On the Impact of Dynamic Memory Management on Software Transactional Memory Performance

Alexandro Baldassin
UNESP – Univ Estadual Paulista
alex@rc.unesp.br

Edson Borin Guido Araujo
UNICAMP – Institute of Computing
{edson,guido}@ic.unicamp.br

Abstract

Although dynamic memory management accounts for a significant part of the execution time on many modern software systems, its impact on the performance of transactional memory systems has been mostly overlooked. In order to shed some light into this subject, this paper reports our first attempt at evaluating the effects of memory allocators on the performance of transactional applications. In general, our results indicate a strong influence of the allocators on the overall performance. In particular, we observed differences ranging from 4% to 171% in the STAMP applications. Our results point to the importance of reporting the allocator utilized in the performance evaluation of transactional memory systems, as the conclusions might change from one allocator to another.

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

General Terms Algorithms, Design, Performance

Keywords Transactional Memory, Dynamic Memory Allocation, Performance Evaluation

1. Introduction

Excessive power dissipation and microarchitectural limitations forced the semiconductor industry to bet its future on multicore processors. The shift from single-core to multicore architectures is causing what has been termed the *concurrency revolution*, in which software plays a critical role [31]. In order to make the most out of current processors, programmers need to explicitly write their code such that it can be executed concurrently on multiple cores. Current abstractions for shared memory parallel programming rely on *locks* and *condition variables* for synchronization, whose drawbacks are well known and include deadlock, priority inversion, and lack of composability [1].

As a result of the many pitfalls raised by lock-based synchronization, researchers have started looking at more abstract alternatives. One promising approach proposes using *transactions* as the unit of concurrency, a strategy more commonly known as Transactional Memory (TM) [16]. A transaction is a sequence of instructions that operates on an *all-or-nothing* fashion: either the entire block of code is executed atomically or none of the instructions appear to take effect. Memory transactions are very much like their database counterparts [15], except that they operate on volatile memory (i.e., durability is not a concern).

The implementation of the transactional mechanism can be done entirely in software (STM), in hardware (HTM), or using a combination of both (HyTM). Despite recent announcements of hardware support for TM in current processors [19, 20, 34], software implementations are still very appealing since they can run on top of the majority of mainstream processors (with no HTM support) and provide an efficient testbed for new ideas. Moreover, existent processors that do provide transactional support implement a best-effort HTM, relying on software to guarantee system progress. As a consequence, software support will play a key role in the future.

Performance has always been the Achilles' heel of software transactional memory. Early reports on STM performance revealed execution time worse than sequential code, deeming STM a *research toy* [3]. Later experiments showed that STM could indeed provide good speedups over sequential execution time by using a more diverse set of benchmarks, a state-of-the-art implementation, and more powerful hardware [8]. Nonetheless, the development of more efficient STM algorithms, implementations and optimizations remains a very active field of research. Recent works have looked into how platform specificities, such as thread mapping strategy and compiler instrumentation, affect the overall runtime performance and scalability of the system [4, 27]. Following on the same direction, we investigate in this paper the impact of dynamic memory management on the performance of blocking STM implementations and applications. Although we only discuss the impact of allocators on STM systems in this paper, we expect that most of the conclusions are valid for HyTMs since they also rely on STMs.

1.1 Motivation

Dynamic memory allocation is among the most expensive and pervasive operations in C/C++ applications. Recent studies conducted with a group of sequential applications have shown that, on average, 30% of the total execution time is spent on dynamic memory management [33]. The advent of multicore processors has intensified the importance of the allocator in deploying high performance systems. As transactional memory becomes mainstream, it should also satisfactorily interact with the memory allocator.

Although the importance of memory management in current software development is clear, surprisingly its impact on the performance of transactional applications has been mostly overlooked. Very few papers have investigated memory management issues in the context of STMs [14, 18]. Unfortunately, most papers do not even mention which allocator is used for performance evaluation.

To illustrate the influence of memory allocators, consider Figure 1 (see Section 4 for details on the experimental setup). The exe-

TRANSACT'14, March 2, 2014, Salt Lake City, Utah, USA.
Copyright © 2014 ACM . . . $15.00.

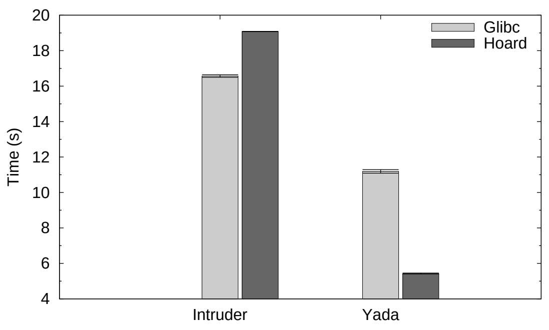

Figure 1. Influence of memory allocators on Intruder (left) and Yada (right) with 8 cores. The best-performing allocator changes from one application to the other.

cution times of two STAMP applications [26] (Intruder and Yada), both using the TinySTM library [10], and two allocators (`Glibc` and `Hoard`), are shown for a configuration with 8 cores. While the `Glibc` allocator performs better for Intruder (left), `Hoard` exhibits a better execution time for Yada (right). The applications binary files have not even been modified: by dynamically changing the allocator at loading time we obtained remarkably different results.

It is important to notice that the performance of memory management in a transactional setting is not only affected by the allocator itself (how objects are layed out in the address space and the allocation algorithms), but it also depends on the specificities of the transactional algorithm: most lock-based implementations require mapping memory addresses to locks for conflict detection. In short, the main research questions we want to address in this paper are: (a) what is the impact of memory allocators in the overall performance of STM systems? (b) should researchers report which memory allocator they have used in their experiments?

1.2 Contributions and organization

To the best of our knowledge this is the first paper to conduct a detailed analysis of the impact of dynamic memory management on the performance of transactional applications. We make the following 2 main contributions:

- A detailed performance analysis of the interaction of different memory allocators and a blocking, word-based STM system, conducted with a synthetic benchmark (Section 5). More specifically, we investigate the influence of allocators on the locking granularity exposed by the STM system, which has a direct impact on the number of false aborts. Although earlier works have explored this issue [7, 10], they have not taken into account the effect of memory allocators. Moreover we show that, at least for some data structures, the best mapping function does depend on the memory allocator.
- A performance characterization of the STAMP applications [26] with respect to a number of different allocators, which reveals that they do interfere in the performance and may lead to wrong conclusions (Section 6). Our results indicate that it is important to report the memory allocator used on the experiments when evaluating STM systems. For instance, we have observed differences ranging from 4% to 171% in the STAMP applications.

