Exploring Garbage Collection with Haswell Hardware Transactional Memory

Carl G. Ritson
University of Kent
C.G.Ritson@kent.ac.uk

Tomoharu Ugawa
Kochi University of Technology
ugawa.tomoharu@kochi-tech.ac.jp

Richard E. Jones
University of Kent
R.E.Jones@kent.ac.uk

Abstract

Intel's latest processor microarchitecture, Haswell, adds support for a restricted form of transactional memory to the x86 programming model. We explore how this can be applied to three garbage collection scenarios in Jikes RVM: parallel copying, concurrent copying and bitmap marking. We demonstrate gains in concurrent copying speed over traditional synchronisation mechanisms of 48–101%. We also show how similar but portable performance gains can be achieved through software transactional memory techniques. We identify the architectural overhead of capturing sufficient work for transactional execution as a major stumbling block to the effective use of transactions in the other scenarios.

Categories and Subject Descriptors D.3.4 [*Programing Languages*]: Processors—Memory management (garbage collection)

General Terms Algorithms, Languages

Keywords Garbage Collection; Transactional Memory; Java; Jikes RVM

1. Introduction

As physical and energy constraints have led to the end of Dennard scaling and ever increasing clock speeds, manufacturers have instead sought to increase performance by delivering increasingly parallel hardware. Garbage collection (GC) designers have taken advantage of parallel hardware in two ways. *Parallel collectors* use multiple collector threads for activities such as marking, sweeping or moving objects, although these collectors may still 'stop the world' (halt all user threads, or mutators) while they work. *Concurrent collectors* allow mutator and collector threads to execute simultaneously, although they briefly stop the world, for example, to scan mutator threads. *On-the-fly* concurrent collectors, on the other hand, never stop the world, but may stop one mutator at a time to scan its roots.

All of these strategies require synchronisation. Parallel collectors require coordination between collector threads, and concurrent collectors between mutator and collector threads so that all threads share a consistent and correct view of the heap. Synchronisation is needed in parallel collectors to ensure that if two collector threads attempt simultaneously to mark objects, e.g. by setting bits in a bitmap, the mark state is updated correctly and no marks are lost. Similarly, parallel copying collectors must ensure that if two collector threads attempt to move an object, precisely one succeeds. Concurrent copying collectors need to ensure that mutator updates are not lost if a collector attempts to move an object at the same time as a mutator is modifying its fields. All concurrent collectors need to ensure that mutator and collector threads share a consistent, if conservative, view of the liveness of objects in the heap.

Coordination between mutators and concurrent and incremental collectors is typically achieved by having the mutator use read and/or write barriers as it loads values from or stores values into object fields; often only pointer values need be barriered. Barrier actions may notify the collector of changes to the connectivity of objects, or may ensure that mutators only see the most up-to-date versions of objects. In many cases, coordination actions can be implemented cheaply using simple loads and stores [24]. However, in some cases, collector and/or mutator threads need to be synchronised, for example by using atomic instructions [11]. However, not only are such instructions more expensive than simple loads and stores, but the instructions sequences required are hard to get right, especially in the face of modern processor memory models.

Recently, there has been considerable interest in *transactional memory* as a simpler yet efficient solution to the problem of writing concurrent software. Inspired by database systems, transactional memory allows a thread to execute a sequence of instructions as a transaction. If no other thread makes a conflicting access to the memory locations used by the first thread, the transaction commits. Otherwise, the transactions aborts and the state of the thread is rolled back to that before the transaction started.

Transactional memory can be implemented in software or hardware. Until recently, hardware transactions have not been available in commodity processors, but this has changed with the release of Transactional Synchronization Extensions in Intel's new Haswell family of processors. We explore whether transactional memory, implemented in hardware or software, can improve the performance of common GC actions. Our context is Jikes RVM [1], a widely used metacircular Java virtual machine. We identify, from an audit of its wide range of GCs, parallel semispace copying collection, concurrent replicating GC and parallel bitmap marking as candidates for transactional memory support. Our results show that:

- Both software and hardware transactional memory techniques can improve the concurrent object copying speeds in the Sapphire on-the-fly collector by 48–101%.
- Hardware transactional memory offers no benefit to parallel copying or bitmap-based mark-sweep collection.
- It is essential perform sufficient work to amortise the cost of a transaction, and to plan activities to do as much work as possible outside the transaction; we demonstrate how to do this.

ISMM '14, 12 June, 2014, Edinburgh, UK.

ACM 978-1-4503-2921-7/14/06…$15.00.
http://dx.doi.org/10.1145/2602988.2602992

2. Haswell

With the release of their latest processor microarchitecture, code-named Haswell, Intel added new *Transactional Synchronization Extensions* (TSX) to their processors' instruction set [12]. These extensions provide *Restricted Transactional Memory* (RTM). Transactional memory allows the atomic manipulation of arbitrary size units of memory. Without transactional memory the largest unit of memory that can be atomically manipulated on x86 architectures is two memory words, 64 bits or 128 bits, using for example a *compare-and-swap* (CAS) operation.[1]

2.1 Programming Model

At an instruction level, Intel's RTM is simple to use. A transaction is initiated with an `XBEGIN` instruction. Computation proceeds normally: memory can be read and written with common instructions and other activities such as branching and arithmetic may also be used. At the end of the transaction an `XEND` instruction commits any changes to memory.

During the transaction, read and write sets are constructed. These sets have cache line granularity and are based on the memory addresses read or written by the transaction's body. If these sets conflict with memory being read or written by other hardware threads then the transaction is aborted. If a transaction is aborted, all changes to memory and registers are discarded and execution jumps to a *fallback handler* supplied to the initial `XBEGIN` instruction. Before invoking the fallback handler, status flags are set in the processor's `EAX` register. These flags allow the fallback handler to determine what caused the transaction to abort.

It is worth clarifying that read and write sets are specific to a given hardware thread (the smallest unit of parallel execution in a computer system). In Intel's Haswell architecture a computer may have multiple processors, each of which has multiple cores, each of which has multiple hardware threads. Hardware threads on the same core share components such as arithmetic units and cache, but have their own registers and instruction pointer.

2.2 Performance

Our early results suggest that the set-up and tear-down cost of a successful memory transaction on Haswell is at least three times the cost of a compare-and-swap [20]. This means that there is no benefit from simply replacing other synchronisation operations with transactions; rather, sufficient work must be available to merit the use of the transaction. Beyond set-up and tear-down costs, we found that reads within a transaction incur a performance penalty of up to 20%. We believe that this arises from the additional coherence constraints placed on reads. Writes normally incur this coherence penalty anyway and thus are not affected. This makes transactions most suited to write and update activity.

3. Methodology

All GC implementations presented in this paper were modifications of Jikes RVM. Performance was evaluated using the 2006 and 2009 DaCapo benchmark suites [3]. All results were obtained from a system with a 4-core Intel Core i7-4770 processor running at 3.4GHz, with 16GiB of RAM. Stock Ubuntu Linux 12.04.3 LTS was used with kernel 3.8.0-25-generic. The processor's Turbo Boost mode and CPU frequency scaling were disabled in the system's BIOS to allow for consistent results regardless of hardware temperature.

