An Object-Aware Hardware Transactional Memory System

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

Abstract

Transactional Memory (TM) is receiving attention as a way of expressing parallelism for programming multi-core systems. As a parallel programming model it is able to avoid the complexity of conventional locking. TM can enable multi-core hardware that dispenses with conventional bus-based cache coherence, resulting in simpler and more extensible systems. This is increasingly important as we move into the many-core era. Within TM, however, the processes of conflict detection and committing still require synchronization and the broadcast of data. By increasing the granularity of when synchronization is required, the demands on communication are reduced. Software implementations of TM have taken advantage of the fact that the object structure of data can be employed to further raise the level at which interference is observed. The contribution of this paper is the first hardware TM approach where the object structure is recognized and harnessed. This leads to novel commit and conflict detection mechanisms, and also to an elegant solution to the virtualization of version management, without the need for additional software TM support. A first implementation of the proposed hardware TM system is simulated. The initial evaluation is conducted with three benchmarks derived from the STAMP suite and a transactional version of Lee's routing algorithm.

1 Introduction

Fundamental limits in integrated circuit technology are bringing about the acceptance that multi-core and, in the future, many-core processors will be commonplace [12, 7, 15]. If general purpose applications are required to exhibit high performance on such processors, it will be necessary to develop new, easy to use, parallel programming techniques. Transactional Memory (TM) is one such programming technique.

TM is an approach to parallel programming where a transactional parallel thread is assumed to execute independently and atomically, with respect to other transactional parallel threads. A thread does not, however, commit any changes it makes to global state until it is verified that no other global state changes have occurred, while it was executing, which might have invalidated its computation. An invalidated transaction must be aborted and restarted to perform the correct computation. The belief is that this provides a way of constructing parallel programs that is simpler and less error prone than existing approaches such as locking. Additionally, unlike lock based programs, transactional sections of a program can be composed. Parallel libraries can be envisaged whose contents can be utilised without any need for knowledge of the internals of the code.

There are two major functions required to implement transactions on top of a multi-threaded multi-core system. 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 between transactions (conflict detection) and the control of which threads are allowed to commit their state and which threads need to be aborted and restarted. This is based on an observation of the intersection between the read and write sets of concurrent transactions.

Transactional computation, particularly in the database world, has been around for some time. Recent interest has centred on using transactions as a more general computational model and, in this environment, the degree of transactional interference is likely to be significantly greater than that encountered in database applications. Implementations of transactional systems are possible using a software layer on top of a conventional multi-core system, and a number of different proposals have been published [16, 10, 11, 6]. However, given the greater degree of interference, many believe that some hardware support for the required functionality is going to be necessary. Indeed the first commercial processors to support Transactional Memory, Azul's Vega [8] and Sun's ROCK [17], include such hardware.

One way to provide memory versioning and conflict detection, in hardware, is to extend existing cache coherence protocols. Memory writes can be held in a local cache and only updated to main memory when it is known that a transaction is being allowed to commit. In addition, the normal

snooping mechanism can be used to observe writes that interfere with data in a cache that is currently being used by another transaction. Proposals that use this approach differ in many details, but they share two weaknesses. The first is that any cache will have a limited size compared to the possible data set of any computation, and mechanisms must be devised to cope with cache overflows. The majority of current hardware TM (HTM) proposals assume that the data set of individual transactions will be sufficiently small that it will be possible to handle this by software without too great an impact on performance. The second weakness is that bus snooping is not extensible beyond a relatively small number of cores. Any mechanism that requires effectively instantaneous global observation of memory operations is not going to be practicable in the many-core era.

1.1 Addressing the Weaknesses

In recognition of the extensibility problem, the TCC approaches [9, 5] have proposed mechanisms that do not rely on conventional cache coherence. In its simplest form, TCC uses a core with local buffers to accumulate a transaction's writes. When a transaction commits, it broadcasts all its writes to all other cores in a single packet transmission. These cores must then check if any of the writes conflict with their data, aborting and restart if necessary. If they do not need to abort, they can use the contents of a broadcast packet to perform a coherence operation. The implementation can broadcast either both addresses and values of all updated store locations or addresses alone, allowing either an update or an invalidate protocol.

Because there is no need to arbitrate for communication of individual memory updates and the packet communication is uni-directional, the TCC approach should be more latency tolerant and thus permit implementations using a more extensible communication mechanism than a global bus. However, published evaluations so far have assumed a bus structure and observed that, in any case, the approach is still not scalable to a large numbers of cores [9]. A more recently published version of TCC [5] describes a way of overcoming this by using a directory based scheme similar to that employed by distributed shared memory systems. A number of distributed directories are employed, each tracking the usage of cache lines in a memory that is physically local to a core but globally addressable. Any request to cache a local copy of a line must access the directory and register its activity. Commits must go via the directory that is then responsible for sending appropriate messages to any sharers to perform the correct action on conflict. This approach, by avoiding global broadcast, is very effective in reducing the communication traffic but comes at the cost of both additional complexity and latency in servicing memory requests.

