Comput Syst Sci & Eng (2009) 5: 303–315

International Journal of
Computer Systems
Science & Engineering

Exploiting object structure in hardware transactional memory

Behram Khan, Matthew Horsnell, Ian Rogers, Mikel Luján, Andrew Dinn and Ian Watson

Advanced Processor Technologies Group, The University of Manchester, United Kingdom.

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 synchronisation and the broadcast of data. By increasing the granularity of when synchronisation 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 recognised 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, 5, 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, 8, 9, 4]. 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 [6] 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 [7, 3] 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 [7]. A more recently published version of TCC [3] 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, 23] 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 minimises 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 live-lock 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], 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 object-aware hardware transactional memory system (OHTM) that recognises the object structure at the hardware level. Its most obvious advantage is an elegant solution to the problem of space virtualization without the need for additional software TM support. Recognising the object structure also leads to novel hardware commit and conflict detection mechanisms.

The remaining sections are organised 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 and conflict detection mechanism between transactions at the object and sub-object level.

Having described the hardware version management and conflict detection mechanism, Section 3 presents how this comes together during transactional execution. 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 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. Section 5 explains the methodology used in order to evaluate OHTM and section 6 presents the evaluation of the complete HTM system. Finally the paper is summarised in Section 7.

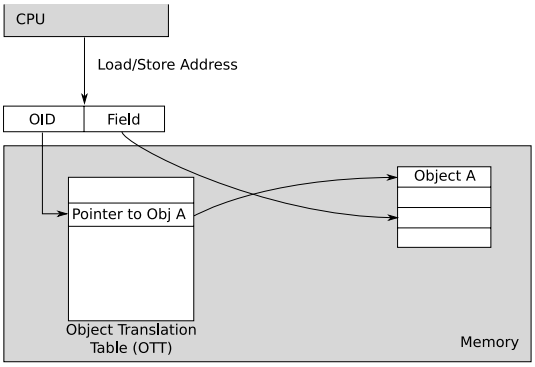

Figure 1 Hardware support for object addressing.

2. TRANSACTIONAL MEMORY AND AN OBJECT-AWARE APPROACH

In this section we explore an approach that recognises the structure of objects at the hardware level and how this approach is employed in order to devise version management and conflict detection mechanism in OHTM. 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 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

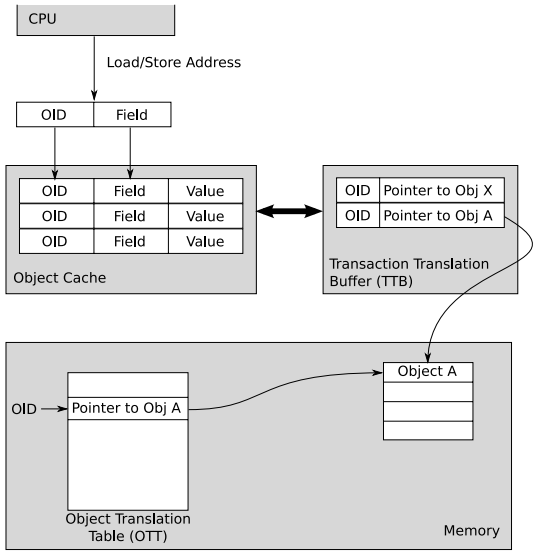

Figure 2 Caching object to address translations in a Translation Buffer allows direct access to recently used translations, reducing the requirements to access the OTT.

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 Object size

It is necessary to define a basic size for the objects manipulated by the system. Size of the objects may have an effect on the overall performance depending on the applications [11]. 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 then 128 bytes 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 then 128 bytes, we simply allocate contiguous OIDs although only 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 bytes objects will appear separate for the purpose of transactional operation.

2.3 Version management

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. Reads of object fields occur via normal translation to the committed object, the translation is cached in the TTB and the field data is cached in the object cache. If a write occurs to the field of

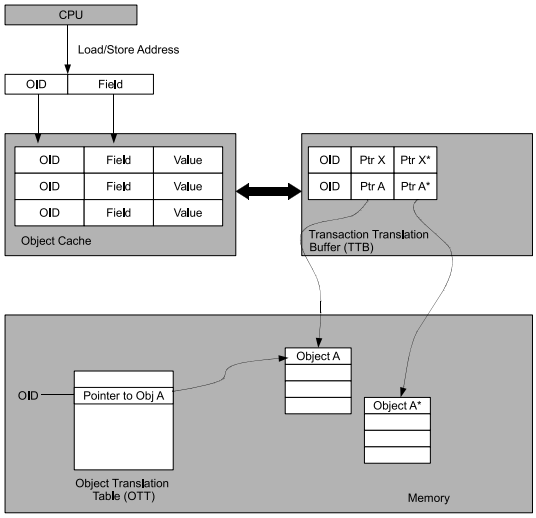

Figure 3 An additional pointer within the TTB allows transactional updates to seamlessly occur in isolation.

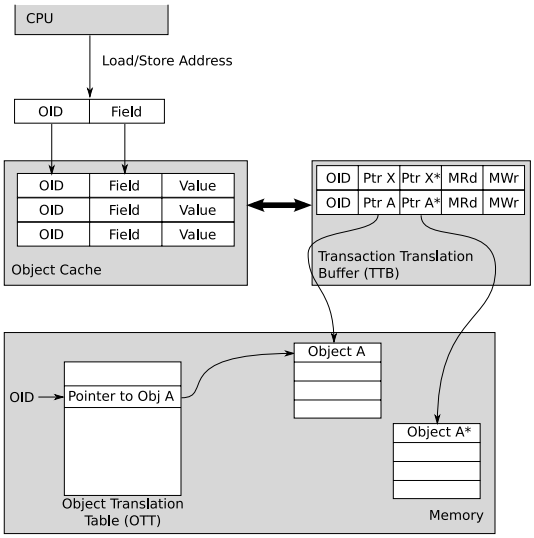

Figure 4 An additional clone pointer along with read and write bitmaps within the TTB.

an object in the object cache, then the object cache line that contains the field is updated without any further action. If the object cache overflows during the transaction and the modified field now needs to be written back, then this write back triggers a *generate clone* event. The modified field cannot be written back to the committed object while the transaction is in progress as this will make the transaction's modifications visible to all the other transactions, thus violating the isolation property. In order to maintain isolation it is necessary to allocate memory space to hold transaction's modified data. A generate clone event generates a clone (Object A*) by copying actual object into the clone space and the address of this clone space is returned and entered into the TTB as the clone pointer (Ptr A*). The evicted object field can now be written to the clone object. Any future reads and writes to that object will be made using the local object pointer.

