Scalable Object-Aware Hardware Transactional Memory

Behram Khan, Matthew Horsnell, Mikel Lujan, and Ian Watson

School of Computer Science, The University of Manchester, UK
{khanb,horsnell,lujan,watson}@cs.man.ac.uk

Abstract. A Hardware Transactional Memory (HTM) aids the construction of lock-free regions within applications with fewer concerns about correctness and potentially greater performance through optimistic concurrency. Object-aware hardware adds a level of indirection to memory accesses, memory addresses become a combination of the object being accessed and the offset within it. The hardware maintains a mapping from objects to memory locations just as a mapping from virtual to real memory is handled through page tables. In a scalable object-aware system the directories are addressed by objects identifiers.

In this paper we extend a scalable object-aware memory system to implement a HTM. Our object-aware protocol permits locks on directories to be avoided for objects only read during a transaction. Working at the granularity of an object allows entries within the directories to be associated with multiple cache lines, as opposed to one, and reduce the amount of network traffic. Finally, our commit protocol dispenses with the need for a centrally controlled transaction ID order.

Keywords: Scalable, Hardware transactional memory, Object-oriented, Multi-core architecture, Concurrency, Atomicity.

1 Introduction

In this paper we propose a novel HTM system and our major contributions are:

- A scalable HTM system allowing parallel independent commits.
- A novel validation and commit protocol that requires only directories containing writes to be locked.
- A HTM system that supports locking within the directory at the granularity of an object (four cache lines).
- A scalable HTM implementation that dispenses with centrally controlled transaction ordering.

Central to our HTM protocol is exposing objects to hardware, we refer to the hardware as being object-aware. An object is addressable by a unique object identifier (OID) and a field offset that is in the region of 0 to 128-bytes (in this paper our proposed object size is 4 L1 cache lines. Greater sized objects can be referenced using multiple object identifiers). The translation of object identifier

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part I, LNCS 6271, pp. 268–279, 2010.

to physical address is done through a lookup table. Such systems have been proposed primarily to improve the performance of garbage collection [9][10]. In this paper we consider the use of our protocol for running object-oriented benchmarks, however, by configuring the object table to resemble a page table the scheme could also be used to run non-object-oriented programs.

For the purpose of describing the HTM implementation we have named it after its central goals of scalability and the use of an object-aware memory protocol. The HTM is called the Scalable Object-aware (SO) HTM. To distinguish object-aware directories from those for regular memory addresses, and because they hold a translation from the object identifier (OID) to a memory address, our directories are referred to as T-Units.

The rest of the paper is laid out as follows. Section 2 provide details of the object indirection extension to the memory hierarchy and how it can be used for TM implementation. Section 3 presents the SOHTM system, the major functional units and protocols. Section 4 presents our experimental methodology and the seven transactional benchmarks used to validate and analyse the HTM. Section 5 evaluates the SOHTM system. We summarise this work in Section 6.

2 An Object-Aware Approach

Previous work [4] has explored an approach to transactional memory that recognises the structure of objects at the hardware level. It argued that such an approach reduces the problems associated with overflow in lazy evaluated TM systems, and that communicating using a concise object representation can cut the bandwidth requirements. For clarity we provide an overview of this approach within the following section.

2.1 Object Addressed Memory

Schemes have been proposed to provide direct hardware support for Object-Oriented (OO) languages [9][10]. 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), 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 update all references to an object that is moved.

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'[9]. An object-addressed cache has cache lines that are tagged directly with (OID, offset) addresses, and thus contain parts of objects rather than block of physical memory. 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 walking the OTT. This latter inefficiency is mostly removed by the provision of a Translation Buffer that caches recently used translation table entries. The

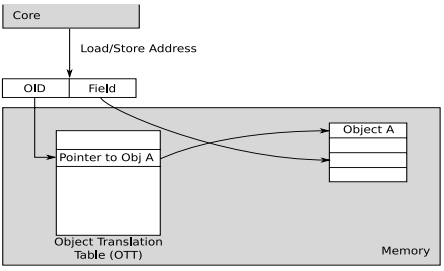

Fig. 1. Hardware support for object addressing

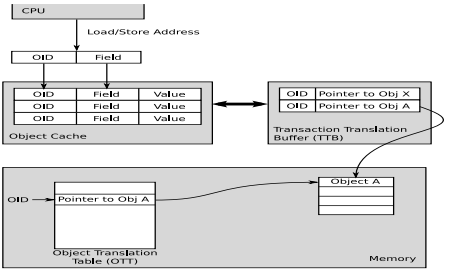

Fig. 2. Caching object to address translations in a Translation Buffer allows direct access to recently used translations, reducing the requirements to access the OTT

OID to memory address mappings can be accessed directly as shown in Figure 2 removing the need for translation via the OTT.

