McRT-Malloc - A Scalable Transactional Memory Allocator

Richard L. Hudson*, Bratin Saha*, Ali-Reza Adl-Tabatabai*, and Benjamin C. Hertzberg**

*Programming System Lab
Microprocessor Technology Lab
Intel Corporation
{Bratin.Saha, Ali-Reza.Adl-Tabatabai, Rick.Hudson}
@Intel.com

**Computer Architecture Lab
Stanford University
Palo Alto California
elektrik@stanford.edu

Abstract

Emerging multi-core processors promise to provide an exponentially increasing number of hardware threads with every generation. Applications will need to be highly concurrent to fully use the power of these processors. To enable maximum concurrency, libraries (such as malloc-free packages) would therefore need to use non-blocking algorithms. But lock-free algorithms are notoriously difficult to reason about and inappropriate for average programmers. Transactional memory promises to significantly ease concurrent programming for the average programmer. This paper describes a highly efficient non-blocking malloc/free algorithm that supports memory allocation and deallocation inside transactional code blocks. Thus this paper describes a memory allocator[1] that is suitable for emerging multi-core applications, while supporting modern concurrency constructs.

This paper makes several novel contributions. It is the first to integrate a software transactional memory system with a malloc/ free based memory allocator. We present the first algorithm which ensures that space allocated in an aborted transaction is properly freed and does not lead to a space blowup. Unlike previous lock-free malloc packages, our algorithm avoids atomic operations on typical code paths, making our algorithm substantially more efficient.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Language Constructs and Features – Dynamic storage management, Concurrent programming structures

General Terms Algorithms, Performance, Languages

Keywords Transactional Memory, Runtimes, Memory Management, Synchronization.

1. INTRODUCTION

Future generations of multi-core processors will provide an increasing number of multi-threaded cores on a single die. To exploit the computational power of these processors, language runtimes must scale to a large number of threads and provide new primitives that simplify concurrent programming [14]. In a cache-coherent shared memory platform, the memory allocator plays an important role in scalability and cache locality optimizations. To enable applications to exploit concurrency to the maximum, the memory allocator needs to employ non-blocking algorithms to avoid becoming a scalability bottleneck. Moreover, transactional programming provides a powerful tool for easing concurrent programming; therefore, the memory allocator must support memory allocation and deallocation in a transactional environment. Past research has investigated memory allocation and transactional memory in isolation, but none has examined the challenges of integrating these two services.

This paper is the first to present a scalable non-blocking memory allocation (malloc/free) algorithm that is tightly integrated with a high-performance software transactional memory (STM) system. Thus our memory allocator is designed for emerging multi-core environments.

This paper makes the following contributions:

- It presents a new non-blocking, scalable memory management algorithm, which we call McRT-Malloc. The malloc and free fast paths avoid expensive compare-and-swap operations by accessing only thread-local data. Since the memory allocator is non-blocking, it avoids becoming a scalability bottleneck and can be used by other highly tuned libraries.
- It is the first to show a transaction-aware memory allocator. For example, memory allocated inside aborted transactions appears never to have been allocated. Moreover, our memory allocator operates correctly in the presence of nested transactions with partial rollback; for example, memory allocation in a parent transaction is not rolled back if a child transaction is aborted. Rather than layer the memory allocator on top of the STM, we tightly integrate the memory allocator with the STM. This gives us several benefits: (1) it improves the performance of malloc and free operations inside transactions by avoiding the overhead of STM; (2) it improves the space efficiency of these operations by recycling memory that is allocated and freed within a transaction; and (3) it allows the STM to perform object-level contention detection by leveraging the memory allocator's metadata.
- It presents a novel algorithm for detecting reachability in the presence of transactions. Unlike prior work, our algorithm incurs a constant overhead per transaction, avoids atomic operations, and therefore has better caching properties.
- It shows how the memory allocator enables the design of an update in place (undo-based) software transactional memory. Current STMs use a write-buffering approach, but an undo-logging based approach provides a significant performance

[1] We use memory allocator instead of memory manager to indicate a malloc/free system as opposed to a garbage collected system.

advantage [17]. Thus this paper shows how a memory allocator may be leveraged for making other parts of a runtime system more efficient; this is among the first papers to demonstrate such synergies.

- It is the first paper to show how to use abort and commit hooks to provide transactional semantics for system services invoked inside transactions. We believe our method can be used for other services, e.g., transactional output.

Some of the problems we tackle here are unique to a non-garbage collected environment. One has to look no further than Linux to see that non-garbage collected environments are widespread in system software and in many application domains, especially inherently concurrent application domains such as graphics, media, gaming, and others.

All our algorithms have been implemented in an experimental multi-core runtime. We present numbers that show our base malloc/free system compares favorably with Hoard, widely considered the best of class malloc/free allocator targeted at multi-threaded environments. We show that the book keeping overheads of our algorithms are acceptable when running applications using our STM.

The rest of this paper is organized as follows. First we discuss the relevant literature. Next we describe the base malloc/free implementation. We then tackle the task of integrating the allocator with our transactional memory system. Analysis and results follow and then we conclude.

2. RELATED WORK

2.1 Concurrent Malloc/Free

The study of thread safe explicit allocation / free algorithms has a long and inspiring history.

Berger, et al. [2] created Hoard, a malloc/free package and showed that speed, scalability, false sharing avoidance and low worse case fragmentation was possible. Michael [11] showed that it was possible to add immunity to deadlock, tolerance to priority inversion, kill-tolerant availability, preemption tolerance, and async-signal safety. We support nine out of the ten properties listed by these authors, making no attempt to encourage the use of async-signals.

Though not explicitly stated in their papers, their algorithms rely heavily on an efficient CAS (compare and swap) hardware instruction for performance. Hoard uses one CAS in its common path for malloc and Michael uses two. CAS instructions on current commercially available processors can be from one to two orders of magnitude slower than similar non-atomic operations. Our algorithm avoids atomic operations in the common paths of the malloc and free routines, and uses them in a non-blocking way in the non-common path code.