The TTB lines can also keep maps of read set (MRd) and write set (MWr) within the object as shown is Figure 4. The TTB here is an extention of the TTB shown in figure 3, as it contains two additional fields: read and write bitmaps. These can be single bit per field of the object or can represent multiple consecutive fields within the object. In OHTM, a single bit of read/write bitmap represents 32 bytes (object cache line size) and the size of the bitmaps is 4 bits. As a result the read/write bitmaps can cover basic object size of 128 bytes. Larger object will span multiple TTB lines. The use of bitmaps can reduce the probability of false conflicts between transactions.

Figure 5 show a simple example of version management in OHTM using read/write bitmaps. Lets assume that at the start of the transaction the object cache and the transaction translation buffer are empty and that the size of the object (Object A) used is less then 128 bytes. For clarity many of the hardware details are not shown in this example. This section only explains the version management mechanism of OHTM using objects. The details of the complete transaction system and its working is explained in the later sections.

In step 1, transaction (T1) loads object A, with OID oA. As oA is not present in T1's TTB, the TTB requests for the translation to physical address, pA. Upon receiving the translation the OID, oA and the physical address pA are installed in the TTB. T1 subsequently sends a read message to a field of object A. On completion of load message read bitmap is set in the TTB to indicate which 32 byte portion of the object A has been read by the transaction.

In step 2, T1 writes to the same field of the object A and this time the write bitmap of the TTB entry is updated.

In step 3, the object cache overflows and the modified field now needs to be written back which triggers generate clone event. A clone space is allocated in the memory (Object A*) and the modified line is written to that clone. Note that generate clone event is different when using read/write bitmaps as compared to generate clone event without using the bitmaps. In case of read/write bitmaps, the clone event does not copy the entire committed object to the newly allocated clone space. The generate clone event only allocates space for the object and writes back only the modified object cache line into that allocated space. Once created the physical address, pA' of the clone is installed in the TTB. At this point if T1 generates a load instruction for the object A's field that had overflowed from the object cache, the decision whether to read the field from the original object or the copy can be made from a simple observation of write bitmaps.

The description of the more detailed system wide mechanism for dealing with commits and transactional conflicts will be explained later. However, at this point it is worth observing that, in order to commit an object, three steps are required.

1. Any field in the write set of the transaction must be written from the cache to the object copy (Object A*).

2. Any unmodified fields should be copied into the object copy (Object A*) from the committed object (Object A). Note that this step is required in case of read/write bitmaps only, where the generate clone event only allocates space to the clone object and does not copy the entire original object into the clone space. In cases where the original object is copied into the clone space during the generate clone event, this phase is no required. Different configurations of generate clone event and their use will be explained later.