Unless otherwise stated, testing on Jikes RVM used 'compiler replay' [7]. For each selected DaCapo benchmark, an initial set of ten warmup runs are used to allow the optimising compiler to reach a stable state. The state of the optimising compiler is then recorded. Results are collected from 20 independent runs in which classes are compiled and initialised using the previously recorded compiler state. A given benchmark run generates multiple data points (one per GC); these are aggregated from all runs and the geometric mean computed. Error bars presented represent the 95% confidence interval computed as per Kalibera and Jones [17].

As we are interested in the performance of individual collector actions (e.g. copying an object) rather than total time spent in GC, a single fixed heap size of 350MiB is used for all tests, rather than conducting experiments with a range of different heap sizes. For stop-the-world collection, the GC is triggered when the available heap is exhausted. For concurrent collection, the GC is triggered as soon as 32MiB have been allocated since the last GC.

We do not explore reattempting failed transactions. While retry of transactions is possible with all the algorithms presented here, if a transaction fails we revert to the defined fallback case. This is based on our earlier observation [20] that only a small percentage of transactions are suitable for the processor to retry. Hence we assume that only a small percentage of failing transactions would succeed if reattempted.

4. Parallel Copying Collection

Copying collection typically creates a copy of all live objects in a logically contiguous area of memory, the *to-space*, discarding the previously used *from-space* and thus de-fragmenting the heap. Copying is usually integrated with tracing of live objects. When an uncopied object is reached, the GC copies the object and updates with the object's new *to-space* address the field in the *to-space* object from which it reached this object. To preserve the topology of the object graph, this address is also installed as a *forwarding pointer* in the *from-space* object in case the trace should reach it again by following other pointers. As part of this trace, unmarked edges from the object are placed onto a work queue. By sharing this work queue between a number of collector threads, copying activity can be performed in parallel [15, 22].

Two or more threads may attempt to copy the same object simultaneously. If unmanaged, this could result in multiple divergent copies of an object in *to-space*. To avoid this, interactions on the same object must be synchronised. One solution is to reserve space *optimistically* for the copy, race to install the forwarding pointer with a compare-and-swap then, if successful, copy the object. Alternatively and more *conservatively*, a GC thread could acquire exclusive access to the object while it copies it. The disadvantage of this method is that any other tracing threads that reach the object must wait until the copy is complete. On the other hand, the disadvantage of optimistic copying is that it may be difficult to unreserve space — for example, the GC thread may have had to acquire a fresh allocation buffer — if a thread loses the race to install the forwarding pointer.

Jikes RVM adopts the conservative approach, assigning objects three distinct monotonic states: *uncopied*, *copying* and *copied*. The designers' assumption was that contention by GC threads was likely to be rare. On reaching an uncopied object the collector threads race to transition the object from *uncopied* to *copying*, using a compare-and-swap operation on the object's header. The winning thread copies the object before installing the forwarding pointer and setting the object's state to *copied*. In the Jikes RVM *semispace* collector, waiting threads spin, testing a field in the object's header. This is potentially expensive as these threads make no progress tracing until the copying thread completes. There is also no protection for the case where the copying thread is context switched by the operating system: other threads may be left busy waiting until the copying thread is resumed.

[1] Compare-and-swap updates a memory location M with a new value Y iff its current value matches another value X, reporting the success or failure of the update.

4.1 Transactional Implementation

The obvious objective for a transactional implementation of this algorithm is to remove the intermediate *copying* state. Instead, an object should transition atomically from *uncopied* to *copied*. This transition should be possible for any collector thread irrespective of the state of the other collector threads in the system. In order to amortize the overhead of the atomic update, a number of these updates must be gathered into a single transaction.

To focus on the costs of transactional and non-transactional copying, we investigated Jikes RVM's simplest copying collector. We modified the Jikes RVM semispace collector to use an optimistic copying strategy. Each object is copied to *to-space* without updating the *from-space* object's state. On completion of the copy, the state of the *from-space* object is updated (and the forwarding pointer installed) using a compare-and-swap operation. If this operation fails then the object has already been copied by another collector thread. This process potentially creates redundant copies which are discarded, creating floating garbage. However, unlike the unsynchronised case, references to these copies are not used and hence the integrity of *to-space* is preserved.

It might be thought that it would be better for the GC thread to optimistically *reserve* space, and only to copy the object if it succeeded in installing the forwarding pointer in the *from-space* object. However, the structure of Jikes RVM makes this approach difficult, and we wanted to minimise the changes that had to be made. Furthermore, as we noted above, it may be difficult to unreserve space if the thread loses the race to install the forwarding pointer. These difficulties are exacerbated if we try to deal with multiple objects in a transaction.

Multiple forwarding pointer installations can be combined within a transaction to reduce the overhead of the atomic operations. This is achieved by buffering forwarding pointer/state updates to *from-space* objects and flushing the buffer within a transaction. Any writes dependent on the final value of these updates must also be buffered in order to maintain heap integrity. These writes include updates to the fields in *to-space* from which the trace discovered the copied objects. If dependent writes are not delayed then *to-space* objects may prematurely become visible.

In Jikes RVM copying to *to-space* occurs while tracing objects. Edges are updated to point at *to-space* as they are traced. These updates depend on the final value of the *from-space* object state. Hence these updates cannot be made until *to-space* address of the object is committed. This preserves the integrity of the heap. Tracing of copied objects is also delayed until the buffer has been committed to avoid traversing cyclic subgraphs.

Within a transaction each buffer entry is processed as follows:

1. Load the buffer entry.
2. Load the state of the *from-space* object.
3. Update the *from-space* object state if it is still *uncopied*.
4. Commit writes dependent on the *to-space* value. If the *from-space* object was updated, this will be the address stored in the buffer entry. Otherwise it will be the forwarding pointer stored in the *from-space* object (by another collector thread).

Only *from-space* object state updates and dependent edge updates are buffered. This minimises the size of the transaction. In the worst case the transaction size is three cache lines per copy: the buffer entry, the *from-space* object state and the dependent write. Thus transactions of up to 85 updates are theoretically possible. However, there are also architectural reasons why some transactions may never be able to complete, such as page faults or insufficient associative space in cache or TLB entries.

When a hardware transaction fails a fallback is needed. In our case we fall back to using atomic compare-and-swap operations to update *from-space* object state. The result of the compare-and-swap determines the value of its dependent writes. It is not possible to determine which memory access caused the transaction to fail, hence if the transaction fails we use this method to flush the entirety of the present transaction buffer.

4.2 Results

Figure 1 shows the copying speed of four copying methods for a selection of DaCapo benchmarks. Space limitations prevent showing all the DaCapo 2006 and 2009 benchmarks, but the results shown here are representative. Performance is derived from the geometric mean of GC times. Results are normalised against the copying speed of the unmodified semispace collector with a single collector thread to show speed up. Within the figure the *std* series represents the unmodified collector. The *opt* series represents the modified collector performing optimistic copying, but not buffering updates. The series *htm* and *cas* are the collector configured to buffer 16 object updates and commit these updates using hardware transactional or compare-and-swap (fallback) mechanisms respectively. A transaction size of 16 objects provides the largest transaction size where the mean rate of transaction failures is less than 0.5% (and 1.5% of objects are duplicated due to buffering).