The rest of this paper is organized as follows. Section 2 presents the context in which this work is inserted, along with a brief description of related works. The memory allocators considered in the analysis are described in Section 3, followed by the experimental setup in Section 4. An analysis of the allocators' impact on the synthetic and realistic benchmarks is done in Sections 5 and 6, respectively. Finally, our conclusions are stated in Section 7.

2. Background and related work

At a higher level there are two main broad categories in which software transactional memory designs can be organized: blocking and non-blocking. For this work we focus on blocking implementations. In particular, our interest is on time-based STMs that operate on the word granularity and are implemented in languages such as C/C++, in which memory management is explicit. Representative designs of such category are among the fastest known implementations, including TL2 [6], TinySTM [10], and SwissTM [7].

In order to correctly track conflicts, the lock-based STMs employed in this work rely on a big lock table commonly known as *Ownership Record Table*, or ORT [1]. We refer to an entry in this table as a *versioned lock*. Memory accessed by an application is divided into stripes, with each stripe being mapped to a versioned lock by means of a *mapping function*. A *lock bit* included in each versioned lock signals whether some transaction is currently modifying the memory stripe protected by the referred versioned lock. An attempt by some other transaction to modify the same memory region is blocked, leading the transaction to either wait or abort. When the lock bit is unset, the versioned lock maintains a timestamp representing the last time that the corresponding memory region was modified. Notice that it might occur for two distinct memory stripes to be mapped to the same versioned lock, resulting in false aborts. The mapping function can be tuned in order to avoid this behavior. Although larger memory stripes increase the likelihood of false aborts, the validation and locking costs are reduced. On the other hand, a small memory stripe will prevent spurious aborts from happening at the cost of larger read/write sets and higher cache pressure. Despite the fact that investigations about the performance implications of the mapping function have been conducted previously [7, 10, 23, 24, 36], none of these works have considered the impact of memory allocators.

For the class of STMs considered in this work, dynamic memory management is not a part of the core design. Instead, it is built around an external allocator interface that provides at least `malloc` and `free` function calls. An allocator wrapper must annotate all transactional allocations (because they must be undone in case of aborts) and defer deallocations to commit time. Hudson et al. [18] investigated the integration of the memory allocator with the transactional algorithm, but we have not heard any further progress of this approach other than it has resulted in the design of the current `TBBMalloc` allocator [21] (in a not transactional setting). Gottschlich and Connors [14] discuss memory management issues in the context of DracoSTM, observing a 20% performance improvement while using a builtin user-configurable memory manager. The use of hardware transactional memory for simplifying the implementation of common data structures related to dynamic memory management is investigated by Dragojevic et al. [9].

3. Dynamic memory management

Dynamic memory management is a fundamental part of any computer system. The interface between the allocator and the application is usually represented by the function pair `malloc/free`. A call to `malloc` is used to request memory from the system's heap, while a call to `free` makes a previously allocated memory block available again. Dynamic memory allocator design is an old topic, dating back to 1961 [11]. A good allocator must provide at least: (i) fast (de)allocation (low latency), and (ii) efficient use of mem-

[1] The reader should notice that not all lock-based STMs are built around an ORT (e.g., RingSTM [30]).

ory space (low fragmentation). Although a bit outdated, Wilson et al. [35] provide an excellent review of the literature about sequential allocators.

With the introduction of multicore processors, allocators are further required to provide good scalability and avoid cache false sharing, a scenario wherein multiple threads accidentally share the same cache line. It is important to notice that the development of a multithreaded allocator requires a new design. For instance, extending an excellent serial allocator with a single global lock to protect each (de)allocation is certainly not a good choice, since it will inevitably serialize all allocations and badly hurt scalability. New multithreaded allocator designs have been proposed recently aimed at providing good scalability [2, 5, 21, 25, 29]. The performance analysis conducted using these allocators show that the choice of the allocator has a big impact on the overall performance. Even then, performance evaluations of transactional systems have mostly omitted the allocator employed in the experiments.

In the following subsections we describe the basic behavior of the four memory allocators studied in this work: `Glibc` [13], `Hoard` [2], `TBBMalloc` [21], and `TCMalloc` [12]. We believe these allocators cover a wide spectrum of allocation strategies and, in addition, are publicly available.

3.1 Glibc

The GNU C library (`Glibc`) memory allocator uses a modified version of Doug Lea malloc (dlmalloc) [22], adapted to support multicore processors by Wolfram Gloger (ptmalloc3) [13]. It is the default allocator distributed with typical Linux systems.

The allocator keeps memory blocks in bins, grouped by size (a technique referred to as binning). For small blocks (usually 128 bytes or less), the allocator uses a caching mechanism wherein freed memory are stored in a very fast type of bin, implemented as a single linked list with no coalescing. Therefore, requests for small blocks are usually resolved very quickly. For larger requests the system memory mapping facility is used. Each memory block has metadata (commonly known as boundary tags) holding size and status information. The minimum allocated block size is 32 bytes on current 64-bit systems.

The multithreading support added by Wolfram Gloger makes use of per-thread *arenas*. An arena is a contiguous block of memory obtained from the kernel (heap area) and managed by the allocator. When a `malloc` is requested by a thread for the first time, the allocator creates a new arena for that thread. To improve locality, subsequent requests by the same thread attempt to use the same arena, but it might not be available, as the allocator does not use private arenas. Therefore, locks are used to avoid having two threads accessing the same arena at the same time.

In order to reduce lock contention, the allocator does not block if a lock is already taken. Instead, a `mutex_try_lock` primitive is firstly used in the hope of acquiring the lock. If it fails, the allocator repeats the procedure for the next arena (they are kept in a circular list). If none of the arenas can be used, a brand new one is created to fulfill the thread request. When a thread frees a block, the allocator returns the block to the arena from which it was originally allocated. Notice that the allocator always requires at least one atomic read-modify-write instruction per (de)allocation, even when there is only one thread executing.

3.2 Hoard

The `Hoard` allocator was proposed by Berger et al. in 2000 [2] and it is still being developed. The allocator is designed to be scalable, avoid false sharing, and exhibit bounded fragmentation.

