MMT: Exploiting Fine-Grained Parallelism in Dynamic Memory Management

Devesh Tiwari, Sanghoon Lee, James Tuck, Yan Solihin
Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, USA
{devesh.dtiwari,shlee5,jtuck,solihin}@ncsu.edu

***Abstract*—*Dynamic memory management is one of the most expensive but ubiquitous operations in many C/C++ applications. Additional features such as security checks, while desirable, further worsen memory management overheads. With advent of multicore architecture, it is important to investigate how dynamic memory management overheads for sequential applications can be reduced.*

In this paper, we propose a new approach for accelerating dynamic memory management on multicore architecture, by offloading dynamic management functions to a separate thread that we refer to as memory management thread *(MMT). We show that an efficient MMT design can give significant performance improvement by extracting parallelism while being agnostic to the underlying memory management library algorithms and data structures. We also show how parallelism provided by MMT can be beneficial for high overhead memory management tasks, for example, security checks related to memory management.*

We evaluate MMT on heap allocation-intensive benchmarks running on an Intel core 2 quad platform for two widely-used memory allocators: Doug Lea's and PHKmalloc allocators. On average, MMT achieves a speedup ratio of $1.19\times$ *for both allocators, while both the application and memory management libraries are unmodified and are oblivious to the parallelization scheme. For PHKmalloc with security checks turned on, MMT reduces the security check overheads from 21% to only 1% on average.*

I. Introduction

Dynamic memory management is one of the most expensive but ubiquitous operations in many C/C++ applications. Many applications, such as factorization algorithms, language processing and translation, object-oriented databases, object-oriented robotics, dataflow constraint solvers, and minimum spanning tree, are highly heap allocation intensive. Previous studies show that some C programs spend up to one third of their execution time in dynamic memory management routines such as `malloc` and `free` [3], [22], [32], [34]. While object oriented programming style improves software reusability and extensibility, it has also made applications more heap allocation intensive. For example, some studies reported that C++ programs may use significantly more dynamic memory allocations compared to C programs [7], [11]. Figure 1 shows that for heap intensive C/C++ benchmarks that we study, on average 30% of their execution time is spent in dynamic memory management when using GNU C library.

This work was supported in part by NSF Award CNS-0834664 and by gifts from Intel. The authors would like to thank anonymous reviewers for their feedback.

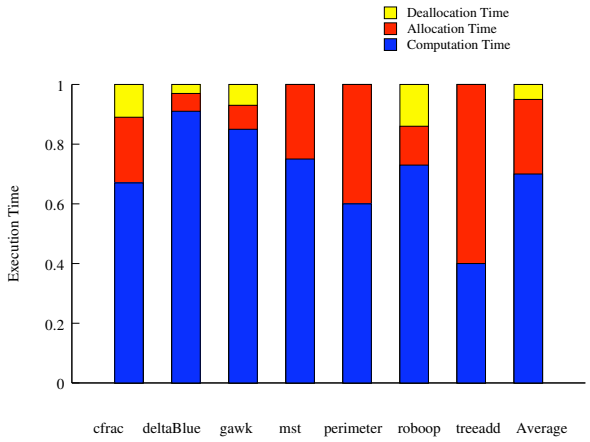

Fig. 1. Fraction of execution time spent in dynamic memory management routines, on an Intel Core2 Quad-Core platform.

Moreover, the time spent in dynamic memory management routines will be even higher if we want to perform safe dynamic memory allocations and deallocations. Adding sanity checks to dynamic memory management routines, such as detecting a deallocation of an invalid pointer and double deallocations, can uncover many dynamic memory management related errors and detect security vulnerabilities [1], [2], [6]. However, such checks incur a high runtime overhead. For example, using the widely used PHKmalloc allocator [18] with extra sanity checks, the benchmarks we tested suffer from an average of 21% execution time overhead.

At the same time, continuing progress in process technology enables the integration of multiple processor cores on a single chip, creating a platform that allows threads of an application to run in parallel on different cores. Considering the high overheads of dynamic memory management, it is important to explore the possibility of reducing dynamic memory management cost on a multicore architecture, especially for *sequential applications* which cannot easily benefit from the multicore architecture otherwise. Therefore, we would like to explore

a new approach to hide the high cost of dynamic memory management by offloading all dynamic memory allocation and deallocation requests to a dedicated *memory management thread* (MMT) thread.

However, translating the potential parallelism of the MMT approach into performance improvement is challenging. First, memory allocation and deallocation are very fine-grained tasks, compared to thread synchronization primitives available in current systems. Figure 2 shows the latency of a pair of Posix thread lock acquisition and release, and a pair of semaphore signal and wait, compared to the average latency for memory allocation, and average time between two consecutive allocation/deallocation requests, for various benchmarks running on an Intel Core 2 Quad system. The figure shows that the latency of lock (uncontended) is roughly the same as the time to perform memory allocation, while the latency of a semaphore is roughly twice as much. Considering that one communication roundtrip between two threads typically involves a pair of semaphores and possibly lock acquisition and release, the overhead of synchronization is multiple times higher than the actual work (memory allocation and deallocation). To make it worse, synchronization latencies are also higher than the time between two consecutive allocation/deallocation requests. Thus, unless these overheads can be reduced significantly, the parallelism from MMT will not offset the overheads to produce a net positive performance benefit.

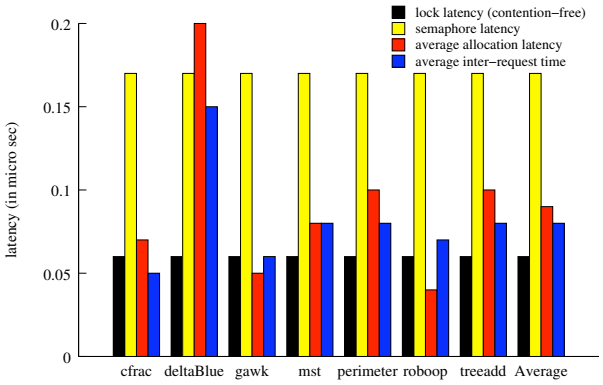

Fig. 2. Latencies of synchronization primitives vs. average memory allocation and time between consecutive allocation and deallocation requests.

Assuming we can overcome the performance challenge, the second challenge is whether the application and memory allocation library can benefit from MMT *transparently*, i.e. without requiring source code modification, hints from programmer, compiler support, library modification, or special hardware support. Achieving such transparency is attractive since the application and the memory allocation library do not need to be redesigned or reprogrammed to enjoy parallelism that MMT provides on a multicore architecture. MMT can be applied, updated, patched, or changed without affecting the application or library code.

Thus, the goal of this paper is to explore how MMT approach can (1) provide performance benefits by exploiting parallelism between main application thread and the dedicated memory management thread, and overcoming the fine-grain task granularity and synchronization overheads, and (2) deliver the speedups transparently from the application and memory management library. This paper makes the following contributions:

First, we propose a new approach (MMT) to accelerate dynamic memory management for sequential applications on multicore architecture, which can speed up heap-intensive benchmarks through several key techniques: speculative memory allocation, bulk memory allocation and deallocation, and eliminating the use of semaphores and locks. Compared to an average slowdown of $2.7\times$ without using these techniques, the techniques combined together deliver a speedup ratio of $1.19\times$, for a set of heap allocation intensive benchmarks running on an Intel Core 2 Quad system.

Secondly, we show that MMT can be designed to be transparent to the application and memory allocation library without modifying any of those. Furthermore, we design MMT to be agnostic to the underlying dynamic memory management library, i.e. MMT does not exploit algorithm or data-structures used by the underlying dynamic memory management library to gain performance improvements. To demonstrate these goals, we use both memory allocation library implemented in GNU C and PHKmalloc library implemented in FreeBSD systems [18]. These two libraries are unrelated, and use significantly different algorithms and data structures. For the same set of benchmarks, MMT delivers an average speedup of $1.19\times$ for both GNU C and PHKmalloc libraries.