LogTM [14, 22] is a proposal for a HTM that uses a different approach to memory versioning. Instead of buffering new versions of data written by a transaction, they are updated directly to memory. A log is kept of replaced old values so that the original state of memory can be restored if the commit fails.

The motivation for this is twofold. Firstly, the assumption is that a transaction is more likely to succeed than fail, therefore it should be more efficient to assume that the new data values will be written and the old values discarded. Using the log minimizes the action needed on commit. The second advantage is concerned with the problem of limited sized buffers. Because the new values are written directly to memory, there is no limit to the size of a transaction's write set. The old values can be stored as address-value pairs in a linear buffer of arbitrary length and do not need to be accessed again unless the transaction is aborted, whereupon they will be scanned by software and reinstated.

A possible limitation of LogTM is that it assumes that aborts are infrequent and that the expensive operation of undoing conflicting memory operations will not be invoked often. A further limitation is that the log approach can only be used with early conflict detection, leading to the need to address livelock and/or deadlock problems.

We are particularly interested in TM as a programming model rather than simply a replacement for locks. The important difference is that a truly transactional program may manipulate large amounts of transactional data, have transactions that run for long periods and create significant amounts of conflict. We recently proposed a real transactional application, which implements Lee's routing algorithm [19, 2], and exhibits all of these properties. In these circumstances, at least one of the assumptions, on which most transactional proposals are based, is violated.

We therefore propose the first HTM system based on objects, with its most obvious advantage being an elegant solution to the problem of space virtualization without the need for additional software TM support. Recognizing the object structure also leads to novel hardware commit and conflict detection mechanisms.

The remaining sections are organized as follows. Section 2 describes the basis of the hardware support for objects. This is similar to paged virtual memory using virtually addressed caches and translation buffers between virtual and real addresses. The description of the proposed hardware also introduces the virtualization mechanism that provides support for object versioning. Having described the hardware version management, Section 3 presents how this comes together during transactional execution (e.g. detecting conflict between threads, how data is committed). Note that the proposed hardware facilitates the execution of object-based programs, which constitutes a large majority of newly developed applications, but the hardware does not

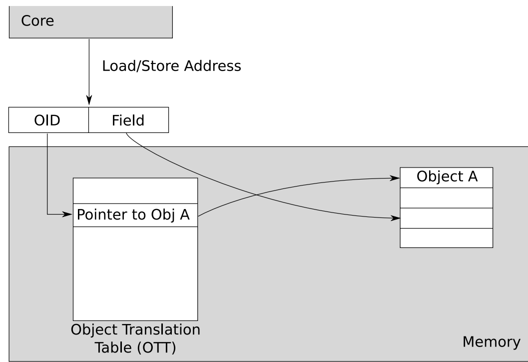

Figure 1. Hardware support for object addressing.

preclude the execution of classic programming languages, such as C or Fortran. Section 4 discusses some extensions that are needed to provide more complete support for transactional memory, but are not evaluated as part of this study. Sections 5 and 6 report the first insights into the scalability of our object-aware hardware TM system. Finally the paper is summarized in Section 7.

2 An Object-Aware Approach

We explore an approach that recognizes the structure of objects at the hardware level. The structure of data within objects makes it possible to implement lazy versioning without the problem of overflow. In addition, information about changes to fields within an object can potentially be communicated in a more concise form leading to lower communication bandwidth.

2.1 Object Caching

Schemes have been proposed to provide direct hardware support for Object-Oriented (OO) languages [20, 18, 21]. The motivation of these proposals was to ease the problems of memory management and garbage collection.

If an object is referred to by the core via an Object Identifier (OID) and field offset, which is translated using an Object Translation Table (OTT) that maps OIDs to memory addresses, then the relocation of objects during garbage collection becomes much simpler. It is necessary only to update a single entry in the OTT rather than all reference containing fields. This indirect object representation, although used in early implementations of OO languages, has generally been abandoned to avoid the inefficiency of an extra load during object accesses, but at the cost of an increase in garbage collection complexity. Figure 1 shows an outline of such a scheme. The inefficiency of an indirect