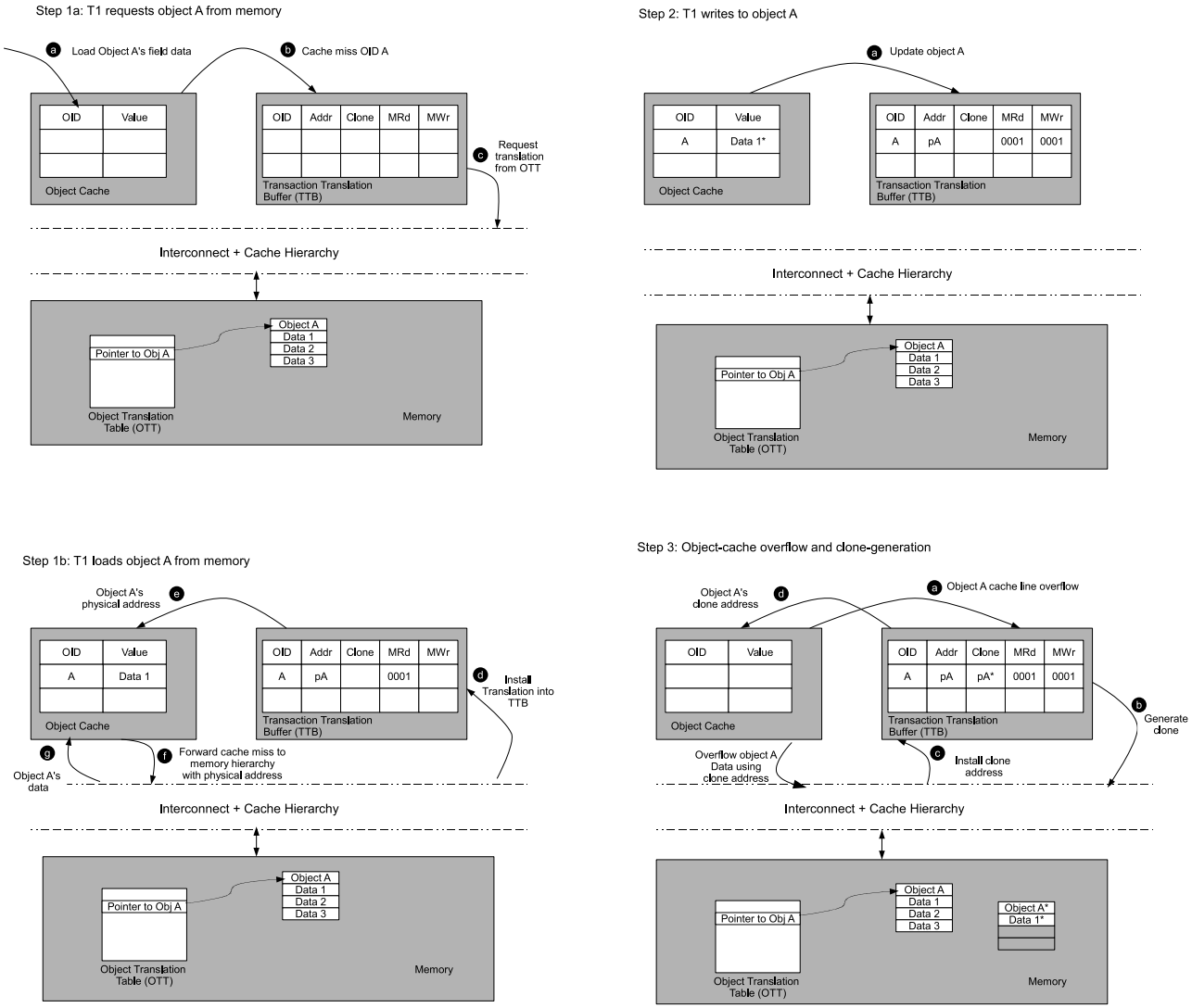

Figure 5 Simple Memory Versioning Example: refer to Section 2.3 for a detailed narrative.

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.

2.4 Conflict detection

It is obviously necessary to detect overlaps between the write sets and the read sets of transactions to detect conflicts. OHTM implements lazy conflict detection as this is more compatible with our aim to support a highly extensible communication structure. There are two ways in which OHTM detect conflict between transactions: conflict at *object level* and conflict at *sub-object level*.

At object level the conflict is detected between transactions by detecting overlaps between read and write set at the OID level. When a transaction wants to commit, it broadcasts all the OIDs of the objects it has written to (the write set of a transaction) to all the other transactions. Observing transactions compare their read sets (the OIDs of the objects they have read from) with the write set of the broadcasting transaction. Any overlap will cause an abort and restart in the observing transaction. The conflict in this case is detected at the object level. Any concurrent transactions that read and write to that same object are conflicting.

One of the issues with object level conflict detection is the increases in the chances of transactions being aborted due to false conflicts. In order to address this issue read/write bitmaps are introduced in OHTM to be able to detect conflicts at sub-object level. At sub-object level conflicts between transactions is detecting by detecting overlaps between the read and write bitmaps of the conflicting transactions. In this case, the committing transaction broadcasts all the OIDs along with their write bitmaps to all the other transactions. Observing transactions then compare their read sets, which includes the OIDs along with their read bitmaps with the write set of the broadcasting transaction. With the introduction of bitmaps we can detect conflicts between transactions at the cache line level (32 bytes) rather then at the object level (128 bytes), thus reducing the possibility of false conflicts.

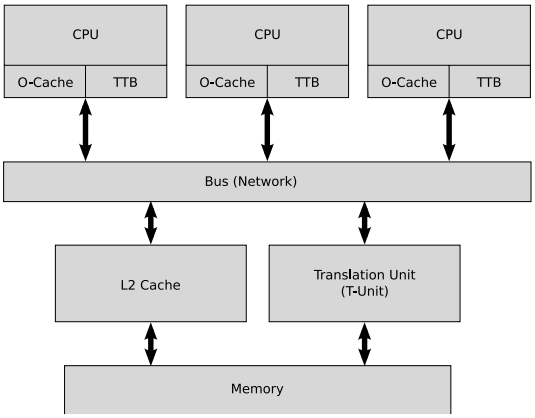

Figure 6 The basic system structure of an object-aware HTM. The T-Unit provides functionality associated with translation, cloning and committing of objects.

3. DESIGN OF COMPLETE OHTM

This section describes hardware structure of OHTM together with the details of how transaction execution takes place using novel object-aware transactional protocols.

3.1 Hardware outline

Figure 6 and 7 show the basic structure of a multi-core OHTM with a single level2(L2) cache and a memory unit. We will later discuss the routes to extensibility. The 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. There are two components that warrant special attention: the transaction translation buffer (TTB), and the translation-unit (T-Unit).

3.2 Transaction translation buffer

In essence the TTB fulfils a similar role to that of a TLB: it caches a number of recent translations, in this case from OID to physical addresses. The TTB is located alongside the L1 object cache and is accessed when object cache lookup misses. To service a cache miss a physical address must be generated and sent on to the memory hierarchy. Either a translation is present in the TTB in which case the object cache request is forwarded immediately using the translation or a translation must be requested from the T-Unit, which will in turn request the OTT if it also contains no valid translation. The TTB also holds translations from OID to physical address of clone objects. Additionally the TTB can hold read and write bitmaps to reduce the number of false conflicts by allowing conflict resolution at a sub-object level.

3.3 Translation-unit (T-unit)

The T-Unit is a global resource in the same way as L2 cache is a global resource and is shared by all cores. T-Unit is essentially

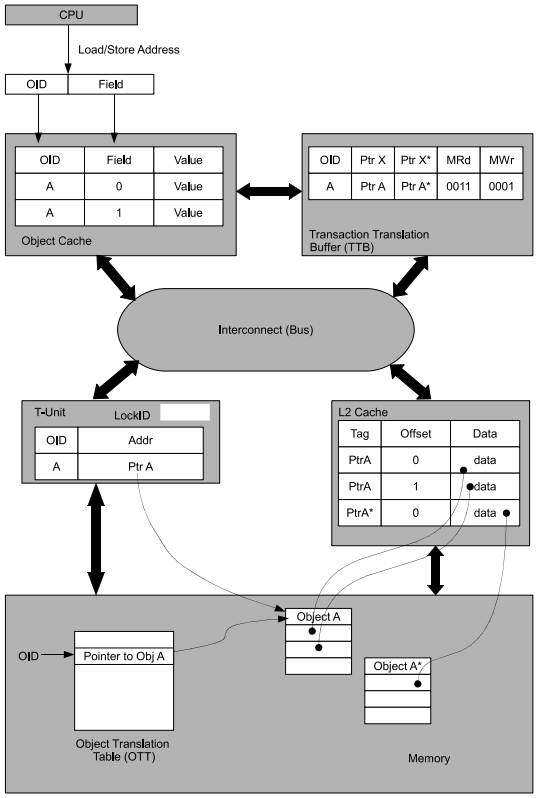

Figure 7 The detailed structure of an object-aware HTM.

a large cache for translations of OIDs to physical addresses, reducing the number of translations that need to be fetched from memory controller or via a software interrupt routine. The T-Unit however does not contain clone translations, as clones are private and unique to each transaction until they are committed.