`Hoard` maintains per-thread heaps along with a global heap. Each heap is assigned to a thread by means of a hash function that maps the thread ID to its heap. When a block of memory is requested, `Hoard` first checks the corresponding thread heap for available memory. If no blocks of the desired size class are found, the allocator retrieves a big chunk of memory from the global heap, called a *superblock* in `Hoard` terminology. A superblock keeps a free list of available blocks, all of the same size class. Size classes are apart from each other by a power of b, bounding internal fragmentation to a factor of b. External fragmentation is reduced by returning superblocks below a given emptiness threshold to the global heap. When a `free` operation is invoked, the block is returned to the superblock from which it was allocated in order to reduce false sharing.

Regarding synchronization, `Hoard` original algorithm required a lock per heap and per superblock. A heap is locked during allocation and deallocation. The deallocation procedure further needs to acquire the lock for the specific superblock. The authors argue that `Hoard` incurs very low contention costs for memory operations in the common case, claiming that the contention for the per-thread heap locks is not a scalability concern and contention for the global heap lock is rare. Recent versions of `Hoard` make use of thread-private local heaps for small blocks (usually 256 bytes or less). These local heaps substantially improve performance since they avoid most of the atomic operations required for locking the per-thread heaps in the original algorithm. Small chunks are also freed locally.

3.3 TBBMalloc

The Intel `TBBMalloc` allocator is part of Intel Threading Building Blocks, a software library designed to take advantage of multicore processors [21]. The basic design of the allocator was carried out during the McRT research program at Intel [28] and is based on a non-blocking memory management algorithm, which was also integrated with a software transactional memory system [18].

The `TBBMalloc` allocator uses thread-private heaps, eliminating the need for costly synchronization if allocation requests can be serviced by the local heap. Like `Hoard`, each heap maintains different superblocks for different size classes. If the allocator cannot find any available block in the local superblock for a given size class, a global heap is accessed and a superblock is transferred to the local heap. If there is no available memory in the global heap, a block of 1MB is allocated using the operating system memory support. This big block is further split into superblocks of 16KB each. To avoid a large memory footprint, empty superblocks are returned back to the global heap.

Freed blocks are returned to the superblocks they were allocated from. This has the advantage of reducing false sharing and may also increase cache locality but, on the other hand, requires some form of synchronization, as a superblock in another thread heap must be accessed. In order to reduce the synchronization cost `TBBMalloc` maintains two separate free lists for each superblock: a public and a private. A `malloc` operation first attempts to grab a block from the private list, which does not required any synchronization. The public list is only inspected when no available blocks are found in the private list. Contrary to the version of the allocator developed by the McRT team, `TBBMalloc` does not employ non-blocking synchronization, using fine-grained locks instead.

3.4 TCMalloc

The Thread-Caching Malloc (`TCMalloc`) [12] is a high-performance multithreaded memory allocator distributed with Google Performance Tools (`gperftools`), along with other tools for heap and CPU profiling. `TCMalloc` is also used by Google Chrome web browser.

The design of the allocator is very close to `TBBMalloc`. Each thread is assigned a local heap (a *thread cache* in `TCMalloc` nomenclature) from which small blocks (usually 256KB or less) are

Allocator	Version	Metadata (tag)	Min Size	Fast Path	Granularity	Synchronization
`Glibc`	2.11.1	Per block	32 bytes	<= 128 bytes	132KB-64MB per arena	A lock per arena. If a thread fails to grab the lock for any of the active arenas, a new one is created.
`Hoard`	3.10	Per superblock	16 bytes	<= 256 bytes	64KB per superblock	Each heap is protected by a lock as is the global heap. A cache is maintained for small block sizes and is accessed without synchronization.
`TBBMalloc`	4.1	Per size class	8 bytes	< 8KB	16KB per size class	The public free lists of a private heap are each protected by a distinct spinlock. Each free list in the global heap is also protected by a separate spinlock. Accessing the private free lists is synchronization-free.
`TCMalloc`	2.1	Per size class	8 bytes	<= 256KB	incremental	Each free list in the central cache is protected by a spinlock. A spinlock is also used to protect the central page heap.

Table 1. Summary of the main attributes of the studied allocators.

allocated without any synchronization overhead. Available blocks are internally stored in separate free lists according to their size. When a `malloc` is requested, the allocator first locates the free list that matches the required allocation size in the thread cache and returns an available block if the list is not empty. Otherwise, the allocator consults a central heap (also called *central cache*) that works as a back store. Inside the central heap, blocks are also segregated by their size and kept in different free lists. Since all threads share the central heap, spinlocks are used to provide consistent access.

In case the requested block is still not found in the central heap, `TCMalloc` uses another sort of allocator called the *central page heap*. This component allocates pages directly from the operating system and serves two main functions: (1) as a back store for the central heap; (2) as an allocator for large chunks. When small blocks are deallocated, `TCMalloc` insert them into the appropriate free list in the current thread's thread cache. Recall that this behavior differs from both `Hoard` and `TBBMalloc`, since they would return the block to the thread cache it was originally allocated from. To avoid external fragmentation, `TCMalloc` runs a garbage collector when a thread cache size exceeds a given threshold, moving unused blocks back to the central heap.

3.5 Discussion

In the description of the allocators presented earlier we focused on how memory (de)allocation is performed, the data layout, and the synchronization aspects. Other important features such as heap corruption protection are out of the scope of this paper.

Table 1 highlights the main attributes of the allocators employed in this work. As can be seen, only `Glibc` maintains metadata information on a per block basis. This choice considerably increases the minimum allocated size (`Min Size` column) on 64-bit machines. Even a `malloc(0)` would cause the `Glibc` allocator to return a pointer to a 32-byte block. Apart from the memory utilization overhead, this choice reduces cache locality and may have a direct impact on the performance of the system. Notice that only 2 memory blocks would fit a cache line on typical L1 caches (considering a standard 64-byte line size), whereas for `TBBMalloc` and `TCMalloc` 8 memory blocks could potentially be placed on the same line.

The `Fast Path` column indicates the block sizes for which the respective allocator provides a fast path (de)allocation. With the exception of `Glibc`, which still requires locking an arena, all remaining allocators implement some sort of local cache that effectively provides synchronization-free (de)allocations. In the fast path, synchronization is required only when a block is not found in the local cache and the allocator must access a global back store. Even then the blocks are segregated by size, allowing the use of fine-grained locks. The `Synchronization` column describes the basic synchronization strategies for each allocator.