There are three cases where the hardware transactional variant outperforms the unmodified collector: *hsqldb* 4-threads, *antlr* 8-threads and *pmd* 8-threads. Of these the optimistic variant has better performance in two cases. Hence there is no clear benefit from using hardware transactional memory in this scenario. Additionally, optimistic copy appears to provide no clear performance or scalability benefit and typically results in reduced performance.

Comparing the hardware transactional and compare-and-swap variants gives an indication of the performance gain from transactional memory as the infrastructure cost is the same. Here hardware transactional memory provides performance increases of up to 20% (e.g. *antlr* 1, 2 and 4 threads) and a mean performance increase of 2%, 5% or 4% for 1, 2 or 4 threads respectively. Using all hardware threads (8 collector threads) degrades hardware transactional performance (mean 2% performance drop); this is to be expected as the transaction buffer (L1 cache) is shared between hardware threads. Given that the performance of the transactional collector does not surpass that of the unmodified collector (with the noted exceptions), it is clear that the architectural cost of buffering updates to form transactions negates its associated performance gains.

5. Concurrent Copying Collection

Section 4 described a collection scenario in which only collector threads are executing during collection: *stop-the-world* collection. However, concurrent GC threads must also synchronise with mutator threads. In this section we investigate explore how transactional memory can be used to accelerate copying in an implementation of the Sapphire concurrent collector [11].

Garbage collection in Sapphire has three distinct phases of activity: *tracing*, *copying* and *flipping*. In the tracing phase all reachable objects in *from-space* are allocated a *to-space* 'shell'. This is similar to stop-the-world copying collection but the object contents are not copied in this phase. An allocation barrier ensures that any newly allocated objects are also allocated shells in *to-space*. Once all objects have been traced the copying phase begins.

During the copying phase the collector copies the fields of each *from-space* object to its *to-space* shell. Mutator threads use a write barrier to replicate updates to both *from-space* and *to-space* versions of the object. Collector thread writes to the *to-space* shells are performed using compare-and-swap operations to ensure that mutator updates will not be overwritten by the collector copying activity. Critically, copying is 'semantic': *to-space* copies point

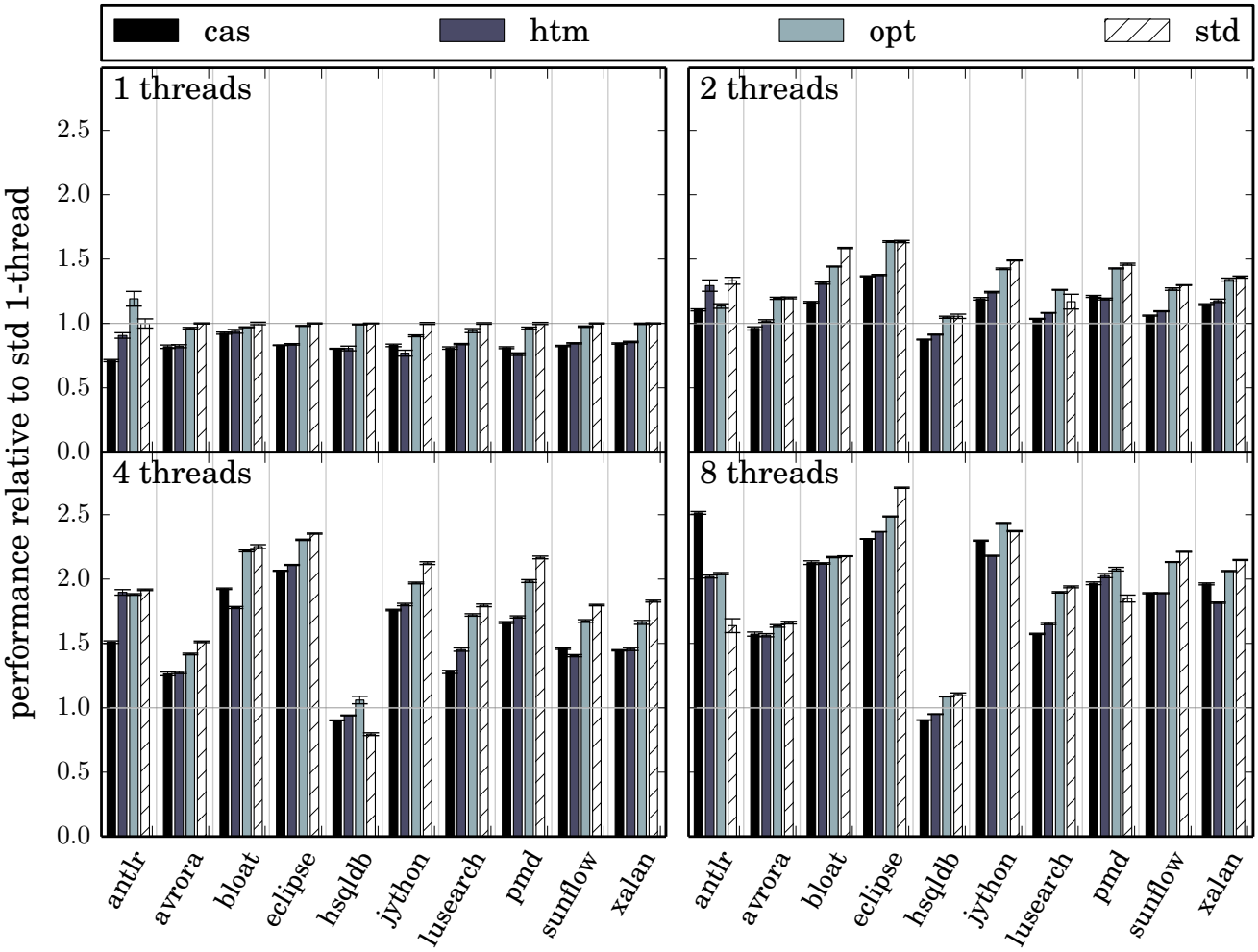

Figure 1: Performance (based on time per collection) of different copying methods. Results are normalised against the unmodified semispace collector (*std*) with one thread (higher is faster). The *opt* series shows optimistic copy, *htm* shows the hardware transactional method and *cas* the compare-and-swap fallback method of executing transactions.

only at *to-space* versions of objects. Once all objects have been copied the flip phase begins.

In the flip phase, references to *from-space* objects are replaced with references to their *to-space* versions, and mutator threads begin reading from *to-space* objects. Flipping is applied to roots and mutator stacks only: references in *to-space* already point at *to-space*. Collection is complete when all references to *from-space* have been flipped. Within the copying phase two significant overheads exist:

- semantic copying requires conditional copying of each field,
- compare-and-swap is significantly slower than simply writing to memory.

Semantic copying can be optimised by data structuring such that fields can be copied in linear word-by-word fashion. We modified Jikes RVM to ensure that reference maps are always ordered, allowing us to process object data word-by-word efficiently. Once all reference fields (and preceding data) have been semantically copied, any remaining data can be copied unconditionally word-by-word. This leaves the use of compare-and-swap as a good candidate for replacement with transactional memory instructions.

5.1 Hardware Transactional Method

The simplest approach is to copy a single object in each transaction. This method is implemented using a version of the semantic copy which does not rely on compare-and-swap. A transaction is started and this method is then invoked. If the transaction fails then we can fall back to the compare-and-swap version.