3.4 Object-aware HTM protocol outline

In this section we describe in detail, the HTM protocol and the commit phase in OHTM. Prior to committing, the transactions run in isolation as described in section 2.3. When a transaction is ready to commit it follows a 6 phase commit protocol.

- **Phase 1: Flush** When a transaction starts the commit process, it first flushes any modified lines from its object-cache into the allocated clone space. Note that at this time, the commit has not completed and there is no need to prevent other transactions from making progress. The clone space is private to a transaction so this process will neither abort another transaction nor will it effect consistency, as no other transaction will read from these unique clones addresses.
- **Phase 2: Lock T-Unit** Once the modified data has been flushed from the cache, a request is made to lock the T-Unit. Once T-Unit is locked, it is no longer accessible by other transactions attempting to fetch the translation. Any such request will receive a retry response. Although a locked T-Unit cannot be accessed by other transactions, it should be noted that this does not prevent access to L2 cache or to the memory, which can occur from a transactions using their locally cached translation in their private TTB.

Once a transaction reaches this phase, it can no longer be aborted. After a transaction acquires the T-Unit lock, it is guaranteed to complete sucesfully.

- **Phase 3: Complete Partial Objects** If OHTM is using bitmaps within the TTB to reduce false conflict between transactions, then at this stage all the incomplete clone objects created during the generate clone event must be completed. This involves committing transaction copying any unmodified fields from the current committed objects into the transactions copy object. 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 single memory unit configuration. Prefetching would increase an object's read set and the associated risk of aborting due to a conflict.

  If the protocol is resolving conflicts at the OID granularity then the clone is created with a complete copy of the original object and the commit process in that case does not include this phase.

- **Phase 4: Broadcast** During this phase all modified objects (OID, optionally bitmaps and new translations) are broadcasted to all the other cores in the system. If a separately running transaction's read set is violated during this phase it needs to abort and restart.

  Conflict detection can take place at the object level which will require broadcasting of OID only or at the sub-object level which will require broadcasting OID along with write bitmaps.

- **Phase 5: Update Translations** Having successfully informed all the other transactions of the modifications that will be made, the new translations are installed in the T-Unit. At this stage the new translations become visible to other transactions. Updating the translations in the T-Unit is not atomic but it is safe, as T-Unit is locked and is not accessible to other transactions. As a result the translation updates seem to be atomic to the rest of the transactions.

  It should be noted that phase 3,4 and 5 can be combined and the protocol will still function correctly. After phase 2 the transaction cannot be aborted and therefore we can combine some of the later phases if it gives us performance advantages.

- **Phase 6: Clear Readset** The final phase clears the read/write sets of the transactions in the TTB. The TTB can now hold the read/write set of new transactions that may start on this core. Unlock message is sent to the T-Unit so that it can be accessed by all the other transactions.

3.5 Simple commit example

Figure 8 show a simple example of two transactions, T1 and T2. T1 commits successfully while T2 is aborted and subsequently restarts. In this example we assume that at the start of the transactions the object caches and the TTBs are empty, and that violations are resolved at the object level detected by conflicting object identifiers (OID). For clarity the object caches and the L2 cache is omitted from the figures.

In step 1, T1 generates a load instruction (address containing OID and offset) for a field in an object A and accesses object cache for the data. As object cache is empty, a cache miss will occur which will trigger access to TTB in order to load physical address of the object. As oA is not present in T1's TTB, an access to T-Unit is required that will look up the OTT and provide the corresponding physical address. The T-Unit returns the translation to a physical address, pA, which is installed inside the T1's TTB. T1 subsequently sends a message to the relevant L2 cache to load the data using physical address pA. Concurrently T2 loads object oB and its physical address pB is installed into its TTB.

In step 2, T1 speculatively stores to oA. Assuming the data is present in the object cache, the store instruction will update the field in the object cache without any further action. Note that the data is within the cache, so a store will not trigger the generate clone event. Concurrently T2 loads object oA, following same procedure as other loads.

In step 3, T1's object cache overflows and needs to write back the modified data of oA. In order to maintain isolation, this write back will trigger the generate clone event. We will assume that the clones are generated by an intelligent memory controller by copying actual object oA into a clone space oA'. Once created the physical address, pA', of the clone is returned, via T-unit to T1 and installed inside its TTB.

In step 4, T1 begins the commit phase, which we donate as com[1-6]. We will assume T2 is significantly longer than T1 and continues transactional execution. T1 enters the first phase of commit (com1), flush, and writes back the modified lines in its object cache to the clone object oA' (physical address pA') into the L2 cache. T1 then enters com2, lock T-Unit, and sends a lock message to T-Unit. T-Unit locks itself and returns a success message to T1. At this point T1 is immune to aborts and is guaranteed to complete successfully. Note that at this time any transaction requiring a translation for any object from the T-Unit will be blocked, however those holding the translation in their TTBs can continue. The access to L2 cache and the memory is not blocked and can be accessed by all the transactions. In this example we skip com3, complete partial objects, as we have no work to do to complete our clones. In com4, broadcast, T1 sends a series of messages containing OID (optionally although not here sub-object bitmaps and new translations) of all its modified objects to all the other transactions. On receiving this message T2 compares the broadcasted OID with the OID in its read set: the OIDs in the TTB marked as read. One comparing the OIDs T2 detects conflict, aborts and restarts.

