A Fast Parallel Dynamic Memory Allocator with the use Intel TSX

Patrick Hughes
B.A.I. Engineering
Final Year Project April 2014
Supervisor: Dr. Jeremy Jones

School of Computer Science and Statistics
O'Reilly Institute, Trinity College, Dublin 2, Ireland
DECLARATION

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university

______________________________________________  _______________________
Name            Date
A Fast Parallel Dynamic Memory Allocator with the use Intel® TSX

by

Patrick Hughes

Submitted to the School of Computer Science and Statistics in partial fulfillment of the requirements for the degree of

Baccalaureus in Arte Ingeniaria

at the

TCD

June 2014

© The University of Dublin 2014. All rights reserved.

Author ..............................................................

School of Computer Science and Statistics

April 2, 2014
A Fast Parallel Dynamic Memory Allocator with the use of Intel® TSX

by

Patrick Hughes

Submitted to the School of Computer Science and Statistics on April 2, 2014, in partial fulfillment of the requirements for the degree of Baccalaureus in Arte Ingeniaria

Abstract

In this thesis, I designed and implemented an algorithm for a fast dynamic memory allocator that in many scenarios out performs an equivalent and respected allocator, ptmalloc used by the GNU C library, on multiple threads. Based on the literature, our algorithm eliminates a highly inefficient characteristic inherent in ptmalloc when allocating small chunk of memory. By eliminating this characteristic we significantly improve the direct memory overhead of our allocator for scenarios such as creating and deleting small data structures, including tiny nodes of a binary search trees or linked lists. Furthermore, we analyse the Intel® Transactional Synchronisation Extensions (Intel® TSX) that were released in June of 2013 on a new family of Intel processors codenamed “Haswell.” These transactional instructions enable us to make further improvements to the speed of our memory allocator by providing an optimistic approach for executing critical sections.

Thesis Supervisor: Jeremy Jones
Title: Associate Professor
Acknowledgments

I would like to express the deepest appreciation to my supervisor, Dr. Jeremy Jones who maintained a personal interest and enthusiasm for my project throughout the year. One of my favourite meetings was when we spent three and a half hours brainstorming the low level structure of the algorithm. Without his continuous enthusiasm, advice and decisiveness this project would not have been possible.
# Contents

1 Introduction ........................................ 9
   1.1 Dynamic Memory Allocators .......................... 10
   1.1.1 The Fragmentation Problem ....................... 11
   1.2 Related Work .................................. 12
   1.2.1 Doug Lea Malloc ............................... 13
   1.2.2 ptmalloc ..................................... 15
   1.2.3 Hoard ....................................... 16

2 Intel® TSX ........................................ 19
   2.1 Background .................................... 19
   2.1.1 What is a transaction? .......................... 20
   2.1.2 Semantics .................................. 20
   2.1.3 Memory Conflicts ............................... 21
   2.1.4 How are conflicts detected? ....................... 22
   2.1.5 Locks and Transactions ......................... 22
   2.2 Testing the Limits ............................... 23
   2.2.1 Test, Results and Conclusions .................. 25

3 The Algorithm - tsxmalloc .......................... 29
   3.1 Super Block .................................... 29
   3.1.1 Doubly Linked-List ........................... 34
   3.2 Locking ....................................... 36
   3.2.1 Operations on Doubly Linked-List ............... 36
3.2.2 Conflicts ........................................... 36
3.3 Transactional approach ................................ 37

4 Proof of Concept ........................................ 41
   4.1 Random Benchmark .................................... 42
      4.1.1 Methodology ....................................... 42
      4.1.2 Results ............................................. 42
      4.1.3 Discussion of Results ............................. 43
   4.2 Serial Benchmark ..................................... 44
      4.2.1 Per-Heap Extension ................................. 44
      4.2.2 Methodology ....................................... 45
      4.2.3 Results ............................................. 45
      4.2.4 Discussion of Results ............................. 46
   4.3 Further Work .......................................... 47

5 Appendix of Code ........................................ 51

6 CD with the contents of this project
Chapter 1

Introduction

Dynamic memory allocation is regarded as a well-established and fundamental aspect of computer science that has been developing since roughly 1960 [3]. There are two aspects of a memory allocator that will define its value; the time taken in allocating and freeing memory, and the efficiency of memory usage. All of the well-known memory allocators today trade-off between these two aspects including dlmalloc [4], ptmlloc [5] and hoard [6]. In this chapter, I introduce you to the aforementioned allocators to identify both the pros and cons of their algorithms, but more importantly by understanding these well-known allocators you can appreciate the structure of the transactional memory allocator, tsxmalloc for short, designed and implemented in this thesis.

Transactional memory\(^1\) (TM) instructions were first proposed by Herlihy and Moss in 1993 as a multiprocessing architecture intended to make lock-free synchronisations as simple to use as conventional techniques based on mutual exclusion [7]. TM provides a simple alternative to handcrafted finely grained locking techniques\(^2\), a skill in its own right that can often introduce subtle bugs. In effect, it allows a processor to determine dynamically whether a thread needs to serialise through a critical section,

\(^{1}\)The primary goal of transactional memory is to make it easier to perform general atomic updates of multiple independent memory words, avoiding problems of locks (priority inversion, convoying, and deadlock). [7]

\(^{2}\)If we could implement fine-grained locking with the same simplicity of course grained, we would never think of building a transactional memory. [8]
and to perform the serialisation only when required [9]. Twenty years later in June of 2013 the first commercial implementation, Intel® Transactional Synchronisation Extensions (Intel® TSX) were included with some versions of the 4th Generation of Intel® Processors codenamed Haswell [9] [10]. It has been shown to provide a 1.31x average bandwidth improvement on network intensive applications when applied to a parallel user-level TCP/IP stack [11] and can improve the performance of database data structures such as index trees [12]. The limitations of a transaction are vague and we are told that they are implementation-specific [9]. In chapter two, we test the limits of these new extensions with various benchmarks to gauge the probability of a transaction succeeding or failing.

In chapter three, we develop and implement an algorithm for a dynamic memory allocator that in many scenarios performs faster, on multiple threads, than the allocator on the GNU C library. The algorithm uses a simple idea from Hoard that restricts large blocks of memory to specific size classes. We introduce a bit mask as a comparatively low book keeping overhead that also reduces and localises the number of read and write operations when freeing memory. Additionally, we integrate the Intel® TSX instructions, hence the name tsxmalloc, to improve the speed of the allocator allowing for a more finely grained lockless approach.

Chapter four concludes this thesis with an evaluation of the allocator, its performance and a discussion of further work that would improve the algorithm and the area of study.

1.1 Dynamic Memory Allocators

In C programming, memory is split into three categories; static, automatic and dynamic. Static memory is allocated during compile time and is located with the associated executable code while automatic memory is allocated from the stack during runtime. However, for both automatic and static memory the size of the memory that is allocated must be constant at compile time. If the size is only known during runtime then this memory must be allocated dynamically from the heap. In the standard C li-
library, dynamic memory is controlled by the functions `void *malloc(size_t size)`\(^3\) and `void free(ptr)`\(^4\). There are many variations of these functions as previously mentioned such as dlmalloc, ptmalloc and hoard that trade-off between speed and memory-usage. You might question; why do we need these functions `malloc` and `free` when we could just allocate memory directly from the operating system? The problem there, is that system calls for memory are relatively much slower by comparison to the `malloc` and `free` routines. To minimise these expensive system calls for memory, allocators request large blocks of memory in advance from the operating system and can then dish out smaller chunks of memory much faster with only a few memory reads and writes.

### 1.1.1 The Fragmentation Problem

By getting large chunks of memory from the operating system and then dishing out smaller chunks, we encounter the problem of fragmentation. This occurs when memory, freed back to the allocator, is in a location bounded by allocated memory.

![Figure 1-1: Fragmentation Problem](image)

In Figure 1-1 steps A to E demonstrate a possible sequence that would result in fragmented memory. The pail green block represents a large block of memory that

\(^3\)The malloc function allocates space for an object whose size is specified by size and whose value is indeterminate. The malloc function returns either a null pointer or a pointer to the allocated space [13]

\(^4\)The free function causes the space pointed to by ptr to be de-allocated that is, made available for further allocation. [13]
has been allocated from the operating system in advance. In steps A, B and C we allocate the brown, blue and green blocks respectively. In step D the blue block is freed and now you can see that the free memory available to the allocator, has become fragmented. The outcome, in step E, is that when the allocator receives a request for memory the size of the red block, even though there is enough free memory to satisfy the request it cannot be used. This is because the free memory is not contiguous or, in other words, it has become fragmented. Consequently, the allocator must make another expensive call to the operating system to extend its block of free memory before it can allocate the requested red block of memory.

Fragmentation is the unavoidable cause for excessive memory usage with dynamic memory allocators. This occurs, as the allocator does not have control over what blocks of memory are freed. Instead, this control is given to the caller of malloc. However, what the allocator does have control over is the location from which memory is allocated. When a request comes for a block of memory, fragmentation can be reduced by attempting to allocate the request from fragmented memory first. The challenge is to find a fragmented block of memory that best fits the request as quickly as possible. Minimising wastage (fragmentation) should be a primary goal of any allocator [3]. By going through the various algorithms below we get an understanding of how well know allocators minimise both fragmentation and increase speed.