Object size is typically less than that of a single cache-line (64-bytes) [6]. This represents the expected write set of a transaction containing a single object. However, semantic copying requires the dereferencing of each reference field of the source object. On a 32-bit machine a single cache-line can hold 16 references, each of which may refer to a distinct object, hence a transaction's read-set may grow to 17 cache lines or 1088 bytes at most. In the best case the object has no reference fields and hence the read-set is only 64 bytes.

Our earlier work indicated that transactions up to 16KiB are possible on Haswell [20]. The estimates of transaction size above show there is scope for copying multiple objects within a transaction. Based on a read and write set total of 1152 bytes, 13 whole objects will fit in a transaction, disregarding other overheads.

Inline copying A multiple-object transaction can either be constructed inline with scanning of the heap, or planned by building a 'to-be-copied' list during the scan. Inline copying starts a transaction (*XBEGIN*) and scans the heap as normal. On visiting each object the scan checks if the object will fit in the transaction. If not, then the transaction is committed (*XEND*) and a new transaction is started. Otherwise the object is copied within the open transaction and its size added to the transaction size. Inline transaction construction has the disadvantage that scanning-related reads will be included in the transaction.

Planned copying removes the open transaction. Each object visited by the scan is added to a to-be-copied list. When the list reaches the desired size, a transaction is initiated and all objects are copied before committing the transaction. Planning removes scanning traffic from the transaction, but has a comparatively more heavy weight implementation and associated overheads. Various other activities may be performed as part of planning the transaction, for example looking up and caching object type information (so that associated reads do not inflate the transaction). In section 5.3 we evaluate different methods of constructing a transaction.

5.2 Software Transactional Method

In addition to a hardware transactional implementation we also tested a software transactional implementation. Rather than apply a general purpose transactional solution we constructed a minimal mechanism just for the copying phase of our Sapphire implementation. This is facilitated by the fact that the *to-space* replica of an object is not read until after the copying phase is complete and thus does not need to maintain consistency during the phase itself.

Our software transactional method comprises copying and verification steps which are performed for each object. The *copying step* semantically copies a *from-space* object to its *to-space* replica without using compare-and-swap. Any reference that has to be resolved as part of the semantic copy is stored in a buffer. A memory barrier (an *MFENCE* instruction on x86) is used to separate the copying step from the verification step. In the *verification step* the contents of a *to-space* replica are semantically compared to the *from-space* object; reference fields in the *from-space* object are compared against the buffer created during copying. If the two objects are consistent, the object has been successfully replicated. If at any point the objects are found to be inconsistent then the object is copied again using the fallback compare-and-swap method.

5.3 Results

To evaluate the performance of the different concurrent copying methods described in this section, we measured the speed of copying objects between *from-space* and *to-space*. We measured this by instrumenting Jikes RVM to record the time taken for the copy phase of collection and the number of bytes copied during this phase. These measurements are used to compute the copying speed of each copying phase independently. The geometric mean of these measurements is then computed from all measurements collected across all runs of a given DaCapo benchmark. Each benchmark may have a different copying speed due to variations in reference density; references will slow copying as they must be resolved. The measurement architecture was the same for all tests and thus any overhead was constant.

First we investigated the multi-object copying methods described in section 5.1 to determine the optimal transaction size for these. We parameterised these methods over the number of bytes written to *to-space* as a proxy for the overall transaction size. While computing the size of the transaction required to copy an object is possible at run time, the overhead incurred would be detrimental to performance. Due to semantic copying each word written to

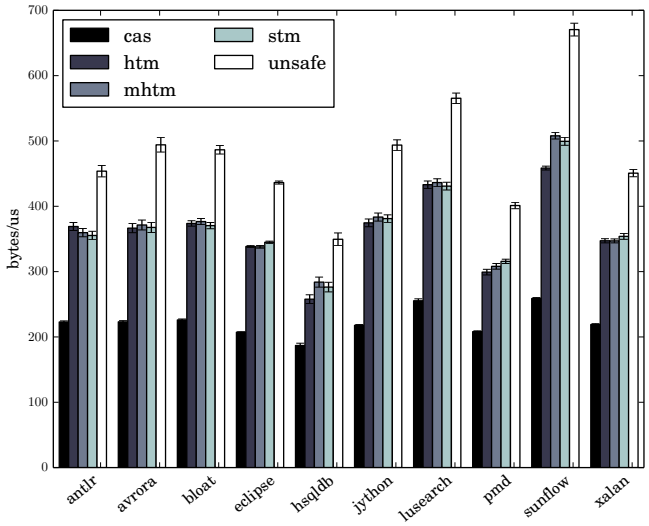

Figure 2: Speed of different copying methods in Sapphire with mutators stopped. The base compare-and-swap method is shown as *cas*. Hardware transactional copy is shown as *htm* and the planned multi-object variant with 256 byte transactions as *mhtm*. Software transactional copy is shown as *stm*. Unsynchronised copying, a method only safe when mutators are stopped, is shown as *unsafe*.

to-space can cause an additional cache line to be added to the transaction's read set. Hence the worst case transaction size required to copy and object is proportional to the *to-space* size of the object.

Figure 3 shows the performance of three different multi-object copying methods across a range of DaCapo benchmarks and transaction sizes. For these tests mutators are not active during the copying phase and hence results represent the *optimal* performance. The inline scanning method is shown as *mhtm inline*. Planned transactions are shown as *mhtm plan* and *mhtm full*. For *mhtm plan* the transaction plan consists of the *from-space* and *to-space* addresses of each object to be copied. With *mhtm full* the plan also includes two words of header information from the object's type information block (TIB) in order to further reduce the number of cache lines referenced during the transaction.

At small transaction sizes the performance of all three variants is very similar. As transaction size increases, a tipping point is reached, for the inline variant at around 128 bytes (approximately 2–3 objects). After this tipping point the performance of the inline variant rapidly degrades, presumably as a result of the read set size becoming too large for most transactions to complete.

Performance of both *mhtm plan* and *mhtm full* is stable once an optimal transaction size has been reached. Some gradual decline in performance can be seen, but a tipping point has clearly not been reached. The *full* variant is always marginally slower that the more basic *plan* variant. This suggests that the overhead of preloading the TIB data is not amortized by any performance gained by this optimisation. Based on these results we only evaluated the basic planning model *mhtm plan* with a transaction size of 256 bytes.