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 buffer 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 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 clone 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 bitmap.

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.

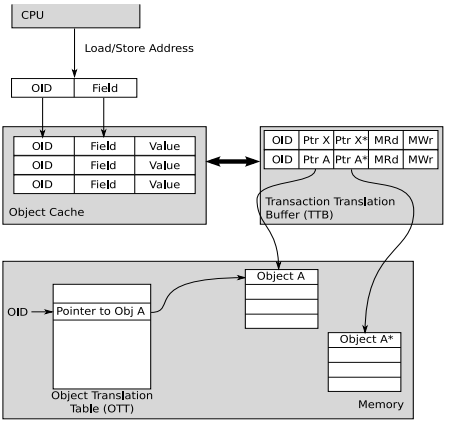

Fig. 3. An additional pointer within the TTB allows transactional updates to seamlessly occur in isolation

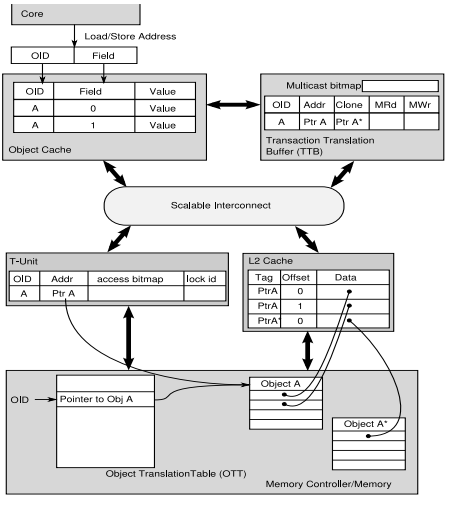

Fig. 4. The basic system structure of an object-aware HTM. The T-Unit provides functionality associated with translation, cloning and committing of objects.

3 Scalable Object-Aware (SO) HTM Protocol

3.1 SO Protocol Overview

To address the limitations of previously proposed object-based hardware transactional memory (HTM) system [4][3], this paper presents a protocol capable of maintaining transactional execution over a scalable memory architecture. This section presents the SO protocol, which no longer relies on a network that supports atomic broadcasts but rather is extendible to function on any scalable network architecture. In SO protocol a committing transaction does not broadcast its updates to all other transactions but multicast its information only to the transactions that share any potentially conflicting data. Additionally the SO protocol allows multiple transactional commit to occur concurrently, as it does not rely on a committing transaction to acquire any global lock.

Hardware Outline. Figure 4 shows the organisation of the hardware required to support the scalable object-aware (SO) HTM protocol. There are two components that warrant special attention: the transaction translation buffer (TTB), and the translation-unit (T-Unit).

Transaction-Translation Buffer (TTB). In essence the TTB fulfills a similar role to that of a TLB: it caches a number of recent translations, in this case from OIDs to physical addresses. The TTB is located alongside the L1 object-cache and is accessed when a cache look-up 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 cache request is forwarded immediately using the translation or a translation must be requested from the T-Unit, which will request in turn from the OTT if it also contains no valid translation.

The TTB also holds translations from OIDs to the physical addresses of clone objects, which are used to provide transactional isolation. When an object is written to during a transaction, the clone translation is used to generate a physical address, or if no clone translation exists a clone area is generated in the OTT and a translation is returned. 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.

The TTB is also responsible for maintaining a multicast bitmap. It is assumed that this bitmap contains a single bit for every core in the architecture. The bitmap is filled during the commit phase of the protocol and represents a list of the cores that are accessing objects that the TTB's associated transaction is modifying. During a commit phase a transaction uses this bitmap to find out which other transactions running on other cores are sharing its data. Using this bitmap a transaction avoids broadcasting its shared data and multicasts its updates only to the transactions that are actually sharing the data.