Some of the data structures used in the malloc implementation are similar to those found in Hoard paper but adapted to be non-blocking. For example, the use a block metadata instead of per-object headers is similar to what Hoard does. The size segregation structures were suggested by Johnstone [12].

2.2 Transactional Memory

This is the first paper to present algorithms that integrate explicit memory management with transactional memory. Previous work on software transactional memory [1], [3], [4],[5], [8] [9], [13], [15] either assumed a garbage collected environment or avoided dealing the optimization and correctness issues related to manual allocation.

Some prior STM work has proposed the use of hazard pointers [16] for memory management. Previous researchers have noted the problem of detecting reachability in the presence of transactions [7] and others have proposed hazard pointers [16] and the repeat offender solution [10] to solve similar problems. Each of these approaches incur an overhead proportional to the number of loads and stores in the transaction.

3. McRT-MALLOC

At initialization, McRT-Malloc reserves a large piece of virtual memory for its heap (using an operating system primitive like the Linux mmap and munmap primitives) and divides the heap into 16K-byte aligned *blocks*[2] that initially reside in a global block store. When a thread needs a block from which to allocate objects, it acquires ownership of one from the block store using a non-blocking algorithm we describe later. After acquiring a block, a thread divides it into equal-sized *objects* initially placed on the block's *private free list*. The thread owning a block also owns the objects in the block.

Each thread maintains a small thread-local array of bins, each bin containing a list of blocks holding the same sized objects. The array contains a bin for every multiple of four between 4 bytes and 64 bytes, and a bin for every power of two between 128 bytes and 8096 bytes. Allocation of objects greater than 8096 bytes is handled by the operating system (e.g., on Linux we use mmap and munmap). Large objects are rare so we don't discuss them further.

When a thread dies, the scheduler notifies McRT-Malloc, which then returns the dead thread's blocks back to the block store. Unowned yet nonempty blocks in the block store are not owned by any thread. These partially filled blocks are segregated based on object size and managed using the same non-blocking algorithms used for the empty blocks. When sufficiently empty a block can be distributed to another thread, which then takes ownership of the block's objects.

The base of each block contains a 64-byte structure holding the block's meta-data. With the exception of the public free list field, the fields in the metadata block are strictly local to the owning thread and are therefore trivially thread safe. To facilitate efficient insertion and deletion the next and previous fields support the doubly-linked list of blocks in each bin. There is a field indicating the size of objects in the block, a field tracking the number of objects in the block, and a field tracking the number of objects that have been allocated but not immediately available for allocation. Per-object headers are not needed by the base McRT-Malloc since all relevant information is held in the block's meta-data.

Each block has fields for two linked lists of freed objects. A private free list field points to a linked list of objects available for allocation; the thread satisfies malloc requests from this list. A public free list field points to a linked list of objects freed by threads that do not own the block (*foreign* threads). There is a bump pointer field used to avoid explicitly initializing the private free list for an empty block. The bump pointer implicitly represents the free list and is used for allocation of objects in newly acquired empty

[2] Some of the literature uses the term super block where we use the term block and block where we use the term object.

blocks. Once the bump pointer reaches the end of the block, the thread allocates objects using the explicit private free list.

3.1 Allocating and Freeing Objects

The algorithm for allocating objects first rounds the requested size up to the next binned size and then allocates from one of the blocks in that bin. Objects whose size is a power of two are aligned to that power of two. The search for a block with free space is key to good performance and must balance the amount of information maintained and timely response to a call to malloc. We discuss this in more detail in Section 3.3.

To allocate an object within a block, the thread first checks the block's bump pointer. If the bump pointer is null, the thread allocates the object from the private free list; otherwise, it increments the bump pointer by the size of the object, checks for overflow, and returns the object if there is no overflow. On overflow, it sets the bump pointer to null and allocates from the private free list. The private free list is accessed only by the owning thread so this allocation algorithm is trivially thread safe.

If the private free list is empty then the thread examines the public free list to see if it has any objects to *repatriate*. A thread repatriates the objects on the public free list of a block that it owns by moving those objects to the block's private free list using a non-blocking thread safe algorithm.

The algorithm for freeing an object depends on whether the thread performing the free owns the freed object. To free an object it owns, a thread places the object on the private free list of the block containing the object, a thread safe operation. To free an object it does not own, a thread places the object on the public free list of the object's block using a more expensive non-blocking operation.

3.2 Non-blocking Operations

The key challenge in the McRT-Malloc algorithms is to minimize the use of CAS instructions in implementing the non-blocking algorithms. We solve this in 2 ways: (1) we use the bits freed up due to the alignment of the blocks to accomplish our non-blocking properties, in particular to avoid the ABA problem, and (2) we maintain two free lists, one private and trivially non-blocking, and another a public version which is managed as a single consumer, multiple-producer queue and implemented as a non-blocking structure.

The ABA problem is a classic concurrency problem, publicly identified in the IBM 370 manual along with the introduction of the CAS instructions. Assume we have a LIFO queue implemented as a linked list with A->B->C on it. Assume thread one inspects A, noting B is the next object on the queue and is then delayed. If thread 2 pops A, pops B, and then pushes A the queue holds A->C. Now if thread 1 CASes the queue it will find an A as it expects but instead of replacing it with C as it should will replace it with B. This assumption that if A does not change its next field does not change is at the root of the problem.

Initially each block in the heap is publicly owned and resides in the block store. The block store manages the blocks using a non-blocking concurrent LIFO queue data structure. The queue is maintained as a linked list threaded through the first word of an aligned 16K block. Any block in a 32 bit address space requires 18 bits to address (the lower 14 bits of a block's address are always zero). To avoid ABA problems the queue uses a versioning scheme and a 64-bit CAS instruction. The version number uses 46 bits leaving 18 for the blocks address. The question then becomes whether 46 bits is sufficient to prevent rollover and the ABA problem.

