Bilal Shirazi
98107317

CS499 Research Paper:
An analysis of concurrent memory allocators

Presented To:
Dr. Peter Buhr

1.0 Introduction:

This project examines concurrent dynamic memory-allocation. Dynamic memory-allocation occurs when a program requests storage other than for local variables on the stack. Dynamic memory-allocation and deallocation is made available via programming language statements such as **new** and **delete**, or via runtime routines such as **malloc** and **free**. Some programming languages provide garbage collection so there is no explicit deallocation.

In a concurrent environment, each task has its own stack so allocation and deallocation of local variables never causes contention among tasks. Furthermore, it is rare to share stack storage among tasks. However, heap allocation and deallocation often occurs among tasks via the shared heap-storage area. This shared resource can cause significant contention during allocation and deallocation. Furthermore, allocated heap storage is often passed around among tasks, resulting in shared access. These issues do not exist in a sequential environment.

Hence, the demands of dynamic memory allocation are more complex in a concurrent environment and can have a significant effect on the performance of dynamic memory-intensive applications. Therefore, it is crucial to have good dynamic memory-allocation in a concurrent system.

2.0 Objectives :

The objectives of this research project are:

1. To understand and validate the claims made by the researchers who created the Hoard memory allocator [1], currently considered the best dynamic memory allocator for concurrent systems.
2. To compare the uC++ memory allocator to the Hoard memory allocator. uC++ [3] is a concurrent system for C++.
3. To improve the uC++ allocator using techniques from Hoard.
4. To determine how each of the techniques in the Hoard allocator contributed to its overall performance.
5. To create a testbed memory allocator in uC++ that allows various memory allocation techniques to be quickly prototyped and tested.

3.0 Goals of a concurrent memory allocator:

The goals of a concurrent memory allocator should be the following:

Speed in sequential applications: The concurrent allocator should perform memory operations close to the efficiency of a sequential memory allocator. This goal ensures good program performance during the sequential components of a threads execution, which often constitute the majority of execution.

Scalability: As the number of processors in a multiprocessor system increase, the performance of the concurrent memory allocator must scale linearly to guarantee scalable performance of multi-threaded applications.

Minimize fragmentation: Memory fragmentation is defined as the total amount of memory allocated from the operating system divided by the maximum amount of memory required by an application. Excessive fragmentation implies that memory is being wasted and leads to performance degradation caused by poor data locality and increased paging.

General performance: A concurrent memory allocator should perform well across a wide variety of sequential and multi-threaded application types and workloads.

Minimize Contention: Since dynamic memory is a shared resource, which may require synchronized access, it can become a bottleneck to performance. A memory allocator must minimize this contention by concurrently servicing allocation and deallocation requests. Failure to do so can potentially serialize the execution of a multi-threaded allocation.

Mimimize false sharing: When multiple processors with separate caches share a common memory, it is necessary to keep the caches coherent by ensuring that any shared value that is changed in one cache is changed throughout all caches. Cache coherence strategies aim to solve the problems associated with sharing data by invalidating processor caches when shared memory is modified. However, each time a processor's cache is invalidated it limits performance as it must now refetch the shared value from the changed cache. “True sharing” occurs when applications explicitly share values that result in cache coherency issues. Some sharing is inevitable in a concurrent program, but many techniques exist to mitigate problems.

“False sharing” occurs when two unrelated data items, each used by a different thread,

appear in memory that is mapped to the same cache line. In systems with coherent caches, if one thread updates its portion of the data, the cache line is invalidated in the other thread even though the data it is using is unaffected. These cache invalidations can significantly degrade the performance of a program. False sharing can be induced by the memory allocator both within the allocator's data structures and the storage returned from an allocator. Although it is impossible to prevent all forms of false sharing, much can be avoided.

4.0 False sharing

It is impossible to avoid false sharing without padding every memory request to the size of a cache line, which wastes space both in memory and in the cache. There are two types of false sharing, actively induced false sharing and passively induced false sharing. Both must be avoided to ensure the scalability of a concurrent memory allocator. [1]

Active false-sharing occurs when memory requests for different threads are satisfied from memory that falls in the same cache line. For example, an allocator may satisfy memory allocations for multiple threads by returning consecutive addresses in memory. The caching mechanism then induces false sharing as each thread invalidates the other's cache without actually ever sharing the data in the cache line.

Passive false sharing occurs when a deallocation allows a future allocation to produce active false sharing. For example, an allocator may satisfy allocation requests from different heaps thus avoiding

active false sharing. However, if one thread passes a piece of storage to another thread to deallocate, this storage may become part of the heap of the deallocating thread and not the thread heap from which the original allocation occurred. This leads to the original active false sharing situation.

An illustration of false sharing is shown in the figure 1. Two threads make allocation requests that are serviced by a concurrent single-heap allocator. Both allocations fit within the size of a 64 byte cache line, so when accessed on separate processors, there is false sharing between caches. That is, an update to either piece of memory results in the invalidation of both processor's caches.

