The Influence of Malloc Placement on TSX Hardware Transactional Memory

Dave Dice
Oracle Labs
dave.dice@oracle.com

Tim Harris
Oracle Labs
timothy.l.harris.@oracle.com

Alex Kogan
Oracle Labs
alex.kogan@oracle.com

Yossi Lev
Oracle Labs
yossi.lev@oracle.com

Victor Luchangco
Oracle Labs
victor.luchangco@oracle.com

Abstract

In this paper, we demonstrate the impact of the placement policies of memory allocators on the performance of applications that use hardware transactional memory. In particular, commonly used allocators such as the default GNU `glib malloc` allocator may place objects in such a way that causes hardware transactions to consistently abort, even when running single-threaded. In multithreaded applications, these consistent aborts can force applications to fall back to using locks, significantly limiting the parallelism. We also show that using index-aware allocators can avoid these pathological memory placements.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel Programming

General Terms Performance, experiments, algorithms

Keywords Concurrency, threads, caches, multicore, malloc, dynamic memory allocation, hardware transactional memory

1. Introduction

As hardware transactional memory (HTM) becomes available on commercial processors, it is important to understand how to use it effectively. One aspect of this is to understand the impact of various system design choices on the performance of applications that use HTM. In this paper, we explore the impact of cache geometry and the placement policies of memory allocators on hardware transactions. Because HTM is implemented by tracking the transactions' read- and write-sets in the caches, infelicitous placement of objects during allocation can prevent transactions from committing, which naturally has a significant negative impact on performance. In particular, we show that the way commonly used allocators such as the default GNU `glib malloc` allocator place objects may cause hardware transactions on Intel's i7-4770 "Haswell" processor [12, 13] to abort consistently, even when running single-threaded. In multithreaded programs, this effect can be much worse: the consistent aborts can force applications to fall back to using locks, significantly limiting the parallelism that can be achieved by optimistic execution using HTM. We also show that allocators that take care to reduce unnecessary conflicts due to cache geometry, such as index-aware allocators, also reduce the likelihood of these problems.

The i7-4770 provides hardware transactional memory (HTM), the implementation of which is similar to that in Sun's *ROCK* processor [6]. Our particular interest is in the use of *Restricted Transactional Memory* (RTM) for *transactional lock elision* (TLE). The critical section body contains unmodified HTM-oblivious legacy code that expects to run under the lock in the usual fashion, but via TLE we can modify the lock implementation to attempt optimistic execution, reverting to classic *physical locking* only as necessary. The i7-4770's HTM implementation tracks the transactional write-set in the L1 (level-1) cache and the read-set in the L1, L2, and L3 caches. At most one cache can have a given line in *modified* or *exclusive* state at any one time—a classic multiple-reader single-writer model. RTM on the i7-4770 uses a requester-wins conflict resolution strategy implemented via the coherence protocol. Eviction or invalidation of a tracked cache line results in a transactional abort. For example if a transaction on CPU C loads address A, and some other CPU writes A before C commits, the write will invalidate the line from C's cache and cause an abort. Similarly, if C stores into A and some other CPU loads or stores into A before C commits, the invalidation of A will cause $C's$ transaction to abort. Read-write or write-write sharing on locations accessed within a transaction results in coherence invalidation and consequent abort.

In addition to coherence traffic, self-displacement via *conflict misses* [10] can also result in aborts. Conflict misses – sometimes called *mapping misses* – arise because of less than ideal associativity and represent imbalanced distribution of active memory lines over the set of available L1 indices.[1] Unlike data conflict aborts, aborts due to conflict misses may occur even in a single threaded execution, and are consistent: that is, retrying the transaction will consistently fail as long as it accesses the same set of memory locations. These consistent aborts may cause a significant performance

[1] We caution the reader about confusing terminology. A *data conflict abort* occurs when a transaction running on CPU A reads a location on some cache line L and another CPU B subsequently – but before $A's$ transaction can commit – writes into L *or* if A writes to L in a transaction and B concurrently reads or writes to L before A commits. Put another way, if CPU A has L in its read or write set, and accesses by CPU B invalidate L from $A's$ cache, then $A's$ transaction will consequently abort. Critically, aborts arising from *conflict misses* are distinct from *conflict aborts*.

degradation due to the wasted futile effort. Furthermore, with a solution such as TLE where the fallback execution method when HW transactions consistently fail is execution under the lock, these aborts lead to serialization of all critical sections that exhibit this behavior, significantly limiting the potential parallelism that TLE may have to offer. Thus, while conflict misses exhibit a non negligible cost even with programs that do not use HW transactions, once HW transactions are involved the cost becomes significantly higher. It's worth pointing out that most application/allocator combinations don't exhibit excessive index conflicts, but for those that do, the performance impact can be significant.