For the queue with a 64 bit CAS, 18 bits needed to address the blocks leaving 46 bits available for versioning. Assuming we allocate at 1 byte per cycle, which is a furious allocation rate, on a 3 Gigahertz machine we would allocate 3, 000, 000, 000 bytes per second. This would require 181,422 blocks per second. Wrap around would take slightly over 12 years and wrap around would only be a problem if a thread was suspended at just the right time and stayed suspended for the requisite 12 years. A 64 bit computer would require a 128 bit (16 byte) double wide CAS instruction. The locked cmpxchg16b instruction provided by the x86 is an example of such an instruction and trivially provides for a full 64 bits of versioning which is sufficient.
We provide the pseudo code for pushing and popping off the block store's queue.

```
 /*
 * Pseudo code for non-blocking push and
 * pop routines for 16K blocks.
 */
void lifoQueuePush(block,  queue)
{
   do {
      oldVal = queue->top;
      [oldCount, oldBlockPtr] =
          unpack(oldVal);
      newCount = oldCount + 1;
      block->next = oldBlockPtr;
      newVal = pack(block, newCount);
   } while(!CAS64(&queue->top,oldVal,
                  newVal));
   return;
}

block lifoQueuePop(queue)
{
    do {
        oldQueue = queue->top;
        [oldCount, blockPtr] =
            unpack(oldQueue);
        if (blockPtr == NULL) {
            /* The queue is empty */
            return NULL;
        }
        newCount = oldCount + 1;
        newBlockPtr = blockPtr->next;
        newQueue =
            pack(newCounter, newBlockPtr);
    } while (!CAS64 (&queue->top,
                     oldQueue,
                     newQueue));
    return blockPtr;
}
```

Figure 1 Non-blocking queues

The public free list holds objects that were freed by foreign threads. Adding an object to a public free list is expensive since it requires an atomic CAS instruction on a cache line that is likely in another processor's cache. To push an object onto the public free list, load the public free list from the blocks metadata, assign the next field in the object to this and do a CAS. If the CAS succeeds we are done, if not, we repeat the process as many times as is required. The pseudo code is provided below.

```
freeListPush (object)
{
    do {
        oldPublicFreeList = publicFreeList;
        object->next = publicFreeList;
    } until CAS(&publicFreeList,
                 object,
                 oldPublicFreeList);
}

repatriatePublicFreeList ()
{
    temp = SWAP(publicFreeList, NULL);
    temp->next = privateFreeList;
    privateFreeList = temp;
}
```

Figure 2 Public Free List

To repatriate the public free list, a thread (1) atomically swaps the public free list field with null (using the IA32 locked exchange instruction), and (2) sets the private free list field to the loaded public free list value.

This is ABA safe without concern for versioning because the concurrent data structure is a single consumer (the owning thread) and multiple producers. Since neither the consumer nor the producer rely on the values in the objects the faulty assumptions associated with the ABA problem are avoided.

3.3 Framework for Managing Blocks

The blocks in any particular size bin are arranged in a circular linked list with one of the blocks distinguished as the *bin head*. As malloc traverses this linked list looking for a free object it collects per bin statistics including how many free objects are encountered upon arriving at a block and how many blocks are on the list. A complete traversal is noted when the thread encounters the distinguished bin head block. The collected statistics are used to set policy for the next traversal. Policy includes whether additional blocks should be allocated to the bin or empty blocks returned to the main store. The process is repeated for each full traversal of the list.

The currently implemented policy is that if less than 20% of the objects are free and available we add blocks and if a block becomes completely empty it is returned to the block store. One final wrinkle is that if we inspect 10 blocks without finding a free object then we add a new block for the bin. This places an upper bound on the time malloc will take before returning an object.

The take away here is that we use the distinguished bin head block as a hook to set policy and the traversal of the blocks to collect the statistics used to set the policy.

4. TRANSACTION AWARE ALLOCATION

The memory allocator is tightly coupled with our previously described software transactional memory (STM) module, [17], that is also part of the runtime. This section discusses how we augmented the memory allocator to handle allocation/deallocation inside transactions.

Our STM uses 2 phase locking for enforcing transactional semantics. All memory locations are mapped to a unique lock; the lock can either be owned by a transaction, or it can contain a version number. Before writing to a location, transactions acquire the lock (referred as write-locks) guarding the location, and log the old value. Before reading a location, transactions check that no writer has acquired the lock (referred as read-locks) for the location, and log the version number of the lock. Prior to commit, the transaction checks that the version number of the read-locks haven't changed. The transaction also releases the write-locks during commit and increments their version number. Since our STM uses optimistic concurrency, transactions may read and execute with stale values (in particular, stale pointer values); nevertheless these transactions will fail to commit.

As explained in [17] the mapping from a memory location to the unique lock can be done either at the granularity of the address or the granularity of an object. To achieve object granularity each object is augmented with an object header placed before the object. To map from an arbitrary address to the start of an object we take advantage of the fact that all objects within a block are the same size and therefore the headers for these objects all fall at known, easily determined, offsets from the base of the block.

For handling allocations inside transactions, McRT-Malloc needs to address 2 key issues: (1) allocations must obey transactional semantics; for example, all memory allocated inside a transaction must be rolled back on an abort. (2) memory must be recycled only when it becomes truly unreachable. Due to optimistic concurrency, some transactions may hold stale pointers into freed memory; McRT-Malloc must ensure such memory is not recycled.