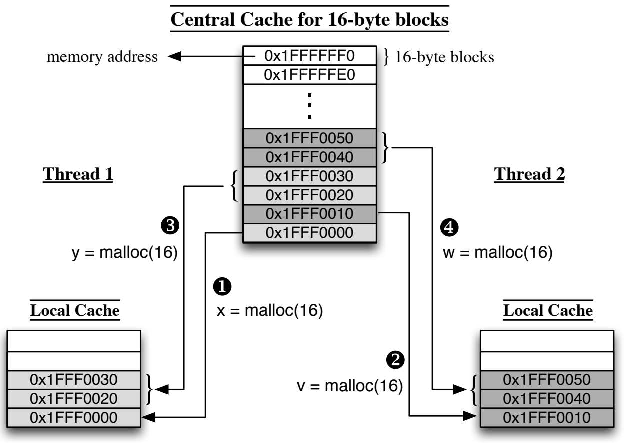

Figure 2. An illustration of false sharing induced by `TCMalloc`.

Finally, the `Granularity` column shows the size of the memory block made available to a thread when the first allocation is requested. For `Glibc`, the minimum arena size is 132KB and is internally split into bins for different size classes. All threads can share the same arena as long as there is no contention. An arena can be further enlarged to a maximum of 64MB on demand. `Hoard` assigns a superblock of 64KB for each requested size class, whereas 16KB blocks are used by `TBBMalloc`. The larger the size the less frequently the allocator has to access the global heap and, consequently, the less the synchronization overhead. On the other hand, larger sizes increase fragmentation. `TCMalloc` employs an incremental approach, wherein all free lists are initially empty and their sizes are incrementally increased on each successive allocation requiring access to the central cache. This behavior can possibly give rise to false sharing scenarios as discussed next.

In Figure 2 we picture a central cache with a free list of 16-byte blocks, showing the respective available addresses inside each block. Notice that the blocks represent consecutive addresses, since a big chunk has been previously allocated from the operating system. We consider only 2 threads for simplicity, also assuming that each local cache is initially empty. When thread 1 requests a memory block, it will not find it in the local cache and therefore the allocator transfers one block from the central cache ❶. After that, thread 2 also requests a memory block and, since its local cache is also empty, the next available block from the central cache is fetched ❷. At this point the two threads will have the variables x and v pointing to consecutive memory locations, 16 bytes apart from each other, possibly causing false sharing. When another block is requested by thread 1, the allocator needs to go to the cen-

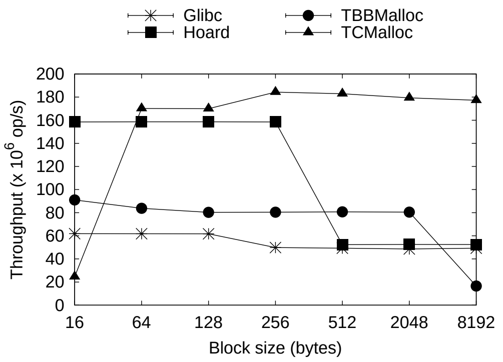

Figure 3. Throughput of the studied allocators for different block sizes (8 threads).

tral cache again but now it transfers 2 blocks ❸ (the third time it will transfer 3 blocks, the fourth time 4 blocks, and so on). Thread 2 also performs another allocation, forcing the allocator to bring the next two memory blocks from the central cache ❹.

To conclude this section we conduct a performance analysis of the allocators described previously. It is difficult to measure the overall performance of a memory allocator since it depends on a host of factors. In this particular analysis we are interested on how fast a pair of `malloc`/`free` operations are processed. Our microbenchmark is the same used by `Hoard` [2], called *threadtest*. In this experiment 8 threads repeatedly do nothing but allocations and deallocations. A memory block is deallocated right after allocation by the same thread. Figure 3 shows the throughput of each allocator for several block sizes.

It is interesting to notice that `TCMalloc` does not perform well for 16-byte blocks due to cache false sharing, as discussed earlier and illustrated by Figure 2. Apart from this behavior, it presented the best overall throughput for this particular experiment. `Hoard` also performs quite well for blocks less than or equal to 256 bytes, as predicted by Table 1 (`Fast Path` column). Afterwards, the local cache is not used anymore and a thread must access its lock-protected heap, decreasing its throughput to a level very close to `Glibc`. Recall that in `Glibc` a thread always needs to go through a per-arena lock in every allocation and deallocation. `TBBMalloc` throughput is kept roughly constant until blocks slightly less than 8KB are requested, in which case the allocator invokes the operating system memory management directly.

4. Experimental setup

In this section we describe our evaluation methodology, tools configuration, and the benchmarks analyzed in this paper. The goal of our analysis is to assess the impact of the different memory allocators on the performance of the STM library and transactional applications.

Our experiments have been carried out on an Intel Xeon machine, whose detailed configuration is given in Table 2. The operating system is a typical 64-bit Linux distribution (Ubuntu server 10.04.3 LTS), with kernel 2.6.32, and Glibc version 2.11.1. The STM library and the applications employed in the analysis were compiled for 64-bit mode using GCC version 4.4.3 with optimization flag `-O3`. For fairness we also avoided GCC specific optimizations targeted at the `Glibc` allocator by using the `-fno-builtin` flags for `malloc`, `calloc`, `realloc`, and `free`. Since the allocators are provided as dynamic libraries, we did not produce different application binaries for each of them. Instead, each allocator is dynamically loaded by setting the LD_PRELOAD environment variable accordingly at runtime. The versions of the allocators are given in the second column of Table 1.

Processor Model	Intel(R) Xeon(R) E5405 @ 2.00GHz
Total cores	8 (2 sockets, 4 per socket)
L1 data cache	32KB, 8-way set associative, 64-byte lines
L2 cache	2x6MB, unified, 24-way set associative
Main memory	4GB

Table 2. Machine configuration used in the experiments.

The main STM library chosen for this work is TinySTM [10] version 1.0.4, released on January 29, 2013. For most of the experiments we did not change any library configuration, using the default ETL (Encounter-Time Locking) design and the `SUICIDE` contention management strategy (the transaction that causes the conflict is aborted and immediately restarted). We did turn on the flag for statistics (number of commits and aborts, mostly) but did not notice any significant overhead. The default size for the ownership record table (ORT) is 2^{20} elements. A given address is mapped to an entry in this table by shifting its 5 less significant bits to the right and taking the rest modulo the size of the ORT. This configuration forces 32 consecutive bytes to be mapped to the same versioned lock in the ORT.