1.2 Related Work

When trying to develop a new algorithm for a dynamic memory allocator we must take into consideration the fact that memory allocators are well established and have been evolving for over 50 years. Thus, it is important to look at other well-known allocators to see how they have optimised the memory usage vs. speed trade-off. Furthermore, we must be critical of other algorithms and aim to identify aspects that can be improved upon. In this section I go into the detail and give a critical analysis of three well known and highly respected memory allocators; dlmalloc, ptmalloc and hoard.
1.2.1 Doug Lea Malloc

Doug Lea began writing a memory allocator (dlmalloc) in 1987 that has been evolving ever since. An extended version, ptmalloc was designed for multiple threads and is part of the GNU C library. As with all the allocators I discuss in this thesis, they request large amounts of memory from the operating system in advance and then dish out smaller chunks as calls to `malloc` are made. Doug Lea maintains a wilderness chunk of free memory at the end of the large block. Memory is only allocated from the wilderness chunk if there are no fragmented blocks of free memory that fit a request. If the wilderness chunk cannot satisfy the request then another expensive system call is made for a large amount of memory. The dlmalloc is split into two core elements; boundary tags and bins of free blocks.

**Boundary Tags**

One of the issues an allocator must overcome based on the C Standard for the function `void free(ptr)`, is that `free` is only passed a pointer to the block of memory that is freed back to the allocator. The size of the block being freed is not given meaning that the algorithm for a memory allocator must have some method to identify the size of a block, given only a pointer to the start of that block. Doug Lea overcomes this problem with the use of boundary tags.

![Figure 1-2: Doug Lea Blocks](image)
In Figure 1-2, boundary tags take up 4 bytes of memory and are used to hold the size and status of every block of memory in the larger block. The status represents whether the block is free or allocated and naturally, the size represents the bytes of memory bounded by the two boundary tags.

With these boundary tags in place when a call to \texttt{free(ptr)} is made, the size of the block that is being freed can be read from the boundary tag located in the 4 bytes before \texttt{ptr}. Additionally by maintaining these boundary tags the neighbouring blocks can also be checked to see whether they are free or allocated. Neighbouring free blocks are coalesced into a larger free block, allowing future allocations to find the smallest block of contiguous free memory to best fit the request giving reduced fragmentation.

**Bins**

Bins are simply, lists of free blocks that are the same size. They allow us to overcome the challenge of finding a fragmented block of free memory that best fits a request quickly by minimizing the number of memory reads. When a free block from a bin coalesces with a block that has just been freed, it must be deleted from that bin and added to a larger bin as the size of the free block will have increased. To enable quick deletions from from a list, these lists of free blocks are maintained as a doubly linked list.

![Figure 1-3: Doug Lea Bins](image)
In Figure 1-3, the 24B bin points to the first free block that is 24B in size. Doug Lea, in fact, uses the space within free blocks to hold the next and previous pointers that maintain the doubly linked list. This means that when a request for a size of memory is made, the bins can be scanned quickly to find the first best fit for the requested size.

However, by maintaining doubly linked lists, it also means that on a 64-bit address space the minimum size of a free block is

\[(2 \times 8B \text{ pointers}) + (2 \times 4B \text{ Boundary tags}) = 24B\]

This means that the minimum size of an allocated block must also be 24B as when an allocated block is freed it must have the space available for two 8B pointer to add it to a doubly linked list. The consequence is a significant 200% memory overhead for 8B allocations.

In Figure 1-2 we see another consequence of this algorithm. The memory allocated can only be guaranteed to be aligned on a 4 byte boundary stemming from the use of 4B boundary tags. While, it’s not a requirement for `malloc` to provide memory aligned on a particular byte boundary, it is often requested to help developers prevent two notorious slow downs known as false sharing and cache line splits.

### 1.2.2 ptmalloc

The ptmalloc algorithm was developed by Wolfram Gloger and is an extended version of dlmalloc for use with multiple threads [5]. It is used on unix systems to fulfill the requirements of the functions `malloc` and `free` from the ISO C standard. Previously with dlmalloc we had just one large block of memory with one set of bin pointers shared between multiple threads. When used with multiple threads,
dlmalloc must be protected by a lock, which prevents it from scaling. Gloger overcomes this problem with the use of multiple sub-heaps that he calls arenas. Each arena is effectively its own Doug Lea allocator that can be protected by a lock. To reduce contention for locks, whenever a thread finds that an arena is locked it moves onto the next arena and try’s to obtain that lock. The last arena locked by a thread is remembered with thread specific data. If a thread finds that all the heaps are locked on a request then it simply creates a new arena and allocates from that one. This is shown in Figure 1-4 where thread 4 attempts to allocate memory from arenas A, B, and C, only to find that they are all locked and must then create a new arena D, from which to allocate.

It is important to recognise that although the thread specific data will help maintain a one-to-one relationship between each thread and its heap that the relationship may be temporary. Consider Figure 1-5 where initially both threads are allocating memory from HEAP 1 without contention. Later Thread 2 attempts to allocate memory and finds HEAP 1 is locked so it create a new heap. However from memory from Thread 2 still remains in HEAP 1. If this memory from HEAP 1 were subsequently freed by Thread 2 then Thread 1 would move onto HEAP 2 if it finds HEAP 1 locked. This shows how threads can share memory in multiple heap. While ptmalloc scales much better on multiple by reducing the sharing of the heap data and contention it also inherits both the positive and negative aspects of dlmalloc such as the poor alignment and 24B minimum size of free and allocated blocks.

1.2.3 Hoard

Hoard, is also a highly scalable memory allocator designed for use on multiple threads. While ptmalloc potentially has one heap for every thread to reduce lock contention, hoard maintains one heap per-processor by hashing thread ids to a particular heap along with a global heap that can be accessed by each thread.
Each heap consists of what they call super blocks, that are large blocks of memory a multiple of the system page size. Each super block is divided into smaller blocks of memory that are all the same size where the free blocks of memory in each super block are maintained in a LIFO linked-list for improved temporal locality, such that memory is always allocated from the last-in free block add to the list.

The heaps maintain partially full super blocks in two sets of doubly linked lists; nearly full ($> \frac{3}{4}$) and empty to $\frac{3}{4}$ full. Memory is always allocated from nearly full super blocks, if one exists to reduce fragmentation. Additionally by allocating from the LRU\(^5\) nearly full super block there is an improvement in temporal locality. To maintain this LRU list, when memory is freed from a nearly full super block it is bumped to the top of the list, such that a call to `malloc` will be allocated memory from the top of the list, i.e. the least recently used super block. By maintaining blocks in LIFO order and maintaining super blocks in LRU order Hoard increases temporal locality, such that super blocks and their corresponding blocks that are accessed are likely to be accessed again soon.

---

\(^5\)Least recently used
Chapter 2

Intel® TSX

2.1 Background

One of the requirements of concurrent programming is to maintain synchronization of shared memory between threads by ensuring that no conflicting data accesses occur. The simplest way of implementing this is with the use of a single lock that protects such critical sections. Consider a linked list, where many conflicting memory accesses may occur such as the concurrent deletion of two neighbouring nodes. The simplest approach to avoid such conflicting accesses, is with the use of a single lock, whereby each thread would lock the entire linked-list during any access to the list. However, this approach will block other threads that try to access the list at the same time and consequently reduce concurrency. A common approach is to provide a lock for every node in the linked list which, for example, would allow for the deletion of a node by locking access to that node and its neighbouring nodes, as opposed to locking the entire linked list. This is an example of taking a coarse grained lock that protects an entire list and making it more finely grained by protecting small sections of that list. Fine-grained locking reduces contention however; it can also introduce subtle bugs due to its complexity [9] [11] [12] and comes with a significant memory overhead.
2.1.1 What is a transaction?

While locks eliminate conflicting memory accesses by serializing critical sections, transactions can execute critical sections in parallel with the ability to detect conflicts dynamically. This is an optimistic approach to executing a critical section and works best when conflicts are unlikely to occur. In the event of a conflict, the transaction will be aborted and the thread executing the transaction will return to the state it was in before the transaction started [9]. At this point a programmer can decide to retry the transaction or take an alternative thread-safe path. A transaction is as simple to implement as a coarsely grained lock but can detect conflicts at the granularity of a cache line, similar to a fine-grained lock. The transaction itself has the advantage of being lock free, meaning that progress is guaranteed regardless of scheduling policies and whether a thread is delayed or even killed [1]. However, with Intel® TSX as transactions are not guaranteed to commit [9], they must always have an alternative thread safe path which, may not be lockless.