The closest related work is that of Baldassin et al. [3], which explores the impact of `malloc` allocator implementations on the performance of applications that use software transactional memory. However, they do not address the interplay between hardware transactional memory and allocator placement.

2. System Description

We now provide some detailed information on the cache geometry of the i7-4770 processor, to better understand the cause of these conflict misses, and explain how index-aware allocators can help avoiding them.

The Intel i7-4770 processor has a relatively simple L1 cache geometry. The L1 data cache is 32KB with 64-byte lines, physically tagged, and 8-way set-associative. There are 64 possible indices (sets). As such the *cache page* size is 4KB – addresses that differ by an integer multiple of 4K will map to the same index (set) in the L1 and compete for the 8 lines within that set. The L1 contains 512 lines. Each core has private L1 and L2 caches, while the L3 is shared by all cores on the chip. The L2 and L3 are unified – able to contain both code and data. The L2 instances are 256KB each and 8-way set-associative, and the single common per-chip L3 is 8MB and also 8-way set-associative.[2] The low-order 6 bits of the address presented to the L1 form the offset into the line, and the next higher 6 bits serve as the L1 index. The MMU base page size is 4KB, so there is no overlap between the virtual page number and the L1 index field in a virtual address. The L1 index field passes through address translation verbatim.[3] As such, operating system-level page coloring [14] is not effective in the L1. (An advantage of this design is that indexing can commence before the virtual address is translated to a physical address, although the cache still ultimately needs the physical address for tag comparison). Some CPUs hash addresses [11] – usually XORing high-order physical address bits into the index bits – in order to reduce the odds of index hotspots and imbalance, but experiments suggest that does not appear to be the case with the i7-4770's L1.

Such simple caches – particularly without the index hashing mentioned above – can be vulnerable to excessive index conflicts, but `malloc` allocators can be made *index-aware* [2] to mitigate and reduce the frequency of index conflicts. Index imbalance results in underutilization of the cache. Some indices will be "cold" (less frequently accessed) while others are "hot" and oversubscribed and thus incur relatively higher miss rates. An index-aware allocator can act to "immunize" an application against some common cases of index-imbalance while typically incurring no additional cost over index-oblivious allocators.

An example for an index-aware allocator is the *CIA-Malloc* allocator by Afek et al. [2]. In addition to being index-aware, CIA-Malloc has a number of useful design properties. It is NUMA-friendly and large-page-friendly. Underlying pages are allocated on the node where the `malloc` operation was invoked. More precisely, the pages underlying a block returned by `malloc` will typically reside on the node where the `malloc` was invoked. The allocator is also scalable with very little internal lock contention or coherence traffic. Each per-CPU sub-heap has a private lock – the only source of contention is via migration or preemption, which are relatively rare. The critical sections are also constant-time and very short. The implementation also makes heavy use of the `trylock` primitive, so if a thread is obstructed it can usually make progress by reverting to another data structure. Remote `free` operations[4] are lock-free. In addition to distributing blocks over cache indices – reducing index imbalance – the allocator also tends to more equitably distribute allocated blocks over coherence planes [9], cache banks and DRAM channels, resulting in reduced channel congestion. Critically, the allocator acts to reduce the cost of `malloc` and `free` operations as well as the cost to the application when accessing blocks allocated via `malloc`. The allocator is also designed specifically to reduce common cases of false sharing : allocator metadata-vs-metadata; metadata-vs-block; and inter-block block-vs-block. Metadata-vs-metadata sharing and false sharing is reduced by using per-CPU sub-heaps. False sharing arising between adjacent data blocks – blocks returned by `malloc` – is addressed by placement and alignment. These attributes prove even more useful when we use CIA-Malloc in conjunction with hardware transactions. Specifically, allocator-induced false sharing results in so-called *coherence misses* in normal execution mode, but in transactional mode those misses translate into aborts, which are typically more expensive than cache misses.