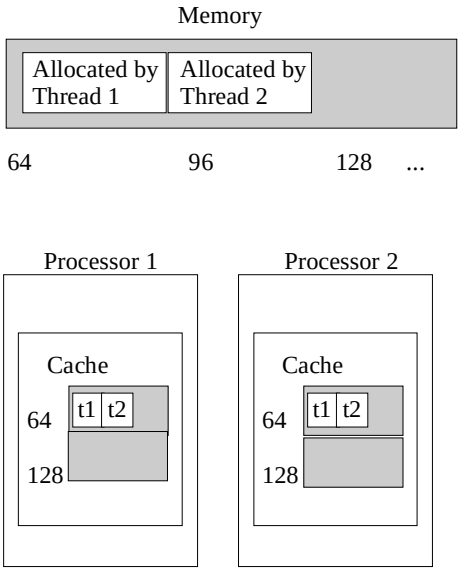

Figure 1: Two processor false sharing.

5.0 uC++ memory allocator:

The concurrent memory allocator built into uC++ has not changed since its inception. The allocator

falls into the class of allocators known as concurrent single-heap. These allocators model the heap as a concurrent data structure such as a concurrent B-tree or a collection of free lists. Many threads may simultaneously operate on this single shared heap. [3]

The uC++ heap is modeled as a contiguous block of memory marked by a beginning and end pointer from which memory is carved into smaller sub-blocks as necessary. As required, the end of the heap is extended by acquiring a global lock on the heap and invoking the UNIX **sbrk()** system call. The uC++ model of the heap is illustrated in figure 2. [3]

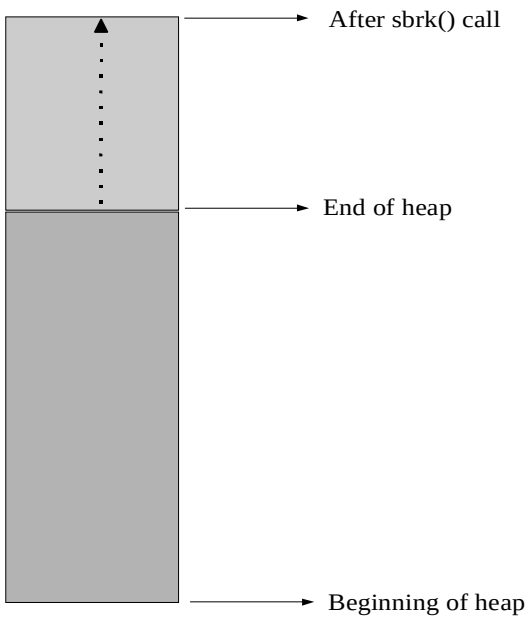

Figure 2: uC++ heap model

The data structure used to manage this storage is an array of structures, which point to free lists of

distinct block sizes ranging from 2^3 to 2^{32} bytes. Each free list is protected by a spin lock which must be acquired during allocation and deallocation. The free list is a chain of elements consisting of a header followed by the available space. The uC++ allocator data structures are illustrated in figure 3. [3]

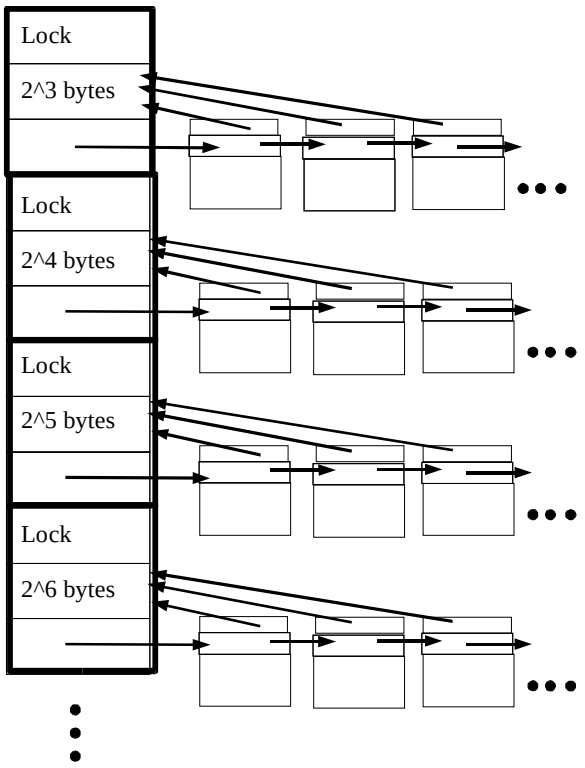

Figure 3: uC++ free lists.