Translation-Unit (T-Unit). The T-Units are global resources in the same way that multiple banks of an L2 cache are global resources, and are shared by all cores. Each T-Unit is essentially a large cache for translations of OIDs to physical addresses, reducing the number of translations that need to be fetched from an intelligent 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.

Alongside each OID translation in the T-Unit is a bitmap with a single bit per core. The bitmap is set when a core's current transaction has used, read from or written to, that OID. There is also a lock identifier field that is set during the commit phase to indicate that a transaction is attempting to update the translation of the object in the T-Unit. When the lock is set, any translation request to the T-Unit is nacked. The lock identifier can be a single lock per T-Unit or it can be a lock per every object translation with in the T-Unit. In case of a single lock, when a committing transaction wants to update the object translation, the entire T-Unit is locked and all requests to the T-Unit from other transactions are blocked. In case of lock per object the committing transaction only locks the line in the T-Unit that contains object translation and only requests to that particular object with in the T-Unit are blocked. The effects of locking at T-Unit level or at the object level is discussed in the later sections.

SO Protocol Outline. In this section we describe the Scalable Object-aware (SO) protocol. Prior to committing, transactions run in isolation as described in Section 2. When a transaction is ready to commit it follows a four phase protocol.

Phase 1: Gather Locks. During this phase a transaction is required to gather locks sufficient to cover all the objects that it has modified. If the locks are gained on a per OID basis then a message must be sent to each relevant T-Unit for each OID modified, if there is only one lock per T-Unit then a single message is sent to each T-Unit that contains an object that the transaction is modifying. The response from the T-Unit to each request contains a copy of the access bitmap for the OID. This is masked on top of the multicast bitmap in transaction's TTB multicast bitmap.

Once a lock is set the OIDs that it covers are no longer accessible by other transactions attempting to fetch the translation. Any such request will receive a retry response.

It should be noted that this phase is liable to run into contention, as the gathering of all object locks is no longer an atomic operation. A discussion of the rules used to overcome this contention is discussed in later sections.

Phase 2: Complete Partial Objects. If the protocol is using bitmaps within the TTB to reduce false sharing within an object, then at this stage the clone objects must be completed. This process involves reading into the clone object any data that is not present. If the protocol is resolving conflicts at the OID granularity then the clone will have been allocated with a complete copy and so this phase is unnecessary.

Phase 3: Multicast. When a transaction enters this phase then it is known that it has gathered all locks and therefore it can complete the commit, as long as no other transaction modifies its read-set. During this phase all modified objects are sent to every core contained in the multicast bitmap. There are two ways in which the multicast can be done. If the conflict is detected at the object level then only the OIDs of the modified objects are sent to the potentially conflicting transactions. If the conflict is detected at the sub-object level, then the OIDs along with the write bitmaps are sent to the conflicting transactions. Detecting conflict at sub-object level can reduce the amount of false conflict between transactions.

Phase 4: Update Translations. Having successfully informed all interested transactions of the modifications that will be made, the new translations are installed in the relevant T-Units, and the bitmaps are cleared for all modified objects along with the locks. At this stage the new translations become visible to other transactions. Access to a locked translation is prevented, a retry message being sent to any transaction attempting such an access. Note that during this phase the transaction can no longer be aborted, and is guaranteed to commit which makes this phase completely safe even tough it is not atomic.

If the transaction receives a multicast message that invalidates its read-set from a transaction in phase 3, it responds with a negative acknowledgement. The nacked transaction will continue to retry until it is aborted or can multicast the message when the blocking transaction commits. In parallel to updating

the translations, the committing transaction also sends a single message to any T-Unit touched by its read to clear its read bits in the T-Units.

Protocol Rules. One complication of the SO protocol is that we wish to be able to process commits concurrently to reduce the limitation imposed on performance by serialised commits. However, in a lazy TM system where resources are distributed this can cause contention as mentioned in the description of phase 1 above, and blocking in phases 3 and 4. We currently handle such contention using simple rules:

When a transaction finds a lock set by another transaction during phase 1, we compare their priority. There is an implicit priority based on the unique hardware identity of the core on which the transaction is executing. If the core ID is lower we say its priority is higher. This is purposefully made simple and arbitrary just to check the correctness and working of the overall SOHTM system. A transaction with a lower priority must release a lock if a transaction of higher priority attempts to gather it. The lower priority transaction must then try to regain the lock. In future we hope to implement more complex conflict resolution methods like resolution based on time stamps [7] or using software contention managers [2].