In this paper we demonstrate the additional cost that conflict misses impose on applications that use HTM. To do that, we adapted the CIA-malloc implementation to the cache geometry of the Intel i7-4770 processor (as originally the CIA-malloc was designed for the SPARC T2+ processor), and compare the benefit of using this allocator with and without HW transactions. We use two micro-benchmarks: one that run single threaded, and the other that run multiple threads using TLE.

3. Evaluation

We now show some examples of the influence of virtual address placement on HTM aborts. All tests were run on an Intel i7-4770 processor with turbo mode disabled and sufficient cooling capacity to avoid any thermal throttling. The i7-4770 has 4 cores with 2 virtual "hyperthreads" per core and runs at 3.4GHz. The system was running Ubuntu 14.10 with a Linux 3.16 kernel. All applications and libraries were written in C or C++ and compiled with gcc 4.9.1 in 64-bit mode.

We first consider a single-threaded microbenchmark which, at startup, allocates a set of 128 nodes. Each node has a *Next* field at offset 0 followed by a 32-bit integer field *Value*. The benchmark allocates each node individually via `malloc` and then organizes the nodes into an intrusively circularly linked list via the *Next* field. Since there is a correlation between allocation order and virtual address, we randomize the order of the nodes with a Fisher-Yates shuffle in order to minimize the impact of automatic hardware stride-based prefetchers.[5] The benchmark then times 10 million traversals of the ring, where each traversal first calls `pthread_mutex_lock` to acquire a lock, traverses the list, and then releases that lock with `pthread_mutex_unlock`. Each step of the enclosed loop body executes the following :

[2] We were unable to determine inclusivity relationships between the L1, L2 and L3 caches.

[3] We assume the x86 segment descriptor base addresses are set to 0.

[4] A `free` operation is considered remote if the block was originally allocated on a different processor.

[5] Such a randomized order can put additional stress on the translation lookaside buffers (TLBs) by increasing the number of page crossing in a given traversal of the ring. However unless otherwise stated, TLB misses are not a dominant influence in our results.

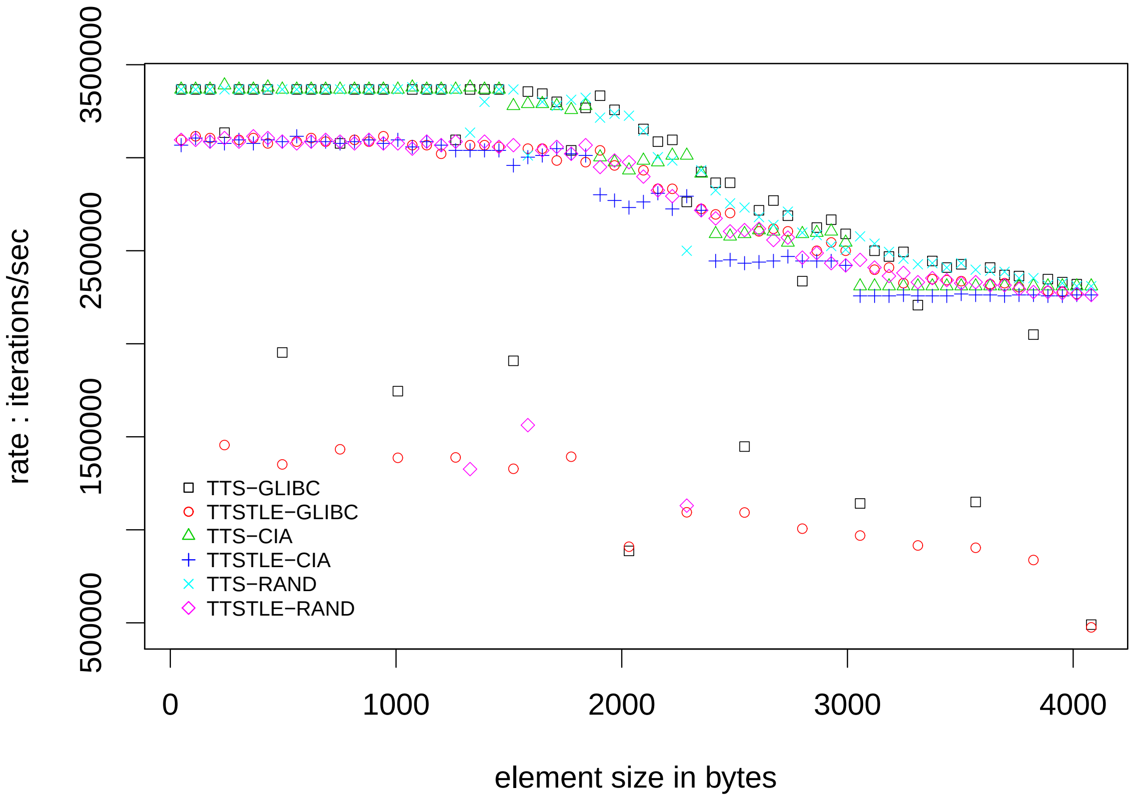

Figure 1: Single-threaded ring traversal rates

```
w = w->Next; w->Value = 0;
```