Thirdly, we enable security checks in PHKmalloc library that has been useful in detecting many errors related to dynamically allocated objects in many applications like `fsck`, `ypserv`, `cvs`, `mountd`, and `inetd`, [28]. Without MMT, these checks incur on average 21% run time overheads. MMT approach reduces such high overheads to just 1% on average, with *no overhead* in most of the benchmarks.

Finally, in the process of designing an efficient MMT approach, we balance many different trade-offs and design choices that have strong implications on the MMT design. Due to the fine-grain task nature of MMT, we believe that such discussion may be useful for other researchers in considering what design issues are important for exploiting fine-grain function parallelism. For example, we show how to design contention-resistant data structures and mechanisms, and avoid using locks while exploiting fine-grain task parallelism, where even "lightweight" Posix thread synchronization mechanism might be too expensive.

We believe that this paper is a first step towards finding useful scenarios in which sequential applications can be sped up through offloading non-essential and meta-computation to separate threads.

The rest of the paper is organized as follows: Section 2 describes related work, Section 3 discusses MMT design issues. Then, Section 4 gives details about our experimental methodology and Section 5 presents the evaluation results. Finally, Section 6 concludes the paper.

II. Related Work

Dynamic memory management for single core systems. In general, past studies have attempted to improve the performance of dynamic memory management in two ways: designing and implementing better memory management algorithms [18], [21], [23], [31], and increasing the cache reference locality of dynamically allocated objects, through various techniques such as changing the heap layout, predicting lifetime and reference pattern of heap objects, using profiling information, hardware support, etc. [8], [9], [10], [13], [15], [16], [20], [22], [24], [29], [30]. This paper looks into an orthogonal question of how dynamic memory management can take the advantage of multicore parallelism.

Custom and pool-based memory allocation techniques have been used for allocation-intensive programs. However, such an approach suffers from significant challenges: (1) customization requires significant programming effort and non trivial source code modifications, (2) programmers must have a priori knowledge of the allocation and deallocation patterns, (3) memory-related debugging and leak detection become challenging, and (4) many applications are not suited for customized or pool based allocation because of irregular allocation/deallocation patterns, and memory fragmentation and footprint issues due to the coarser allocation and deallocation granularities. Moreover, Berger et al. [3] show that a general purpose allocator such as Doug Lea's allocator performs competitively with custom memory allocators. Therefore, we believe that it is crucial to speed up applications that rely on common memory allocation libraries, such as Doug Lea's GNU C allocator and PHKmalloc library.

Dynamic memory management for parallel and multi-threaded applications. There have been many attempts to design memory allocators for parallel and multithreaded applications, such as Hoard [4], MAMA [17], Streamflow [27], and TCMalloc [14]. Compared to allocators for sequential applications, these allocators attempt to address scalability, avoid false sharing, and restrict memory footprint as first order goals. Currently, MMT targets parallelization between sequential applications and sequential memory allocation libraries. While it is possible to apply our MMT approach for multithreaded memory allocation libraries, it is outside the scope of this work and hence we leave it as future work.

Heap Server. Kharbutli et al. [19] proposed splitting the memory management functions into a separate *process* to decouple the heap meta data and heap data storage in different address spaces in order to improve heap security. While MMT shares the spirit of splitting the memory management functions from the main application, MMT's goal is to improve performance by exploiting multicore parallelism. Consequently, there are very important differences. First, HeapServer requires the use of a specific (bitmap based) memory allocator that is organized differently than commonly used allocators. In contrast, MMT can be used in conjunction with practically any memory allocators. Secondly, Heap Server slows down, rather than speeds up, the performance of heap intensive applications for better heap security. In contrast, MMT improves the performance of sequential heap-intensive applications. Finally, inter-process and inter-thread synchronization and communication protocols differ significantly in their nature and trade-offs. Therefore, designing and implementing a dedicated thread for memory management functions provides a different set of challenges, opportunities, and design issues.

III. MMT Approach and Design

In this section, we will describe our MMT approach, its basic design, and how it achieves the transparency goal. Then, we discuss how we exploit fine-grained parallelism between main application thread and MMT by performing speculative memory management and delaying some memory management tasks to perform them during idle MMT cycles, hiding the cost of such requests from the main application thread, while keeping the MMT design contention resistant and completely agnostic to underlying memory management library implementation. Later, we illustrate how to effectively reduce communication and synchronization costs while offloading fine-grain tasks.

A. MMT Approach

Traditional memory management lets the application directly interact with memory management library by calling functions such as `malloc` and `free` (Figure 3(a)). One way to offload these functions onto a separate thread is by using a fork-join parallelism model in which the application dispatches work to a memory management library thread each time a memory allocation or deallocation request occurs in the application (fork) and the library thread returns the result at the end of processing the request (join), as illustrated in Figure 3(b). However, a fork-join approach requires changes to the memory allocation library and the application as they have to be able to communicate and synchronize their progress, as well as it goes against the transparency goal of MMT. In addition, the approach is inflexible since the order and frequency of when the allocation and deallocation requests are dispatched cannot be changed. Furthermore, the library thread can only be reactive, and perform allocation and deallocation only when requested, instead of being proactive.

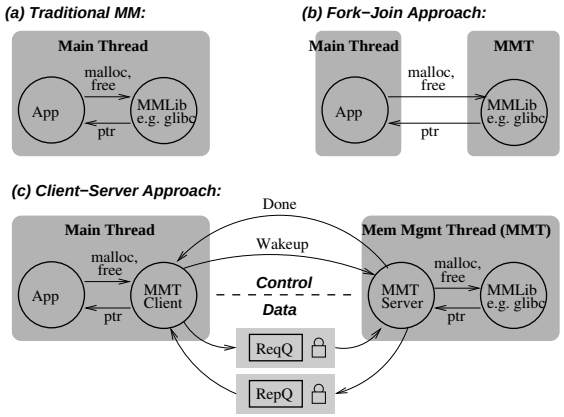

Fig. 3. Traditional Memory Management (a), Fork-join MMT (b), versus Client-Server MMT approach (c).

In order to avoid all the limitations above and achieve the transparency goal of MMT, we propose a client-server model

instead, as illustrated in Figure 3(c). The figure shows that MMT functionalities are split into the client side and the server side. The MMT client, located at the main thread, acts like a memory management library to the application, accepting regular memory allocation and deallocation requests. On a separate thread, the MMT server acts like the application program, generating calls to memory allocation and deallocation routines of the memory management library. The MMT client and server communicate and synchronize to coordinate which requests should be passed on to the library. With this client-server approach, both the application and the memory management library are completely unmodified and are oblivious to the fact that they run on two different threads. Hence, the application can enjoy the parallelism while using a regular memory management library. In addition, the library can be upgraded, patched, or replaced without affecting the way MMT works.

Finally, the client-server approach is flexible in the sense that the MMT client and server determine the order and timing of when allocation and deallocation requests are passed to the library. This allows MMT to apply interesting techniques to achieve performance benefits, such as having the MMT perform speculative dynamic memory management using the memory management library or even delay some management tasks to be performed out of the critical path of the main application thread to hide dynamic memory management costs from main thread.

B. Basic MMT Design

As shown in Figure 3(c), an MMT client places memory allocation and deallocation requests into a *request queue*, and uses a *wakeup* signal to tell the MMT server that there is a request to serve. The MMT server performs the request, and places the reply in a *reply queue*, and uses a *done* signal to tell the main thread of the completion of the work. The reply may contain a pointer to a newly allocated chunk in the case of memory allocation, or no reply in the case of memory deallocation. This is illustrated in Figure 4(a).