Figure 2 shows the copying speed of all methods across a selection of DaCapo benchmarks with a single collector thread. These tests represent the *optimal* case as mutator threads are stopped during the copying phase: no interference is present. The base case using compare-and-swap is represented by the *cas* series. The hardware transactional method is shown by *htm* and multi-object hardware transactional by the *mhtm* series. The software transactional

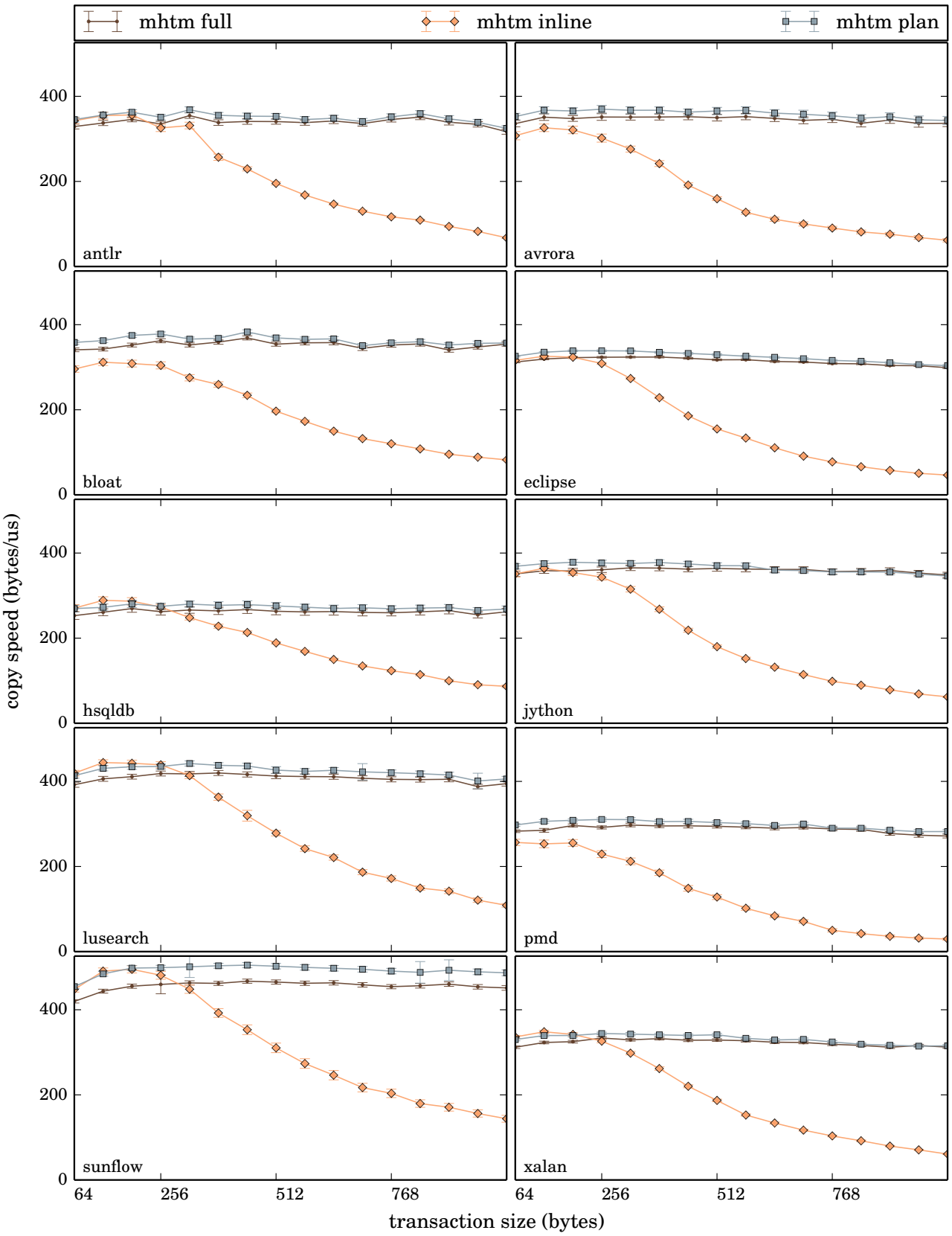

Figure 3: Speed of different transaction size transactions with multiple object copying. Inline transaction construction is shown as *mhtm inline*. Planned transaction construction is shown as *mhtm plan*. Planned transaction construction including TIB pre-loading is shown as *mhtm full*.

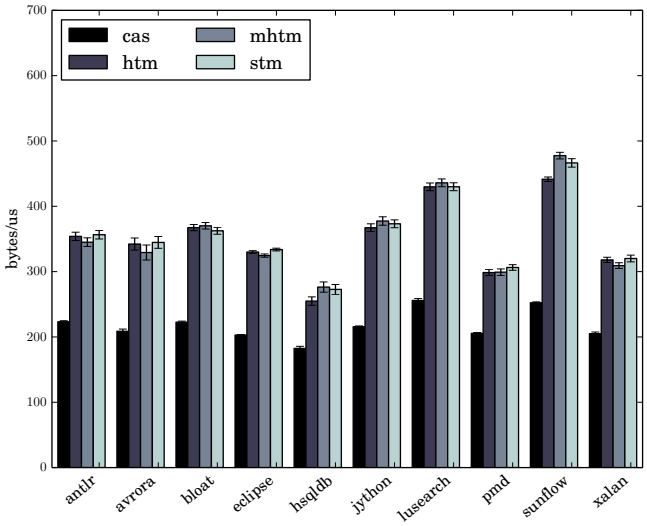

Figure 4: Speed of different copying methods in Sapphire with mutators running. Copying methods are the same as those shown in figure 2.

method corresponds to the *stm* series. As mutators are stopped it is also possible to use an unsynchronised method of copying: this is corresponds to the *unsafe* series. This series is unsafe because it does not prevent inconsistencies arising from concurrent mutator activity. Copying with transactional methods is 60-80% faster than the base *cas* variant. However, there is still a significant difference in copying speeds between synchronised variants and the unsynchronised case (*unsafe*). It is significant that *stm* and *htm* variants have comparable performance. This suggests that transactional performance gains can be made without hardware support.

Figure 4 reproduces the tests shown in figure 2 with results from the realistic configuration with mutator threads active during the copying phase. The transactional methods, both hardware and software, continue to show substantial performance improvements over the original method of copying objects using CAS instructions. Results for some benchmarks with many mutator threads, in particular *sunflow* and *xalan*, show some reduction in performance of the transactional methods. While the pattern of performance is broadly similar to the case with mutators stopped, there is a slight indication that the *mhtm* method is affected more by mutator contention.

Figure 5 shows the speed of different copying methods with increasing numbers of collector threads. While the processor used for testing has four cores, each of these cores has two hardware threads, so we explore scaling up to eight collector threads. For *cas*, *htm* and *stm*, peak performance is reached with six or seven collector threads. As most benchmarks are not heavily multi-threaded this may correspond to the case where there are one or two mutator threads executing in parallel with the collector threads. Adding further collector threads causes context switching with the mutator and hence does not yield any further gain in performance.

Significantly, the multi-object hardware transactional method (*mhtm*) does not scale as well as other methods. Haswell uses the processor's L1 cache to hold a transaction's read and write sets. Hardware threads on the same core share the L1 cache. As *mhtm* creates larger transactions there is greater contention on L1 cache, leading to more transaction failures with higher numbers of collector threads and reduced copying speeds. In particular *antlr* shows an oscillation in performance. This may correspond to the mapping of collector threads to hardware threads on the same or different cores; this requires further investigation.

6. Bitmap Marking

All tracing collectors need some mechanism to mark the objects that they have visited. Non-moving mark-sweep collectors do so by setting a mark either in the object's header or in a separate side table. In principle, a single bit is sufficient for the mark, and there is usually space for it in an existing header word. However, some collectors use a small number of bits so that the value of a mark used for the current collection is different from that used for the next collection. This removes the need for the sweep phase to clear the marks from live objects and hence reduces the number of objects modified in the cache.

The alternative to storing the mark bit in object headers is to use separate bitmaps. The size of the bitmap depends on two factors: the alignment of objects in the heap and the number of bits used to represent a mark. For example, a system that allocates objects on double-word boundaries will require a mark bitmap half the size of that required by a system that allocates on single-word boundaries.