One way to handle memory management inside transactions is to treat the entire allocation/deallocation code sequence as part of the transaction; that is, treat the malloc code as ordinary loads and stores and reuse the STM to provide transactional semantics. The other option is to make the allocation routines implement transactional semantics independent of the STM. The memory allocator takes the later approach for the following reasons: (1) We did not want to slow down the memory allocation sequence for non-transactional code by using read and write barriers throughout the allocation sequence. We believe that the common case is most allocation would happen outside transactional regions. To avoid paying any overhead for non-transactional allocations, we would need two versions of every function used in the allocation sequence; one used for non-transactional allocation and another used for transactional allocation. Unfortunately, STMs do not (yet) guarantee strong atomicity; that is, transactional code is not atomic with respect to non-transactional code which means that a non-transactional thread could not allocate concurrently with a transactional thread. (2) Making the allocation transactional would help only with undoing allocation activity if a transaction aborts. It would not help in detecting when memory becomes unreachable and ready for recycling. (3) We would still need special actions to prevent memory freed inside a transaction from being reused inside a non-transactional thread through the global allocation pool. (4) The STM itself needs to allocate memory for its internal structures. This creates a cyclic dependency where the malloc uses the STM, and the STM reuses the malloc. The dependency can be resolved, but it makes the design messier.

4.1 Transactional Allocation

We will use the following terms to denote transactional allocation/deallocation. (1) Speculatively allocated: this is memory allocated inside a transaction. The allocation is valid if the transaction commits, but must be rolled back if the transaction aborts. (2) Speculatively freed: This is memory allocated outside a transaction, but freed within a transaction. If the transaction aborts, the freeing must be rolled back, and the memory must still remain valid. (3) Balanced allocated: The memory is allocated and freed inside a transaction. No special action is needed on a commit or abort since the memory is essentially transaction local.

To enforce transactional semantics, allocation actions must appear to other threads only at transaction commit; for example, memory is presented to the free routines discussed in section 4.2 only at transaction commit. This means that the memory allocator needs to provide some guarantees against space blowup. The memory allocator ensures that balanced allocations and repeated speculative allocations do not lead to space blowup. The transaction in Figure 3 shows a loop holding both a malloc and a free. A naïve STM that delays the freeing until the transaction completes risks space blowup and depleted memory resources.

```
transaction
{
  for (i=0; i<big_number; i++) {
    foo = malloc(size);
    free(foo);
  }
} /* is safe for space */
```

Figure 3 Space Blowup

On the other hand if the STM does not free the malloced object shown in Figure 4 one could also encounter space blowup.

```
transaction
{
  foo = malloc(size);
  abort();
} /* is safe for space */
```

Figure 4 Speculative Allocation

Speculatively freed memory can increase the space usage; however, in most cases, the increase in space usage is bounded by the amount of memory that was allocated prior to the transaction. There are some producer-consumer use scenarios where space blowup may be an issue. This involves one thread allocating memory non-transactionally (or allocating transactionally and then committing the transaction), while another thread tries to free the memory transactionally.

4.1.1 Nested Transactions

The STM supports nested transactions with partial aborts. Nested transactions are executed in the same thread as the parent. It supports a closed nesting model; that is, the commit of child transactions is contingent upon the commit of the parent transaction. Updates made by a child transaction are visible to other threads only when the parent transaction commits; moreover, when a parent transaction aborts, it also rolls back the actions of the child transaction. However, an abort of a child transaction does not roll back the actions of the parent transaction.

Nested transactions make it more difficult to detect balanced allocations (and detecting balanced allocations is crucial for providing space guarantees). In the code sequence in Figure 5, the inner transaction could get aborted which means that the `free(foo)` is not balanced.

```
transaction {
  foo = malloc(size1);
  goo = malloc(size2);
  transaction {
     free(foo); /* not balanced */
     bar = malloc(size3);
     free(bar); /* balanced */
  }
  free(goo); /* balanced */
}
```

Figure 5 Speculative Free

Our first attempt was to use transaction nesting depth to detect speculatively freed and balanced objects, but unfortunately this is expensive to implement:

```
transaction {
   foo = malloc(size);
}

transaction {
   free(foo);
}
```

Figure 6 Simple Transactional Malloc and Free

When a toplevel transaction commits, we would have to adjust the nesting depth of all allocated objects so that a subsequent transactional region can detect speculatively freed objects.

We use a novel sequencing approach to detect balanced allocations. Every transaction increments a thread-local ticket when it starts. Nested transactions have a different ticket number from their parents. A transaction creates a sequence number by concatenating the thread id and the ticket number. Every object allocated inside a transaction is tagged with its sequence number. If an object was allocated in a different thread (detected from the sequence number), then any transactional free is a speculative free. If the object was allocated by the same thread and its sequence number is less than the sequence number of the transaction in which it is being freed, then the transactional free is a speculative free. Figure 7 is an annotated example showing how sequence numbers are used to detect balanced frees.

```
transaction { // seqNumber=1
  transaction { // seqNumber=2
    transaction { // seqNumber=3
      foo = malloc(size); // seqNumber=3
      bar = malloc(size); // seqNumber=3
    }
  }
  transaction { // seqNumber=4
    free(foo); // 3 < 4,speculatively free
  }
  free(bar); // 3 > 1, balanced free
}
```

Figure 7 Nested Allocation and Freeing

4.1.2 Handling Deferred Actions

Our STM exports callbacks that are invoked when a transaction is committed or aborted. McRT-Malloc uses these callbacks to handle deferred actions. The callbacks process two auxiliary data structures: an undo log and a commit log. On a transactional allocation, the sequence number of the current transaction is added to the object header, and the object is placed in the undo log. On a transactional free, we first check whether the free is speculative or balanced. For a balanced free, we remove the object from the undo log (the transactional allocation would have added the object to the undo log), and add it to the block freelist (just like a normal free). For a speculative free, we place the object in the commit log.