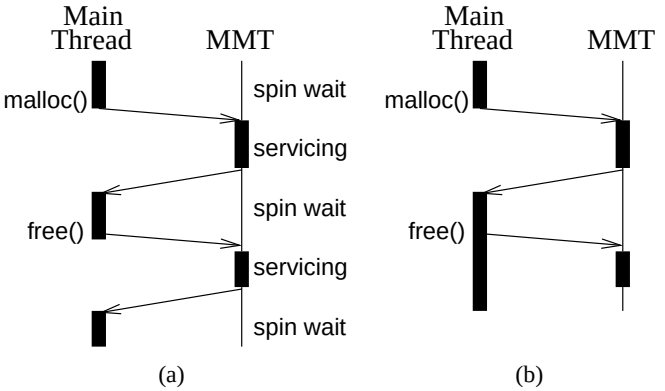

Fig. 4. Protocol for MMT to perform synchronous allocation and deallocation (a), and for MMT to perform synchronous allocation but asynchronous deallocation (b).

This basic design uses a semaphore to signal the MMT server to service a request, and another to signal the MMT client of the completion of the request. In Figure 4(a), both allocation and deallocation calls are *synchronous*, i.e. the main thread is stalled until MMT finishes serving the request. However, since memory deallocation does not return a value to the main thread, it does not need to be synchronous. We can take advantage of this fact by making all memory deallocation calls asynchronous, as shown in Figure 4(b). *Asynchronous deallocation* helps in hiding the cost of deallocation by overlapping it with the main thread and hence, exploiting the parallelism between MMT and main application thread.

While simple, we found that the basic design with asynchronous deallocations slows down, rather than speeds up, applications by $1.9\times$ on average (Section V-A). One fundamental reason for this is that the latency of communication and synchronization relative to the time to perform memory allocation or deallocation is very high. For example, since allocation is synchronous, the main thread is delayed by the round-trip communication latencies between itself and MMT, which include semaphore signal/wait, and queue entry insertion and deletion latencies. As shown in Figure 2, communication and synchronization latencies exceed the latency of the actual work (allocation and deallocation). Therefore, we need to reduce communication and synchronization costs, and at the same time, look for further opportunities to exploit parallelism between main thread and MMT as asynchronous deallocation alone is not sufficient to achieve any performance benefits.

C. Speculative Memory Allocation

One way to reduce the high communication and synchronization costs is to avoid offloading allocation requests since they are synchronous and incur most of the communication and synchronization costs. If allocation requests are performed at the main thread (by the MMT client), we can avoid the communication and synchronization overheads associated with dispatching them to the MMT. Unfortunately, this scheme results in a new type of synchronization overhead. Since allocation and deallocation requests are performed by different threads, the threads can potentially modify the underlying dynamic memory management data structures (e.g. free lists) simultaneously. For example, while the main thread attempts to recycle a free chunk from the free list, the MMT may be adding newly freed chunks to the free list. Therefore, the free list now needs to be protected by a lock, and this causes extra lock synchronization overheads. Contention for the free list lock can be quite severe because asynchronous deallocation calls at MMT compete for locks with synchronous allocation requests performed at main thread. Moreover, such a fine grain task partitioning might potentially reduce the cache locality because dynamic memory management data structures are shared and modified by two different threads. Finally, requiring accesses to the free lists to be wrapped by lock acquisitions and releases breaks the transparency goal of MMT, since the memory management library code must now be modified.

In order to deal with the communication overheads problem without breaking the MMT transparency goal and causing additional synchronization overheads, we go back to letting the MMT to perform not only deallocations, but also allocations. However, to hide the latency of communication and the actual

allocation work, MMT also *speculatively preallocates* heap chunks proactively, in anticipation of the actual allocation requests by the application, during the MMT's *idle* cycles. The key to how preallocation can succeed is our observation that applications tend to allocate many objects of the same size. Therefore, MMT can dynamically *preallocate* an object before the main thread places the allocation request of the object's size, and hence, extract more parallelism between MMT and main application thread in addition to asynchronous deallocation. Done correctly, preallocation can completely hide the allocation latency, hide some part of the communication and synchronization overheads, and reduce MMT contention as preallocation is performed during the MMT's idle cycles.

Note that preallocation requires the MMT to predict the next object size that the main thread will ask to allocate. If the prediction is correct (the size matches), the pointer to the newly created object is returned to the main thread. If the prediction is incorrect (the size mismatches), the MMT has incurred a fragmentation equals to the size of the preallocated object. However, as long as in the future the predicted size is requested, the fragmentation is only temporary. Thus, in general, the MMT does not need to undo its preallocation when it mispredicts the next allocation size during preallocation.

There are several key design questions that need to be addressed in preallocation: *(1) what size of dynamic objects should be pre-allocated at MMT? and (2) when should the MMT preallocate an object?*

To answer the first question, we profiled allocation size distribution for allocation-intensive benchmarks. We found that most dynamically allocated objects are of small size (less than 512 bytes). This observation has an important implication on our preallocation scheme design. Small allocation requests ($\leq$ 512 bytes) are not only common, but are also repeated many times. On the other hand, large requests ($>$ 512 bytes) are rare, less repeated, less predictable, and pre-allocating them result in a higher space fragmentation. Thus, we only preallocate for small allocation request sizes.

To answer the next question of when to preallocate for a particular allocation size, there are several possible approaches that we can consider. In the first approach, preallocation can be triggered *conservatively* when the MMT has seen a high number of allocation requests for a particular size. Such a conservative approach tries to keep unnecessary preallocations to a minimum. However, the approach may not be an optimal choice because (1) keeping track of the number of allocation requests to different size incurs extra occupancy at the MMT, making MMT contention more likely, (2) the cost of miss-speculation is a small amount of fragmentation if preallocation is only made for small request sizes, and (3) the loss of preallocation opportunities during the learning period.

The second approach is more *aggressive*. Rather than waiting until a pattern is established, we start preallocating for a given size the very first time an allocation request for that size is seen. Since the cost of miss-speculation for small chunks is very small, the aggressive approach avoids the cost of employing a learning algorithm and avoids missing performance opportunities to preallocate. Thus, we choose the latter approach for the MMT design.

D. Bulk Memory Allocation and Deallocation

While MMT performing preallocation and asynchronous deallocation are effective techniques for hiding allocation and deallocation latencies through extracting parallelism between MMT and main application thread, unfortunately they are not sufficient for MMT to give a net positive performance benefit. The fundamental reason is that the latency of semaphore signal/wait and lock acquisition is higher than the time between allocation and deallocation requests (see Figure 2). Since each request involves a pair of semaphore signal/wait and queue entry insertion and deletion latencies, requests are generated by the application at a much faster rate than they can be dispatched to the MMT. Therefore, we must reduce communication and synchronization frequencies, reduce their latencies, or both. In this section, we discuss how to reduce their frequencies by performing memory management requests in bulk, and leave reducing their latencies to Section III-E.

1) Bulk Deallocation: Asynchronous deallocation exploits parallelism between MMT and main thread, but to reduce high communication and synchronization frequencies, we aggregate multiple deallocation requests and dispatch them to the MMT server as a single request. We refer to this scheme as *bulk deallocation*. To support bulk deallocation, we design the request queue such that while one part of it still sends a request to the MMT server on each allocation, another part allows deallocation requests to be accumulated and sent to the MMT server as a single request when they fill up the part of the queue.