In com5 T1 sends a series of messages each containing an OID and the clone translation to the T-Unit. Upon receiving the message T-Unit installs the new translation. Finally in com6 T1 sends unlock message to the T-Unit and clears the read and write set in its TTB and object cache. T1 has then completed the transaction and can continue from the next instruction.

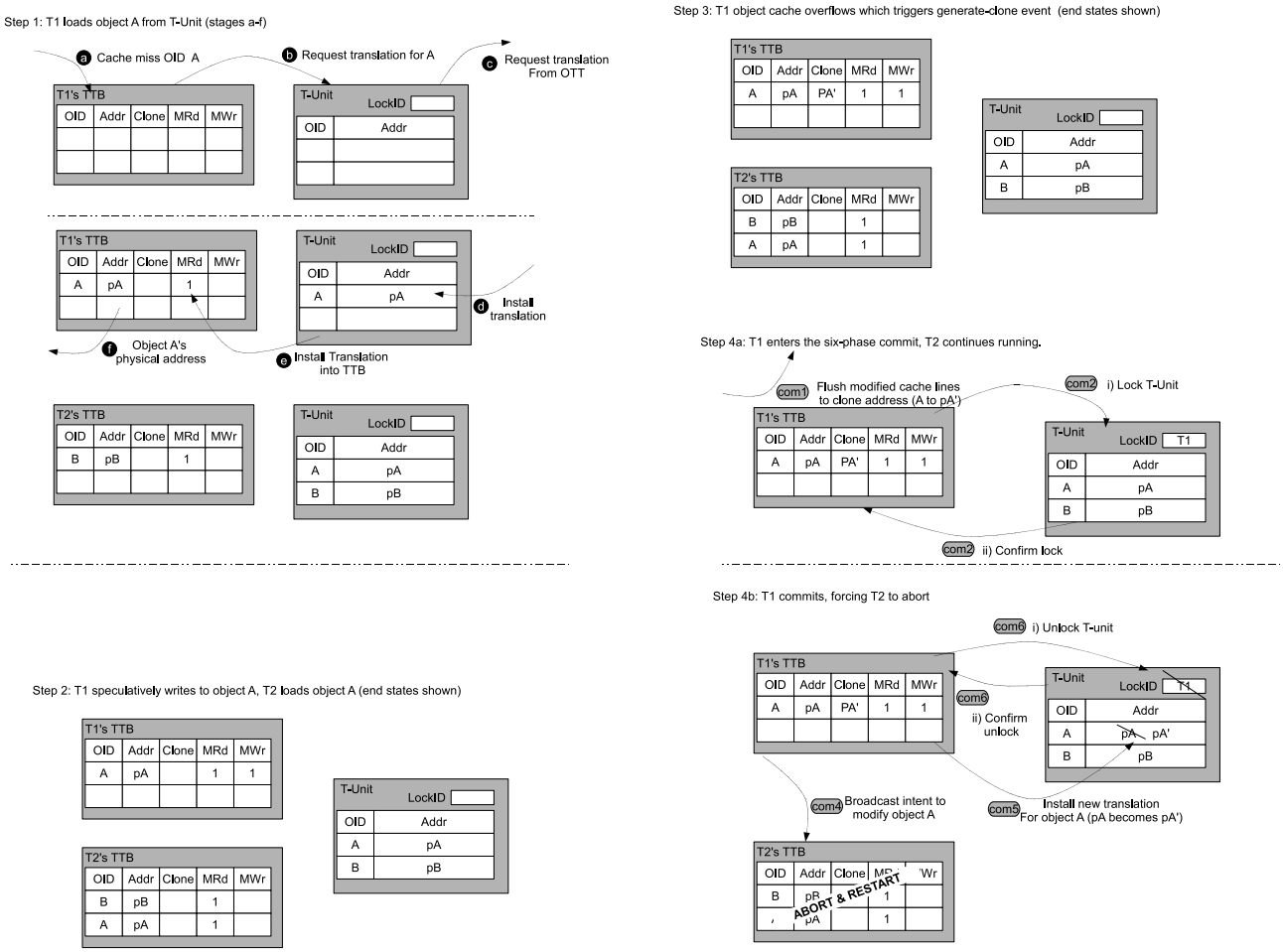

Figure 8 Simple Commit Example: refer to Section 3.5 for a detailed narrative.

4. OTHER DESIGN ISSUES AND OPTIMISATIONS

Similar to the earlier versions of other HTM systems [14, 3, 7], 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 optimised 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 [2], 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 optimisation 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 optimisations 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 form of Bloom filter [1] 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]; Lee-TM.

5.1 Simulation platform and java runtime

As in related HTM studies [13, 14], we use an event driven simulation platform for the evaluation purposes. In order to implement OHTM, an object-aware memory architecture was first simulated by modifying the JAMSIM simulator [22, 10]. The simulator was further extended to implement object transactional memory. OHTM is simulated using a single-bus JAMAICA CMP with an object transactional memory model. Single context JAMAICA cores are used 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 byte-code, 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.

Table 1 Simulation parameters.

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.

5.2 Benchmark applications

We exercise our transactional system by executing three applications (Kmeans, Genome and Vacation) derived from the STAMP benchmark suite, RBTree 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 optimise 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 and also Lee-TM-ER, an early-release implementation.

6. EVALUATION

In this section we will evaluate two configurations of OHTM. One configuration involves detecting conflicts at object level,

Table 2 Parameters for the benchmarks.