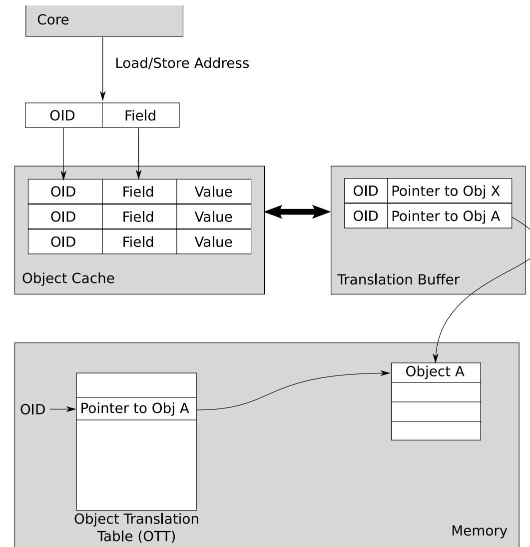

Figure 2. Caching object to address translations in a Translation Buffer allows direct access to recently used translations, reducing the requirements to access the OTT.

representation 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 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 incurring the penalty of two memory accesses. This latter inefficiency can largely be removed by the provision of a Translation Buffer that caches recently used translation table entries. The OID to memory address mappings can be accessed directly as shown in Figure 2 removing the need for translation via the OTT.

It should be clear that this approach is very similar to hardware support of paged virtual memory using a virtually addressed cache and a TLB. However, the optimum page size will be different and it is necessary to handle both small and large objects.

2.2 Transactional Object Caching

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 3. Assume the object cache and the Transaction Translation Buffer (TTB) are empty at the start of a transaction. The

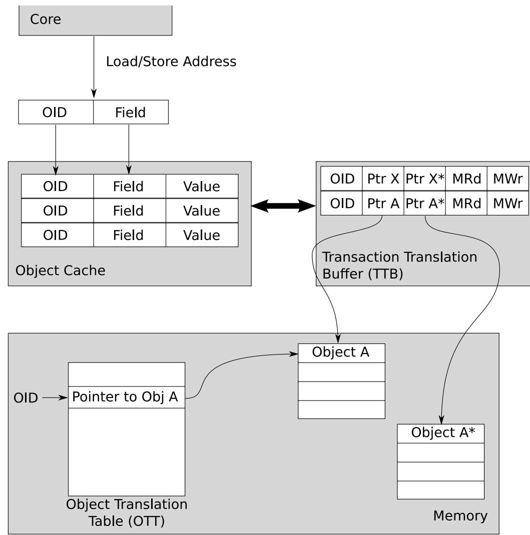

Figure 3. An additional pointer within the TTB allows transactional updates to seamlessly occur in isolation.

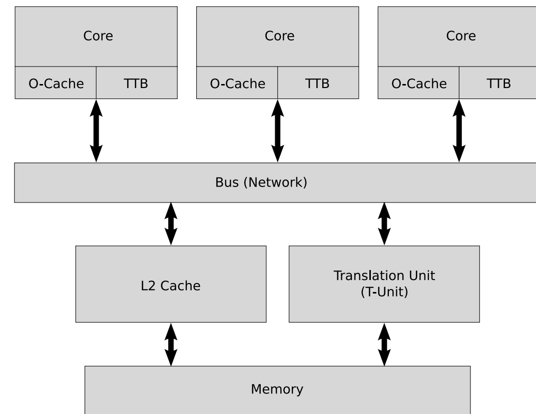

Figure 4. The basic system structure of an object-aware HTM. The T-Unit provides functionality associated with translation, cloning and committing of objects.

TTB lines also keep maps of the read set (MRd) and write set (MWr) of the copy. These can 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 entered into the TTB as the copy pointer (Ptr A*). On the first write a cache entry is made and thus any future reads and writes to that field will obtain the current local object cache value. A book-keeping entry is also made in the write set.

If, during the execution of a transaction, it is necessary to displace a modified object field from the cache, that field is evicted from the cache and written back into the object copy (Object A*) in memory. 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 direct hardware support for 'virtualization' of version information. This results from the object-aware scheme because we are dealing with objects of known size rather than arbitrary collections of store locations.

The next section will describe in more detail the system wide mechanisms for dealing with commits and transactional conflicts. However, at this point it is worth observing that, in order 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).
3. The pointer in the memory based OTT (Ptr A) must be replaced by the pointer to the copy (Ptr A*). It is worth noting here that, for a single object, this can readily be achieved as an atomic action as it is a single write operation.

There are a number of implementation possibilities for an object-aware system that are described in the following section.

3 An Object-Aware Hardware Transactional Memory System

Section 2 described the basic functionality of an individual core and transactional object cache system. This section considers the structure of a more complete system together with details of how transactional execution occurs. Figure 4 shows the basic structure of a multi-core system with a single Level 2 (L2) cache and memory unit. We will later discuss routes to extensibility. 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) that provides a number of functions associated with the translation of object addresses, object cloning and committing.

The movement of functionality in the T-Unit, away from the core and toward the memory system, is a deliberate exploitation of the principle of 'intelligent memory'. This

becomes both feasible and sensible as we move to higher levels of integration. It is beyond the scope of this paper to consider how conventional virtual memory fits within the overall system. For the purposes of this evaluation the memory address space is real.