At the end of the run the microbenchmark reports the iteration rate. (In this context, an "iteration" refers to the act of acquiring the lock, traversing the full circumference of ring, and finally releasing the lock). Crucially, there there are no allocations or deallocations during the measurement interval. Instead, the benchmark times accesses to a set of objects that were previously allocated via `malloc`. Note that only the *Next* and *Value* fields are accessed. The remainder of the element is not accessed during the measurement interval. Such access patterns are not uncommon and can be found in various lookup structures where headers are iterated over but the larger body of an object is less likely to be accessed.

In Figure 1, we plot 6 sets of points corresponding to various combinations of 3 `malloc` allocators and 2 lock implementations. **TTS** is an LD_PRELOAD library that interposes on the `pthread_mutex` family of operators and implements a simple test-and-test-and-set spin lock. **TTSTLE** is just **TTS** augmented with simplistic TLE. All coherence conflict aborts are retried indefinitely. Unresolvable aborts – such as those arising from conflict misses underlying the L1 write set – revert to the slow path and traditional **TTS** locking. To avoid the *lemming effect* [7] we use unbounded spinning to wait for the lock before trying or retrying the fast path. **GLIBC** is the default GNU `libc` `malloc` allocator. **CIA** is an implementation of the index-aware allocator described in [2] but has been modified to use the L1 geometry of the i7-4770 and to use size classes that are prime multiples of the cache line size (64 bytes). This helps avoid both intra- and inter-size class index conflicts [6]. **CIA** is implemented as an LD_PRELOAD interposition library. Finally, **RAND** is an interposition library that intercepts `malloc` calls and probabilistically adds a small number to the requested size, and then passes control to the underlying `malloc` in **GLIBC**. Such randomization can intentionally introduce irregularity into the spacing of blocks and act to reduce index conflicts. We include **RAND** because it provides some degree of relief with rather trivial overheads and an extremely simple implementation.

Each point in the figure reports the median of 5 separate runs. The x-axis is the node size, which can be controlled via a command-line argument. The y-axis reflects the traversal rate expressed as iterations per second of the ring. (The microbenchmark also reports additional details such as distribution of node base addresses over the L1 indices). Since there is just one thread, the lock is never contended and the thread never waits.

We can see in Figure 1 that although most points for all allocator and lock combinations cluster at the top of this graph, there are several points where **TTS-GLIBC** significantly underperforms **TTS-CIA** and the main sequence. Degraded performance occurs

[6] The original **CIA** allocator used so-called *punctuated arrays* but for these experiments our implementation avoided that technique and instead depends on the prime-based size class policy noted above.