2.1.2 Semantics

We describe the semantics of a transaction in the C programming language. In the example (Figure 2-4) _xbegin() is called to specify the start of a transactional execution. If the transaction begins successfully then the condition in line 2 will be true and the critical section will execute transactionally. Once the critical section is complete _xend() is called to try and commit the transaction. If successful, the transaction will appear to other processors as atomic. If the transaction aborts at any time, execution will return to line 2 where the condition will now

```c
int status = _xbegin();
if(status == _XBEGIN_STARTED){
    //critical section
    _xend();
} else{
    //fallback path
}
```

Figure 2-1: C - Transaction
be false, hence the fallback path will be taken.

2.1.3 Memory Conflicts

Memory conflicts are the primary cause for a transaction to abort as they signify that the transaction has not been executed atomically. For a transaction to retain its atomicity, it maintains a **read-set** and a **write-set** at the granularity of a cache line. As can be seen in Figure 2-2 a conflict occurs if the read-set of a transaction is written from outside the transaction or if the write-set of a transaction is either written or read from outside the transaction.

---

**Figure 2-2: Conflicting Accesses**

---

\(^1\)When an operation or set of operations appear to other threads as if they executed instantaneously.
In Figure 2-2 the address space is represented with 32B increments. Note, the conflict that occurs between threads 9 and 10. Even though the memory accessed is unrelated the conflict occurs as Intel® TSX detects data conflicts at the granularity of a cache line and the memory accessed is on the same cache line.

2.1.4 How are conflicts detected?

When we say that a transaction detects conflicts at the granularity of a cache line, this is because memory conflicts are detected by extending cache coherence protocols. Any cache coherence protocol capable of detecting accessibility\(^2\) conflicts can also detect transactional conflicts at no extra cost [7]. They ensure that exclusive cache lines can both be read and written, whereas shared cache lines only allow read operations.

If we consider the MESI\(^3\) protocol, when processor A needs to write to a location, it must first gain exclusive access to that location by invalidating any copy present in another cache, be it modified, exclusive or shared. If another processor B then reads from that location, processor A’s exclusive access is revoked as cache A observes a bus read from a modified location. It is easy to see how accessibility conflicts can indicate transactional conflicts. The idea proposed by Herlihy and Moss is that a cache would hold all the transactional writes provisionally and only allow them to propagate to other processors on a successful commit of the transaction. Otherwise, those cache lines would be invalidated [7].

2.1.5 Locks and Transactions

Intel® TSX specifies that transactions are not guaranteed to succeed and must always have an thread-safe fall back path [9] [10]. The fall back path may be to serialize the critical section that was aborted with the use of a lock. Any threads executing the same critical section transactionally must also be aborted when this lock is acquired. This can be done by including the lock in the read-set of a transaction.

---

\(^2\)Memory locations can have: (1) No cache access as in memory (2) Non-exclusive access, i.e. shared by other processors. 3) Exclusive access by one processor.

\(^3\)Modified, Exclusive, Shared and Invalid
Consider the scenario in figure 2-3 where two threads begin a transaction by reading a shared lock to check that it hasn’t been acquired. Neither of the threads will abort as the lock is only read, which is not a conflicting memory access. Both threads continue to execute their critical sections in parallel, however for some reason the transaction in Thread 1 is aborted and must now take an alternative path by serialising the critical section. To do this Thread 1 acquires the lock by writing to it atomically. As the lock written to by Thread 1 remains in the read-set of Thread 2’s transaction, a memory conflict occurs and that transaction will also be aborted ensuring that the critical section is serialised.

In short, by reading a lock at the beginning of a transaction we ensure that the corresponding critical section will be serialised if the lock is acquired during that transaction.

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>_xbegin()</td>
</tr>
<tr>
<td>2</td>
<td>if(lock==1){_xabort()}</td>
</tr>
<tr>
<td>3</td>
<td>//critical section</td>
</tr>
<tr>
<td>4</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>abort!!</td>
</tr>
<tr>
<td>6</td>
<td>acquire lock (success)</td>
</tr>
<tr>
<td>7</td>
<td>//critical section</td>
</tr>
<tr>
<td>8</td>
<td>acquire lock (success)</td>
</tr>
<tr>
<td>9</td>
<td>//critical section</td>
</tr>
<tr>
<td>10</td>
<td>release lock</td>
</tr>
<tr>
<td>11</td>
<td>release lock</td>
</tr>
</tbody>
</table>

Figure 2-3: RTM Lock Elision

2.2 Testing the Limits

The primary cause for a transaction to abort is due to conflicting memory accesses, however “aborts may also occur due to limited transactional resources, for example the
amount of data accessed in a region may exceed and implementation specific capacity.”

In this section, we examine the success rate of a transaction when there are no conflicting memory accesses given different sized reads, writes and duration.

**Environment**

For all the benchmarks performed in this section we use:

- An Intel Core® i7-4770 processor with a clock speed of 3.4Ghz and 4 CPUs (8 when hyper-threaded).
- Windows 7 Enterprise.
- Microsoft Visual Studio 2012 Ultimate both in release mode and 64-bit mode.

**Success or Failure**

The code in Figure 2-4 shows how to check if a transaction has succeeded or failed.

The fallback path at line 7 which, usually holds a serialised version of the transaction, instead just returns 0 to indicate that the transaction has failed. If we get to line 9 we know that the transaction was committed as the fallback path returns from `foo()` and does not continue to the success path hence, when `foo()` returns 1 we know the transaction committed successfully.

```c
bool foo(){
  int status = _xbegin();
  if(status == _XBEGIN_STARTED){
      //test
      _xend();
  } else{
    return 0;
  }
  return 1;
}
```

Figure 2-4: Commit or Abort
2.2.1 Test, Results and Conclusions

Each test is performed on a single thread to ensure that there are no conflicting memory accesses. To find probability of a transaction succeeding we repeat a transaction 1000 times for a given test. For example, when finding the probability of a transaction succeeding given that the transaction writes $X$ Kilobytes of memory, as described below in the Memory Writes sub-section, we repeat a transaction 1000 times for each 0.5KB increment of $X$ where $1KB \leq X \leq 64KB$.

**Duration**

Transactions will abort due to some system events. Figure 2-5 examines the probability of a transaction committing on the y-axis given the duration of the transaction on the x-axis. By running a transaction that does nothing other than waste time we are testing the effect that system events have on transactions committing.

![Figure 2-5: x-time](image)

Each transaction repeatedly writes to a single location to waste clock cycles. To increasing the duration of a transaction, we increase the number of times the write
is repeated. The duration of a successful transaction is recorded with performance
counters for improved accuracy. From the results, we can see that for transactions of
extended duration the probability of a system event causing an abort is quite high.

Memory Writes

Figure 2-6 examines the success rate of a transaction on the y-axis given that the
transaction writes X kilobytes of memory. As expected a transaction commits nearly
100% of the time when it only writes small amounts of memory up to 10KB and even
has a reasonable commit rate all the way up to 18KB. For transactions that write
over 18KB the success rate is unacceptable dropping down to 80% with much greater
variance. Notice that transactions have a 0% success rate after 32KB. This shows
that transactional writes are completely dependent on the L1 cache, which is 32KB
in size.
Memory Reads

Figure 2-7 examines the success rate of a transaction on the y-axis given that the transaction reads X kilobytes of memory. When transactions read more than 32KB we observe a sudden drop in the probability of the transaction committing. While this again shows that there is a direct relationship between transactions and the 32KB L1 cache, it is clear that Intel® TSX must be supported by more that just the L1 cache as transactions that read 400KB can still succeed. On the Intel Core® i7-4770 there is an L2 cache of 0.5MB local to each core. This suggests that for transactional reads Intel® TSX is supported by L2 caches that are local to cores.

![Figure 2-7: x-read](image)

In summation, with the aforementioned test conditions, a transaction may read up to 32KB of non conflicting memory or write up to 18KB of non conflicting memory with an acceptable probability that the transaction will succeed. Additionally that decreases rapidly for transactions of extended durations.
Chapter 3

The Algorithm - tsxmalloc

In this chapter, we take a bottom up approach to explaining the algorithm that was implemented in this thesis.

3.1 Super Block

Memory allocators request large amounts of memory in advance from the operating system and then dish out smaller chunks as request for memory are made. [4] [5] [6] The tsxmalloc starts with the concept of a super block, which is a large block memory restricted to a size class, meaning that it can only dish out blocks of memory that are of a specific size class. For example, a super block with a size class of 8B can only allocate 8B blocks of memory. Hence, a request for 16B of memory cannot be satisfied with two 8B blocks and similarly a request for 8B of memory cannot come from a super block with a size class of 16B.

For clarity, I explain the structure of the super block for a size class of 8B. Each superblock is split into 512 blocks of its size class. The size of a super block is given by:

\[
\text{sizeof}(\text{super block}) = 512 \times \text{size class}
\]

Hence a super block with an 8B size class is:

\[
\text{sizeof}(\text{super block}) = 512 \times 8 = 4KB
\]
We maintain a header for each super block that is 104B in size containing bookkeeping information to describe the memory available with some additional pointers. The status of each 8B block of memory, whether it is free or allocated, must be maintained. We introduce a bit mask for this purpose. The status for each of the 512 blocks in the super block is represented by a single bit in the bit mask, which is 64B (512 bits) in size, i.e. the size of a cache line. If the bit is set, then the corresponding block is allocated and if the bit is cleared then the corresponding block is free and can be allocated on request.