We also allow any multicast (from a transaction in phase 3) to be blocked by a transaction in phases 4 whose read-set would be violated by the implied effect of the multicast.

4 Experimental Methodology

To evaluate the SO 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 7 applications detailed in Section 4.2.

4.1 Simulation Platform and Java Runtime

As in related HTM studies [5][6], we opt for an event driven simulation platform with an IPC of 1 for all but memory operations, observing that transactional

Table 1. Simulation parameters

Feature	Description
L1 object cache	32KB, private, 4-way assoc., 32B line, 1-cycle access.
TTB	32KB, private, 12-way assoc., 8-12B lines, 1-cycle access.
Network	Cross bar.
L2 cache	4MB, shared, 32-way assoc., 32B line, 32-cycle access. 8 banks
T-Unit	4MB, shared, 32-way assoc., 8-12B lines, 32-cycle access. 8 banks
Memory	100-150 cycle off-chip access.

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. When objects are greater than 128-bytes they span adjacent OIDs.

As described with the protocol, different configurations are possible for the locking of the T-Unit, either individual objects may be locked or the T-Unit as a whole may be locked. For the multicast, violations may be detected either at the object-level or at a finer granularity with the object and bitmap. The bitmap is described in Section 3.1 and comprises 1-bit per cache line. Table 2 summarises these configurations.

Table 2. TM system configuration

Lock,multicast configuration	Description
Object,Object (O-O)	Individual OIDs are locked, multicasts only include the OID.
Object,Sub-Object (O-S)	Individual OIDs are locked, multicasts include the OID and bitmap.
T-Unit,Object (T-O)	Locking at the T-Unit bank level, multicasts only include the OID.
T-Unit,Sub-Object (T-S)	Locking at the T-Unit bank level, multicasts include the OID and bitmap.

4.2 Benchmark Applications

We exercise our architecture using two variants of the Kmeans and Vacation benchmarks from the STAMP benchmark suite [5], two variants of the Lee-TM benchmark [8][1] and a transactional Red-Black (RB) tree. The STAMP benchmarks were ported to Java by converting structs to objects. The benchmarks contained self verification tests.

Table 3 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

Table 3. Parameters for the benchmarks

Benchmark	Parameters
Kmeans-Low	-m130 -n130 -t0.05 -i random-n4000_d12_c130
Kmeans-High	-m80 -n80 -t0.05 -i random-n4000_d12_c130
Vacation-Low	-n2 -q90 -u98 -r16384 -t4096
Vacation-High	-n4 -q60 -u90 -r16384 -t4096
Lee-TM	75×75×2 grid, 573 routes
Lee-TM-ER	75×75×2 grid, 573 routes
RB-Tree	33% Updates

early-release implementation. The RB-Tree has a synthetic parallel workload where reads are more frequent than updates by a given ratio.

5 Evaluation

5.1 Concurrent Commits

Central to the motivation of our approach is that concurrent commits are likely in HTM systems. We measured how many transactions are attempting to commit concurrently for each of our configurations. An attempt to commit may be aborted or succeed leading to variations when different SO HTM configurations are simulated. Due to space constraints we summarise the results in Table 4.

Table 4. Transactions concurrently committing (number of transactions in commit phase 128 cores)

Application	Object,Object		Object,Sub-Object		T-Unit,Object		T-Unit,Sub-Object	
	Mean	Max	Mean	Max	Mean	Max	Mean	Max
RB-Tree	8.69	64	9.23	106	7.13	103	7.33	79
Kmeans-Low	1.36	14	2.3	31	1.99	20	2.65	28
Kmeans-High	6.0	73	15.34	85	5.62	54	5.36	41
Vacation-Low	8.5	24	9.15	22	6.01	20	5.78	19
Vacation-High	7.57	21	8.11	22	5.49	17	5.42	18
Lee-TM	1.6	18	2.04	25	1.57	18	1.73	26
Lee-TM-ER	1.47	11	1.35	7	1.56	13	1.25	7

The largest average number of commits occurs for the high contention Kmeans benchmark for the object based lock and object multicasting SO HTM configuration. On average 15.34 transactions were attempting to concurrently commit. With lower contention fewer threads were attempting to commit concurrently for the Kmeans benchmark.