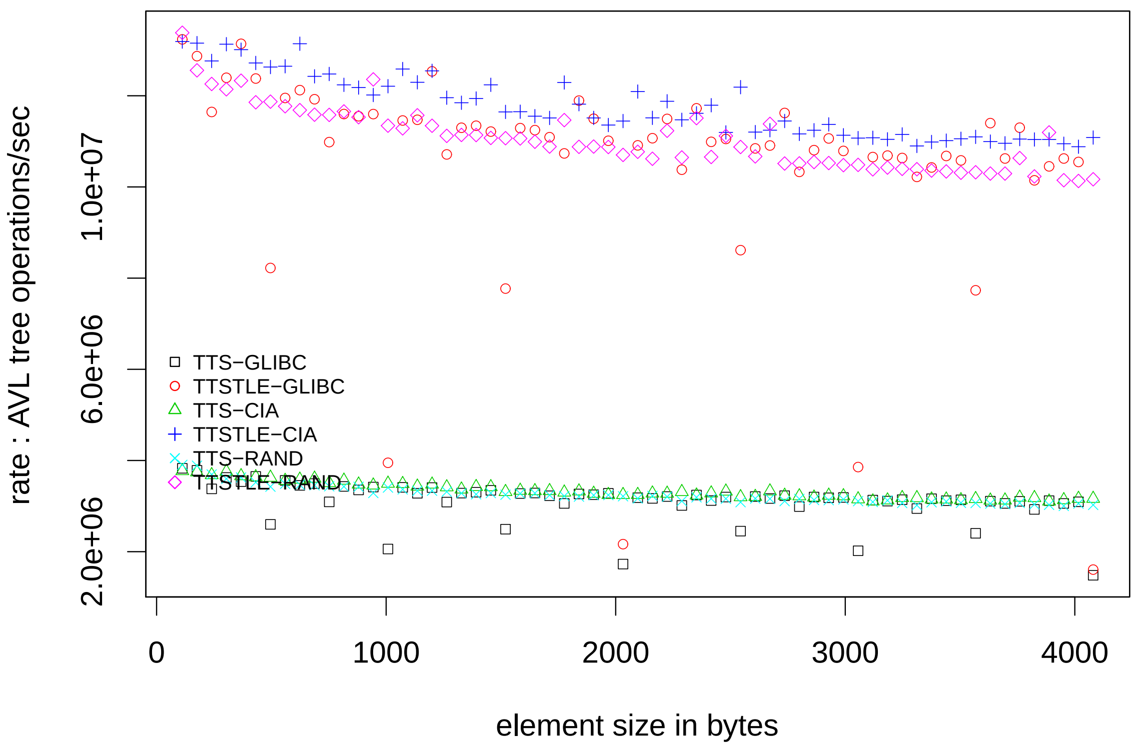

Figure 2: AVL tree throughput with 4 threads

near element sizes of 512, 1K, 1.5K, 2K, 2.5K, 3K and 4K bytes, for instance. Using hardware performance counters, we find that the degraded performance correlates with increased L1 miss rates, supporting our claim that those sizes are index-unfriendly and result in index imbalance and underutilization of L1. When the stride between nodes is 1K, for instance, node base addresses map to just 4 of the possible 64 L1 indices, resulting in potential imbalance and under-utilization of the L1. In more detail, say we have a collection of N elements, each of which was allocated via `malloc(`S`)`. The allocator may place those N objects in a contiguous fashion such that the values returned by `malloc(`S`)` differ by $\bar{S}$. $\bar{S}$ may be greater than S because of quantization and potential per-block `malloc` metadata headers and footers. If $\bar{S}$ – the effective stride – happens to be 1K, for instance, then the N blocks may fall on just 4 of the possible 64 L1 indices, resulting in conflict misses as we traverse the collection.

We also observe that **TTSTLE-GLIBC** underperforms **TTS-GLIBC** at those same points. Under **TTSTLE**, conflict misses cause the fast path transaction to abort with an "internal buffer overflow" error code [7]. This abort is not generally retryable, so **TTS-TLE** reverts to the non-transactional normal mode slow path. The transactional attempt was futile and constituted wasted effort. In particular, unresolvable aborts are more costly than cache misses.

[7] To the best of our knowledge, this abort code indicates self-displacement of read-set or write-set elements

Generally, the **CIA** forms outperform **RAND** which in turns outperforms **GLIBC**.

The inflection point in the main sequence at about 2000 bytes arises from level-1 data TLB misses. For the default 4KB page size, the i7-4770 has a 64-entry 4-way set associative level-1 data TLB (L1-DTLB) and a 1024-entry 8-way set associative level-2 unified TLB (L2-TLB). The L1-DTLB thus has a maximum "span" of 256KB (64 TLB entries $*$ 4KB pages). With 128 elements of 2000 bytes each, the best-case minimum TLB footprint of the ring is 256KB, matching the capacity of the L1-DTLB. All the allocators provide reasonably dense and compact placement of the ring elements. The cache footprint of the ring – the number of lines underlying the *Next* and *Value* fields – depends in part on the alignment of blocks returned by `malloc`. **CIA** always returns addresses aligned on 64-byte boundaries while the default **GLIBC** allocator – and consequently the **RAND** allocator – return addresses only guaranteed 8-byte alignment. Under **CIA** both the *Next* and *Value* fields reside on the same cache line, and the cache footprint is simply the number of elements in the ring multiplied by the cache line size of 64-bytes. That is, the cache footprint of the ring is independent of the element size. With 128 elements, the cache footprint is just 8KB, or 1/4 of the L1's capacity. With worst-case pessimal alignment, under **RAND** and **GLIBC** the *Next* and *Value* fields will be split and reside on two adjacent lines. In that case the cache footprint would be 16KB or 1/2 the L1's capacity. If the L1 were fully associative, the ring would fit comfortably in the L1, and subsequent traversal could be completed with any misses. But because of index conflicts, traversals may be subject to conflict misses.

