processing cores to make them aware of potential conflicts. *OID broadcast* accounts for less than 5% of the traffic even at 32 cores.

4. REFERENCES

[1] http://www.cs.manchester.ac.uk/apt/projects/TM/.

[2] L. Hammond *et al.* Transactional Memory Coherence and Consistency. In *Proc. of ISCA*, 2004.

[3] Jim Larus and Ravi Rajwar. *Transactional Memory.* Morgan & Claypool Publishers, 2007.

[4] K.E. Moore *et al.* LogTM: Log-based transactional memory. In *Proc. of HPCA*, 2006.

[5] I. Watson *et al.* A Study of a Transactional Parallel Routing Algorithm. In *Proc. of PACT*, 2007.

[6] I.W. Williams. *Object-Based Memory Architecture.* PhD thesis, University of Manchester, 1989.

[7] G. Wright *et al.* An Object-aware Memory Architecture. *Science of Computer Programming*, 62(2):145–163, 2006.