Again for the object based lock and object multicasting version of SO HTM, the maximum number of concurrent commits occurs for the RB-Tree benchmark. 106 transactions were committing concurrently. In other words 82.8% of the transactions were in their commit phase.

Whilst for some benchmarks and configurations the average number of concurrently committing transactions is small, these results highlight that for some applications a large number of concurrent commits can occur both on average and in bursts. In such situations arbitrating for a shared resource or contending on a shared locked component would serialise execution and limit performance. The results in Table 4, show the importance of allowing concurrent commits in any TM system.

5.2 T-Unit Contention

There are two ways in which T-Unit contention can take place in SOHTM. The SOHTM configurations can be divided into two groups depending on the way

they lock objects in the T-Unit during the commit phase of the transactions. These two groups are the O-O,O-S configuration and the T-O,T-S configuration.

In O-O and O-S configurations the committing transactions lock individual objects within the T-Unit that form their write-set. When the objects are locked, any request for the translations of those objects made by the other transactions are rejected by the T-Unit and the requesting transactions have to retry their request.

In T-O and T-S configurations the committing transactions lock the entire T-Unit bank that contains the objects present in the transaction's write-sets.

Figure 5 show the distribution of the T-Unit access of all the benchmarks using four SOHTM configurations with 128 core simulations. For both O-O and S-S configurations, all the benchmarks have T-Unit contention of less than 10% (overall average of 2.98% only). On the other hand the T-Unit contention for all the benchmarks using T-O and T-S configuration is greater then 32% with some benchmarks experiencing T-Unit contention of upto 80% (overall average of 60.14%).

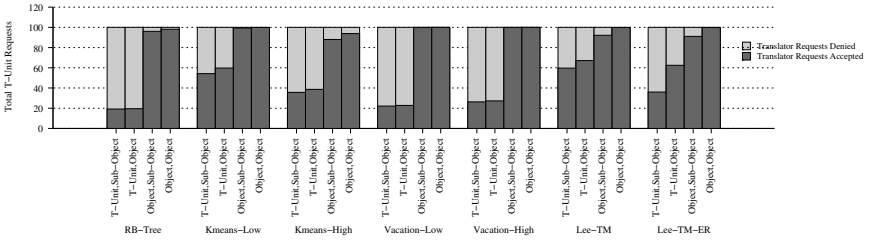

Fig. 5. T-Unit access composition of all benchmarks using four SOHTM configurations with 128 cores

5.3 Overall Performance

The graphs in figures 6 and 7 present the speedups achieved from execution of each benchmark on the SOHTM using the O-O and O-S configurations, scaling from 1 to 128 processors. In the O-O configuration the maximum speedups range from 28.11 times for RBTree to 67.80 times for Vacation-High. For the O-S configuration the maximum speedup is achieved by Lee-TM-ER, 68.17, while the RB-Tree achieves a speedup of 28 times. In both the O-O and O-S configurations, all applications scale to 64 processors (arithmetic mean speedup of 49.8 for the O-O configuration and 52.03 for the S-S configuration), while RBTree speedup begins to fall between 64 to 128 cores. One result that is immediately noticeable from the graphs is that the performance of Kmeans-High drops dramatically for 128 cores. This is artificial limit imposed by the application parameters. For Kmeans-High the data set is limiting in exposing parallelism beyond 80 threads.

Figures 8 and 7 show the speedups of the benchmarks for the T-O and T-S configurations. For both T-O and T-S configurations only three benchmarks

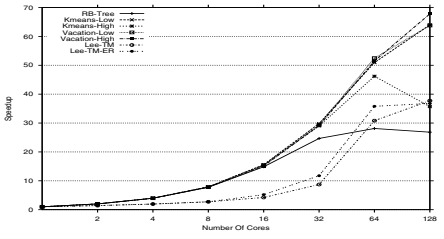

Fig. 6. Speedups over sequential code. (Object,Object).

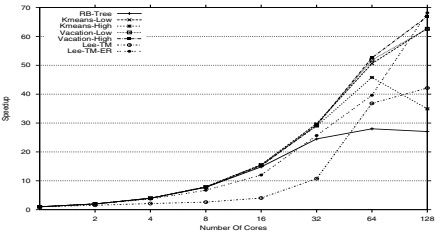

Fig. 7. Speedups over sequential code. (Object,Sub-Object).