The number of chunks that are deallocated in one go (the *bucket size*) is an important determinant of performance. Recall that deallocation latency is hidden by MMT because it is performed asynchronously during the time period in which the MMT server would otherwise be idle, so it tempting to set the bucket size large. But idle cycles are scattered as holes located between synchronous allocation requests made by the main thread and bulk deallocation may take too long to fit in any contiguous idle cycles. If an allocation request arrives while the MMT is busy performing bulk deallocation (a *collision*), the allocation request must be delayed until the bulk deallocation completes, which directly exposes the deallocation latency to the main thread. While a too-small deallocation bucket size does not reduce communication and synchronization frequencies sufficiently, a too-large deallocation bucket size exposes deallocation latency. Hence, choosing a good preallocation bucket size is important for optimum performance. Through experiments, we found that, a bucket size of 200 provides a good balance between the competing goals of reducing communication and synchronization frequencies and keeping deallocation latency hidden, across all applications (more discussion in Section V-E).

2) Bulk Speculative Allocation: The principle of aggregating requests cannot be applied directly to allocation requests because they are synchronous. However, we can apply them to preallocation since it is performed speculatively. To achieve that, MMT can preallocate a bucket of chunks of a particular size in parallel with main application thread rather than preallocating one chunk at a time. Furthermore, this group

of preallocated chunks can be sent directly to the MMT client, which will store them locally. When an allocation request of a particular size occurs, and the MMT client has a preallocation bucket for the size, it retrieves one chunk from the bucket and returns it to the application, *without involving any communication or synchronization with the MMT server*. If the bucket of a particular size is empty, the MMT client sends an allocation request to the MMT server.

In designing bulk speculative preallocation, there are several important issues to consider, such as *(1) when preallocation should be initiated, (2) how many buckets there should be, and (3) what preallocation bucket size is appropriate for each bucket*. We will discuss each of the issue in more details next.

The first design issue is when preallocation should be initiated. Suppose that an allocation request of size x is received by the MMT server. The MMT server allocates and returns the requested chunk, and with bulk speculative preallocation, it starts to preallocate N chunks of size x by repeatedly calling `malloc` to the memory management library. If later an allocation request for size x is received again, the speculation is confirmed to be correct, and the MMT server immediately sends off the preallocation bucket to the MMT client. From this point on, the main thread will start to consume chunks from the bucket residing on MMT client until the bucket becomes empty. During this time, the MMT server can take one of the following actions. In a *conservative* approach, after half of chunks in the preallocation bucket are used up, the MMT client sends a *resume-preallocate* signal to the MMT server so that it can start preallocating the next bucket of N chunks. Hopefully by the time the main thread has used up all chunks in the current bucket, the MMT server has finished preparing the next bucket. This conservative preallocation is illustrated in Figure 5(a). An alternative is to use a more *aggressive* approach. The MMT can proactively preallocate the next N chunks immediately after giving a preallocation bucket to the main thread, as shown in Figure 5(b).

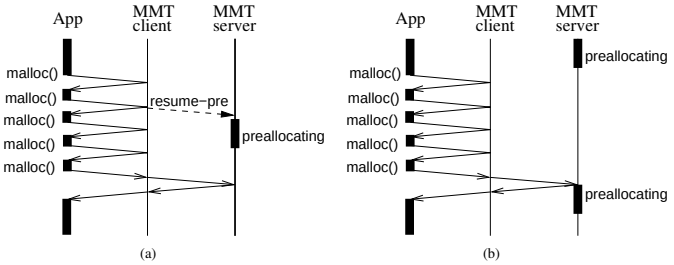

Fig. 5. Preallocation and bulk deallocation (a) conservative preallocation (b) aggressive preallocation

The conservative approach seems reasonable because it reduces the probability of unnecessary preallocations and can hide the allocation cost if preallocations are completed before the next request arrives. However, it suffers from the risk of not being able to preallocate timely as the MMT may be busy when the resume-preallocate signal arrives, in which case preallocation is delayed until the MMT becomes free. On the other hand, the aggressive approach has a better chance of finishing its next preallocation bucket before the main thread needs it. Through experimentation with both approaches, we confirmed that the aggressive approach always outperforms the conservative approach for the same reason. Hence, MMT uses the aggressive preallocation approach.

The next design issue to consider is whether to associate the preallocation buckets with the allocation request size (which is determined by the application) versus the allocated chunk size (which is determined by the memory management library). The actual chunk size allocated is typically the request size rounded up to the nearest multiple of some (e.g. 8) bytes. Thus, there are a lot more possible request sizes than there are chunk sizes. Keeping one bucket for each chunk size helps in reducing the total number of preallocation buckets to be managed but it breaks the transparency goal because it requires the knowledge of the chunk size classes as determined by the memory management library. Furthermore, recall that during preallocation, the MMT server calls `malloc` at the library repeatedly until it obtains the target number of chunks. For preallocation to completely hide allocation latency, the rate at which a bucket is drained by the application must be lower than the rate at which a bucket can be filled by the MMT server, which depends on allocation rate the memory management library can sustain. We found that sometimes there are similar allocation request sizes (e.g. 26 and 28 bytes) made by the application that correspond to a single chunk size (32 bytes). These allocation requests are bursty, and during bursts, the bucket can be drained quicker than a new one can be filled. Keeping two buckets, one for 26 byte request containing 32 byte chunks and another for 28 byte request also containing 32 byte chunks, helps tolerate such burstiness. Therefore, we reduce the contention at MMT by keeping one preallocation bucket for each allocation request size both at MMT client and server side, without breaking the transparency goal of our MMT approach. In order to avoid keeping too many buckets, we only perform preallocation for request sizes smaller than 512 bytes.

Finally, as with bulk deallocation, the choice of preallocation bucket size is important for performance. Choosing too small bucket size might not reduce synchronization cost significantly, on the other hand choosing too large bucket size might expose the preallocation latency to the main application thread. Through experiments, we find that bucket size of 400 offers good performance across all applications we tested, except in few cases where applications generate memory allocation requests in bursts, and the bursts between different allocation sizes coincide. For example, suppose that there are allocation requests for size x, y and z performed in a loop. If the same preallocation bucket size is used for all allocation request sizes, preallocation buckets for them will be filled and drained at the same time. This creates a bottleneck at the MMT server due to increased contention at MMT server such that it must refill several buckets at once, making an allocation request for size y wait the completion of preallocation for size x. This is obviously detrimental to performance since the allocation latencies are now largely exposed to the main thread. To avoid contention at MMT, we vary the bucket sizes for different allocation request sizes, according to the following formula: $BucketSize_i = BaseSize + k \times i$, where i is the allocation request size, $BaseSize$ and k are constants

that are applied to all request sizes. Through experiments, we find that $BaseSize = 400$ and $k = 4$ provide good performance across a wide range of scenarios without any application-specific tuning.

Note that bulk deallocation and bulk preallocation are only coordinated between the MMT client and server. Both the application and the memory management library are oblivious of the bulk deallocation and preallocation, and all the library sees are individual calls to `malloc` and `free` made by the MMT server. MMT approach changes the ordering of allocation and deallocation requests, however that does not affect the correctness or semantics of the programs. Interestingly, changing the ordering of requests and doing them in bulk, on a dedicated thread, improves the cache and TLB performance of the applications we tested (Section V-C). Moreover, we note that cost of miss-speculative preallocations is only small amount of fragmentation and no roll-back is required unlike other speculative techniques.

E. MMT Synchronization Mechanism

In the previous section, we have discussed how to reduce the synchronization frequency. In this section, we will show techniques to reduce the latency of each synchronization in MMT. Bulk speculative preallocation and deallocation decrease synchronization and communication frequency, but increasing the bucket size for them beyond a certain point degrades performance due to increased MMT contention. Therefore, it is important to reduce the latency of synchronization mechanisms to achieve performance improvements.

To reduce the cost of synchronization, we can implement faster synchronization primitives, or avoid the use of semaphores and locks altogether. We will detail them next.