Benchmark	Parameters
RBTree	-w 33%
Kmeans-Low	-m40 -n20 -t0.05 -i random1000_12
Kmeans-High	-m20 -n20 -t0.05 -i random1000_12
Genome	-g256 -s32 -m16384
Vacation-Low	-n2 -q10 -u80 -r65536 -t4096
Vacation-High	-n4 -q10 -u80 -r65536 -t4096
Lee-TM-t	75×75×2 grid, 481 routes
Lee-TM-ter	75×75×2 grid, 481 routes

where generate clone event makes complete copies of objects during the clone phase and the broadcast phase involves broadcasting OIDs only in order to detect conflict between transactions. The second configuration involves detecting conflicts at sub-object level, where transactions make partial copies of objects during the clone phase and complete those partial objects during the commit phase. This configuration also requires broadcasting of OIDs and the bitmaps to detected conflict between transactions. Table 3 summarises these configurations.

Table 3 OHTM system configuration.

Copy,broadcast configuration	Description
Object,Object (O-O)	Complete object copied during clone, broadcasts only include OID.
Sub-Object, Sub-Object (S-S)	Partial object copied during clone, broadcasts include OID+bitmap.

6.1 Transaction profiles

Understanding the performance of a TM system requires knowledge of the selected applications used for evaluation. In Table 4 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, Lee-TM-ER and Vacation create significantly large working sets that they overflow the 32KB L1 object cache.

6.2 Performance analysis

6.2.1 Speedups

The graphs in figures 9 and 10, present the speedups achieved from execution of each benchmark on the OHTM when scaling from 1 to 32 processors. In O-O configuration the maximum speedups range from 6.97 times for Genome upto 23.39 times for Kmeans-Low. For S-S configuration the maximum speedup is achieved by Lee-TM-ER, 25.8 times, while Genome speedups to 6.47 times. In both configurations, all applications scale to 16 processor cores (arithmetic mean speedup of 13.82 for O-O configuration and 14.45 for S-S configuration), while Genome and RBTree speedups begins to fall between 16 to 32 cores. One result that is immediately noticeable from the graphs is that the performance of Kmeans-High for 32 cores dramatically drops. This is an artificial limit imposed by the application parameters.

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

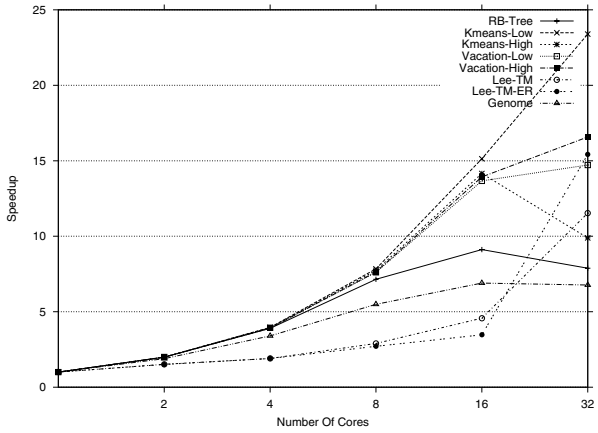

Figure 9 Speedups over sequential code. (Object,Object)

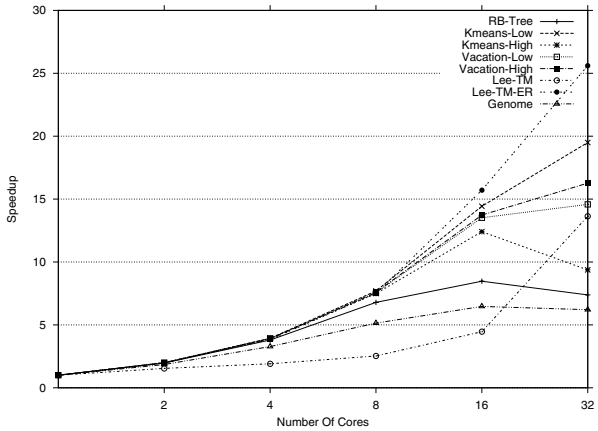

Figure 10 Speedups over sequential code. (Sub-Object,Sub-Object)

On 32 cores, 32 concurrent transactions attempt insertion (modifications) into 20 clusters (shared objects) on each iteration, so at least 12 transactions have to abort which negatively effects the speedup of the benchmark. Lee-TM and Lee-TM-ER are different from other transactional benchmarks as the ordering of commits has an effect on the solution. The Lee benchmarks are non-deterministic therefore changing the number of concurrent threads by increasing the number of cores may result in different solutions. It is therefore difficult to meaningfully compare the speedups achieved by Lee-TM and Lee-TM-ER by increasing the number of cores.

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

6.3 Execution time composition

Figures 11 and 12 show the execution time breakdown of all the applications running on OHTM. The execution time is broken down into four components that indicate how processor spend their time. The first component is *Idle+Wait* cycles. Wait represents the time spent waiting at barriers and synchronisation points while the idle time represents the time spent by the processors waiting for other processors to finish their tasks. The ideal time actually shows the work imbalance between processors running the benchmarks. The second component *Commit*, is

Table 4 Transaction profile statistics for all benchmarks.

Application	#Tx	#Insts	Object readset/Tx		Object writeset/Tx		Inst/Tx		Overflow	
			Mean	COV	Mean	COV	Mean	COV	Txs	Lines
RBTree	64004	43464721	20.55	0.16	2.02	1.61	679.1	0.45	0	0
Kmeans-Low	6695	2210380	7.4	0.21	3.5	0.29	330.1	0.54	0	0
Kmeans-High	6695	2210380	7.4	0.21	3.5	0.29	330.1	0.54	0	0
Genome	9905	60965589	35.5	0.92	7.62	0.5O	6155	1.86	0	0
Vacation-Low	4099	98711006	141.86	0.23	80.7	0.23	24081.7	0.27	6	7.72
Vacation-High	4099	138694470	189.2	0.30	107	0.29	33836.2	0.34	55	106
Lee-TM-t	1447	540892284	118.5	2.85	78.8	3.08	373802.545	4.12	292	407365
Lee-TM-ter	1447	540897094	19.32	1.32	78.8	3.09	373805.87	4.12	293	407773