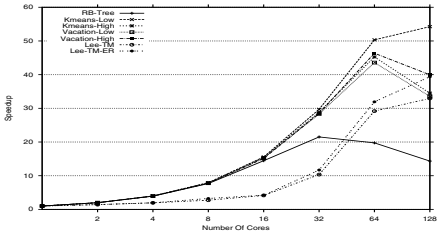

Fig. 8. Speedups over sequential code. (T-Unit,Object).

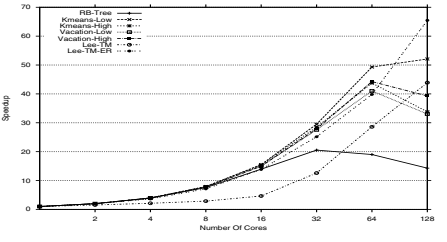

Fig. 9. Speedups over sequential code. (T-Unit,Sub-Object).

Kmeans-Low, Lee-TM and Lee-TM-ER show any speedups beyond 64 cores. Kmeans-High and the Vacation benchmarks show significant drop in performance at 128 cores while the performance of RBTree drops beyond 32 cores. Benchmarks running on the T-O and TS configurations have an overall arithmetic mean speedup of 40.46 times and 44.42 times respectively. The reason for the drop in performance of Kmeans-High is obvious, as there is limited parallelism available beyond 80 processors due to an artificial limit imposed by the application parameters.

6 Summary

Centralised resources and locked resources are problematic for scaling HTMs. Using an object-aware memory system allows for efficient locking at the object level. It also allows an efficient multicast mechanism. Such a system can be further extended to avoid false sharing within the object.

The SO HTM has extended object-aware HTM design to tackle the problem of scalability. Finer grain locking is measured to have a performance improvement from 10% to 91% over locking at a directory (T-Unit) level on 5 benchmarks for 128 cores.

Central to the SO HTM design are privatizing written data using the object indirection, and arbitrarily priorizing transactions based on their core identity. Such a scheme is extendable to support regular address spaces as well as potentially allowing other object-based optimizations. The SO HTM shows that

being object-aware is complementary to scalability and should be considered for future processor implementations.

Acknowledgment

Dr. Mikel Luján holds a Royal Society University Research Fellowship.

References

1. Ansari, M., Kotselidis, C., Watson, I., Kirkham, C., Luján, M., Jarvis, K.: Lee-TM: A non-trivial benchmark suite for transactional memory. In: Bourgeois, A.G., Zheng, S.Q. (eds.) ICA3PP 2008. LNCS, vol. 5022, pp. 196–207. Springer, Heidelberg (2008)
2. Scherer III, W.N., Scott, M.L.: Advanced Contention Management for Dynamic Software Transactional Memory. In: 24th ACM Symp. on Principles of Distributed Computing (2005)
3. Khan, B., Horsnell, M., Rogers, I., Luján, M., Dinn, A., Watson, I.: A First Insight into Object-Aware Hardware Transactional Memory. In: Proc. SPAA (2008)
4. Khan, B., Horsnell, M., Rogers, I., Luján, M., Dinn, A., Watson, I.: An Object-Aware Hardware Transactional Memory System. In: Proc. HPCC (2008)
5. Minh, C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., Olukotun, K.: An effective hybrid transactional memory system with strong isolation guarantees. In: Proc. ISCA, pp. 69–80 (2007)
6. Moore, K., Bobba, J., Moravan, M., Hill, M., Wood, D.: LogTM: Log-based transactional memory. In: Proc. HPCA, pp. 258–269 (2006)
7. Rajwar, R., Goodman, J.R.: Transactional Lock-Free Execution of Lock-Based Programs. In: Proc. of the Tenth Intl. Conference on Architectural Support for Programming Languages and Operating Systems (2002)
8. Watson, I., Kirkham, C., Luján, M.: A study of a transactional parallel routing algorithm. In: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques, pp. 388–398 (2007)
9. Williams, I.: Object-Based Memory Architecture. PhD thesis, Department of Computer Science, University of Manchester (1989)
10. Wright, G., Seidl, M., Wolczko, M.: An Object-aware Memory Architecture. Science of Computer Programming 62(2), 145–163 (2006)