1) Exploring Minimalist Synchronization Primitives: Our first attempt to reduce synchronization cost is to replace Posix thread (Pthread) synchronization primitives with our own "minimalist" implementation. Pthread primitives can cause a thread to block if it waits for a synchronization event to complete. In a minimalist approach, to implement a lock, we use non-volatile variables, atomic (compare and swap) instructions, and a spin loop for waiting in case the lock is not free. A similar implementation is used for the minimalist semaphores. The synchronization latencies for the minimalist primitives, compared to those of Posix, are shown in Table I. In general, the minimalist primitives' latencies are 39-60% lower compared to Pthread primitives. However, in addition to the serialization introduced by locks, their costs are still non-negligible for frequent use by MMT (Section V-B shows results). Thus, we seek to eliminate the synchronization costs entirely.

2) Avoiding Semaphores: The purpose of the semaphore is to signal the MMT that the main thread has a new request for the MMT to work on. The main thread increments the semaphore, while the MMT decrements it (if positive). The semaphore variable is protected by a lock (or by atomic increment and decrement) and that is where the serialization and contention come from. To avoid this bottleneck, we let each thread keep its own local counter. Each thread can write only to its own local counter, but it can read others' local counters. The main thread (MMT client) increments its "wakeup" counter when it has a request for the MMT to serve, while the MMT server increments its "served" counter when it has served a request. The MMT (server) discovers about a pending work if the difference between the wakeup counter and the served counter is positive. Since now each counter only has a single writer, a lock or an atomic increment/decrement over the counter is no longer necessary. It does not assume any kind of support from underlying hardware either.

TABLE I
LATENCY OF PTHREAD VS. MINIMALIST SYNCHRONIZATION PRIMITIVES, REFER SECTION IV FOR MACHINE CONFIGURATION DETAILS.

	Contention-free	2-thread Contention
Pthread lock	$0.05\mu s$	$0.10\mu s$
Minimalist lock	$0.02\mu s$	$0.06\mu s$
Pthread semaphore	–	$0.17\mu s$
Minimalist semaphore	–	$0.07\mu s$

3) Avoiding Locks: Locks are used by MMT client and server to protect against simultaneous modifications to the request and reply queues, where the MMT client inserts requests into the queue, while the MMT server removes requests from the queue. There are two ways to eliminate the use of locks entirely. One way is to use optimistic concurrency (lock-free) approach such as using software transactional memory (STM). However, such an approach suffers from increased latency under a low contention situation due to the multiple passes involved (marking node addresses, acquiring nodes, and committing). Since the goal of avoiding locks is to improve performance, the extra latencies in optimistic concurrency can potentially defeat the purpose. Thus, we use a different approach. We can eliminate locks entirely if we can ensure that at any given time, only one thread writes to the queue. To achieve that, we split the queue into two, and define a time period (epoch) in which the MMT client always modifies one queue while the MMT server always modifies the other queue. In the next time epoch, the role is reversed. This is illustrated in Figure 6. We do not know if such an approach is new (it probably is not), but we argue it is essential for parallelization involving very fine grain tasks, in order to avoid regular lock overheads and critical section serialization.

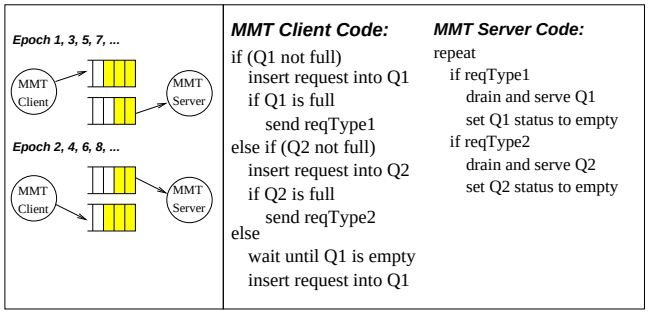

Fig. 6. Illustration and pseudo-code of our lockless protocol.

F. Putting it All Together

Handling Allocation Requests. To handle allocation requests, both the MMT server and MMT client keep a bucket (array) of memory pointers of preallocated chunks associated with each allocation request size. On an allocation request, if the MMT client has a non-empty bucket for the request size, it retrieves an object from the bucket and returns it to the application. Otherwise, it places an allocation request in the allocation request queue and increments the wakeup counter. The MMT server continuously checks the difference between the wakeup and served counters. If it is positive, there is an outstanding allocation request. If it already has a new preallocation bucket filled up for for the requested size, it passes the bucket to the MMT client, and signals the MMT client to resume execution. The MMT server then starts to preallocate a new bucket to replace the one it just passed. Preallocation buckets on either side do not need to be protected by locks as at any time each bucket only has one writer. Note that preallocation is only applied to allocation sizes that are 512 bytes or smaller. Larger allocation request sizes are handled synchronously.

Handling Deallocation Requests. To handle deallocation requests, two bulk deallocation requests are shared between MMT client and server. These deallocation queues are used for all deallocation chunk sizes. The main thread adds a deallocation request to one queue and MMT serves deallocation requests from the other queue, in order to avoid using locks for synchronization, reduce contention at MMT, and exploit more parallelism between MMT and main thread (Section III-E3). The MMT client uses a flag to indicate which queue it has recently filled up. MMT then drains the deallocation requests from one queue while main thread is filling the other queue without stalling. In the rare occasion in which both queues fill up, the main thread waits until the queue it waits for becomes free. With two bulk deallocation queues, not only lock synchronization is avoided, but there is almost no stall since most of the time both the main thread and MMT work in parallel on different queues.

Interaction between Allocation and Deallocation Request Handling. The design for preallocation and deallocation affects the performance of one another positively. Bulk deallocation can potentially reduce the cache locality of the program because recently deallocated chunks have a smaller reuse probability, but bulk preallocation technique increases the cache locality of the program by allocating the chunks before they are needed. Bulk preallocation bucket needs to be refilled in one continuous window of idle cycles at MMT which are scattered, fortunately applying bulk deallocation in conjunction increases the probability of finding one large continuous window of idle cycles as MMT does not get constantly interrupted by single asynchronous deallocation calls. Similarly, bulk preallocation helps bulk deallocations in finding large continuous window of idle cycles. Bulk deallocation might make the MMT busy for a long period of time, which increases the waiting time for allocation requests if they have to be serviced synchronously. However, preallocation reduces the occurrence of this case since most allocation requests can be served from the preallocation buckets. Moreover, when the MMT server has an outstanding request to serve, it prioritizes serving an allocation request over deallocation requests since allocation requests are in the critical path of the program execution. Therefore, overall synergy between bulk deallocation and preallocation helps in achieving higher performance in our MMT approach.

IV. Experimental Methodology

Machine configuration. We evaluated MMT on a 2.4GHz Intel Core2 quad processor, which runs Linux kernel version 2.6.18. For all experiments, only the application and MMT run, plus regular OS processes and daemons, and no other user applications run. Each core has small private L1 instruction and data caches, and the cores share a 4MB L2 cache.

Benchmarks. We use seven heap allocation intensive benchmarks which perform high number of allocation and deallocation calls per unit time (Table II): cfrac, deltaBlue, gawk, mst, perimeter, roboop, treeadd. These benchmarks have been widely used in past dynamic memory management studies [4], [11], [15], [19], [20], [26], [33]. Each benchmark is compiled with `GCC` 4.1.2 with `-O3` optimization flag and with Posix thread version 2.5. For each experiment, each benchmark is run three times, and the average execution time is used for reporting.