![Super Block Structure](image)

Figure 3-1: Super Block Structure

With the bit mask in place, if a call comes to allocate a block from a super block we only need to:
- find the first bit in the bit mask that is cleared (i.e. free)
- flip that bit to indicate that the corresponding block is now allocated
- calculate the pointer to the allocated block using the index of the bit that was flipped and the pointer to the start of the super block.
- return the pointer of the allocated memory to the calling function

Finding the first set bit is relatively fast and straightforward with BSR\(^1\) from the Intel instruction set enabling you to scan 64 bits at a time with just a single

\(^{1}\)Searches the source operand for the most significant set bit. [14]
instruction. We also maintain the bit mask in an address space that is on a 64-byte boundary to ensure that scanning the bit mask only requires reading a single cache line from memory.

While allocating memory from a super block is quite simple, if a call to \texttt{free(ptr)} is made, we are only given a pointer to the start of the block that is being freed without any size information about that block. To free the block we must flip the bit corresponding to that block in the bit mask, contained in header at the start of the super block. The question that remains; how do you find the start of the super block given only a pointer to a location within the super block? We accomplish this by aligning each super blocks on a 4K boundary.

![Figure 3-2: Freeing from Super Block](image)

Figure 3-2 shows a superblock aligned on a 4K boundary with a 16-bit address space for clarity. Note that all the pointers within the superblock lie between 0xA000 and 0xB000 and that the 12 least significant bits of these pointers are cleared. Thus, when a call to \texttt{free(ptr)} is made, the pointer to the corresponding header can be calculated by clearing the least significant 12 bits of the \texttt{ptr} passed to \texttt{free}.
pointer to head = ptr & 0xF000 // Clear 12 LSBs
block offset = ptr - (pointer to head)
blob size = head->blob size // get blob size in header
block index = offset/blob size
flipbit(index) // flip the corresponding bit to free the block

Figure 3-3: freeing pseudo code

The above code in Figure 3-3 is for freeing a block of memory when `free(ptr)` is called where `ptr = 0xAFD8`, as in Figure 3-2. Initially, we find the pointer to the head (0xA000) of the super block by clearing the least significant 12 bits of the `ptr` passed to `free`. In line 2 we calculate the offset in bytes between the block being freed and the start of the super block. We then read the block size class from the header to calculate the index of the block being freed at line 4. Finally, the corresponding bit to free the block is flipped to indicate that it is free.

There is a distinct advantage that the structure of this super block brings by comparison to dlmalloc and ptmalloc when allocating small amounts of memory. As previously mentioned, both dlmalloc and ptmalloc have a direct memory overhead of 200% when allocating 8B blocks of memory. However, with tsxmalloc, for 8B memory allocations the direct memory overhead comes from the header of the super block and is just 2.54%.

While this structure has a significant advantage for small memory allocations, there remains a disadvantage for large memory allocations. The previous example is for a size class of 8B resulting in a 4KB super block. However, if the size class of the super block were 32B, as in shown in Figure 3-4 then the size of the corresponding super block would be 16KB ($512 \times 32$). When a call to free memory from location 0xC180 is made, then clearing the least significant 12 bits would bring you to the previous 4K boundary at 0xC000 as opposed to the header of the super block at 0xA000. To allow for super blocks of different size classes we place a pointer on every 4K boundary pointing to the start of the super block. Each pointer takes up an
entire block in the super block. Consequently, super blocks with a large size class have greater direct memory overhead, the worst case being 48% in tsxmalloc.

To reduce this direct memory overhead we could increase the size of a super block to contain a multiple of 512 blocks. For example, a super block could be aligned on a 16K boundary and contain 2048 blocks of a size class instead of 512. With this change to tsxmalloc there would only be a 12% direct memory overhead in the worst case. This would come at the expense of reading up to three extra cache lines every time a request to \texttt{malloc} is made.

The need for a pointer on every 4K boundary is a result of variable sized super blocks. If all super blocks were of a specific size, for example 32KB and aligned on a 32K boundary then the bit mask to represent all the blocks would be \((32\text{KB} \div \text{size class})\) bits. A bit mask of 512B (8 cache lines) would represent the smallest size class of 8B, while super blocks with a size classes of 64B would have a bit mask 64B in size.

The basic high-level idea of our super block is similar to Hoard, such that each super block is divided into equal smaller blocks of memory. However, Hoard maintains a linked-list of free blocks in the super block requiring read and write operations through-out the super block. The list is maintained in a LIFO order to improve upon temporal locality. As all the read and write operations of our super block structure occur within the first two cache lines of the super block for both calls to \texttt{free} and \texttt{malloc}, there is much greater locality of reference within each super block by comparison to Hoard’s super block. We can further state that, as all references to

![Figure 3-4: Freeing from Super Block](image)
a super block are spatially local, temporal locality for accesses to the super block is maximized by comparison to Hoard.

### 3.1.1 Doubly Linked-List

In the header of the super block are two pointers for maintaining a doubly linked list of partially full super blocks that are of the same size class. The lists are maintained in order with the fullest super block at that start of the list and the emptiest at the end, as seen in Figure 3-5. Full super blocks are not on the list, as they have no memory to allocate. The credits of a super block represent the number of free blocks that are available in the super block to allocate. A characteristic of the functions `malloc` and `free` is that we can decide where memory is allocated from but we cannot decide what memory is freed. To reduce fragmentation, memory is always allocated from the super block that is closest to being full, at the top of the list. That way, full super blocks are kept full, while over time the other super blocks become emptier. If a super block eventually becomes completely empty, it is dropped of the end of the linked list and returned to the operating system. In the event that memory is requested and there are no super blocks for that size class then a super block is created and added to the list so that memory can then be allocated from that superblock.

![Figure 3-5: Super Block Linked List](image)

Figure 3-6 shows the structure of the heap that maintains in total 52 pointers, one for each size class. Every request for a block of memory is rounded up to the nearest
size class. Small size classes between 8B and 128B are in multiples of 8B, medium size classes between 128B and 512B are in multiples of 32B and large size classes between 512B and 2048B are in multiples of 64B. If a request for memory greater than 2048B is made then it is allocated directly from the operating system.

An advantage of tsxmalloc is that for size classes, which divide evenly by a number $2^n$ are aligned on a boundary of $2^n$. This is a result of the super block structure being both aligned on a 4K boundary and sliced evenly into 512 blocks of its size class. The result that small size classes are guaranteed to be aligned on an 8B boundary, medium size classes on a 32B boundary and large size classes on a 64B boundary. It was previously explained how dlmalloc and ptmalloc only align on a 4B boundary. While this is not a requirement of the function `malloc`, as in the ISO standard [13], programmers often need to align memory on a byte boundary in order to avoid major slow downs such as false sharing and cache line splits.

![Figure 3-6: Heap Structure](image)

35
3.2 Locking

The pointer of each size class points to the top of a list of partially full super blocks of that size class. As the lists are maintained in order of fullest to emptiest, superblocks regularly move up and down the list while memory is allocated and freed. Thus, operations on the linked list must occur atomically when concurrently allocating and freeing. We maintain a lock for each size class; hence, tsxmalloc can run in parallel at the granularity of a size class.

3.2.1 Operations on Doubly Linked-List

Four major operations occur on the doubly linked-list of super blocks:

**Pop**: When a super block becomes entirely full (i.e. all its memory is allocated) it is popped from the top of the list as the list only contains partially full super blocks.

**Push**: When a block of memory is freed from a full super block, that super block will be the fullest, partially full super block and is pushed onto the top of the list.

**Demote**: When a block is freed from a partially full super block $A$ and as a result, the next super block $B$ in the list has more of its memory allocated, then block $A$ must be demoted down the list so that the next block has more free memory and the previous block has less. This ensures that the list is ordered from fullest to emptiest super blocks.

**Drop**: When a block becomes entirely empty it is dropped from the end of the list and returned to the operating system.

3.2.2 Conflicts

While memory conflicts may occur when allocating and freeing in parallel, they are also quite unlikely. This is shown by example in figure 3-7 with 3 non-conflicting parallel operations, $A$, $B$ and $C$ consisting of one call to malloc and two calls to free memory respectively from the same linked-list.
A: The `malloc` in this example allocates a block from the super block at the top of the list, as memory is always allocated from the top of the list. The number of credits\(^2\) in that block is decremented resulting in 10 credits. No other changes are necessary, hence all the memory accesses to the list for this operation are local to a single super block.

B: In this situation, when a block of memory is freed, the super block’s credits are incremented to 59. As the list must be maintained in order, the super block is demoted down the list because the next super block only has 58 credits. All the parts of the list colored green involve memory writes that could result in conflicts. However, there are no conflicting memory accesses in this case.