When a transaction commits, the commit callback processes the commit log and any speculatively freed objects that have now become balanced may be freed. If the toplevel transaction commits, then we reset the undo log. (In our implementation, we free the commit log objects only at the top level commit. At the top level we don't need to check whether an object was allocated balanced, we can free all the commit log objects. This creates the possibility of a space blowup, but it should be obvious that our framework supports eager freeing at nested transactions.)

When a transaction aborts, the abort callback processes the undo log and frees the speculatively allocated objects. This traversal is efficient since objects are added to the undo list as they are allocated. Hence all the objects allocated in the aborted transaction are contiguous and at the end of the undo list. We also restore the commit log to its state before the aborted transaction started. This rolls back all the speculative frees; the commit log restore is similar to the general STM mechanism for handling read/write sets for nested transactions [17].

4.1.2.1 Pitfall: Bump Pointers

When allocating in a fresh block McRT-Malloc uses a bump pointer to implicit represent the free list. Once all the objects in the block have been allocated we revert to a traditional free list scheme. If we note the bump pointer at the start of a transaction then objects allocated beyond this original bump pointer would only be visible to the local transaction. This would provide us with a quick check to detect if the objects are visible only to the transaction. If the objects do not escape the transaction until commit then we can elide the transactional logging used to detect conflicts. We have left this optimization for future implementation.

We also naively assumed that on aborts simple resetting the bump pointer to its value when the transaction started would be a nice optimization. We did not use such a bump pointer rollback scheme since it introduces other overhead which defeats fast rollbacks.

```
transaction {
  foo = malloc(size);
  free(foo); // foo appended to freelist
  abort(); // bump pointer restored
}
```

Figure 8 Bump Pointer Problem

In Figure 8 to reuse `foo` within the transaction, we need to add it to the freelist. If the transaction aborts and we rollback the bump pointer, we would then need to process the freelist and delete the entry for `foo`.

4.1.3 Implementation considerations

We wanted to minimize the overhead in the following cases: (1) non-transactional allocation, and (2) transactions without any allocation. To minimize non-transactional allocation overhead, the memory allocator checks at the beginning (via a TLS access) whether we are inside a transaction, and if not, executes the vanilla memory management code. Thus, non-transactional allocation only requires an additional access to a TLS field. Further, the STM maintains a transaction local structure in the TLS (called the transaction descriptor) that holds transaction related metadata. We added extra fields in the descriptor to note whether there was any allocation activity inside a transaction. If there was no allocation activity, the commit/abort hooks remain disabled, and the transaction commit/abort code does not pay any overhead.

4.2 Critical Sections != Transactions

We will now discuss the tension between the explicit malloc/free and STMs highlighting the following interesting problem. This problem shows up in languages such as C and C++ where transactions can not simply replace mutual exclusion critical sections without consideration for how explicit malloc/free memory allocation is handled.

Figure 9 shows the simplified code sequence to delete a node in a linked list. Similar code written using a mutex and a critical section would work fine. The function traverses a list to find a node with a given key and then unlinks it from the list.

```
nodeDelete(int key) {
    ptr = head of list;
    transaction {
        while( ptr->next->key != key ) {
            ptr = ptr->next;
          } /* end while */
     temp = ptr->next;
     ptr->next = ptr->next->next;
    } /* validate & end transaction */
    free(temp); /* Anyone using temp? */
}
```

Figure 9 Speculative Access to Freed Objects Possible

Assume multiple threads try to delete a node with a given key. Since the transaction is optimistic, multiple threads could get a pointer to the same node (same value of temp). One of the transactions is going to commit, while the remaining will ultimately abort and presumable retry. The committing transaction unlinks the node, but unfortunately does not know when it is safe to free the node since some active transaction may still have a pointer to it. The same problem occurs if we have a non-blocking algorithm because multiple threads may try to perform a task, and some threads may obtain a stale pointer. Previous researchers have proposed solutions based on the repeat offender problem [10], or hazard pointers [16], or reference counting. Essentially, all of these approaches do additional work at every read and write (including atomic operations) to record that an object is being accessed; reference counting associates an access count and changes that at accesses, while the hazard pointers and the repeat offender add the accessed pointers to a global root set that is traversed before freeing any structure.

Our solution works as follows:

After an object is freed, the memory allocator needs to wait till the object becomes truly unreachable. Only transactions that were started before the object is freed can have pointers into it; transactions started after the object is freed can not contain any pointers into it. Therefore, the memory allocator needs to detect when all transactions that are outstanding at the time an object is freed have ended.

For expository purposes assume the memory allocator uses a daemon thread for this. The daemon thread maintains two variables: *currentFreeEpoch* and *safeToFreeEpoch*. Every transaction reads the variable *currentFreeEpoch* at the beginning and stores it in the TLS in a thread local variable called *transactionEpoch*. The daemon thread wakes up at intervals, traverses the TLS of the threads, and finds the earliest *transactionEpoch*, E, of all the currently active transactions. It then sets the *safeToFreeEpoch* to be E-1, and increments the value of *currentFreeEpoch*.

When an object is freed (inside or outside a transaction) it is added to the head of a thread local FIFO speculative free list, and tagged with the current value of *currentFreeEpoch*. If the speculative list grows beyond a threshold, then the thread examines the objects at the tail of the list, and frees (for real) objects whose tagged epoch is earlier than the current value of *safeToFreeEpoch*. This is safe since, by design, all currently active transactions started after the *safeToFreeEpoch*, while the objects were freed before the *safeToFreeEpoch*.

We still need to guard against long running transactions, or transactions containing an infinite loop. If some thread runs out of memory, or finds that its speculative free list has grown beyond a high water mark, it can validate all active transactions. If a transaction has a pointer to speculatively freed memory, then the transaction is aborted and cleaned up. If a transaction does not have a stale pointer to freed data, then it can not acquire such a pointer in future. Thus, our algorithm prevents a space leak, even though an application may end up consuming somewhat more memory.