Broadly, the **GLIBC** allocator exhibits reduced performance at certain pathological sizes. At such problematic sizes, **TTSTLE** underperforms **TTS** by a significant margin because of cycles wasted on futile transactions. **RAND** provides some benefit relative to **GLIBC** but in some cases yields poor performance. **CIA** avoids the pathological sizes completely. We note in passing that two horizontal "bands" appear in the figure on the upper left-hand side of the graph. Transactional executing appears to be slightly slower than normal execution. We believe this is an artifact of higher latencies associated with TSX than occur with the normal atomic operations used to acquire and release a mutex.

In Figure 2 we show how the problem of aborts arising from conflict misses is amplified when using TLE with multiple concurrent threads. In particular, conflict misses cause aborts, which force the lock to use the classic slow path, greatly reducing the opportunities for concurrency.

For the benchmark presented in Figure 2 we use a single shared AVL tree [4, 17] where insert, delete and update operations are protected by a single `pthread_mutex` instance. The AVL tree is based on the implementation used in OpenSolaris [1]. The only changes to the AVL tree code itself were to (a) insert padding (alignment constraints) between the frequently updated element count field and other fields in the tree descriptor structure; and (b) to move the update of that element count to the end of the critical section. These were the only concessions to make the AVL tree code more transaction-friendly. Sequestering the count field as the sole occupant of its own cache sector acts to reduce false sharing and consequent transactional aborts [8]. In addition, new structural and content integrity check routines were added. These are used after a run to check the validity of the tree. The tree is intrusively linked and each node is individually allocated and freed via `malloc` and `free` calls. These allocation and deallocation operations are executed within the critical section. The AVL tree implements a key-value store, with the key and value both being 32-bit integers. Each AVL tree node contains AVL tree linkage, a key field, a value field, and a variable size area. (That area is never accessed). The size of the AVL tree node is controlled by a command-inline argument and is reflected in the x-axis. The benchmark spawns 4 concurrent threads, each of which loops by generating random numbers – via a thread-local uniform pseudo-random number generator – to control the operation type and key [9] : 30% of the time the loop will insert a new key (if the key already exists in the tree, its value is updated); 30% of the time a key is deleted and 40% of the time a lookup is performed. The key range is [0 – 65536). The tree is initially populated to half capacity (32767 elements) with a random set of keys. At the end of a 10 second measurement interval the benchmark reports the aggregate operation rate. (An operation is an iteration of the loop that inserts, deletes or looks up keys in the tree). This rate is shown on the y-axis and expressed in operations per second. We report the median of 5 independent runs.

Not surprisingly, the **TTSTLE** forms outperform the **TTS** forms, as more concurrency is available. But again, for **TTSTLE-GLIBC** we see the same set of pathological sizes as was found in Figure 1. At about 2000 bytes, for instance, **TTSTLE-GLIBC** with 4 threads actually performs worse than the best of the serialized **TTS** forms. This reflects the compounding effect of restricted concurrency and wasted cycles in futile transactions that end in abort.

4. Conclusion

We have shown that the use of index-aware allocators can avoid certain pathological cases where index conflicts cause misses, aborts, and potentially restrict concurrency under TLE.

Put simply, allocator placement can influence conflict miss rates, which in turn influence abort rates, which in turn can force threads to abandon fast-path TLE execution and revert to serialized execution under a lock, restricting parallelism. An index-aware allocator can provide some relief against this phenomenon. Absent such an allocator, randomization of sizes at either the allocation size or in the allocator (**RAND**) may provide benefit by disrupting regularity in placement.

Programming with hardware transactional memory is in its infancy, so the degree to which programs might be afflicted by aborts arising from index conflicts is unknown. Generally, we expect the problem to be infrequent, but when it does manifest, the impact can be surprising and significant. We suggest index-aware allocators as a way to reduce the odds of encountering the problem.

Hardware-based remedies to reduce the rate of conflict misses were suggested Seznec [16] (skew-associative caches) and later by by Gonzales [8] and Wang [18] and Sanchez [15]. All require changes to the hash function that maps addresses to cache indices. By acting to reduce conflict misses, they would also reduce aborts arising from such misses.