C: A block of memory is freed from the super block and as a result its credits will be incremented to 391. In this operation all memory accesses are also local to the super block, which is very unlikely to cause a conflict.

### 3.3 Transactional approach

In the above example it is simple to demonstrate how many threads could be allocating and freeing from a single list of super blocks concurrently without conflicting accesses, however this example is quite restricted for demonstration purposes. In practice, the number of super blocks in a linked list would greatly exceed the above example, making conflicts much less likely to occur.

It is clear that concurrency can be improved when multiple threads need to access the same list of super blocks. As it is unlikely that conflicts will occur it makes

\(^2\)Credits are the number of free blocks available in a superblock.
sense to execute critical sections optimistically in parallel using transactional memory. Previously, any access to a list of super blocks of a specific size class had to be serialised with the use of a lock protecting the entire list. With transactional memory, we allow for a more fine-grained approach, reducing the granularity to the size of a cache line. However, we must remember that transactions may not succeed and must have an alternative thread safe path. In tsxmalloc the alternative path is to lock the entire list of a size class. As previously mentioned, the lock and pointer to each size class is maintained in the heap structure with 52 different size classes ranging from 8B to 1984B. Transactions are incorporated into tsxmalloc by eliding the lock that protects a list of super blocks corresponding to a size class. The lock is elided in the transaction by adding the lock to the transactions read-set instead of acquiring the lock. Hence, if during the transaction the lock is acquired by writing to it from outside the transaction a memory conflict will occur and the transaction will be aborted.

A simple mutual-exclusion spin lock is used as it has been shown to provide better scalability across a range of configurations for transactions by comparison to more complex read-write locks [12]. As transactions operate at the granularity of a 64B cache line, each lock must be padded out to 64B and aligned on a 64B-boundary to ensure that the locks do not share a single cache line. The corresponding size class pointer takes up 8B of the padding. In total this results in an indirect memory overhead of $52B = 416B$ for maintaining the locks and size class pointers in the heap structure.

<table>
<thead>
<tr>
<th>time (sec)</th>
<th>free tsx</th>
<th>malloc tsx</th>
<th>Locked free</th>
<th>Locked malloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.62</td>
<td>0</td>
<td>0</td>
<td>49%</td>
</tr>
<tr>
<td>2</td>
<td>2.53</td>
<td>0</td>
<td>1</td>
<td>48%</td>
</tr>
<tr>
<td>3</td>
<td>2.59</td>
<td>1</td>
<td>0</td>
<td>7%</td>
</tr>
<tr>
<td>4</td>
<td>2.53</td>
<td>1</td>
<td>1</td>
<td>6%</td>
</tr>
<tr>
<td>5</td>
<td>2.51</td>
<td>0</td>
<td>2</td>
<td>48%</td>
</tr>
<tr>
<td>6</td>
<td>2.51</td>
<td>1</td>
<td>2</td>
<td>6%</td>
</tr>
</tbody>
</table>

Figure 3-8: tsx attempts
The table in Figure 3.3 shows the performance improvement that results for transactionally executing the \texttt{free} and \texttt{malloc} routines. This particular test allocates a range of 1B to 64B on 4 threads running in parallel. The test is based on the random benchmarks discussed in chapter 4. The \texttt{free tsx} and \texttt{malloc tsx} columns show the number of attempts at transactionally executing the respective routine before serialising with the use of a lock. The \texttt{locked free} and \texttt{locked malloc} columns show the percentage of operations that were serialised through a lock. Hence, in row one when no attempt at transactional execution is made, 100\% of the operations use a lock. The number of transactional retries were incremented for each test until there was no further improvement in speed. While there is a 4.2\% speed up in rows 5 and 6 when attempting more transactions, the best results across a range of tests were observed with a single attempt at a transactional \texttt{malloc} as in row 2 where on average there was a 4\% speed up.
Chapter 4

Proof of Concept

In this chapter we identify both the strengths and weaknesses of tsxmalloc with respect to a well known and well established allocator ptmalloc, that is currently used as part of the GNU C library on Unix operating systems. The benchmarks make two strong yet opposite assumptions of random and serialised freeing of memory. All benchmarks are performed on:

-Intel i5-4200 processor, which has two CPUs (4 with hyper-threading) and a clock speed of 1.6GHz.
-Ubuntu 13.10
-Eclipse with Linux GCC

<table>
<thead>
<tr>
<th>Cache Info</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>32K</td>
<td>256K</td>
<td>1MB</td>
</tr>
<tr>
<td>Line size</td>
<td>64B</td>
<td>64B</td>
<td>64B</td>
</tr>
<tr>
<td>Shared</td>
<td>Hyper Threaded Core</td>
<td>Hyper Threaded Core</td>
<td>All Cores</td>
</tr>
<tr>
<td>Associativity</td>
<td>8-way</td>
<td>8-way</td>
<td>8-way</td>
</tr>
<tr>
<td>#sets</td>
<td>64</td>
<td>512</td>
<td>4096</td>
</tr>
</tbody>
</table>


```
1 begin timing
2 while operations are not complete
3     index = rand()
4     if (A[index] == NULL)
5         A[index] = malloc(MEM)
6     else
7         free(A[index])
8     A[index] = NULL
9     operations++
10 stop timing
```

Figure 4-1: Benchmark Pseudo Code

4.1 Random Benchmark

4.1.1 Methodology

In this benchmark, each thread continuously calls the functions `malloc` or `free`. We eliminate locality of reference by pseudo-randomly deciding the order in which memory is freed. This has as drastic effect on ptmalloc and as a result, tsxmalloc (using just a single heap) outperforms ptmalloc (using one heap per-thread) on multiple threads.

The benchmark is described using pseudo-code in Figure 4-1. Each thread holds an array of void pointers that are initialised to NULL. An index to the array is chosen at random. If there is a NULL pointer at that index, then `malloc` is called and the pointer returned is placed in the array at that index. Otherwise, if there is a pointer in the array that points to allocated memory, then the pointer is passed to `free` and the index of the array is replaced with NULL. This benchmark is performed using on 1, 2 and 4 threads with variable sized memory requests for both tsxmalloc and ptmalloc.

4.1.2 Results

In figure 4-2, the time taken to run a set amount of operations that are either `malloc` or `free` is on the y-axis and the number of threads used is on the x-axis. We run the benchmark with four different types requests defining benchmark 1, 2, 3, and 4. In the first and third benchmark we allocate exactly 8B and 1KB for calls to `malloc`,
while in the second and fourth benchmarks the memory allocated is randomly chosen from a range of 1B to 64B and 1KB to 1.5KB respectively. To state the obvious, the first two benchmarks examine the performance of the allocators when requesting small amounts of memory and the second two tests examine the performance for large memory allocations.

### 4.1.3 Discussion of Results

It is clear that for random memory allocations, tsxmalloc is faster on multiple threads. You might consider this strange as tsxmalloc has only one heap structure, while ptmalloc can have up to one heap per thread. However, threads in ptmalloc will only hold temporary relationships with a heap as discussed in the critical analysis in chapter 1. While beneficial for memory allocations that exhibit some temporal locality of reference (for example allocating an arbitrary number of blocks at the same time and then later freeing those blocks at the same time) when this locality is
eliminated, as it is here, ptmalloc experiences excessive sharing of the heap between multiple threads.

In benchmarks 1 and 3 where memory is only allocated from a single size class, we can see that tsxmalloc has much better performance on multiple threads by comparison to when variable sizes of memory are requested, as in benchmarks 2 and 4. You might think that tsxmalloc should scale better when allocating variable sizes of memory, considering conflicts only occur within a size class. However, the deciding factor for speed is not the contention for locks, but the locality of reference.

In ptmalloc the boundary, tags for each individual block of memory must be written to, for every call to malloc and free resulting in excessive line caching and hence, cache evictions. In tsxmalloc there is the advantage that all memory operations for a super block occur in the header of the super block thus, the memory accesses of 512 blocks of memory are confined locally to just two cache lines. Additionally, the coarse granularity of ptmalloc with respect to tsxmalloc is an added factor and demonstrated.

4.2 Serial Benchmark

4.2.1 Per-Heap Extension

To enable tsxmalloc to scale well we extend the algorithm to incorporate multiple heaps similar to Hoard and ptmalloc. This maintains a one-to-one relationship between a thread and a per-processor heap. When malloc is called, the current thread id is passed to a hash function, which maps the thread to a heap. The number of heaps maintained corresponds to the number of processors on the machine. In this case, up to four heaps are created, as there are four CPUs on the benchmarking machine with hyper-threading enabled. By using multiple heaps, we reduce the sharing of both the heap data structure and the headers of the corresponding super blocks. This drastically improves the scalability of tsxmalloc as the heap data structures are not shared between threads.
4.2.2 Methodology

In this benchmark, each thread serially allocates memory by calling `malloc` a number of times and then frees that memory in the order it was allocated. Contention for locks is minimal, as the memory allocated and freed is specific to each thread and each thread always has access to one heap. Additionally, by serially allocating and freeing, we increase the temporal locality of reference for tsxmalloc while increasing both temporal and spatial locality of reference for ptmalloc.