In a full implementation we would expect to provide for both object-aware and non-object-aware caching and memory access. However, to simplify the description we will assume an 'all transactions all of the time' mode of operation, as in TCC [9].

3.1 Transactional Memory Access

An implementation in its simplest form assumes that a core will flush its caches so that dirty objects are not present at the start of a transaction. On accessing an object for the first time, a cache miss will occur in both the object cache and the TTB. An access to the T-unit is required that will look up the OID in the OTT and provide the corresponding memory address. An entry will be made into the TTB with the read and write set bitmaps cleared. When an object is modified the changes are made in the cache, and the write set is updated accordingly. Should a modified line overflow the object cache it is necessary to allocate memory space to hold the transaction's modified copy. This could be implemented in software, but we assume the T-Unit provides this function. A request is therefore sent to the T-Unit to allocate space for a copy of the committed object, and the evicted line is written to the allocated space. The copy address is returned and stored in the TTB.

Store accesses will then continue during the transaction with cache entries being made and cache overflows being handled as described in Section 2.2.

3.2 Transaction Commit

Assume that the transaction completes without conflict. The transaction first flushes any modified lines from its object cache into the allocated copy space. Note that, at this time, the commit has not completed and there is no need to prevent other transactions 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, it should be noted that this does not prevent access to the L2 cache or memory, which can occur from a core that is using a locally cached translation. If software wished to reduce the time spent copying unmodified fields, it can prefetch the fields prior to transaction commit using conventional loads. However, locking of the T-Unit without prefetching is believed to be an efficient solution for a single memory unit configuration. Prefetching would increase an object's read set and the associated risk of aborting due to a conflict. A more complex distributed protocol is required for extensible systems but a description is beyond the scope of this paper.

3.3 Conflict Detection

It is obviously necessary to detect overlaps between the write sets and read sets of transactions to detect transactional conflict. We have chosen to implement lazy detection as this is more compatible with our aim to support a highly extensible communication structure. It also avoids problems of livelock.

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 sets 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 that broadcasts all addresses that 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 would be to attach a directory to the translation unit to keep track of object sharing. It may also be possible to determine object sharing in software to avoid unnecessary communication. We are exploring these alternatives.

3.4 Object and Cache Line Size

It is necessary to define a basic size for the objects manipulated by the system. Although it would be possible to use multiple sizes, we currently use only one. Statistics from OO programs suggest that the average object size is of the order of hundreds of bytes [21]; currently we use a basic object size of 128 bytes.

If an object is smaller than this then we will waste virtual object identifier space. However, this does not result in a waste of real address space as appropriate sized units are allocated and manipulated. If an object is larger than 128

bytes we simply allocate contiguous OIDs although only the first will contain an object header. This allows normal access to indexed objects, such as arrays, although the physical memory allocated does not need to be contiguous and the 128 byte objects will appear separate for the purpose of transactional operation.

The object cache and system as described so far has assumed that each field appears as a separate entry in the cache. In practice, it is possible to use the common mechanism of allocating multiple entities to a cache line, saving tag space at the expense of the possibility of false sharing. This will appear in the transactional model as false conflict. Currently we are using a 32 byte quantity that is regarded as eight 32 bit fields. The read and write set maps in the TTB are also held at a 32 byte unit level, and hence conflicts are detected at the granularity of cache lines.

Clearly there are many options as to size and exact organization of objects, cache entries and maps. Extended studies are required to determine optimum values.

4 Other Design Issues and Optimizations

Similar to the earlier versions of other HTM systems [14, 5, 9], in this initial implementation of the object-aware HTM we do not allow transaction suspension, migration or context switches, however, in this section we discuss details of how these concepts can be achieved.

4.1 Self Validation

The OO structure makes it relatively straightforward for a transaction to validate itself at any point. Its TTB holds information on the objects in its read set and the address of the object in memory. If a transaction queries the T-Unit, it can compare its view of the address of the object with that stored in the OTT. If they are different then the object has been committed since the transaction started and it should be aborted. However, this is detecting conflict at the object level and cannot easily be optimized with bitmaps to refine the granularity to check at the field level, as in a broadcast scheme, to avoid false sharing. For this reason, we regard the broadcast scheme as the primary conflict detection mechanism. However, there may be circumstances where self validation is useful. For example, when transactions are suspended and later resumed. An object level validation on resumption invoked by the runtime environment can ensure correct operation but at the cost of an increased possibility of aborting. Several alternatives are available to avoid pitfalls with real addresses allocated to objects ‘wrapping around’ ranging from simple, abort transactions upon transaction commit generating wrap around, to more elaborate ones involving extra bits of ‘version number’ combined with addresses.