TABLE II
The benchmarks used for evaluation

Benchmarks	Input	malloc/free ops per sec (Doug Lea's allocator)
cfrac	a 35 digit number	9,388,037
deltaBlue	1000000	37,908
gawk	large.awk	3,308,227
mst	8192 nodes	3,441,900
perimeter	13 levels	6,083,674
roboop	bench	9,057,914
treeadd	27 levels	13,557,346

Memory management library. To demonstrate MMT's transparent design, we use two widely used memory allocation libraries. The first library implements Doug Lea's allocator [21], which is considered to be among the fastest and most space efficient allocators [3], [5], and also forms the basis of GNU C library. The second library is PHKmalloc allocator designed for FreeBSD operating systems [18]. PHKmalloc differs significantly from Doug Lea's allocator (details in Section V-C).

In addition, PHKmalloc provides memory allocation and deallocation security checks which can be turned on by specifying malloc options, but they result in high performance overheads [18]. However, the security checks have been shown to be highly useful in both identifying both bugs and detecting attacks [18], [28]. We use secure PHKmalloc allocator to illustrate how MMT can reduce the overheads from security checks.

Detailed profiling. We use Oprofile [25], a low overhead profiler, for collecting L2 cache statistics, branch mispredictions and data TLB miss statistics for the process we run. These statistics are collected using a different run than the one used for reporting performance numbers, to avoid any profiling perturbation and noise.

V. Evaluation

In this section, we evaluate the performance of MMT on two different memory allocators (Doug Lea's and PHKmalloc) plus security check-enabled allocator (PHKmalloc), to gain insights into the performance of MMT. Instead of reporting the performance of the final design of MMT, we will show and discuss the performance of various incremental designs of MMT in order to observe the contributions of various design aspects.

A. Basic MMT Design Performance

In this subsection, we evaluate following two designs:

Base	No MMT used. The application calls the Doug Lea's allocator library directly
Design 1	Synchronous allocation and deallocation at MMT
Design 2	Synchronous allocation, bulk asynchronous deallocation at MMT

Figure 7 shows the execution time for various benchmarks normalized to the base case in which MMT is not used, broken down into the time spent for computation or for memory allocation and deallocation. The base case uses Doug Lea's memory allocator. For MMT in Design 1 and Design 2, the execution time is from the main thread, hence the allocation and deallocation time components represent the application waiting for allocation and deallocation requests being performed at the MMT server, or in some cases at the MMT client (e.g., when chunks are extracted from a preallocation bucket). In all designs, we use the minimalist synchronization primitives discussed in Section III-E2, unless otherwise noted.

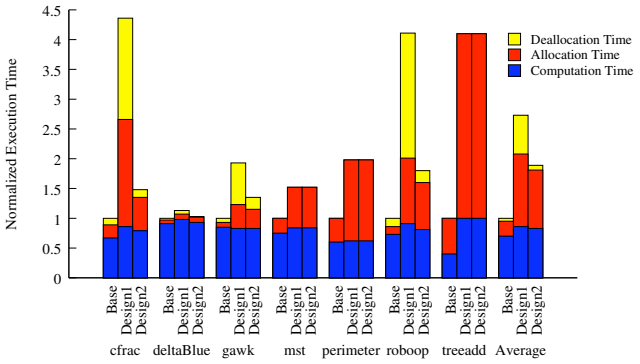

Fig. 7. The performance of basic MMT design.

The Base bars in the figure show that on average, the benchmarks spend 30% of their execution time in memory allocation and deallocation routines. Typically, allocation time is much higher than deallocation time. Comparing Design 1 and the Base case, we can see that offloading each allocation and deallocation request to the MMT significantly slows down all benchmarks, increasing the execution time by 173% on average. In Design 1, since each request is serviced synchronously, it is clear that there is no parallel execution between the main thread and the MMT, hence speedups for Design 1 should not be expected. However, the magnitude of slowdown confirms our data in Figure 2 in that the additional communication and synchronization overheads for each request greatly outweigh the latency to perform the actual memory allocation and deallocation.

In Design 2, we use bulk asynchronous deallocation with a single deallocation queue of size 200 to reduce contention at MMT due to frequent deallocation requests. This reduces the average slowdown from 173% in Design 1 to 89% in Design 2. The reduction in slowdown is primarily due to the 88% reduction in the deallocation time component going from Design 1 to Design 2, demonstrating the ability of bulk asynchronous deallocation in hiding deallocation latency. However, even after the reduction, the deallocation time component in Design 2 is still higher than that in the Base case. Overall, even with asynchronous deallocation in Design 2, MMT still does not get any speedup, and the overall slowdown is still very high.

B. Task Partitioning and Synchronization

From the previous section, we have observed that the main performance problem of MMT in Design 2 is the high allocation time component. Therefore, we need to focus on reducing the allocation cost. One intuitive solution is to perform allocation at the main thread itself instead of at the MMT, as discussed in Section III-C. Thus, in this section, we will evaluate several designs centered in an approach in which the MMT only performs asynchronous deallocation, while allocation is performed at the main thread. The designs are:

Design 3, 4, 5	allocation at main thread, asynchronous deallocation at MMT
Design 3	Pthread synchronization
Design 4	Minimalist synchronization (Section III-E1)
Design 5	Design 4 + bulk deallocation with a single queue

Figure 8 shows the execution time for various designs normalized to the base case in which no MMT is used. Comparing Design 3 with Design 2 from Figure 7, it is clear that moving allocation requests back to the main thread reduces the slowdown significantly, on average from 89% to 46%.

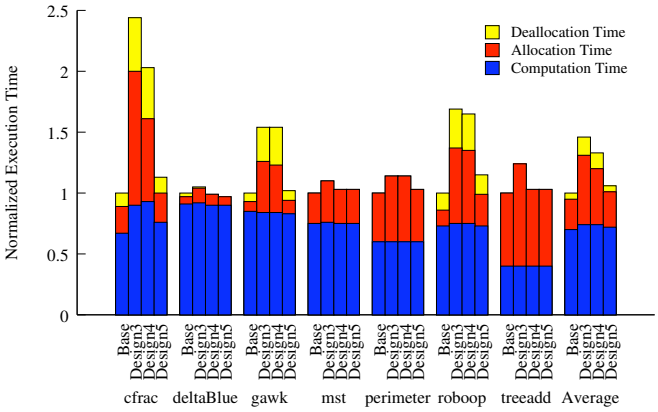

Fig. 8. Performance of task partitioning and synchronization implementation.

Comparing Design 3 and the base case, we can make some interesting observations. First, the allocation time of the base case roughly doubles on average in Design 3 (and quintuples in cfrac and gawk!). Both the base case and Design 3 perform allocation at the main thread by directly calling memory allocation library. Thus, the increase may not seem intuitive. The primary reason is that an allocation request attempts to recycle a recently-freed chunk from the free list. At the same time, deallocation requests at the MMT also attempts to add deallocated chunks into the free list. As a result, the free list must be protected by a lock, and the allocation time suffers from the lock overheads and serialization due to critical sections. This outcome can also be confirmed by looking at the deallocation time which triples, because deallocation also suffers from the same overheads. In addition to these lock overheads, this design breaks the transparency of MMT by making it aware of the internal data structure of the allocation library, i.e. the free lists.

Furthermore, the computation time component increases slightly from the base case to Design 3, especially noticeable in cfrac and roboop, caused by a more subtle phenomenon: the reduction in cache temporal locality when the allocation and deallocation routines run on different cores. For example, the free list is read and written by both the allocation and deallocation routines, so data in the free list is ping-ponged between the cores through invalidations and subsequent L1 cache misses (note that the L2 cache is shared by both cores).

Since synchronization overheads dominate in Design 3, Design 4 replaces Pthread synchronization with our minimalist synchronization implementation (Section III-E1), which roughly halves of the latency of Pthread synchronization primitives. As a result, Design 4's average slowdown compared to Base is only 33%, through a 21% reduction in allocation time and 13% reduction in deallocation time compared to Design 3. However, Design 4 still suffers from serialization cost of critical sections, and from the bad cache temporal locality. Moreover, even with minimalist synchronization primitives, synchronization latencies are still too high.