The benchmark is described using pseudo-code, in Figure 4-3. Each thread holds an array of void pointers that are initialised to NULL. A call to `malloc` is made for each index of the array and the pointer returned is stored in the array at that index. This is followed by all the pointers in the array being freed in the same order they were allocated. Each thread repeats this process a number of times divided by the number of threads. Hence, one thread will execute the same number of operations as two or four threads running the same test. In the pseudo-code `MEM` is the amount of memory requested and varies for different tests. The test is performed on 1, 2 and 4 threads for various memory requests.

4.2.3 Results

The results of the benchmark in Figure 4-4 show the duration for a set number of operations on the y-axis where the operations are to serially allocate and free memory as discussed above. On the x-axis is the number of threads used in the benchmark. We examine results for variable sizes of memory allocations. In benchmarks 1 and
3 all calls to `malloc` request exactly 8B and 1KB of memory respectively while in benchmarks 2 and 4 the requests are chosen randomly between 1B-64B and 1KB-1.5KB respectively.

![Serial Benchmark](image)

**4.2.4 Discussion of Results**

In the first two benchmarks performed, ptmalloc is faster than tsxmalloc. However, we can see from the graphs that for these benchmarks tsxmalloc scales marginally better with multiple threads.

An interesting result is shown in the second serial benchmark, where there is a slow down in speed with an increased number of threads. The scalability is equivalent to the random benchmark. It is not obvious why this occurs, considering that this benchmark results in high locality of reference and minimal contention for both memory allocators. A possible cause might be cache evictions due memory access
being in the same set.

For large memory allocations, in benchmarks 3 and 4,tsxmalloc demonstrates both superior scalability and better performance on multiple threads. As you can see, there is a speed up for both memory allocators when moving from one to two threads. However, on four threads ptmalloc slows down while tsxmalloc has another increase in speed. The slow down in ptmalloc is a consequence of extra memory operations required to maintain large bins\(^1\) and the cache architecture used in this benchmark.

Bins of ptmalloc smaller than 512B are maintained in LIFO order to improve temporal locality where each bin holds a list of free memory corresponding to a specific size. However, bins greater than 512B are not restricted to a specific size and must be maintained in order of size, hence large bins have no associated temporal locality. Furthermore, maintaining these bins in order requires extra memory operations that have a higher probability cache misses due to the lack of locality. The expense of extra cache misses and added memory operations may result in additional cache evictions. Coinciding with this inefficient use of the cache, the L1 and L2 caches on the machine used in the benchmark are shared between two cores when hyper-threading is enabled. The effect of reduced cache efficiency can be seen when each cache is shared by more than one thread, hence when 4 threads are running in parallel we see a slow down in ptmalloc.

\[4.3 \text{ Further Work}\]

Equivalent transactional benchmarks were subsequently published \[^2\] and show similar probability rates for the transactional-write benchmark. However, for the transactional read-benchmark they showed no commits past 32KB of memory reads. This is likely a result of the specific machine used in the benchmarks. Further work should be done quantify the effect that the architecture of a machine has on a transaction’s probability of committing.

\[^1\]doubly linked lists of free memory
Further benchmarking needs to be performed to test the robustness of the tsx-malloc, such as in a producer-consumer setup. External benchmarks performed show that for eight benchmarks on just four threads, ptmalloc is faster in three, Hoard in two and for the remaining three there is negligible difference [1]. Note; The Larson benchmark performed in this paper does not eliminate locality as only 1024 different blocks are referenced in each thread. Testing performance against the Hoard allocator with the two benchmarks from this thesis would be interesting especially for the pseudo-random benchmark that eliminates locality of reference.

In this thesis, we claim that locality is the main contributing factor towards the speedup of an allocator. Further work needs to be done to gauge this relationship by examining a cache trace for various benchmarks on the allocator.

Allocators with per-processor heaps aim to maintain a one-to-one relationship between each thread and its heap. However, in a producer-consumer environment the thread freeing the memory may not belong to the heap from which the memory was allocated resulting in shared memory operations. Further work on an allocator that queues addresses passed to \texttt{free} when they do not belong to the thread’s heap, such that a thread belonging to the heap later free the addresses could prove significant as it would ensure an actual one-to-one relationship between a heap and its threads.
References


Chapter 5
Appendix of Code

//============================================================================
// Code for tsxmalloc including the benchmarks performed
//============================================================================

#include "helper.h"

using namespace std;

/******
 * BENCHMARK VARIABLES
 * if mine is defined tsxmalloc os used, otherwise the gnu c malloc is used
 * MEM is the memory allocated in the benchmarks
 * if serial is defined the serial benchmark is performed, otherwise the pseudo-random benchmark is performed
 * XFREE and XMALLOC defines the number of transactional attempts
 *

******

#define mine

#define numthreads 1

//#define MEM 8
#define MEM (rand()%64) + 1
//#define MEM 1024
//#define MEM (rand()%512) + 1025

#define BLOCKS 50000/numthreads
#define totalops 4000000/numthreads
#define CYCLE 2
#define SERIAL
#define XFREE 0
#define XMALLOC 0
#include <cstdlib>
#include <unistd.h>

#ifdef mine
#define MALLOC mymalloc
#define FREE myfree
#else
#define MALLOC malloc
#define FREE free
#endif

void spin(UINT64 *lock, volatile UINT64 newval){
    volatile UINT64 oldval;
    until:
        oldval = 0;
        newval = 1;
        cmpxchg( lock, oldval, newval, until );
}

procheap *tempHeap[4] = {(procheap*)mmap(NULL, 52*(64), PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0),
    (procheap*)mmap(NULL, 52*(64), PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0),
    (procheap*)mmap(NULL, 52*(64), PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0),
    (procheap*)mmap(NULL, 52*(64), PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0)};
thread_id[4] = {0, 0, 0, 0};

//function to set the thread affinity for testing
int setTA(int core_id){
    int num_cores = sysconf(_SC_NPROCESSORS_ONLN);
    if(core_id< num_cores){
        return 0;
    }
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(core_id, &cpuset);
    pthread_t current_thread = pthread_self();
    return pthread_setaffinity_np(current_thread, sizeof(cpu_set_t), &cpuset);
}

//decrement the credits of a super block and pop from the top of the list if full
void decCredits(header *head, sizeclass *size){
    head->Descriptor.credits--;
    if(head->Descriptor.credits == 0){
        pop(head, size);
    }
}

//increment the credits of a super block and pop from the top of the list if full
bool incCredits(header *head, sizeclass *size){
    head->Descriptor.credits++;
    if(head->Descriptor.credits == 1){
        push(head, size);
    }
    if(head->next!=NULL){
        if(head->Descriptor.credits > head->next->Descriptor.credits){
            demote(head, size);
        }
    }
    if(head->Descriptor.credits == head->Descriptor.maxcredits){
        push(head, size);
        return false;
    }
    return true;
}

/************************************************************************************
 * The bit mask is a 64B block in the header of every super block as shown in the
 * structure of the super block. All the functions below take and/or return bits
 * corresponding to the diagram. Hence, MSB is always bit 511 and LSB is always bit
 * 0, as opposed to bit 512 and 1 respectively.
 * Each bit represents the state of each block in the super block; FREE: 0, ALLOC: 1
 * The most significant few bits are used to keep the header of the super block
 ************************************************************************************/
* * * ------MSB (bit 511)
* * 
* * byte8[7] --> 0x0000000000000000
* * byte8[6] --> 0x0000000000000000
* * byte8[5] --> 0x0000000000000000
* * byte8[4] --> 0x0000000000000000
* * byte8[3] --> 0x0000000000000000
* * byte8[2] --> 0x0000000000000000
* * byte8[1] --> 0x0000000000000000
* * byte8[0] --> 0x0000000000000000 <-- LSB (bit 0)
* *
************************************************************************************/

void setBitInMask(mask *mask, UINT64 bit){
    UINT64 index = bit/64;
    bit = bit%64;
    UINT64 set = 1;
    mask->byte8[index] = (mask->byte8[index] | (set<<bit));
}

void clearBitInMask(mask *mask, UINT bit){
    UINT index = bit/64;
    bit = bit%64;
    UINT64 clear = 1;
    mask->byte8[index] = mask->byte8[index] & ~(clear<<bit);
}

void clearBitMask(mask *mask){
    for(int i=0; i<8; i++){
        mask->byte8[i] = 0x0;
    }
}

void setBitMask(mask *mask){
    for(int i=0; i<8; i++){
        mask->byte8[i] = 0xffffffffffffffff;
    }
}

int findFirstFreeBlock(mask *mask){
    int ret = 0;
    int i = 0;
    for(i=0; ret==0 && i<8; i++){
        ret = __builtin_ffsll(~(mask->byte8[i]));
    }
    if(ret == 0){
        return -1;
    }else{
        return (ret-1)+(64*(i-1));
    }
}

void printBitMask(mask *mask){
    for(int i=0; i<8; i++){
        cout << hex << mask->byte8[7-i] << endl;
    }
}