4.2 Transaction Suspension and Migration

Any practical transactional scheme will need to deal with the possibility that the transactional thread can be suspended in mid execution. In addition, it is almost certainly desirable in a multi-core system to allow threads to migrate across physical resources, e.g. to suspend on one core and resume on another. It is always possible to abort and restart but this is likely to severely hinder forward progress in a system with long running transactions. The saving of transactional state and subsequent resumption can be a problem for HTM systems. However, in an object-aware HTM scheme, it is always possible to flush the data to object copies at any point. As long as we maintain the TTB state, the transaction can be restarted. As discussed above, it may be necessary for the transaction to validate itself before proceeding.

4.3 TTB Overflow

As well as overflows in the data cache, it is also possible that the TTB may overflow. Although the TTB can be relatively large and slow, as it is only accessed on a cache miss in a similar manner to virtual cache TLBs [4], overflows will still occur. As the TTB is the place that holds information local to an executing transaction, this is a potential problem. Specifically, a TTB entry holds the pointer to an object in memory, read and write maps for the object and a pointer to a temporary object copy if a write to the object has occurred.

Considering the function of an individual TTB entry, the pointer to the object is held only as an optimization to avoid accessing the object table for each field access. This can be fetched again, from the T-Unit, if needed in the future. The read and write maps are optimizations that allow the granularity of conflict to be observed at field level but can be discarded at the expense of occasional false conflicts. However, the pointer to an object copy must be remembered if a TTB entry is displaced otherwise it will not be available either for future speculative object writes or to replace the original object during commit. One way to do this is to extend the TTB into memory using a hash table or software routines. However, we must also consider the content of the TTB as a whole. It contains, at object level, the complete read and write set of the transaction. The complete write set must be available to broadcast and the complete read set must be available so that it can be compared against any other broadcast write sets.

The simplest way to do this is to keep overflow bits associated with each TTB line. These can be separated into read and write overflows allowing a pessimistic approximation to the read and write sets to be constructed. This is in fact a

Feature	Description
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.
T-Unit	4MB, shared, 32-way assoc., 12B lines, 16-cycle access.
Memory	100-200 cycle off-chip access.

Table 1. Simulation parameters.

Benchmark	Parameters
Kmeans-Low	-m40 -n20 -t0.05 -i random1000_12
Kmeans-High	-m20 -n20 -t0.05 -i random1000_12
Genome	-g256 -s16 -m16384
Vacation-Low	-n4 -q10 -u80 -r65536 -t4096
Vacation-High	-n8 -q10 -u80 -r65536 -t4096
Lee-TM-t	75×75×2 grid, 481 routes
Lee-TM-ter	75×75×2 grid, 481 routes

Table 2. Parameters for the benchmarks.

form of Bloom filter [3] that can be integrated with the TTB structure. The exact amount of information that needs to be kept depends on the frequency of overflow. The evaluation of this mechanism is beyond the scope of this paper.

5 Methodology

To evaluate our object-aware HTM system a prototype platform has been developed, comprising an event-driven simulator and an associated static Java compiler and runtime system. We exercise the hardware using applications derived from the STAMP benchmark suite [13] and a transactional version of Lee's routing algorithm [19, 2]; Lee-TM.

5.1 Simulation Platform and Java Runtime

As in related HTM studies [13, 14], we opt for an event driven simulation platform with an IPC of 1 for all but memory operations, observing that transactional performance is essentially memory bound. 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.

In addition to the simulation platform a static Java compiler and runtime system has been implemented. The compiler is conventional; Java source is compiled into Java bytecode, translated into machine code and then linked into an executable alongside the runtime system code. The static runtime system has been extended to support the T-Unit for allocation of objects. In the current system objects are allocated in a reserved, high-addressed, region of the address space.

5.2 Benchmark Applications

We exercise our transactional system by executing three applications (Kmeans, Genome and Vacation) derived from the STAMP benchmark suite and Lee-TM. For the purposes of this study the STAMP benchmarks have been implemented in Java by converting C structs into Java objects, and inserting calls to the runtime system in order to start, end and abort transactions. It should be noted that no efforts have been made to change or optimize the structures from the original C version. The outputs of the benchmarks are verified against the C counterparts. When available the benchmark's self verification test has also been ported.

Table 2 presents the parameters used for the benchmarks. For Kmeans and Vacation we evaluate both high- and low-contention versions. For Lee-TM we evaluate the transactional implementation Lee-TM-t and also Lee-TM-ter, an early-release implementation.

6 Evaluation

This preliminary evaluation focuses on the scaling characteristics of the object-aware HTM system to investigate its feasibility.