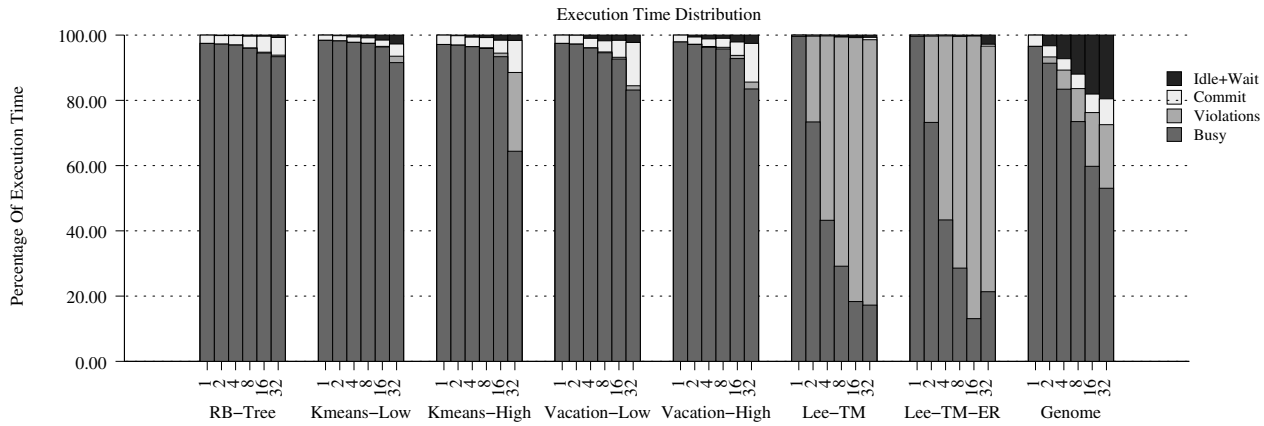

Figure 11 Composition of execution time during transactional execution. (Object,Object)

the time spent by the transactions in the commit phase. The third component *Violations*, is the execution time wasted by the transactions that eventually abort. Finally, the fourth component, *busy* cycles, refer to the time spent executing instructions of the committed transactions. A greater proportion of time busy is better.

Genome is the only benchmark with barriers, which result in waiting cycles in the execution graphs. In O-O configuration 12.18% of the total execution time is spent waiting in barriers as compared to 13% in barriers for S-S configuration using 32-core simulations. Increase in the waiting cycles can result in reduction in the overall performance of the benchmark. Apart from Lee-TM-ER in S-S configuration, Genome also has higher number of idle cycles as compared to other benchmarks (7.33% for O-O and 6.6% for S-S configuration using 32-core simulation), which indicates greater work imbalance in Genome, compared to other benchmarks. These two components are one of the reasons why Genome does not get greater speedups in the speedup graphs.

The cause of the drop off in performance of Kmeans-High using 32 cores is clearly visible as the time spent in aborted transactions increases, from approximately 1% at 16 cores upto 24% and 19% at 32 cores in both O-O and S-S configuration respectively.

Lee-TM show a significant amount of violations, with aborted transactions accounting for 25 to 82% of the total execution time for both O-O and S-S configurations. These violations are due to false conflicts within the readset when observed at the algorithmic level ((as explained in [19])). Using 'early release' [9] mechanism reduces the read set of the transactions from average read set of 118.5 to 19.32, which results in significant reduction in the time wasted due to violations in S-S configuration. Using S-S configuration the aborted transactions account for only 2 to 9% of the total execution time for 2 and 32 cores respectively. Due to false conflicts between transactions using object level conflict detection, the O-O configuration does not take advantage of the early release mechanism and the percentage of time spent during violations approximately remains the same.

A last interesting point is that average time per commit increases as number of cores goes up. Especially the biggest jumps are observed when going from 16 to 32 cores in Kmeans-High and Vacations benchmarks, with a 2-4x increase. With increase in the number of processors, T-Unit contention increases, as T-Unit lock request in the commit phase of the transactions in denied due to other committing transactions. Updating of the object table which requires access to the memory and T-Unit also take longer due to T-Unit contention which can also result in the increase in total commit time.

6.4 Bus utilisation

Figure 13, 14 show 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*, *Clone Request* 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 *Clone Request* traffic is an artifact of our current preliminary implementation, as any requests to the T-Unit while locked triggers continual retries un-

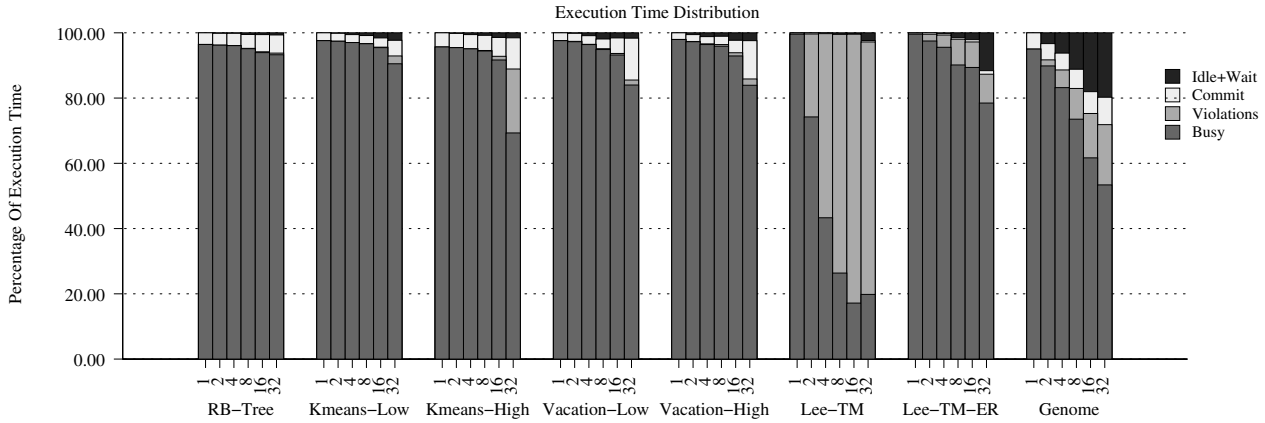

Figure 12 Composition of execution time during transactional execution. (Sub-Object,Sub-Object)