The simplest design is a single bitmap to represent the entire heap. However, in a block-structured heap, a separate bitmap can be used for each block. The latter organisation has the advantage that no space is wasted if the heap is not contiguous. This organisation also permits varying alignment of objects per block, e.g. for 'big bag of pages' schemes, further increasing the density of bitmaps. Per-block bitmaps might be stored in the blocks. However, placing the bitmap at a fixed position in each block risks degrading performance as the bitmaps contend for the same sets in a set-associative cache. A solution is to vary the position of the bitmap in the block using some simple hash of the block's address to determine an offset for the bit map. Alternatively, the bitmap can be stored to the side, somehow indexed by the block, again perhaps by hashing [4].

Mark bitmaps have a number of potential advantages over mark bits in object headers. A bitmap stores marks more densely. Marking with a bitmap modifies only the bitmap, not objects; conservative collectors use bitmap marking for this reason: as they are not type accurate, they dare not modify data in the heap. Sweeping need neither read nor modify live objects. Moreover, given the tendency of objects to live and die in clusters [9, 16], use of a bitmap allows a sweeper to test the liveness of several objects at a time in the common case that every bit is set or every bit is clear. A corollary is that it is simple to determine from the bitmap whether a complete block is garbage. Overall, bitmap marking might be expected to dirty fewer cache lines than header marking.

However, bitmaps also have disadvantages. A collector with parallel marking threads must ensure that GC threads do not interfere with each other as they update a bit map word: the update must be indivisible. Either the bitmap must be updated with an atomic instruction or bytes rather than bits must be used for marks. In contrast, parallel collector threads can use plain stores to mark header words since the action of setting the mark bit(s) is idempotent. However, this may not be the case for a concurrent collector in which both mutator and collector threads are active simultaneously. If mark bits share the same header words as other runtime structures such as lock or hashcode bits, both mutator and collector operations must be synchronised.

In this section we ask, can transactional memory techniques reduce the synchronisation overheads required by bitmap marking in Jikes RVM on Intel's Haswell architecture?

6.1 Transactional Method

Updating a single mark in a bitmap is insufficient to amortize the cost of transaction setup and shutdown: a transaction must set multiple marks to be efficient. As with concurrent copying (section 5) there are two potential methods for constructing a transaction: *inline* and *planned*. We evaluated a method of constructing the transaction inline with heap tracing and found that transaction formation

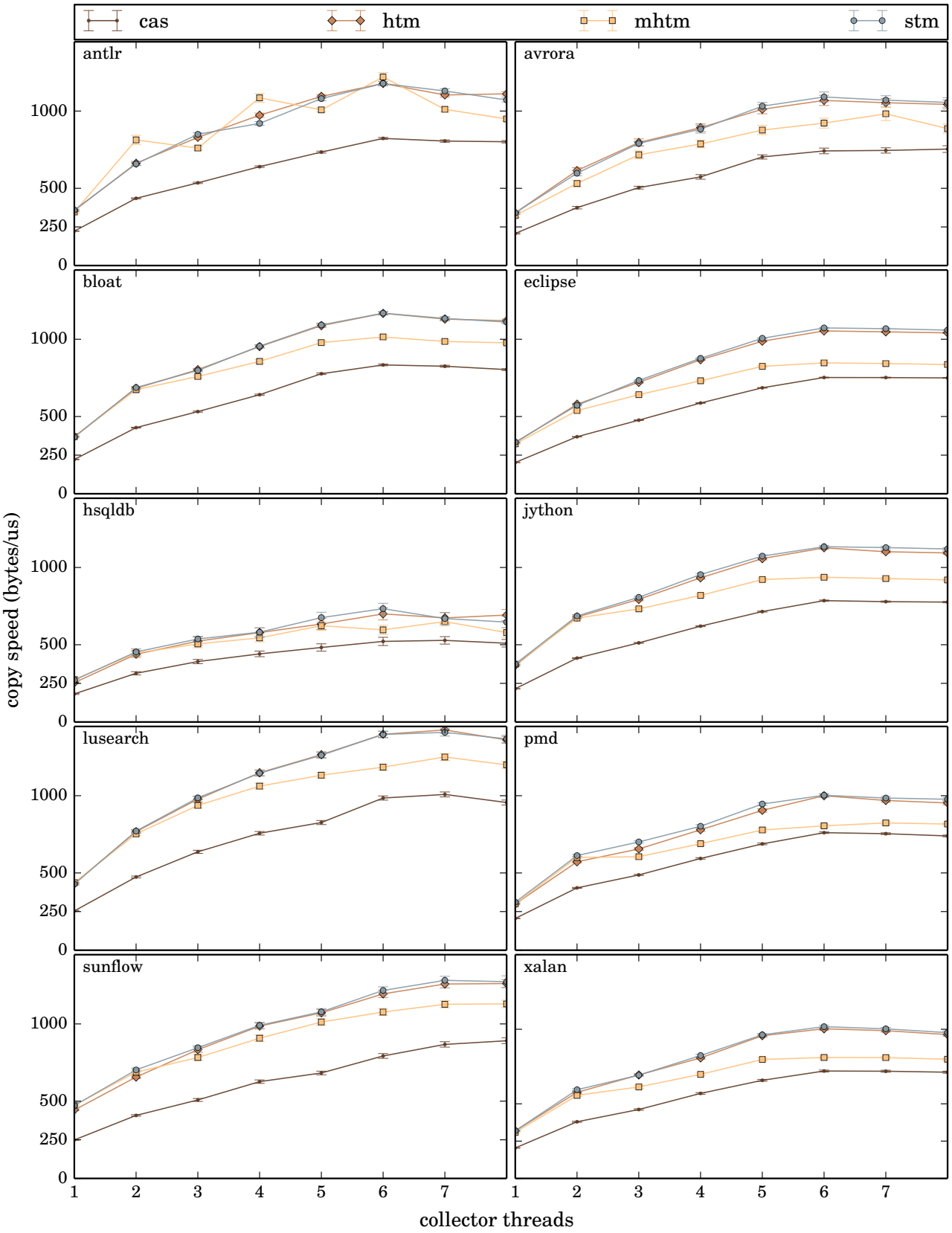

Figure 5: Speed of copying methods with increasing numbers of collector threads. The series are the same as figure 2.