6.1 Transaction Profiles

Understanding the performance of a TM system requires knowledge of the selected applications used for evaluation. In Table 3 we present transaction profile statistics for the seven applications used. The transactions range in size from a few hundred to a few hundred thousand instructions. Larger transactions help to amortise the transaction start and commit overheads but also generate larger read and write sets that must be isolated from global state prior to committing. We include both the arithmetic mean and coefficient of variance (COV)[1] for read sets, write sets and instructions per transaction. COV provides a comparison of the variation in the transactions compared to the mean. For Lee-TM in particular there is a significant variation in the length of transactions and hence the size of the working sets. Lee-TM-t, Lee-TM-ter and Vacation create significantly large working sets that they overflow the 32KB L1 object cache.

6.2 Performance Analysis

The graph, Figure 5, presents the speedups achieved from execution of each benchmark on the object-aware HTM system when scaling from 1 to 32 processors. The

[1] COV is calculated by dividing the standard deviation by the arithmetic mean, and produces a dimensionless value that can be compared across distributions.

Application	#Tx	#Insts	Object readset/Tx		Object writeset/Tx		Inst/Tx		Overflow	
			Mean	COV	Mean	COV	Mean	COV	Txs	Lines
Kmeans-Low	6695	2210121	7.4	0.21	3.5	0.29	330.1	0.54	0	0
Kmeans-High	6695	2210110	7.4	0.21	3.5	0.29	330.1	0.54	0	0
Genome	6596	33859054	44.7	0.85	8.5	0.53	5133.3	1.39	0	0
Vacation-Low	4099	149656758	203.1	0.32	108.6	0.30	36510.6	0.35	176	224
Vacation-High	4099	149656758	266.8	0.40	146.0	0.37	39893.6	0.41	561	1068
Lee-TM-t	1447	540904041	117.5	2.87	78.8	3.07	373810.671	4.12	287	407255
Lee-TM-ter	1447	540908851	15.8	1.35	78.8	3.07	373813.995	4.12	291	407683

Table 3. Transaction profile statistics for all benchmarks.

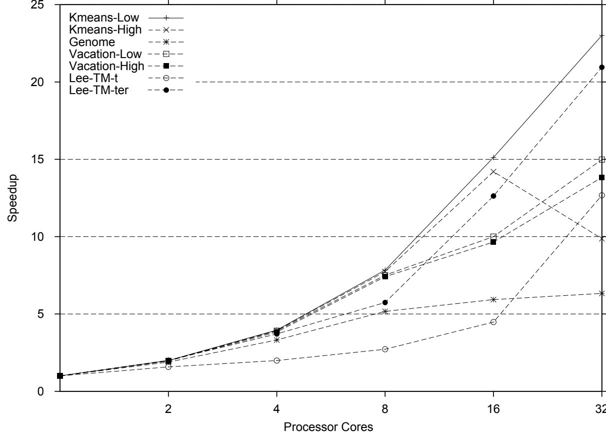

Figure 5. Speedups over sequential code for the object-aware HTM system.

maximum speedups range from 6.33 times for Genome upto 23.01 times for Kmeans-Low. All applications scale to 16 processor cores (arithmetic mean speedup of 10.29), while Genome speedup begins to tail-off between 8 and 16 cores. One result that is immediately noticeable from the graph is that the performance of Kmeans-High for 32 cores dramatically drops. This is an artificial limit imposed by the application parameters. On 32 cores, 32 concurrent transactions attempt insertions (modifications) into 20 clusters (shared objects) on each iteration, so at least 12 have to abort. Such scenarios are ideal candidates for adaptive concurrency control [1].

The scaling characteristics of the object-aware HTM using STAMP benchmarks are similar to those TM systems presented in [13]; i.e. where the STAMP suite was introduced.

A breakdown of the total execution time is shown for all benchmarks in Figure 6. This time is composed of *idle* cycles (due to work imbalance), *commit* cycles (the commit overhead), *violations* cycles (time spent executing within aborted transactions) and finally *busy* cycles (the useful committed work). A greater proporion of time busy is better. The cause of the drop off in performance in Kmeans-High is clearly visible as the number of busy cycles remains constant from 16 to 32 cores but the time spent in aborted transactions increases, from 1% at 16 cores upto 23% at 32 cores. Genome is the only benchmark to exhibit a significant amount of work imbalance, 22% at 32 cores, an artifact of the amount of barrier synchronization within the application.

Lee-TM-t shows a significant amount of violations, with aborted transactions accounting for 25 to 73% of the total execution time for 2 and 32 cores respectively. These violations are due to false conflicts within the read set when observed at the algorithmic level (as explained in [19]). One of the phases of Lee-TM-t reads a large number of elements into the read set, accounting for the high average of 117.5 objects, most of which are discarded during the immediate phase of the transaction. We use this observation in the Lee-TM-ter version of the application to replace all those reads that will be discarded with non-transactional reads, essentially 'early-release' [11]. This algorithmic change reduces the average read set to 15.8 objects, and increases the speedup at 32 cores from 12.67 to 20.95 times.