We use both synthetic and realistic benchmarks in our experiments. The synthetic one has a configurable number of threads updating (inserting or deleting) or searching for elements in a given data structure. Three different data structures were used: a sorted linked-list, a hashset, and a red-black tree. The number of elements is kept nearly constant by forcing insertions and deletions to take turns: the next element to be removed is the last one inserted. This sort of benchmark has been extensively used in previous works (see for instance [17] and [10]) and provided us an excellent and simplified workload to understand the behavior of the allocators. The configuration exploited in the experiments uses a set with 4096 elements, random numbers in the range of $[0, 8192)$, and three different update rates: read-only, read-dominated (20% of updates), and write-dominated (60% of updates). Due to space constraints, we only discuss the write-dominated configuration because its performance is more sensitive to the allocators.

We also use the STAMP benchmark suite [26] in our evaluation[2]. STAMP is comprised of 8 different applications, each with different behavior concerning time in transaction, level of contention, size of read/write sets, and transaction length. The configurations used in the experiments are the ones suggested in the original paper with large data sets. For two of the applications, `Kmeans` and `Vacation`, there are two recommended configurations, but we only use one of them in this paper (the one with the highest contention level and working set size) due to space constraints.

Finally, our results are presented as a mean of 50 (synthetic benchmark) or 30 executions (STAMP). In order to provide statistically significant values, our figures also show error bars representing a 95% confidence interval. Besides throughput and execution time, some of the experiments required more detailed information about cache events (such as number of accesses and miss ratio). In order to collect those events we utilized the PAPI interface [32] version 5.2, an abstraction layer for accessing hardware performance counters.

[2] Since STAMP is not maintained by the original research group anymore, we used the one supported by the community and available at `http://code.google.com/p/tm-benchmarks/`. This version has several bug fixes.

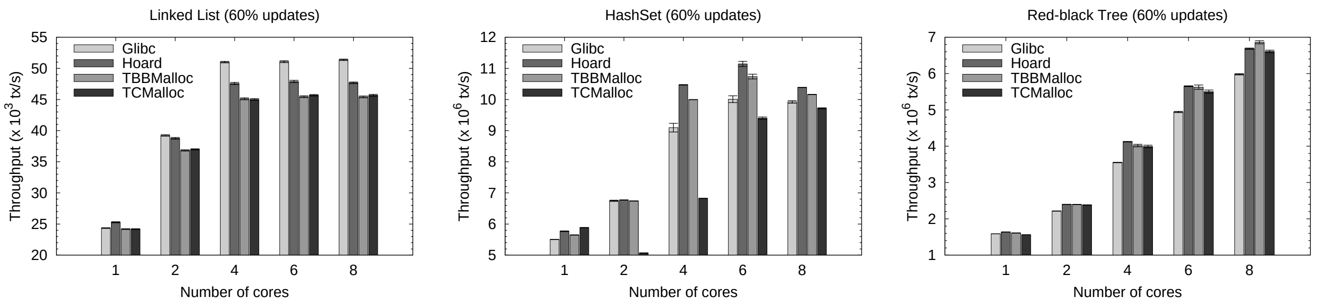

Figure 4. Throughput of the different data structures: Sorted LinkedList (left), HashSet (middle), and Red-black Tree (right). The results are for the write-dominated workload (60% updates).

Application	Best	Worst	Speedup	Threads
Linked-list	`Glibc`	`TBBMalloc`	13.12%	8
HashSet	`Hoard`	`TCMalloc`	18.52%	6
RBTree	`TBBMalloc`	`Glibc`	14.76%	8

Table 3. Best and worst allocators for each data structure, relative speedup and respective thread number (write-dominated configuration).

#P	`Glibc`		`Hoard`		`TBBMalloc`		`TCMalloc`	
	aborts	L1miss	aborts	L1miss	aborts	L1miss	aborts	L1miss
1	00.0%	4.6%	00.0%	3.2%	00.0%	3.2%	00.0%	3.2%
2	10.4%	5.0%	17.4%	3.4%	17.3%	3.3%	17.4%	3.3%
4	30.9%	5.2%	45.4%	3.6%	45.0%	3.5%	45.1%	3.5%
6	44.1%	5.2%	61.2%	3.6%	60.8%	3.5%	61.0%	3.5%
8	**55.7%**	5.3%	**70.6%**	3.6%	**70.2%**	3.5%	**69.7%**	3.5%

Table 4. Percentage of aborted transactions and L1 data cache misses for the write-dominated configuration (sorted linked list).

5. Synthetic benchmark analysis

We start off our performance analysis considering the synthetic benchmark described previously. Figure 4 presents throughput results for the different data structures and the write-dominated workload. It is surprising to notice that the performance of the different allocators vary considerably among the three data structures, as revealed by Table 3. This table also shows the relative speedup between the best and worst allocators, as well as the thread configuration that produced the maximum throughput. In the following subsections we investigate what is behind this behavior and how the allocators and the STM library affect the overall performance.

5.1 Sorted linked list

In this microbenchmark each node of the list is composed of a 64-bit value field and a pointer to the next node, amounting to 16 bytes. A transaction allocates a node through `malloc` when inserting a new element, and deallocates the node via `free` when deleting it. Finding an element in the linked list may require scanning a lot of nodes. Since insertions and deletions in a sorted list first need to locate the previous element, many memory locations are touched during the traversal, resulting in large transactional read sets.

One important difference among the allocators is that `Glibc` will allocate 32 bytes for each node (minimum block size), whereas all other allocators will reserve the exact amount (16 bytes). Despite the worse cache locality introduced by `Glibc`, it is intriguing to observe that it displayed the best overall results. To further investigate this issue we measured the fraction of the total transactions that have been aborted and also the L1 data miss rate. The values are displayed in Table 4. As expected, the cache locality is worse for `Glibc` but, on the other hand, many more transactions are being aborted with the other allocators. Our findings reveal that the good results achieved by `Glibc` are due to the 32-byte aligned addresses and the way the STM library maps addresses to versioned locks in the ORT. We use Figure 5 to illustrate the problem in detail.