Design 5 adds bulk deallocation with a single deallocation queue to Design 4 in order to reduce the synchronization and communication frequencies. The bulk deallocation queue has 200 entries, which provides a good balance between synchronization frequency reduction and MMT contention. Compared to Design 4, the lower synchronization frequency reduces both allocation and deallocation time components significantly such that on average Design 5 performs similarly to the Base case. However, in some programs (cfrac and roboop) the slow down is still significant.

C. Bulk Preallocation and Deallocation

Some important lessons from the previous section are that the synchronization frequency for allocation is still high, and that even minimalist synchronization primitives cannot completely remove synchronization cost since the serialization due to the critical sections still remains. Therefore, the final design to explore is one in which the MMT performs bulk speculative preallocation, and synchronization primitives are avoided. As discussed in Section III-C, preallocation increases parallelism by performing many allocations at once during MMT idle cycles. In this section, our final MMT design employs both bulk deallocation and bulk speculative preallocation, as well as communication and access protocols that avoid semaphores and locks (Section III-E2 and III-E3).

To demonstrate MMT's transparent design, we apply MMT to two widely used memory allocation libraries: Doug Lea's allocator and PHKmalloc library. The two allocators are significantly different in their design philosophy, algorithms and data structures: (1) PHKmallloc is aware of the virtual memory system while Doug Lea's allocator is not, (2) PHKmalloc tries to minimize TLB misses and page working set, (3) PHKmalloc does not store metadata for small objects, (4) PHKmalloc rounds up small object requests (less than 2KB) to the nearest power of two versus multiple of 8 bytes in Doug Lea's allocator, etc. Using two widely used allocators that are significantly different is suitable for demonstrating the generality of our MMT approach, since MMT is designed without taking into account the underlying algorithms and data structures employed in both allocators. In addition, both allocators are unmodified and are oblivious to MMT and MMT's parallelization schemes. The performance of MMT for Doug Lea's and PHKmalloc allocators are shown in Figure 9. PHKmalloc's base case performance is close to that of Doug Lea's library (within 3% on average across all the benchmarks we tested), except for deltaBlue in which PHKmalloc outperforms Doug Lea by 30% due to a better virtual memory performance. We could not get one benchmark (gawk) to run with PHKmalloc library, therefore we do not report results for gawk.

The figure shows that despite being designed without taking into account the knowledge of the underlying algorithms and data structures of the allocators, MMT reduces the average execution time by 16% in both allocators, resulting in a an average speed up ratio of $1.19\times$. On an average, MMT reduces the allocation and deallocation time components by between 50-70%, which is a very significant improvement. Moreover, the speedups are highly uniform across all benchmarks, except for deltaBlue on the PHKmalloc allocator. The reason for this is that memory management time is no longer significant in deltablue due to the excellent cache and virtual memory performance in PHKmalloc library.

The most important source of performance is the parallelism between MMT and the main thread. However, that is not the only source of performance improvement, since MMT also reduces the computation time component of some applications. To dig into this deeper, we profile the number of L2 cache misses and TLB misses for both MMT and main thread application thread (the number of TLB misses is the sum over both cores) when we apply our MMT approach to Doug Lea's allocator (Figure 10). We find that the number of L2 cache misses is reduced by 23% in the best case, and 5% on average. Similarly, the number of TLB misses is also reduced by 18% on average. The reason why MMT improves the L2 cache and TLB performance is that the code and cache behavior of regular computation in the benchmarks and the memory allocation library are different enough that they produce mostly

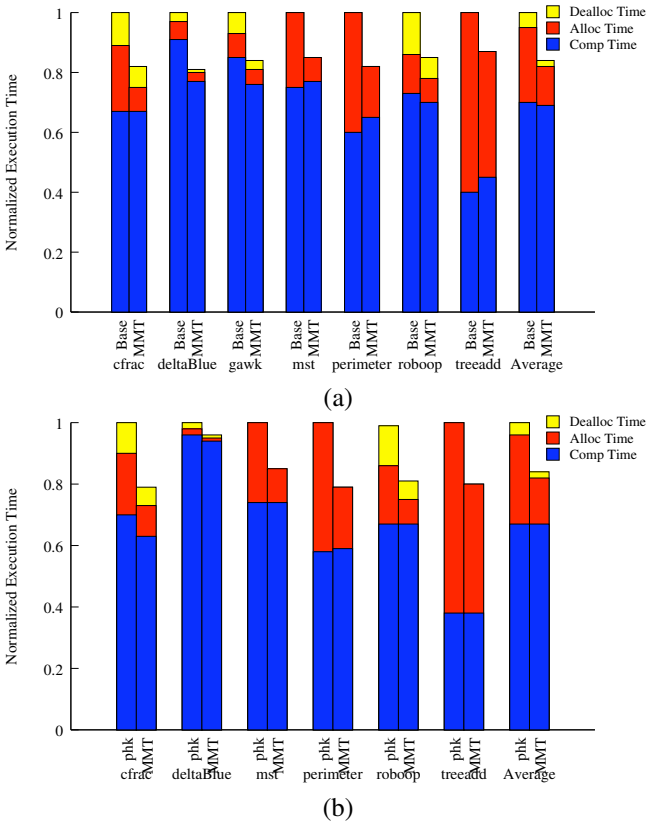

Fig. 9. Performance of the final MMT design on Doug Lea's allocator (a), and PHKmalloc allocator (b).

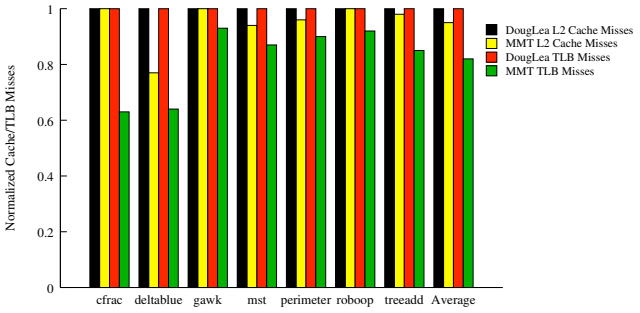

Fig. 10. Last level Cache and TLB Misses (Base: Doug Lea allocator).

negative interference. When they run in separate cores with the MMT approach, they no longer compete and interfere for core resources or cache resources. This observation strengthens the appeal of decoupling the memory management from the main application.

We note that speculative bulk allocation should have positive effect on cache locality, on the other hand bulk deallocations may have negative effect. Interestingly, speculative preallocation and bulk deallocation work synergistically so that overall locality of the program is significantly improved (Figure 10). We also experimented with different techniques improving the cache locality by short-circuiting the deferred deallocations, and found that all programs have better or at least same level of locality compared to sequential base case even without short-circuiting deferred deallocations because of positive synergy between speculative preallocations and bulk deallocations. Though two applications do get significant extra benefits due to short-circuiting. More detailed discussion on the same can be found here [12].

D. Safe Memory Management using MMT

In this subsection, we will show that MMT can also be very effective in hiding the high overhead of security checks in memory management library. For that, we enable extra security features in the PHKmalloc library, which include among others zeroing newly allocated chunks, detecting whether pointers being deallocated are valid, detecting double frees, etc. These checks were designed to cover security vulnerabilities that have been reported in many applications [1], [2], [6], and have been shown to detect bugs or attacks on many applications such as `fsck`, `ypserv`, `cvs`, `mountd`, and `inetd`, etc [28].