6.3 Advantages of Object-Aware HTM

One of the advantages of the object aware approach is virtualization of version management. Lee-TM-t, Lee-TM-ter and Vacation all overflow the 32KB L1 cache and it can be assumed that the working set of some transactions will always overflow the provided cache resources. As transactions are allocated a separate copy of any object that they modify, overflows of modified data from the object cache are handled by line evictions as in a regular cache architecture and similarly the impact on execution performance in the common case is insignificant.

Figure 7 shows the composition of bus traffic during execution. 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 cur-

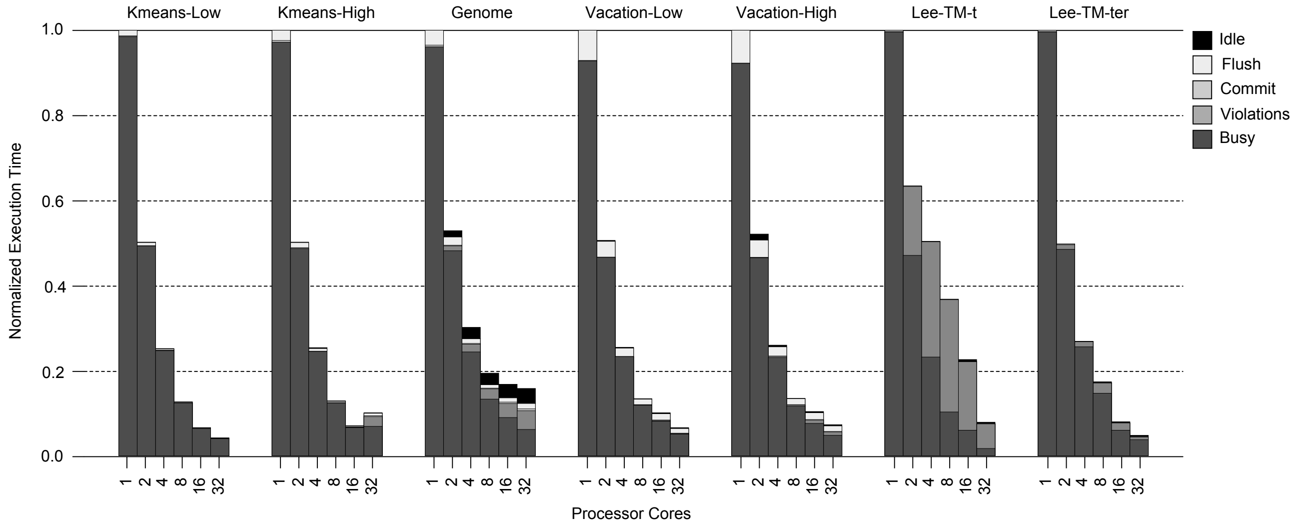

Figure 6. Composition of execution time during transactional execution.

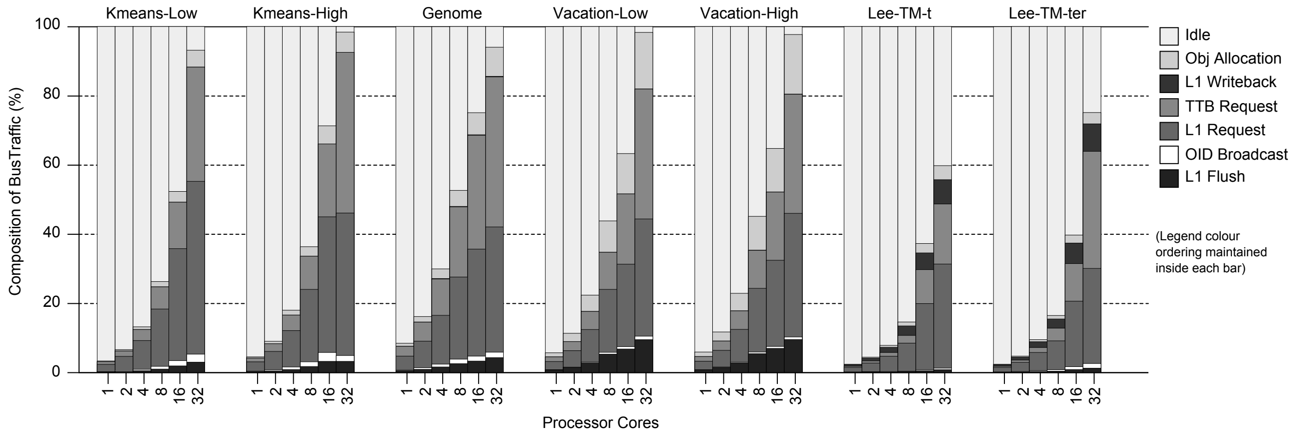

Figure 7. Composition of bus traffic during application execution.