The following figures show the pseudo code for the mechanisms:

The daemon thread:

```
while (1) {
  For all active transactions {
    Find the earliest epoch at which
      any transaction started, (call it E).
    safeToFreeEpoch = E – 1
  }
  currentFreeEpoch++;
  park();
}
```

Figure 10 Daemon Thread Mechanism

The freeing mechanism:

```
Tag block being freed with the current value
of currentFreeEpoch

Add freed block to head of FIFO speculative
free list

FreeLabel:
while (there are entries in the
         speculative free list) {
  if (epoch of block < safeToFreeEpoch)
     Free block;
  else break;
}
If length of speculative free list
      > highThreshold {
   Validate currently active transactions;
   If (transaction does not validate)
       abort transaction;
   Goto FreeLabel;
}
```

Figure 11 The Freeing Mechanism

The current implementation does not use a daemon thread; instead it uses a data structure that consists of fixed sized buffers that are linked together. When a buffer fills the mutator thread performs the actions safeToFreeEpoch and currentFreeEpoch updates and performs the appropriate free actions.

5. ANALYSIS / RESULTS

We now analyze our system with a few goals in mind. First we validate our base malloc approach by comparing it to Hoard, considered a best of class allocator. Second we show the advantages of our approach to avoiding the use of CAS in the common case. Third we show the overhead of making the algorithms transaction aware. Finally we show the overhead of making the algorithms transactional aware.

The measurements presented here were gathered on a 16-processor IBM x445 SMP system with Xeon© MP 2.2 GHz processors running Windows Server 2003. The SMP system is arranged in clusters of 4 processors, with processors within each cluster sharing a 64MB L4 cache. Each processor has private L1 (8KB), L2 (512KB), and L3 (2MB) cache. Each processor runs a single hardware thread.

The Hoard system used for comparison consisted of binaries found at www.hoard.org that print out the following *“This software uses the Hoard scalable memory allocator (version 3.3.0, libhoard). Copyright (C) 2005 Emery Berger, The University of Texas at Austin, and the University of Massachusetts Amherst. For more information, see http://www.hoard.org”.*

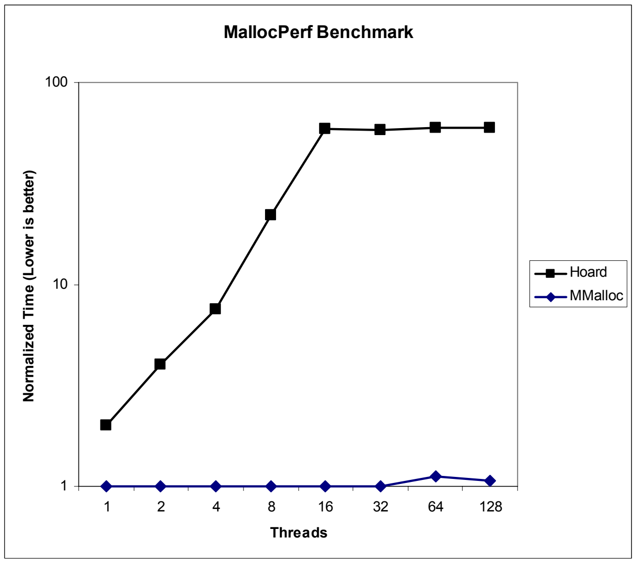

Figure 12 MallocPerf

Figure 12 shows results from the MallocPerf benchmark found in the Hoard distribution. MallocPerf is what can be referred to as an embarrassingly parallel benchmark. Using a random number generator the benchmark mallocs and frees thread local objects. There is no sharing of malloced objects between threads and very little work other than randomly selecting objects for mallocing and freeing the objects. Each thread does the same amount of work so the 4 thread run does 4 times as much total work as the 1 thread run.

The X axis of the graph indicates the number of software threads involved. The Y axis indicates time normalized based. Time (*T*) is the wall clock time; Base (*B*) is the time it takes one McRT-Malloc thread to complete the task; *SWT* is the number of software threads; *HWT* is the number of hardware threads, in our case 16. To calculate the normalized time (*NT)* we use the following formula.

$$NT = (T/B) / Max(1, SWT/HWT)$$

Linear scaling would produce a flat line at 1. A flat tail from when SWT equals HWT on indicates that the software is resilient in the presence of thread delay.

As we can see from Figure 12, the avoidance of atomic instructions by McRT-Malloc leads to linear scaling for the embarrassingly parallel program MallocPerf. It also shows that McRT-Malloc compares favorable with the best of class Hoard allocator. In a recent private communication with Emery Burger he indicated that Hoard had independently adopted a private free list approach.

We have also measured the cost of this mechanism using the Machias benchmark. The Machias benchmark rapidly allocates and frees objects. The object is written to immediately after each malloc and read from immediately before each free. This is meant to mimic a very simple consumer producer pattern. Knobs on the Machias benchmark allows us to adjust what percentage of the malloc/frees pairs are local to a thread and what percentage is malloced in one thread but freed in a randomly selected thread. If all frees are done by the allocating thread then we have 0% sharing. If the thread that does the free is randomly chosen then we have 100% sharing. If we have only one software thread performing the allocation and freeing then the amount of sharing is meaningless since only one thread is active.

We have plotted three graphs, Figure 13 showing results from Hoard, Figure 14 McRT-Malloc showing result from the base line McRT-Malloc and Figure 15 showing the results from McRT-Malloc with the STM enhancements. Each line represents a different degree of sharing and all the lines are relative to the base line single thread results.