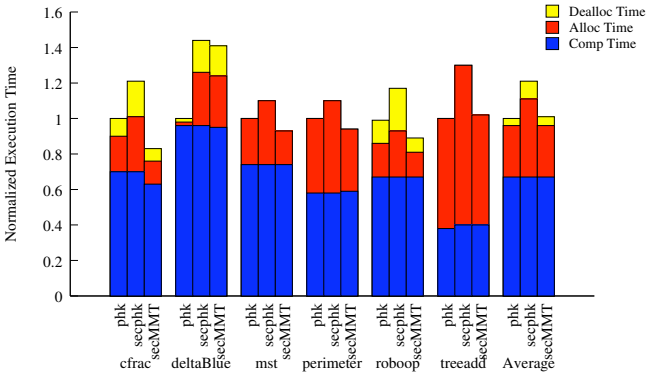

Fig. 11. Security overhead using MMT approach (Base: PHKmalloc allocator)

Note that we do not claim that the security checks are novel, efficient, or complete. They are simply what are provided in the PHKmalloc library. Our focus is that these security checks and features are very expensive, slowing down our benchmarks by 21% on average, and by 44% in the worst case (Figure 11). The figure shows that MMT can hide the overheads almost completely, with an average slowdown of only 1%. Only one benchmark (deltaBlue) still suffers from 41% slowdown. All remaining benchmarks are either faster or just as fast as the base PHKmalloc without security features turned on. Overall, this result demonstrates that the parallelism that MMT has relatively broad benefits. It can be used for improving memory management performance, or for hiding overheads of various security functionalities, all achieved without modifications to the memory management libraries.

E. Sensitivity Analysis

As previously discussed in Section III-D2 and Section III-D1, choosing the right bucket size for bulk preallocation and deallocation is important for performance. We experimented with different bucket sizes for all applications and find that a preallocation bucket size of between 200 and 400 works consistently well for all applications. Similarly, the bulk deallocation queue also gives good performance when the size is between 200 and 400. There is little performance

variation for bucket sizes between 200 and 400. However, bucket sizes less than 50 or larger than 1000 show noticeably worse performance. Because of the stability of performance for bucket sizes between 200 and 400 across all applications tested, MMT does not need to be tuned for specific applications.

F. Memory Footprint

Both bulk preallocation and deallocation may increase the heap memory footprint due to the storage fragmentation of preallocated chunks and delayed storage reclaiming of freed chunks. However, we find that the increased memory footprint for all benchmarks is negligible because the storage overheads of the preallocation and deallocation buckets are small and bounded, compared to the total heap size of the applications.

VI. Conclusion

In this work, we have presented a new approach for exploiting multicore parallelism in dynamic memory management for sequential applications. Dynamic memory management functions are offloaded to a separate thread called MMT. We have presented a thorough discussion of the design issues that are needed in order to exploit parallelism and improve the performance of sequential heap allocation-intensive applications and provide low-overhead safe dynamic memory management.

Our study resulted in several interesting findings. First, it is possible to accelerate dynamic memory management using a separate thread, despite many challenges that need to be overcome for parallelism involving fine grain tasks. Secondly, we show that MMT can be designed to be transparent to the application and memory allocation library without requiring source code modifications, hints from programmer, compiler support, library modification, or special hardware support for them. Furthermore, MMT design is agnostic to the underlying dynamic memory management library algorithms or data structures.

Using heap allocation-intensive benchmarks, we evaluate MMT on an Intel core 2 quad platform. On average, our MMT approach achieves a speedup ratio of $1.19\times$ for both Doug Lea's allocator and PHKmalloc allocator. We also showed that for PHKmalloc allocator, MMT approach can allow security features and checks to be provided without any performance loss on average, which otherwise result in 21% execution time overheads.

We believe that the findings in our paper can be useful to other researchers in considering what design issues are important for exploiting fine-grain function parallelism. We also believe that MMT can be useful for offloading other high overhead tasks, such as debugging, tracing, and profiling.

References

[1] US-CERT Vulnerability Note VU 650937. http://www.kb.cert.org/vuls/id/6509371.

[2] US-CERT Vulnerability Note VU 866472. http://www.kb.cert.org/vuls/id/866472.

[3] E. D. Berger. Reconsidering custom memory allocation. In *OOPSLA,2002*.

[4] Emery D. et al. Berger. Hoard: a scalable memory allocator for multithreaded applications. *ASPLOS, 2000*.

[5] Aniruddha Bohra and Eran Gabber. Are mallocs free of fragmentation? In *USENIX Annual Technical Conference, 2001*.

[6] US-CERT Advisory CA-1993-15. http://www.cert.org/advisories/CA-1993-15.html.

[7] B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 1994.

[8] J.M. Chang and E.F. Gehringer. A high performance memory allocator for object-oriented systems. *IEEE Transactions on Computers*, 1996.

[9] J.M. Chang, Y. Hasan, and W.H. Lee. A high-performance memory allocator for memory intensive applications. In *High Performance Computing in the Asia-Pacific Region, 2000*.

[10] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *PLDI, 1999*.

[11] D. Detlefs, Al. Dosser, and B. Zorn. Memory allocation costs in large C and C++ programs. *Software Practice and Experience*, 1994.

[12] Devesh Tiwari, Sanghoon Lee, James Tuck and Yan Solihin. Memory management thread for heap allocation intensive sequential applications. In *Proceedings of the 10th MEDEA workshop on Memory Performance*. ACM, 2009.

[13] Yi Feng and Emery D. Berger. A locality-improving dynamic memory allocator. In *Workshop on Memory System Performance, 2005*.

[14] S. Ghemawat and P. Menage. Tcmalloc : Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[15] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. *SIGPLAN Not.*, 28(6):177–186, 1993.

[16] D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Softw. Pract. Exper.*, 1990.

[17] Simon Kahan and Petr Konecny. MAMA!: a memory allocator for multithreaded architectures. In *PPoPP, 2006*.

[18] P.-H. Kamp. Malloc(3) revisited, http://phk.freebsd.dk/pubs/malloc.pdf.

[19] Mazen Kharbutli, Xiaowei Jiang, Yan Solihin, Guru Venkataramani, and Milos Prvulovic. Comprehensively and efficiently protecting the heap. *ASPLOS, 2006*.

[20] C. Lattner and V. Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *PLDI, 2005*.

[21] Doug Lea. A memory allocator, http://g.oswego.edu/dl/html/malloc.html.

[22] W. Li, S. Mohanty, and K. Kavi. "a page-based hybrid (software-hardware) dynamic memory allocator". *IEEE Computer Architecture Letters*, 2006.

[23] Mark Moraes. Csri malloc, ftp://ftp.cs.toronto.edu/pub/moraes/malloc.tar.gz.

[24] J. Morris Chang, W. Srisa-an, and C.-T.D. Lo. Architectural support for dynamic memory management. In *ICCD, 2000*.

[25] Oprofile. http://oprofile.sourceforge.net/.

[26] Anne et al. Rogers. Supporting Dynamic Data Structures on Distributed-Memory Machines. *ACM Trans. Program. Lang. Syst., 1995*.

[27] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *ISMM, 2006*.

[28] R. Seacord. Secure Coding in C and C++. Addison-Wesley Professional, 2005.

[29] Matthew L. Seidl and Benjamin G. Zorn. Predicting references to dynamically allocated objects. *CU-CS-826-97, University of Colorado at Boulder*.

[30] Dan N. Truong, Francois Bodin, and Andre Seznec. Improving cache behavior of dynamically allocated data structures. In *PACT*, 1998.

[31] KP Vo. Vmalloc: A general and efficient memory allocator:. In *Software: Practice and Experience, 1996*.

[32] P. R. Wilson, Mark S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management*, 1995.

[33] Q Zhao, R Rabbah, and Weng-Fai Wong. Dynamic memory optimization using pool allocation and prefetching. *SIGARCH Comput. Archit. News*, 2005.

[34] Benjamin Zorn and Dirk Grunwald. Empirical measurements of six allocation-intensive C programs. *SIGPLAN Not.*, 1992.