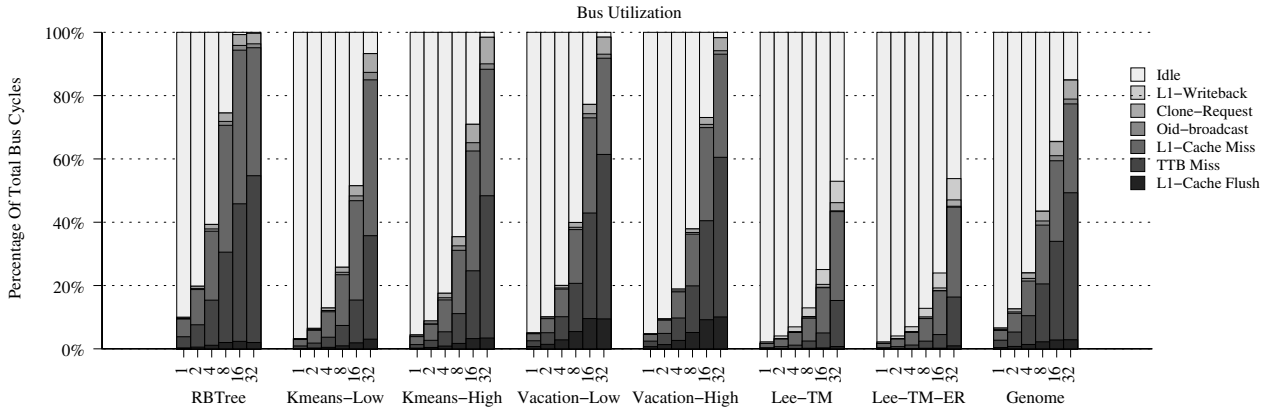

Figure 13 Composition of Bus Traffic During Application Execution. (Object,Object)

til 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. *OID broadcast* only in case of O-O configuration and *OID, bitmaps broadcast* in case of S-S configuration. The broadcast traffic on the bus 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 3% of the traffic even at 32 cores for both O-O and S-S configurations. As the object-aware HTM only broadcasts OIDs 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.

The bus bandwidth requirement for Kmeans(H) and Vacation(L-H) and RBTree is very high with 32 processors. In case of RBTree the bus idle time is less then .7% for 16 cores and less then .3% for 32cores for both O-O and S-S configuration. This can result in high bus contention for RBTree benchmarks and can result in degrading the overall performance of the benchmark. Thus one of the reasons for not getting greater speedups in the RBTree can be the limitation of the hardware used to run the benchmark rather then the characteristics of the benchmark itself.

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 recognised and harnessed to solve this weakness. 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 3% of bus bandwidth showing the potential of the proposed object-aware HTM.

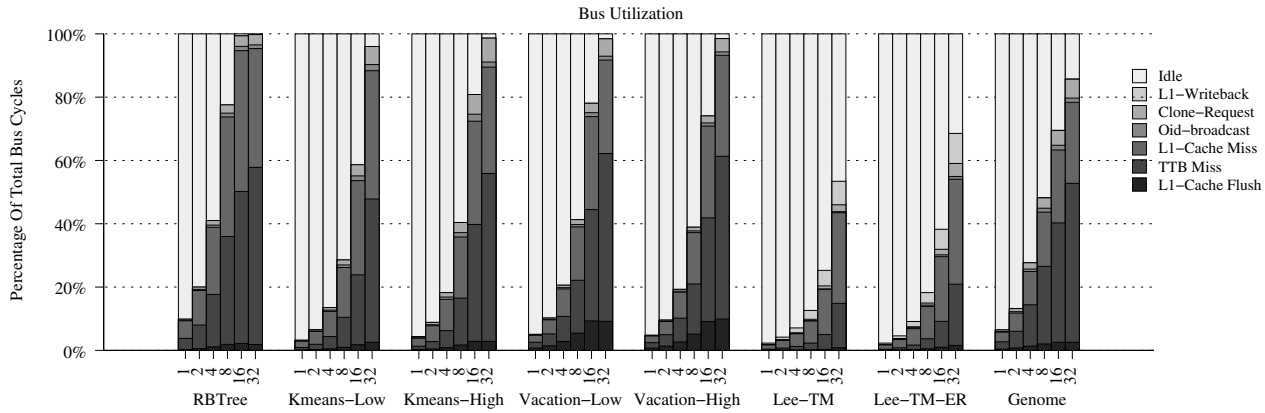

Figure 14 Composition of Bus Traffic During Application Execution. (Sub-Object,Sub-Object)

Acknowledgements

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

REFERENCES

1. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
2. M. Cekleov and M. Dubois. Virtual-address caches. Part 1: problems and solutions in uniprocessors. *IEEE Micro*, 17(5):64–71, 1997.
3. 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.
4. D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the 20th Intl. Symposium on Distributed Computing*, Sept 2006.
5. M. Flynn and P. Hung. Microprocessor design issues: thoughts on the road ahead. *IEEE Micro*, 25(3):16–31, 2005.
6. B. Goetz. Optimistic Thread Concurrency: Breaking the Scale Barrier. Azul Systems Whitepaper, `http://www.azulsystems.com/products/whitepapers.htm`, January 2006.
7. 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.
8. 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.
9. 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.
10. M. Horsnell. A chip multi-cluster architecture with locality aware task distribution. *PhD thesis, Department of Computer Science, The University of Manchester*, January.
11. K. Li, C. Shen, H. Francis, and W. Zhang. Multimedia Object Placement for Transparent Data Replication. *IEEE Transactions On Parallel And Distributed Systems*, 18(2), 2007.
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. G. Wright. A single-chip multiprocessor architecture with hardware thread support. *PhD thesis, Department of Computer Science, The University of Manchester*, January.
23. 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.