Before spawning the threads to perform the operations on the linked list, the main thread allocates all the nodes and inserts them in the list. Pick two nodes allocated in sequence, say x and y. When the `Glibc` allocator is used, these nodes are 32 bytes apart (minimum allocation size). Assume these addresses are `0x18000020` and `0x18000040`, respectively. Now assume transaction 1 is performing a write operation on node x (e.g., changing its next pointer to insert a new node) and transaction 2 is traversing the list and reads node y (Figure 5a). Transaction 1 sets the lock bit for x in the ORT and, since the address of y is mapped to a different entry in this table, there is no conflict and both transactions proceed (recall that the mapping function simply right-shifts the address by 5 and takes the rest modulo the ORT size). Now consider the same scenario for the other allocators (Figure 5b). Since in this case addresses are 16 bytes apart (`0x18000020` and `0x18000030` in the example), transaction 2 will mistakenly be aborted.

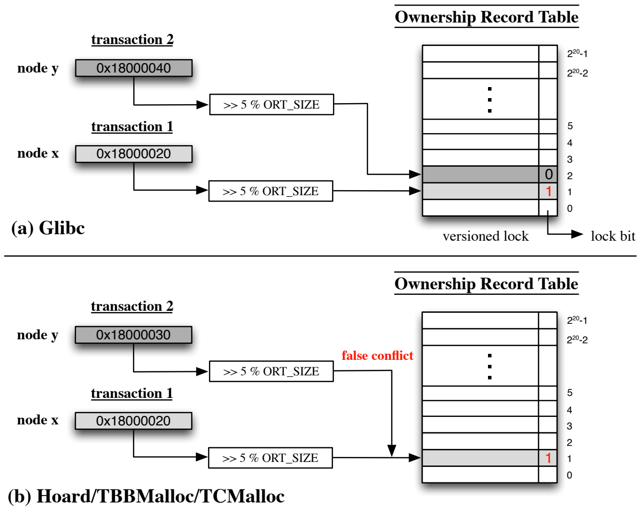

Figure 5. The interaction between the allocator and the STM library may cause false aborts. While nodes are 32-byte apart when `Glibc` is used and the system advances naturally (a), the combined effect of 16-byte blocks and the STM mapping function causes false aborts for `Hoard`, `TBBMalloc`, and `TCMalloc` (b).

5.2 HashSet

Differently from the linked list microbenchmark, operations on the HashSet are very fast as a hash function is used to directly calculate the target addresses. Therefore, transactions are short and have relatively small read/write sets. When there is a collision in the hashtable the nodes are linked linearly. However, the likelihood of a collision is very low since the hash table has 128K entries and the set is 4K long. The size of a node is also 16 bytes, but since the transactions do not have to traverse them in a linear fashion, the issue depicted previously for the linked list is not a concern here. Nevertheless, one can see that some allocators did not perform very well, in particular `TCMalloc`.

Looking at the fraction of aborted transactions we noticed that both `Glibc` and `TCMalloc` exhibited much larger numbers when compared to `Hoard` and `TBBMalloc`. On closer inspection we realized that when transactions were about to perform an insertion and requested memory for a node, `TCMalloc` was returning adjacent memory addresses (16-byte apart) due to the behavior described previously (see Figure 2). The impact on throughput is twofold: first, there is false sharing due to distinct nodes in different threads sharing the same cache line; second, the STM library will map two contiguous nodes to the same versioned lock in the ORT. The combined effect is the increased number of aborted transactions and consequent reduced throughput as observed in Figure 4. The worst case happens when only two threads are active. When one of them locks the ORT to insert/delete a node in the hashset it might have to wait for the invalid cache line to become available[3]. Meanwhile, the other transaction will continuously abort. Indeed, looking at the execution log we noticed a huge number of repeated aborts.

The problem that caused a subpar performance with the `Glibc` allocator is a bit different but also stems from allocator-induced false aborts. Each transaction will most likely allocate memory from a different arena and, since arenas have a 64MB alignment, the mapping function will discard the higher bits of the addresses and map the result to the same versioned locked. As an example, consider that the nodes allocated by any two transactions are at locations `0x18000000` and `0x1c000000`. There is no false sharing at the cache level here, but both addresses will be mapped to entry 0 in the ORT. This problem does not occur with `Hoard` and `TBBMalloc` because their superblocks are aligned at 64KB and 16KB boundaries, respectively.

5.3 Red-black tree

The red-black tree benchmark has some key differences compared to the previous two. First, each node of the tree is 48 bytes long. Both `Glibc` and `Hoard` do not provide a class with the exact same size, thus using the 64-byte class. It is curious to notice that a 48-byte block might cause false conflicts with the default shift value (5), as its last 16 bytes will be mapped to the same versioned lock of the first 16 bytes of the next contiguous node. Because a 64-byte block is used in the case of `Glibc` and `Hoard`, this would not be possible. We did not observe this trend directly in the results, although `Hoard` did exhibit the less percentage of aborted transactions among all allocators. For `Glibc` we actually found a relatively larger fraction of aborted transactions, probably due to arena-induced conflicts. Also, recall that the access to the arenas requires an atomic instruction for grabbing its lock, which also contributes to the inferior overall throughput of this allocator.

Another important difference is that in this benchmark a transaction can deallocate a block of memory allocated by other transactions. When a transaction tries to delete a node that it last allocated, the node actually deleted might have a different address (this is due to the nature of tree deletions that might rearrange the disposition of the nodes and copy the values around). Also, the size of the write set tends to be larger for insertions and deletions (the other microbenchmarks only perform a single write).

[3] Having a microarchitectural store buffer will not help here because the next step of the commit protocol requires a memory barrier before the phase responsible for unlocking the write set starts.

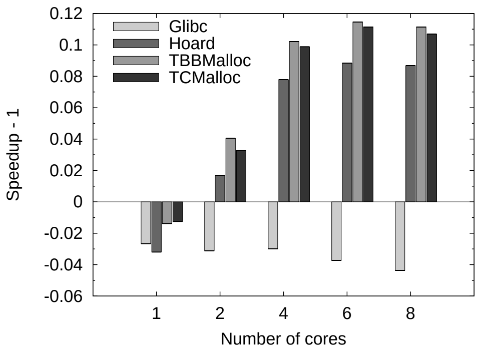

Figure 6. Relative speedup (-1) of the sorted linked list with a shift amount of 4 bits for the write-dominated workload (with regard to a shift amount of 5).