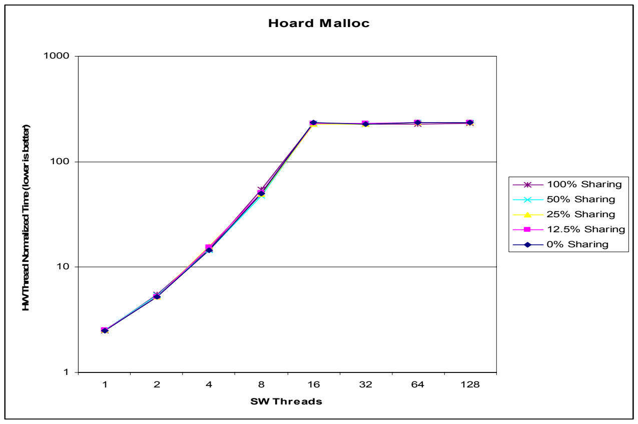

Figure 13 Hoard Malloc

The first thing to notice is that the cost of mallocing and freeing objects in Hoard is independent of sharing resulting in all five of the lines lying on top of each other.

All the graphs have the same log scale on both axis, both the McRT-Malloc and the McRT-STM-Malloc graphs fall below the Hoard lines. This is an indication that the performance of both McRT-Malloc as well as McRT-STM-Malloc is more than acceptable, in fact it is quit good when compared with Hoard.

In Figure 14 the bottom line shows that when there is no sharing of objects between threads we see close to linear scaling. As the sharing increases the number of CAS instructions required increases and we have several lines separated roughly by the increase in overhead. When the number of software threads exceeds the number of hardware threads the flattening of the line indicates that we are getting reasonable scaling. We note that the Hoard line in Figure 13 shows worse scaling as we increase the number of software threads until the number of software threads equals the number of hardware threads. Further increases in software threads show linear scaling.

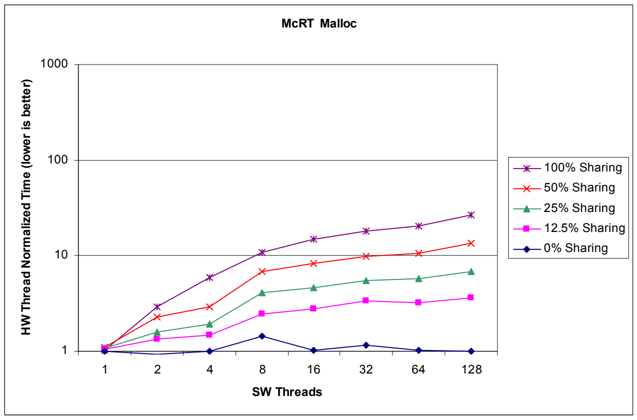

Figure 14 McRT-Malloc

Figure 15 the McRT-STM-Malloc graph below shows the cost of applying the transaction aware algorithms from Section 4 under different sharing loads. Again performance improves as sharing decreases.

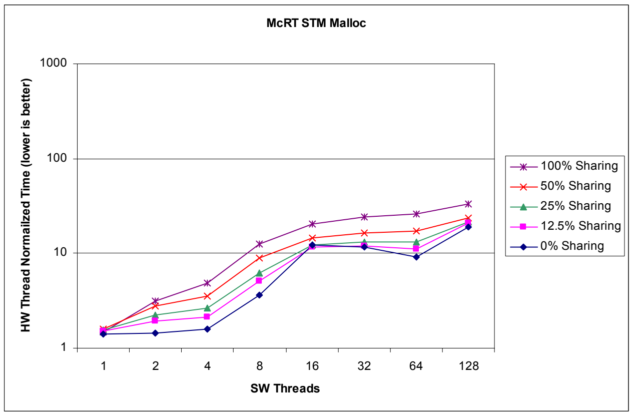

Figure 15 McRT STM Malloc

Figure 16, the Average STM Overhead Graph below summarizes and compares the McRT-Malloc and the STM graphs above. The top line indicates the operation of our transaction aware algorithm described in Section 4. This algorithm tracks mallocs and frees and if they are balanced and in the same transaction then the free is performed "for real" and the space can be reused before the end of the transaction. It also delays the actual freeing of the objects until it is safe in our STM. The bottom line is the normal McRT-Malloc code.

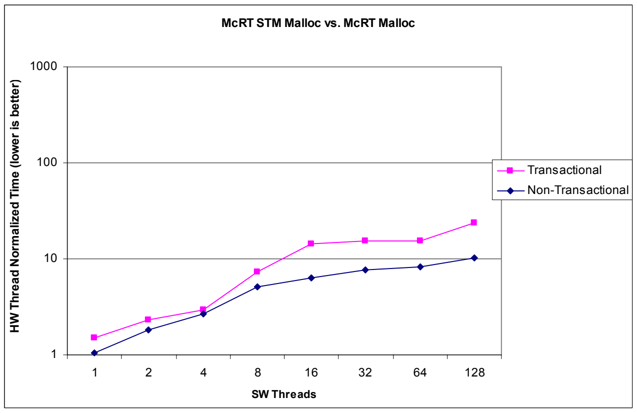

Figure 16 Average STM Overhead

As we can see the cost of ensuring safety by delaying frees has acceptable overhead. These graphs represent an upper bound since the Machias benchmark is focused completely on the mallocs and frees while a typical application will have a mix of application computations and malloc and frees.

Of course there is also a memory footprint affect of using McRT-STM-Malloc. Figure 17 compares running Machias starting with a single thread and continuing through 128 threads. This is then repeated for the different sharing levels. The lower line is the memory used by McRT-Malloc. The memory used quickly goes up at the number of threads are added since each thread does the same amount of work. Since the current implementation of McRT-Malloc does not return memory to the OS the line does not decrease. The increase in memory usage is a result of delaying the actual release of the data. If memory is released in a timely fashion then the amount of overhead is relatively small but as the number of threads increases objects are retained longer waiting for every thread to enter an epoch beyond the epoch when the object was freed.