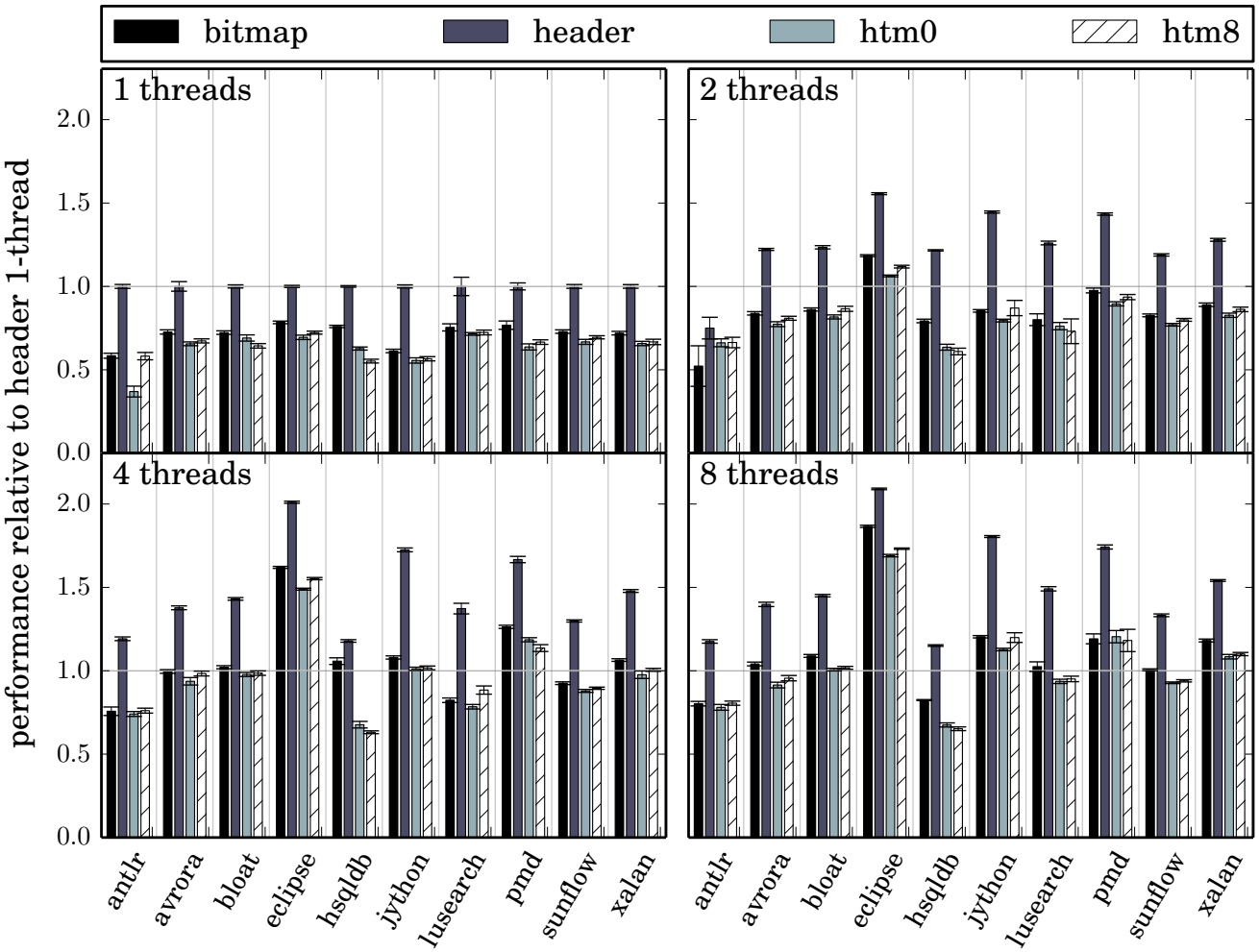

Figure 6: Performance (based on time per collection) of different marking methods. Results are normalised against the unmodified mark-sweep collector using header marking (*header*) and one thread. The *bitmap* series shows the same collector using bitmap marking. The hardware transactional architecture without using transactions is shown as *htm0*, and with transactions of eight marks as *htm8*.

and mark queue interaction interfered. Pushing and popping from a shared queue of gray objects requires thread synchronisation operations which cannot be performed within a transaction. These operations are non-trivial to remove so we abandoned an inline approach.

A planned transaction can be constructed by buffering updates to the marking bitmaps. During the trace when an object is to be marked, the bitmap is not updated; rather, the relevant bitmap address and mark bit are stored in a buffer. Once the buffer reaches a given size, it is flushed by writing the relevant bitmap words within a transaction. In the event of transaction failure the buffered updates are applied using compare-and-swap.

This planning scenario produces an almost ideal transaction as the read and write sets of the transaction need only contain the buffer and the bitmap words updated. However it is important to consider that each bitmap word updated by the transaction may lie on a different cache line. Hence the worst case transaction size is 64 bytes for each mark, plus 64 bytes buffer bytes per eight marks. We determined by testing (results not shown) that optimal performance was reached with transactions of eight marks.

6.2 Results

We tested the performance of transactional marking using Jikes RVM's mark-sweep collector. Figure 6 compares the performance of different marking methodologies. As with parallel copying in section 4, our performance metric is GC time.

The default configuration of the mark-sweep collector is to use header marking, rather than bitmap marking. Header marking (labelled *header*) does not require synchronisation and thus represents the unsynchronised case; we normalise results from other configurations against this base case. Standard bitmap marking with a CAS operation is represented by the *bitmap* series.

Results for the transactional method are shown by the *htm8* series. To evaluate the impact of the architectural changes required for transactional marking, we include results for the case where marks are captured during the trace, but are committed immediately using compare-and-swap (*htm0* series).

Clearly performance of bitmap marking is significantly lower than header marking (30–40% slower). Some of this performance decrease can be attributed to the need for synchronisation on bitmap marking; it is this that we hope to reclaim by using transactions.

Comparing *htm0* to *bitmap*, we see that introducing our architecture for buffering marking reduces performance by a further 10%. A key architectural cost of buffering marks is the introduction of branching and indirect call costs on each mark operation. Clearly these have a significant impact on the pipelining activity of the processor and thus overall performance.

Once margins of error have been considered there is no clear case where the transactional implementation (*htm8*) surpasses the performance of synchronised bitmap marking. This suggests that architectural costs have not been amortized by performance gains. Furthermore this performance margin is maintained with increased numbers of collector threads. Hence we can also conclude that the transactional method does not have any improved scaling behaviour. However, we did not explore different organisations and placements of the bitmaps in order to reduce the risk of contention or false sharing of cache lines.

7. Related Work

The earliest work on GC transactions discussed it in the context of transactional systems, such as reliable distributed systems [5], object-oriented databases [2] or persistent object stores [25]. Here the main issues were how to expose rollback mechanisms to the collector. in contrast, we are interested in using transactional memory to support GC.

Transactional memory was first proposed by Herlihy and Moss [10] as a hardware architecture intended to make lock-free synchronization as efficient (and easy to use) as conventional techniques based on mutual exclusion. Hardware transactional memory (HTM) has been provided by Sun Microsystems' Rock processor (although this was never released as a product), IBM's BlueGene/Q and Azul Systems Vega processors. In 2009 AMD proposed the Advanced Synchronization Facility (ASF), a set of x86 extensions to provide a limited form of hardware transactional memory, but has yet to announce whether it will be used in products.

However, Intel's Transactional Synchronization Extensions (TSX) have been implemented in some of its new Haswell commodity processors (2013). Ritson and Barnes evaluate in detail the performance of this restricted transactional memory on a Core i7-4770 processor and examine its potential application for the implementation of communicating process architectures [20].

The idea of implementing transactional memory in software alone (STM) is due to Shavit and Touitou [21]. Researchers have proposed exposing STM to the programmer through language extensions such as `atomic` blocks with `retry` actions [8].

7.1 Transactional memory for copying

To the best of our knowledge, the first use of software transactional memory to support GC was by McGachey et al. [18]. Their concurrent GC supported a version of Java extended with an `atomic` construct. Read and write barriers were used both for transactional code and to provide strong atomicity. Each object stores a transaction version number in its header, which is used by both the mutators and the GC. In contrast to (our transactional memory implementations of) the Sapphire on-the-fly copying GC which places no restrictions on mutators, McGachey et al. must put an object into exclusive mode before writing to it. The GC notes the version number before copying then object and checks that the number has not changed afterwards. If it has, the copy aborts and has to be retried. Only one object is copied in a transaction.