5.4 Discussion

The experiments with the synthetic microbenchmarks revealed that allocators may interfere with the way that word-based STM libraries map addresses to versioned locks in the ORT, changing the likelihood of false aborts. In the experiments performed previously we use a shift amount of 5 bits, forcing a region of 32 bytes to be mapped to the same versioned lock. We repeated the experiment for the linked list with a shift amount of 4 in order to examine the behavior of the allocators. The relative speedup is showed in Figure 6 for the write-dominated workload. In general, we observed that reducing the shift amount increased the L1 data cache misses (more entries in the ORT need to be accessed) and, as a result, all allocators showed a performance loss with only 1 core (there are no conflicts in this case). As more cores are added, the question is whether the gain obtained with the reduced number of false aborts will overcome the extra overhead. Notice that for `Glibc` there is no conflict to be avoided (the allocator returns 32-byte blocks) and hence its performance falls off. On the other hand, the remaining allocators display some improvements since the issue illustrated in Figure 5 does not happen anymore.

The tuning of the shift amount parameter has been studied in earlier works [7, 10, 23, 24, 36]. Although the optimal value for this parameter is application specific, a value of 5 is usually accepted (4 bit on 32-bit machines) as this configuration provides the best overall results. However, these earlier works did not consider allocator-specific interferences. As our results have showed, for the linked list microbenchmark a value of 5 is optimal for `Glibc`, but 4 presents the best results for `Hoard`, `TBBMalloc`, and `TCMalloc`.

6. STAMP analysis

Instead of analyzing the details of the interaction between each allocator and the STM system as we did with the microbenchmarks, we take a holistic approach with the STAMP applications due to their complexity. We are particularly interested in contrasting the speedup achieved by each allocator. In the introduction (Figure 1) we showed an example where the execution time varied according to the allocator used. Here we extend this analysis to other STAMP applications and allocators.

Figure 7 shows the speedup of the STAMP applications for each of the allocators studied. Each speedup number represents the result

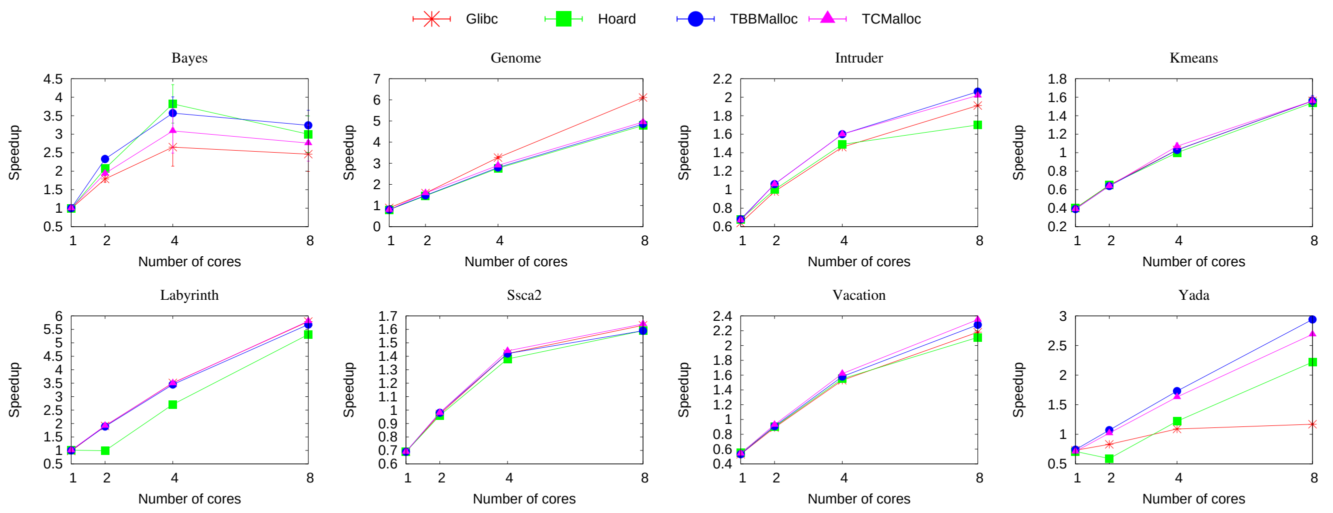

Figure 7. Speedup with different allocators for the STAMP applications.

a research group would achieve if the corresponding allocator was used. In other words, the normalization is applied independently for each allocator using the respective sequential execution time. This allows us to see the discrepancies in scalability among the allocators. Important disparities can be noticed for Bayes [4], Genome, Intruder, Vacation, and Yada. The remaining applications (Kmeans, Labyrinth, and SSCA2) are not heavily affected by the allocators, although we noticed that `Hoard` did not perform as well as the others with Labyrinth.

An interesting behavior is displayed by Genome. At first sight it seems that `Glibc` is more scalable, but a closer look reveals a completely different story. The data layout pattern produced by `Glibc` causes many cache misses and the sequential execution time (from which the speedups are calculated) is about 30% worse than the other allocators. This loss is attenuated when multiple cores are used as the working set can fit nicely in each available cache. Therefore, the 6x speedup attained with `Glibc` is misleading since it was an artifact of the allocator rather than the result of the STM library design.

Another noteworthy behavior is presented by Yada. Notice that its speedup curve reveals very different trends. While it practically does not scale with the `Glibc` allocator, it performs reasonably well with the other 3 allocators. Yada displays a high abort rate and, moreover, it is highly dependent on the memory allocator. Although we have not further investigated the sources of the `Glibc` inefficiency at this point, we suspect it is related to its high synchronization overhead (per-arena locks).

The speedup numbers show the scalability attained with each allocator but they do not allow us to compare the allocators' relative performance (the speedup curve is highly dependent on the sequential execution time of each allocator). Similarly to what was done with the synthetic benchmark, Table 5 displays the best and worst allocator for each application along with the relative performance gain and thread configuration that exhibited the best results (we omitted Kmeans because no important difference among the allocators was noticed). Observe the impressive performance gain achieved with `TCMalloc` when compared to `Glibc` for the Yada application. It is interesting to notice that `Glibc` did not present the best performance for any of the STAMP applications. If we assume that the majority of the performance evaluations of software transactional memory systems has been carried out using `Glibc` (the default Linux allocator), we can observe that some of the results might not reflect the real performance of STM systems, as shown by the behavior of Genome and Yada.