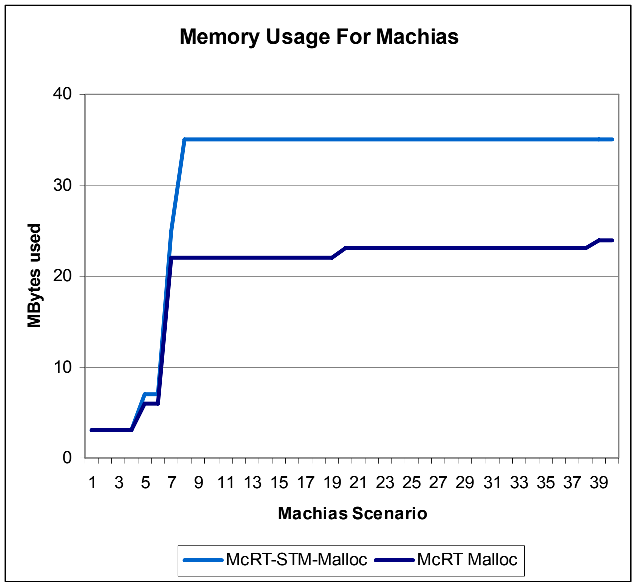

Figure 17 Memory Usage Overhead

6. CONCLUSIONS

This paper presents a non-blocking transaction aware memory allocator. Our memory allocator performs better than the Hoard memory allocator, a best-in-class memory allocator for multi-threaded applications. The memory allocator is also transaction aware and correctly handles memory allocation activity inside transactions, including nested transactions with partial rollback. We show that the memory allocator pays little overhead for handling transactions correctly.

This paper presents a number of novel algorithms. We show a high performance allocation algorithm that avoids atomic operations in most cases. We show how to detect balanced transactional allocations to avoid space blowup. We also show how to safely reclaim memory in the presence of optimistic transactions. Finally, we show how commit and abort hooks may be used to impart transactional semantics to system services such as memory management.

7. ACKNOWLEDGEMENTS

The authors wish to thank the other members of our group, Tatiana Shpeisman, Vijay S. Menon, Brian R. Murphy, Leaf Petersen, and Anwar Ghuloum for their lively discussions and support. We would also like to thank Jesse Fang for creating the exciting environment where such work is encouraged and supported. We would also like to thank our reviewers for their time and valuable feedback.

8. REFERENCES

[1] Allan, E., Chase, D., Luchango, V., Maessen, J., Ryu, S., Steele Jr., G., Tobin-Hochstadt, S. The Fortress language specification, version 0.618. Sun Microsystems Technical Report, April 2005.

[2] Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R. 2000. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth international Conference on Architectural Support For Programming Languages and Operating Systems* (Cambridge, Massachusetts, United States). ASPLOS-IX. ACM Press, New York, NY, 117-128. DOI= http://doi.acm.org/10.1145/378993.379232

[3] Charles, P., Donawa, C., Ebcioglu, K., Grothoff, C., Kielstra, A., von Praun, C., Saraswat, V., Sarkar, V. X10: An object oriented approach to non-uniform cluster computing, *OOPSLA,* October 2005.

[4] Cray Inc. The Chapel language specification, version 0.4. Technical Report, Cray Inc. Feb 2005.

[5] Ennals, R. Cache sensitive software transactional memory. *Technical Report.*

[6] Gray, J. and Reuter A. Transaction processing: concepts and techniques.

[7] M. Greenwald. Non-blocking Synchronization and System Design. Ph.D. Thesis July 1999. Also available as *Stanford University Technical Report* STAN-CS-TR-99-1624

[8] Harris, T.L. and Fraser, K. Language support for lightweight transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications* (Anaheim, California, USA, October 26 - 30, 2003). OOPSLA '03. ACM Press, New

York, NY, 388-402. DOI= http://doi.acm.org/10.1145/949305.949340

[9] Harris, T.L., Marlow, S., Peyton Jones, S., Herlihy, M. Composable memory transactions. *Proceedings of the Principles and Practice of Parallel Programming, PPoPP,* June 2005.

[10] Herlihy, M., Luchangco, V., and Moir, M. 2002. The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures. In *Proceedings of the 16th international Conference on Distributed Computing* (October 28 - 30, 2002). D. Malkhi, Ed. Lecture Notes In Computer Science, vol. 2508. Springer-Verlag, London, 339-353.

[11] Michael, M. M. 2004. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation* (Washington DC, USA, June 09 - 11, 2004). PLDI '04. ACM Press, New York, NY, 35-46.

[12] Johnstone, M. S. and Wilson, P. R. 1998. The memory fragmentation problem: solved?. In *Proceedings of the 1st international Symposium on Memory Management* (Vancouver, British Columbia, Canada, October 17 - 19, 1998). ISMM '98. ACM Press, New York, NY, 26-36. DOI= http://doi.acm.org/10.1145/286860.286864

[13] Rajwar, R., Herlihy, M., and Lai, K. Virtualizing transactional memory. *Proceedings of the International Symposium on Computer Architecture*, June 2005.

[14] Rattner, J. Multicore to the masses. Parallel Architectures and Compilation Techniques, Keynote. September 2005.

[15] Marathe, V. J., Scherer, W. N., and Scott, M. L. 2004. Design tradeoffs in modern software transactional memory systems. In *Proceedings of the 7th Workshop on Workshop on Languages, Compilers, and Run-Time Support For Scalable Systems* (Houston, Texas, October 22 - 23, 2004). LCR '04, vol. 81. ACM Press, New York, NY, 1-7. DOI= http://doi.acm.org/10.1145/1066650.1066660

[16] Michael, M. M. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, page 21-30, July 2002.

[17] Saha, B., Adl-Tabatabai, A., Hudson, R., Minh, C., Hertzberg, B., A High Performance Software Transactional Memory System For A Multi-Core Runtime. In *Proceedings of the Principles and Practice of Parallel Programming, PPoPP,* New York, NY, March 2006.