/*******************************************************************************

* initialise a super block of a size class X, where X%8=0
* set the bit mask so that the most significant 96/X bits = 0 and every 4096/X bit = 0
* the rest are set to 1 to indicate that they are free
* Set the descriptor CREDITS:512 STATE:3 (-> empty) TAG:0 (lockless ABA, TSX?)
* A size class must be of the power of 2
* 
*******************************************************************************/

header *createSuperBlock(UINT sizeClass, sizeclass *size){
    size_t SBSIZE = (sizeClass/8) * 0x1000;
    header *SB = (header*)getPageAligned(SBSIZE);
    UINT credits = 512;
    SB->start = SB;
    SB->next = NULL;
}
SB->prev = NULL;
SB->size = size;
for(int i=0; i<104; i+=sizeClass){
    credits--; 
    setBitInMask(&SB->Mask), 511-(i/sizeClass));
}
  int test = 0;
  for(UINT64 i=0x1000; i<SBSIZE; i+=0x1000){
    if(((SBSIZE-i)%sizeClass) > 0){test = 0;}else{test = 1;}
    credits--; 
    setBitInMask(&SB->Mask), (SBSIZE-i)/sizeClass)-test;
    header **start = (header**)((UINT64)SB + i);
    *start = SB;
  }
SB->Descriptor.credits = credits;
SB->Descriptor.maxcredits = credits;
SB->Descriptor.sizeclass = sizeClass;
return SB;
}/************************************************************************************
*  Because every super block is aligned on a 4K boundry and the start of every 4K of
*  a super block has a pointer to the start of the super block, given any pointer
*  from a super block we can clear the last twelve bits of that pointer to get a
*  pointer to the pointer to the start of the super block.
*  ************************************************************************************/
header *getSBHead(void *ptr){
    UINT64 head = (UINT64)ptr;
    head = head & MASK_4K_ALIGNED;
    UINT64 *point = (UINT64*)head;
    if(*point == 0){
        UINT64 size = *(point+1);
        munmap((void *)head, size);
        return 0;
    }
    header **ret = (header**)head;
    return *ret;
}/************************************************************************************
*  A doubly linked list is maintained for each size class of super blocks. The
*  objective is to try and get as many full block and empty blocks as possible. Thus,
*  reducing the sparseness of the memory. We then return empty super blocks to the
*  operating system. Where memory is freed to is out of our control, however we
*  can decide where memory is allocated from. The linked list is kept in order of size
*  and memory can only be allocated from the active super block, which is the block
*  closest to being full. This mean that we are always trying to make the fullest
*  partially full super block full.
*  _sizeclass
*  * FULL | ACTIVE | PARTIAL
*  * | SB |---SzCl | SB |---SzCs | SB |----| SB |----| SB |----| SB |----| SB |----| SB |---NULL
*  * | SB |<---NULL<---| SB |------| SB |------| SB |<---| SB |<---| SB |<---NULL
*  ************************************************************************************
void push(header *push, sizeclass *size){
    if(size->top == NULL){
        push->next = NULL;
        size->top = push;
    }else{
        push->next = size->top;
        size->top = push;
        push->next->prev = push;
    }
    size->top->prev = NULL;
void pop(header *pop, sizeclass *size){
    //cout << "POP" << endl;
    //pop1++;
    if(size->top->next!=NULL){
        size->top = pop->next;
        size->top->prev = NULL;
    }else{
        size->top = NULL;
    }
    pop->prev = NULL;
pop->next = NULL;
}

void drop(header *drop, sizeclass *size){
    //drop1++;
    if(drop->prev!=NULL){
        drop->prev->next = NULL;
        drop->prev = NULL;
    }else{
        drop->prev = NULL;
drop->next = NULL;
        size->top = NULL;
    }
    //returnSBtoOS(drop);
}

void demoteDrop(header *remove, sizeclass *size){
    if(remove == size->top){
        size->top = remove->next;
        remove->next = NULL;
        remove->prev = NULL;
    }else{
        remove->next->prev = remove->prev;
        remove->prev->next = remove->next;
        remove->next = NULL;
        remove->prev = NULL;
    }
}

void demote(header *demote, sizeclass *size){
    //demote1++;
    header *insert = demote;
    while(insert->next!=NULL && insert->next->Descriptor.credits < insert->Descriptor.credits){
        insert = insert->next;
    }
    demoteDrop(demote, size);
    demote->prev = insert;
    if(demote->prev->next == NULL){
        demote->prev->next = demote;
        demote->next = NULL;
    }else{
        demote->next = demote->prev->next;
        demote->prev->next = demote;
        demote->next->prev = demote;
    }
    if(demote == size->top){
        size->top = demote->next;
    }
}

//******************************************************************************
******
* This function is used to print the header of a super block to ensure that the
* state of the header is exactly as we expect. The mask is printed in hex from the
* MSB to the LSB. The same as seen in comment for the bit mask functions.
* *****************************************************************************/

void printHeadInfo(header *head){
    cout << "---Descriptor---\nMax Credits:" << dec << head->Descriptor.maxcredits
}
/* Finds the first bit in the bit mask that is cleared
 * Calculate the pointer to that bit
 * return pointer to calling function
 */
*******************************************************************************

void *allocateFromSB(sizeclass *size){
    header *head = size->top;
    int free = findFirstFreeBlock(&head->Mask);
    //update descriptor, credits, state
    if(free>=0){
        setBitInMask(&head->Mask), free);
        decCredits(head, size);
        return (void *)((511-(UINT64)free) * (UINT64)head->Descriptor.sizeclass)+(UINT64)head);
    }else{
        cout << "error: no block found; top points to a full super block" << endl;
        return NULL;
    }
}

/*******************************************************************************
 * allocateFromSizeClass(sizeclass *size){
 * header *newSB;
 * void *ret;
 * int attempt = 0;
 * while(attempt < XMALLOC){
 *     int status = _xbegin();
 *     if (status == _XBEGIN_STARTED) {
 *         if(size->spinlck!=0){_xabort(0xff);} //this section must be executed using a lock as createSuperBlock calls mmap which
 *         if(size->top == NULL){
 *             _xend();
 *             return ret;
 *         }
 *         attempt++;
 *     }
 *     spin(&size->spinlck, 0x1);
 *     if(size->top == NULL){
 *         if(size->top == NULL){
 *             newSB = createSuperBlock(size->size, size);
 *             push(newSB, size);
 *         }
 *     }
 *     ret = allocateFromSB(size);
 *     size->spinlck = 0;
 *     return ret;
 * }
 */
void myfree(void *ptr){
    header *head = getSBHead(ptr);
    if(head == NULL){return;}//head will equal NULL if sizeclass>=2048
    sizeclass *size = head->size;
    int attempt = 0;
    while(attempt<XFREE){
        int status = _xbegin();
        if (status == _XBEGIN_STARTED) {
            if(size->spinlck!=0){_xabort(0xff);}
            UINT64 diff = (UINT64)ptr - (UINT64)head;
            UINT bit = diff/(head->Descriptor.sizeclass);
            if(incCredits(head, size)){
                clearBitInMask(&(head->Mask), 511-bit);
            }
        }
        _xend();
        return;
        attempt++;
    }
    spin(&size->spinlck, 0x1);
    UINT64 diff = (UINT64)ptr - (UINT64)head;
    UINT bit = diff/(head->Descriptor.sizeclass);
    if(incCredits(head, size)){
        clearBitInMask(&head->Mask), 511-bit);
    }
    size->spinlck=0;
    return;
}

sizeclass *getSizeClassMalloc(size_t request, procheap *procHeap){
    int index;
    if(request<=128){index = (request/8)-1;}
    else if(request<=512){index = ((request-128)/32)+15;}
    else{index = ((request-512)/64)+27;}
    //cout << "req: " << request << "tindex: " << index << endl;
    sizeclass *ret = &procHeap->Sizeclass[index];
    if(ret->size == 0){
        ret->size = request;
    }
    return ret;
}

bool init = true;

size_t roundup(size_t request){
    if(request < 121 && (request%8 != 0 || request == 0)){
        request = request - (request%8) + 8;
    } else if(request < 481 && request%32 != 0){
        request = request - (request%32) + 32;
    } else if(request < 1985 && request%64 != 0){
        request = request - (request%64) + 64;
    }
    return request;
}

void *sysalloc(size_t request){
    UINT64 *ret = (UINT64*)mmap(NULL, request, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    *ret = 0;
*(ret+1) = request;
    return (void*)(ret+2);
}

int heap_id(){
    pthread_t t_id = pthread_self();
    int heap = 0;
    for(int i=0; i<4; i++){
        if(thread_id[i]==0){
            thread_id[i] = t_id;
        }
        if(thread_id[i] == t_id){
            heap = i;
            i+=4;
        }
    }
    return heap;
}

void *mymalloc(size_t request){
    request = roundup(request);
    if(request>=2048){return sysalloc(request);} // get sizeclass pointer from process heap
    sizeclass *size = getSizeClassMalloc(request, tempHeap[heap_id()]);
    // no need for atomicity above here
    return allocateFromSizeClass(size);
}