We note in passing that under a requester-wins conflict resolution strategy—as found in the current members of the "Haswell" family—it is useful to shift stores, to the extent possible and reasonable, of frequently accessed shared variables toward the end of a transaction [5]. This might be accomplished by hand, or a transaction-aware compiler or just-in-time compiler (JIT) can perform some of the transformations. Shifting reduces the window of vulnerability during which the store resides in the transaction's write-set. (Active transactions are vulnerable in both time and space). But the asymmetry in the i7-4770, where the write-set is tracked in the L1 and the read-set in the L1, L2 and L3, gives us yet another reason to shift stores toward the end of a transaction. Consider a transaction that executes a store followed by large number of loads. Those loads may displace the store from the L1 and cause an abort. If we shift the store to the end of the transaction, the same set of accesses (just reordered) can succeed without abort. The store may displace a loaded line from the L1, but the L2 and L3 can still track the line.

References

[1] Solaris AVL Tree Implementation, 2015. URL `https://github.com/illumos/illumos-gate/blob/master/usr/src/uts/common/sys/avl_impl.h`.

[2] Y. Afek, D. Dice, and A. Morrison. Cache Index-aware Memory Allocation. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, 2011. URL `http://doi.acm.org/10.1145/1993478.1993486`.

[3] A. Baldassin, E. Borin, and G. Araujo. Performance Implications of Dynamic Memory Allocators on Transactional Memory Systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, 2015. URL `http://doi.acm.org/10.1145/2688500.2688504`.

[4] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. ISBN 0070131511.

[8] Because of the *adjacent sector prefetch* facility, we align to 128 bytes instead of 64 bytes, even though 64 bytes is the line size throughout the cache hierarchy and the unit of coherence. The Intel manuals also recommend 128 bytes for the purposes of avoiding false sharing

[9] We opted to use 4 threads to avoid hyperthreaded execution – with 4 threads we have just 1 thread per core

[5] D. Dice. Reducing Transactional Abort Rates Using Compiler Optimization Techniques, 2008. URL http://www.google.com/patents/US20100169870. US Patent Application – US20100169870.

[6] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, 2009. URL http://doi.acm.org/10.1145/1508244.1508263.

[7] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski. Early Experience with a Commercial Hardware Transactional Memory Implementation, 2009. URL https://blogs.oracle.com/dave/resource/smli_tr-2009-180.pdf. Sun Labs Technical Report SMLI TR–2009–180.

[8] A. González, M. Valero, N. Topham, and J. M. Parcerisa. Eliminating Cache Conflict Misses Through XOR-based Placement Functions. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, 1997. URL http://doi.acm.org/10.1145/263580.263599.

[9] R. Hetherington and S. Phillips. Multiple Independent Coherence Planes for Maintaining Coherency, 2008. URL http://www.google.com/patents/US7353340. US Patent 7,353,340.

[10] M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. *IEEE Trans. Comput.*, 38(12), Dec. 1989. URL http://dx.doi.org/10.1109/12.40842.

[11] R. Hund, C. Willems, and T. Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *Security and Privacy (SP), 2013 IEEE Symposium on*, May 2013. URL http://dx.doi.org/10.1109/SP.2013.23.

[12] Intel Corporation. Transactional Synchronization in Haswell, 2012. URL https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/. [online; retrieved 2015].

[13] C. G. Ritson and F. R. Barnes. An Evaluation of Intel's Restricted Transactional Memory for CPAs, 2013. URL http://www.wotug.org/papers/CPA-2013/RitsonBarnes13/RitsonBarnes13.pdf.

[14] T. Romer, D. Lee, B. N. Bershad, and J. B. Chen. Dynamic Page Mapping Policies for Cache Conflict Resolution on Standard Hardware. In *In 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 255–266, 1994.

[15] D. Sanchez and K. Christos. The ZCache: Decoupling Ways and Associativity. In *MICRO 43: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2010.

[16] A. Seznec. A Case for Two-way Skewed-associative Caches. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, 1993. URL http://doi.acm.org/10.1145/165123.165152.

[17] Wikipedia. AVL Tree, 2015. URL http://en.wikipedia.org/wiki/AVL_tree.

[18] W. Zhenghong and R. B. Lee. A Novel Cache Architecture with Enhanced Performance and Security. In *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, 2008. URL http://dx.doi.org/10.1109/MICRO.2008.4771781.