Collie [13] is an on-the-fly, concurrent collector that uses the HTM provided by Azul Systems Vega processors to provide wait-free compaction; again, to the best of our knowledge, it is the first GC to exploit HTM. Collie uses a hardware supported read barrier to ensure that mutators always access to-space replicas of objects, unlike Sapphire which uses from-space objects until its root-flip phase. Collie is mostly compacting: an object that cannot be physically copied wait-free ("individually transplanted") is virtually copied by mapping its from-space page to the same physical memory as its mirrored to-space page. In order to provide wait-freedom, objects referenced from roots or accessed by mutators while Collie is trying to move them are not compacted. During its mark phase [23], Collie constructs conservative "referrer sets" of objects holding references to each object. During the compaction phase, read and write barriers mark objects as non-transportable, replacing references to them (including those in their referrer set) with references to their corresponding address on the mirrored to-space page. In contrast, our copying collectors never pin objects, although Jikes RVM does allocate some object in non-moving spaces. Like our implementations, Collie tries to reduce the size of transactional read and write sets. To transplant an individual object, Collie copies its contents before it starts a transaction. Inside the transaction, it checks the references in the referrer set: if any point to the mirrored to-space, the transaction (and the copy) is aborted. Otherwise the transaction commits.

8. Conclusion and Further Work

In this work we have explored the basic application of hardware transactional memory to GC. Our results show that transactional memory can provide performance gains if sufficient and concise work is available for embedding in a transaction. The work must be *sufficient* to amortize the cost of transaction setup and any other architectural costs, and *concise* enough not to inflate the work that must be performed within the transaction with dependent activity that could be performed outside it.

Our results also suggest that while appropriate work may be available for transactional execution, the architectural overheads of capturing this work within existing collectors may outweigh the benefits of using transactions (section 6.2). Further work is required to establish how collectors may be designed to produce packets of work suitable for transactional execution. From our work here transactional memory is most applicable to concurrent and on-the-fly collectors where synchronisation requirements are complex. Whilst we present results only for Jikes RVM on Intel's Haswell architecture, the requirements for sufficient and concise work make it unlikely that different conclusions could be drawn for other GCs or architectures.

While not fully explored in this work, software transactional approaches can be applied to all the algorithms presented here. Where tested, our software transactional approaches performed as well or better than hardware transactional memory, suggesting that many of the benefits of transactions can be obtained without hardware support. One reason is that we can weaken the requirements for consistency within a software transaction.

The existence of software transactional solutions to the algorithms explored here suggests that the full potential of hardware transactional memory is not being exploited. Hardware transactional memory has the potential to solve problems which cannot be overcome with traditional synchronisation mechanisms as it can provide strong consistency for reads and writes to distributed memory locations. We suggest further work to explore this design space for virtual machines and GC.

Acknowledgments

We are grateful to Rick Hudson and Intel for a license to implement Sapphire in Jikes RVM, and Laurence Hellyer for his work on Sapphire. We are also grateful for the support of the JSPS KAKENHI Grant Number 25330080, the EPSRC through grant EP/H026975/1, Google's Summer of Code and the University of Kent Sciences Faculty Research Fund.

References

[1] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN Notices 34(10), pages 314–324, Denver, CO, Oct. 1999. ACM Press.

[2] L. Amsaleg, M. Franklin, and O. Gruber. Garbage collection for a client-server persistent object store. *ACM Transactions on Computer Systems*, 17(3):153–201, 1999.

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN Notices 41(10), pages 169–190, Portland, OR, Oct. 2006. ACM Press.

[4] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, 1988.

[5] D. L. Detlefs. *Concurrent, Atomic Garbage Collection*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 15213, Nov. 1991.

[6] S. Dieckmann and U. Hölzle. A study of the allocation behaviour of the SPECjvm98 Java benchmarks. In R. Guerraoui, editor, *13th European Conference on Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 92–115, Lisbon, Portugal, July 1999. Springer-Verlag.

[7] A. Georges, L. Eeckhout, and D. Buytaert. Java performance evaluation through rigorous replay compilation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN Notices 43(10), pages 367–384, Nashville, TN, Oct. 2008. ACM Press.

[8] T. Harris and K. Fraser. Language support for lightweight transactions. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN Notices 38(11), pages 388–402, Anaheim, CA, Nov. 2003. ACM Press.

[9] B. Hayes. Using key object opportunism to collect old objects. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN Notices 26(11), pages 33–46, Phoenix, AZ, Nov. 1991. ACM Press.

[10] M. P. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20th Annual International Symposium on Computer Architecture*, pages 289–300, San Diego, CA, May 1993. IEEE Press.

[11] R. L. Hudson and J. E. B. Moss. Sapphire: Copying garbage collection without stopping the world. *Concurrency and Computation: Practice and Experience*, 15(3–5):223–261, 2003.

[12] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, June 2013.

[13] B. Iyengar, G. Tene, M. Wolf, and E. Gehringer. The Collie: a wait-free compacting collector. In McKinley and Vechev [19], pages 85–96.

[14] R. Jones and S. Blackburn, editors. *7th International Symposium on Memory Management*, Tucson, AZ, June 2008. ACM Press.

[15] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Applied Algorithms and Data Structures. Chapman & Hall, Aug. 2012.

[16] R. Jones and C. Ryder. A study of Java object demographics. In Jones and Blackburn [14], pages 121–130.

[17] T. Kalibera and R. Jones. Rigorous benchmarking in reasonable time. In E. Petrank and P. Cheng, editors, *12th International Symposium on Memory Management*, Seattle, WA, June 2013. ACM Press.

[18] P. McGachey, A.-R. Adl-Tabatabi, R. L. Hudson, V. Menon, B. Saha, and T. Shpeisman. Concurrent GC leveraging transactional memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 217–226, Salt Lake City, UT, Feb. 2008. ACM Press.

[19] K. McKinley and M. Vechev, editors. *11th International Symposium on Memory Management*, Beijing, China, June 2012. ACM Press.

[20] C. G. Ritson and F. R. Barnes. An evaluation of Intels Restricted Transactional Memory for CPAs. In *Communicating Process Architectures*, 2013.

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

[22] F. Siebert. Limits of parallel marking collection. In Jones and Blackburn [14], pages 21–29.

[23] G. Tene, B. Iyengar, and M. Wolf. C4: The continuously concurrent compacting collector. In H. Boehm and D. Bacon, editors, *10th International Symposium on Memory Management*, pages 79–88, San Jose, CA, June 2011. ACM Press.

[24] X. Yang, S. M. Blackburn, D. Frampton, and A. L. Hosking. Barriers reconsidered, friendlier still! In McKinley and Vechev [19], pages 37–48.

[25] J. Zigman, S. M. Blackburn, and J. E. B. Moss. TMOS: a transactional garbage collector. In G. N. C. Kirby, A. Dearle, and D. I. K. Sjoberg, editors, *9th International Workshop on Persistent Object Systems (Sept., 2000)*, volume 2135 of *Lecture Notes in Computer Science*, pages 116–135, Lillehammer, Norway, 2001. Springer.