float totaltime = 0.0f;
pthread_mutex_t mtx;

void *testRand(void *n){
    //int request = *(int*)n;
    //cout << "request:" << request << endl;
    void *ptrs[BLOCKS];
    for(int i=0; i<BLOCKS; i++){
        ptrs[i] = NULL;
    }
    int ops = 0;
    Timer t;
    t.start();
    while(ops<totalops){
        if(ptrs[choice] == NULL){
            ptrs[choice] = MALLOC(MEM);
        }else{
            FREE(ptrs[choice]);
            ptrs[choice] = NULL;
        }
        ops++;
    }
```cpp
void testSerial(void *n){
  int totalOps = 20/numthreads;
  int blks = BLOCKS*numthreads;
  void *ptrs[blks];
  Timer t;
  t.start();
  int ops = 0;
  while(ops<totalOps){
    for(int i=0; i<blks; i++){
      ptrs[i] = MALLOC(MEM);
    }
    for(int i=0; i<blks; i++){
      FREE(ptrs[i]);
    }
    ops++;
  }
  t.stop();
  pthread_mutex_lock(&mtx);
  totaltime+=((float)t.duration()/1000) << endl;
  //cout << "lockedpath: " << locked << "\tnew: " << new1 << "\tdrop: " << drop1 << "\tpush: " << push1 << "\tpop: " << pop1 << "\tdemote: " << demote1 << endl;
  return NULL;
}

#include "<pthread.h>

int main() {
  #ifdef mine
    cout << "mymalloc:" << endl;
  #else
    cout << "gnu lib c malloc: " << endl;
  #endif
  int size[4] = {0, 1, 2, 3};
  pthread_t trd[numthreads];
  for(int i=0; i<CYCLE; i++){
    for(int i=0; i<numthreads; i++){
      pthread_create(&trd[i], NULL, &FUNCTION, (void *)&size[i]);
      setTA(1<<i);
    }
    for(int i=0; i<numthreads; i++){
      pthread_join( trd[i], NULL);
    }
  }
  cout << "Average: " << totaltime/(CYCLE*numthreads);
  cout << "!!!COMPLETE!!!" << endl;
  #endif
```
/* helper.h
 * Created on: 13 Mar 2014
 *      Author: patrick
 *
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/time.h>
#include <pthread.h>
#include <cmath>

#define PAGE_SIZE 0x1000
#define MASK_4KAligned 0xfffffffffffff000
#define CLEAR 0xffffffffffffffff
#define UINT64 unsigned long long int
#define UINT unsigned int
#define MASKOFFSET 0x20

// TSX code is from glibc-2.18
#define _XA_EXPLICIT 0
#define _XA_RETRY 1
#define _XA_CONFLICT 2
#define _XA_CAPACITY 3
#define _XBEGIN_STARTED (~0u)
#define _XABORT_EXPLICIT (1 << _XA_EXPLICIT)
#define _XABORT_RETRY (1 << _XA_RETRY)
#define _XABORT_CONFLICT (1 << _XA_CONFLICT)
#define _XABORT_CAPACITY (1 << _XA_CAPACITY)
#define _XABORT_DEBUG (1 << 4)
#define _XABORT_NESTED (1 << 5)

#define _XABORT_CODE(x) (((x) >> 24) & 0xff)
#define _ABORT_LOCK_BUSY 0xff

#define __force_inline __attribute__((__always_inline__)) inline

static __force_inline int _xbegin(void)
{
    int ret = _XBEGIN_STARTED;
    asm volatile (".byte 0xc7,0xf8 ; .long 0" : "+a" (ret) : "memory");
    return ret;
}

static __force_inline void _xend(void)
{
    asm volatile (".byte 0x0f,0x01,0xd5" : ":memory");
}

static __force_inline void _xabort(const unsigned int status)
{
    asm volatile (".byte 0xc6,0xf8,%P0" : ":1" (status) : "memory");
}

static __force_inline int _xtest(void)
{
    unsigned char out;
    asm volatile (".byte 0x0f,0x01,0xd6 ; setnz %0" : "+r" (out) : "memory");
    return out;
}
/**
 * Structure of Super Block for size class X
 *******************************************/

// Linux kernel source code for cmpxchg with fall back instruction

#define cmpxchg(ptr, _old, _new, fail_label) {
    __ptr = (volatile uint32_t *)ptr;
    asm goto("lock; cmpxchg %1,%0 \t\n"
             : /* empty */
             : "m" (*ptr), "r" (_new), "a" (_old)
            : "memory", "cc"
        : fail_label);
}

/* Structure of Super Block for size class X */

typedef struct
descriptor{
    UINT64 credits:10, maxcredits:10, sizeclass:12, tag:32;
};

struct
descriptor{
    descriptor Descriptor;
    mask Mask;
};

struct
header{
    header *start://8
    header *next://8
    header *prev://8
    descriptor Descriptor://8
    mask Mask://64
    sizeclass *size://8
};

// aligned to a 64byte boundry for use in transactions
struct
sizeclass{
    header *top;
    long long int size;
};
struct procheap {
    sizeclass Sizeclass[52];
};

class Timer
{
    timeval timer[2];

public:
    timeval start()
    {
        gettimeofday(&this->timer[0], NULL);
        return this->timer[0];
    }
    timeval stop()
    {
        gettimeofday(&this->timer[1], NULL);
        return this->timer[1];
    }
    int duration() const
    {
        int secs(this->timer[1].tv_sec - this->timer[0].tv_sec);
        int usecs(this->timer[1].tv_usec - this->timer[0].tv_usec);
        if(usecs < 0)
        {
            --secs;
            usecs += 1000000;
        }
        return static_cast<int>(secs * 1000 + usecs / 1000.0 + 0.5);
    }
};

void push(header *push, sizeclass *size);
void pop(header *pop, sizeclass *size);
void drop(header *drop, sizeclass *size);
void demote(header *demote, sizeclass *size);
// Transactional Time Benchmark
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <math.h>
#include <conio.h>
#include <thread>
#include <Windows.h>

using namespace std;
#define UINT64 unsigned long long int
#define MAXSIZE 0xffff

int overload(double size);
void task1(double size);
int DIYRand();
UINT64 writeme[MAXSIZE];

int _tmain(int argc, _TCHAR* argv[])
{
    for(UINT64 i = 0; i<MAXSIZE; i++){
        writeme[i] = MAXSIZE/8 - i;
    }
    for(double size = 1<<10; size<<= 1<<17; size+= 1<<9){
        thread t1(task1, size);
        SetThreadAffinityMask(t1.native_handle(), 4);
        SetThreadPriority(t1.native_handle(), HIGH_PRIORITY_CLASS);
        t1.join();
    }
    _getch();
    return 0;
}

volatile UINT64 read;
void task1(double size)
{
    int commit = 0;
    int start = 0;
    cout << size/1024;
    for(int y = 0; y<=10; y++){  
        for(int i=0; i<1000; i++){  
            start++;
            commit = commit + overload(size);
        }
    }
    cout << "," 
    for(int i=0; i<size/(8) ; i++)
    {
        cout "=commit/start)*100; 
        commit = 0;
        start = 0;
    }
    cout << endl;
}

int p = 28;
int DIYRand(){
    return (((p = p*214013L + 2531011L)>>16)&0x7fff);
}

int overload(double size){
    int status = _xbegin();
    if (status == _XBEGIN_STARTED) {
        for(int i = 0; i<=size/(8) ; i++)
        {
            writeme[i] = i;
        }
    }
//transactional read-write benchmark
#include "stdafx.h"
#define UINT64 unsigned long long int
#define MAXSIZE 0xfffff

int overload(double size);
void task1(double size);
int DIYRand();
UINT64 writeme[MAXSIZE];

int _tmain(int argc, _TCHAR* argv[])
{
  for(UINT64 i = 0; i<MAXSIZE; i++){
    writeme[i] = MAXSIZE/8 - i;
  }
  for(double size = 1<<10; size<= 1<<17; size+= 1<<9){
    thread t1(task1, size);
    SetThreadAffinityMask(t1.native_handle(), 4);
    SetThreadPriority(t1.native_handle(), HIGH_PRIORITY_CLASS);
    t1.join();
  }
  _getch();
  return 0;
}

volatile UINT64 read;
void task1(double size)
{
  int commit = 0;
  int start = 0;
  cout << size/1024;
  for(int y = 0; y<=10; y++){
    for(int i=0; i<1000; i++){
      start++;
      commit = commit + overload(size);
    }
    cout << " , " << ((double)commit/start)*100;
    commit = 0;
    start = 0;
  }
  cout << endl;
}

int p = 28;
int DIYRand(){
  return(((p = p*214013L + 2531011L)>>16)&0x7fff);
}

int overload(double size){
  int status = _xbegin();
  if (status == _XBEGIN_STARTED) {
    for(int i = 0; i<=size/(8) ; i++){
read = writeme[i];
}
xend();
return 1;
}
else{return 0;}
}