rent preliminary implementation, as any requests to the T-Unit while locked triggers continual retries until the T-Unit is unlocked. The retries can easily be eliminated by using 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 writeset 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. As the object-aware HTM only bradcasts OIDs and the writeset map rather then the cache lines, the size of the data that needs to be broadcast in object-aware HTM can be considerably smaller as compared to other HTM systems when dealing with transactions involving large objects. One example is Lee-TM where the number of OIDs broadcast is less than 2.5 times the number of cache lines that need to be broadcast. The maximum saving is bound by the basic object size, 128 bytes in our evaluation.

7 Summary

One way to provide memory versioning and conflict detection in HTM is to extend existing cache coherence protocols. The majority of proposed HTMs following this approach have one fundamental weakness, rooted on the assumption that the data set of individual transactions will be sufficiently small that it will be possible to handle overflows in software, without too great an impact on performance.

This paper has described the first HTM where the object structure is recognized and harnessed to solve this weak-

ness. Our approach is very similar to hardware support of paged virtual memory using a virtually addressed cache and a TLB. Objects are accessed through OIDs and field offsets rather than memory addresses. To avoid double indirection for each object access, object caches and TTBs have been introduced. Furthermore the TTB (similar to a TLB) together with T-Units allow overflows from the object cache and enable a novel commit and conflict detection mechanism. Both Lee-TM and Vacation benchmarks have exhibited overflows that previously would have had to be handled by software with an associated great impact on performance. The initial evaluation has shown, through simulation, that an object-aware HTM allows an elegant solution to the problem of cache overflow within a transaction. It has provided an insight into the scalability characteristics of the object-aware HTM. The broadcast of OIDs and write sets accounts for less than 5% of bus bandwidth showing the potential of the proposed object-aware HTM.

8 Acknowledgements

This work has been supported by the EPSRC grant EP/E036368/1.

References

[1] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. Adaptive Concurrency Control for Transactional Memory. In *First Workshop on Programmability Issues for Multi-Core Computers*, January 2008.

[2] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Luján, and K. Jarvis. Lee-TM: A non-trivial benchmark suite for transactional memory. In *Proceedings of the 8th International Conference on Algorithms and Architectures for Parallel Processing, ICA3PP*, volume 5022 of *Lecture Notes in Computer Science*, pages 196–207, 2008.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[4] M. Cekleov and M. Dubois. Virtual-address caches. Part 1: problems and solutions in uniprocessors. *IEEE Micro*, 17(5):64–71, 1997.

[5] H. Chafi, J. Casper, B. Carlstrom, A. McDonald, C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A Scalable, Non-blocking Approach to Transactional Memory. In *Proceedings of the 13th Annual International Symposium on High Performance Computer Architecture*, pages 97–108, 2007.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the 20th Intl. Symposium on Distributed Computing*, Sept 2006.

[7] M. Flynn and P. Hung. Microprocessor design issues: thoughts on the road ahead. *IEEE Micro*, 25(3):16–31, 2005.

[8] B. Goetz. Optimistic Thread Concurrency: Breaking the Scale Barrier. Azul Systems Whitepaper, `http://www.azulsystems.com/products/whitepapers.htm`, January 2006.

[9] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotum. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 102–113, 2004.

[10] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, 2003.

[11] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 92–101, 2003.

[12] D. Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, 30(9):37–39, 1997.

[13] C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 69–80, 2007.

[14] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th Annual International Symposium on High Performance Computer Architecture*, pages 258–269, 2006.

[15] Semiconductor Industry Association. *The International Technology Roadmap for Semiconductors*, 2005.

[16] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213. 1995.

[17] M. Tremblay and S. Chaudhry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread CMT SPARC processor. *IEEE International Solid-State Circuits Conference*, pages 3–4, 2008.

[18] N. Vijaykrishnan, N. Ranganathan, and R. Gadekarla. Object-Oriented Architectural Support for a Java Processor. *Proceedings of the 12th European Conference on Object Oriented Programming*, pages 330–354, 1998.

[19] I. Watson, C. Kirkham, and M. Luján. A study of a transactional parallel routing algorithm. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 388–398, 2007.

[20] I. Williams. *Object-Based Memory Architecture*. PhD thesis, Department of Computer Science, University of Manchester, 1989.

[21] G. Wright, M. Seidl, and M. Wolczko. An Object-aware Memory Architecture. *Science of Computer Programming*, 62(2):145–163, 2006.

[22] L. Yen, J. Bobba, M. Marty, K. Moore, H. Volos, M. Hill, M. Swift, and D. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th Annual International Symposium on High Performance Computer Architecture*, pages 261–272, 2007.