[4] As observed by other groups, this application presents high variability. We decided to include it for completeness.

Application	Best	Worst	Speedup	Threads
Bayes	`Hoard`	`Glibc`	47.6%	4
Genome	`TBBMalloc`	`Glibc`	14.4%	8
Intruder	`TBBMalloc`	`Hoard`	24.2%	8
Labyrinth	`TCMalloc`	`Hoard`	9.6%	8
SSCA2	`TCMalloc`	`TBBMalloc`	4.1%	8
Vacation	`TCMalloc`	`Hoard`	24.1%	8
Yada	`TCMalloc`	`Glibc`	170.9%	8

Table 5. Best and worst allocators for each STAMP application, relative speedup and respective thread number.

As revealed by our experiments, the allocator has a direct impact on the performance and should be mentioned along with the evaluation of STM systems. Finally, our results with the STAMP benchmark indicate that either `TBBMalloc` or `TCMalloc` displayed the best overall results and should be preferred for evaluating STM systems.

7. Conclusions

This paper reported our first attempt at measuring the impact of dynamic memory allocators on the performance of software transactional memory applications. By using a synthetic microbenchmark we conducted a detailed analysis of the interference of allocators with the mapping function of blocking software transactional memory systems, showing that allocators do not agree on a single function. We further showed that the scalability of the transactional applications included in the STAMP benchmark is also affected by the chosen memory allocator. Our results highlight the importance of reporting the allocator employed in the evaluation of transactional systems.

Acknowledgments

We would like to thank Tim Harris for earlier discussions and the anonymous reviewers for their valuable feedback and suggestions. We did our best to incorporate as many of them as possible in this final version. This work is supported by FAPESP under grant number 2011/19373-6.

References

[1] A.-R. Adl-Tabatabai, C. Kozyrakis, and B. Saha. Unlocking concurrency. *Queue*, 4(10):24–33, Dec./Jan. 2006-2007.

[2] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM SIGPLAN Notices*, 35(11):117–128, Nov. 2000.

[3] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Communications of the ACM*, 51(11):40–46, Nov. 2008.

[4] M. Castro, L. F. W. Goes, C. P. Ribeiro, M. Cole, M. Cintra, and J.-F. Mehaut. A machine learning-based approach for thread mapping on transactional memory applications. In *Proceedings of the 2011 18th International Conference on High Performance Computing*, pages 1–10, Dec. 2011.

[5] D. Dice and A. Garthwaite. Mostly lock-free malloc. In *Proceedings of the 3rd International Symposium on Memory Management*, pages 163–174, June 2002.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *20th International Symposium on Distributed Computing*, pages 194–208, Sept. 2006.

[7] A. Dragojevic, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 155–165, June 2009.

[8] A. Dragojevic, P. Felber, V. Gramoli, and R. Guerraoui. Why STM can be more than a research toy. *Communications of the ACM*, 54(4): 70–77, Apr. 2011.

[9] A. Dragojevic, M. Herlihy, Y. Lev, and M. Moir. On the power of hardware transactional memory to simplify memory management. In *Proceedings of the 30th Annual Symposium on Principles of Distributed Computing*, pages 99–108, June 2011.

[10] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*, pages 237–246, Feb. 2008.

[11] J. George O. Collins. Experience in automatic storage allocation. *Communications of the ACM*, 4(10):436–440, Oct. 1961.

[12] S. Ghemawat and P. Menage. TCMalloc : Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html. [Last accessed November, 2013].

[13] W. Gloger. Dynamic memory allocator implementations in Linux system libraries. In *Internationaler Linux Kongre' in Wrzburg*, May 1997.

[14] J. E. Gottschlich and D. A. Connors. DracoSTM: A practical C++ approach to software transactional memory. In *Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 52–66, Oct. 2007.

[15] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

[16] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2 edition, June 2010.

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

[18] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg. McRT-malloc: A scalable transactional memory allocator. In *Proceedings of the 2006 International Symposium on Memory Management*, pages 74–83, June 2006.

[19] *Intel Architecture Instruction Set Extensions Programming Reference*. Intel Corporation, Feb. 2012.

[20] C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for IBM system z. In *Proceedings of the 45th ACM/IEEE International Symposium on Microarchitecture*, pages 25–36, Dec. 2012.

[21] A. Kukanov and M. J. Voss. The foundations for scalable multi-core software in intel threading building blocks. *Intel Tecnology Journal*, 11(4):309–322, Nov. 2007.

[22] D. Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

[23] S. Mannarswamy and R. Govindarajan. Making STMs cache friendly with compiler transformations. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, pages 232–242, Oct. 2011.

[24] S. S. Mannarswamy and R. Govindarajan. Variable granularity access tracking scheme for improving the performance of software transactional memory. In *Proceedings of the International Symposium on Parallel and Distributed Processing*, pages 455–466, May 2011.

[25] M. M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, June 2004.

[26] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 35–46, Sept. 2008.

[27] W. Ruan, Y. Liu, C. Wang, and M. Spear. On the platform specificity of STM instrumentation mechanisms. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 1–10, Feb. 2013.

[28] B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. L. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang. Enabling scalability and performance in a large scale CMP environment. In *Proceedings of the 2nd European Conference on Computer Systems*, pages 73–86, Mar. 2007.

[29] S. Seo, J. Kim, and J. Lee. SFMalloc: A lock-free and mostly synchronization-free dynamic memory allocator for manycores. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, pages 253–263, Oct. 2011.

[30] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: Scalable transactions with a single atomic instruction. In *Proceedings of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 275–284, June 2008.

[31] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, Sept. 2005.

[32] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting performance data with PAPI-C. In M. S. Mller, M. M. Resch, A. Schulz, and W. E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 157–173. Springer Berlin Heidelberg, 2010.

[33] D. Tiwari, S. Lee, J. Tuck, and D. Solihin. MMT:exploiting fine-grained parallelism in dynamic memory management. In *Proceedings of the International Symposium on Parallel and Distributed Processing*, pages 1–12, Apr. 2010.

[34] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 127–136, Sept. 2012.

[35] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, pages 1–116, 1995.

[36] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee. Kicking the tires of software transactional memory: Why the going gets tough. In *Proceedings of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 265–274, June 2008.