The pseudo-code for the uC++ concurrent allocator is:

```
malloc(sz)
1. align the size request to the appropriate block size
2. acquire the lock for the free list of that block size
3. if the free list is not empty
4.       remove an empty block from the front of the list
5.       release the free list lock
6. else
7.       release the free list lock
8.       carve a block from the heap, possibly growing the heap
9. set the block header pointer to the appropriate free list
```

```
10.return the pointer to the block space
```

free(ptr)

```
1. skip back from the block space to get to the block header
2. extract the block pointer from the header
3. acquire the free list lock for the list to return space to
4. add the block to the front of the free list
5. release the free list lock
```

The allocation routine is simple and easy to follow. Once the block size has been computed, the lock for the free list is acquired. If space is available it is allocated to the thread; otherwise it is must be carved off of the heap. The deallocation routine is just as simple to understand. The block header contains a pointer to the free list where the block needs to be chained to. Once the lock for the list is acquired, the space is inserted at the head of the free list.

The approach used by the uC++ allocator is both fast and efficient. However, it fails to address the problems of contention and false sharing avoidance. Multiple threads allocating from the same free list contend for the same lock and this leads to serialization of memory access. False sharing occurs in two ways: at the level of the allocator data structures, and in the user's code from the way in which blocks are allocated from the heap area. Both of theses points are discussed in detail during performance analysis of the uC++ allocator.

6.0 Hoard memory allocator:

The Hoard concurrent memory allocator falls into the class of allocators known as private heaps with thresholds. These allocators provide each thread with its own heap but have a bound on unused memory in a heap. [1]

Hoard augments per-thread heaps with a global heap that every thread can access. Each thread allocates and deallocates from its thread heap and occasionally accesses the global heap. Hoard uses Doug Lea's

allocator (dlmalloc) [2] to acquire memory from the underlying operating system rather than directly managing the heap with the **sbrk()** routine. The memory model used in the Hoard allocator is illustrated in figure 4. A hashing mechanism routes a thread allocation request to its local thread heap. Excess free memory is returned in the local thread heap to the global heap once an upper bound is reached within a thread heap.[1]

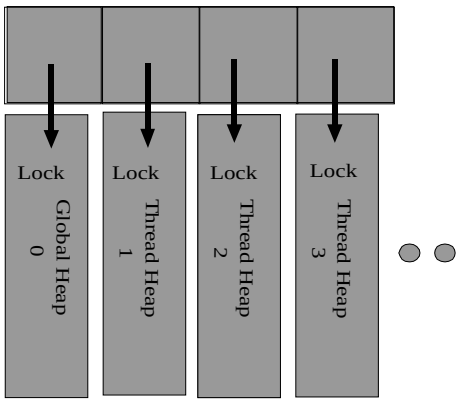

Figure 4: Hoard heap model

Each thread heap manages a set of superblocks, which represents the storage allocated by a thread. The thread heap stores all of the superblocks it owns in an array, sorted first by size class and then by fullness. Each thread heap contains a lock that must be acquired before a request can be satisfied. The Hoard allocator hashes a thread-ID and uses it as an index, where a thread-ID is thread-system dependent. The hashed index is used to locate the appropriate thread heap. An illustration of a thread heap is shown in figure 5. [1]

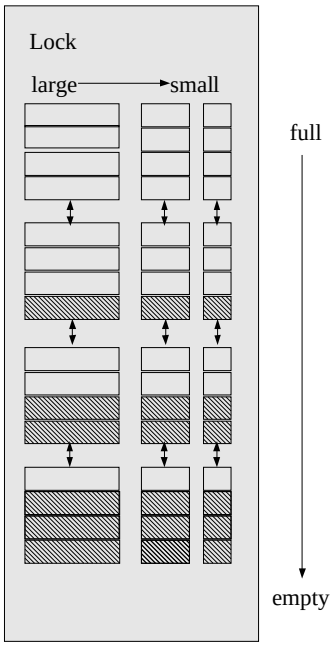

Figure 5: Thread heap structure: three superblock sizes sorted by fullness

All superblocks are 8 Kilobytes in size. Each superblock contains a complex set of data structures to manage a set of fixed-size allocation blocks. Hence, each superblock manages block sizes from the same size class. For example, allocation requests for 32 bytes are serviced from the same superblock. There can be multiple superblocks for a class size. If an upper bound on the total amount of free memory in the thread heap is exceeded, it is possible for an empty superblock to be moved from its local thread heap to the global heap where it can be be used by other threads heaps.

Each superblock contains a lock that must be acquired before operations are performed within it, such as allocation, deallocation, and transfer. Size classes range from 2^3 to 2^{11} bytes, the granularity between size classes ranges from 8 bytes to 128 bytes. Size requests larger than 4K, half the size of a

superblock, are serviced directly from the underlying storage allocator (dlmalloc) since the overhead in managing such a large superblock outweighs its benefits. A simplified block and superblock relationship is illustrated by figure 6. The block header contains all the relavent information for each superblock such as size and fullness. The blocks themselves are contiguous addresses in memory, and contain a pointer to the superblock header and a pointer to the next available free block. [1]

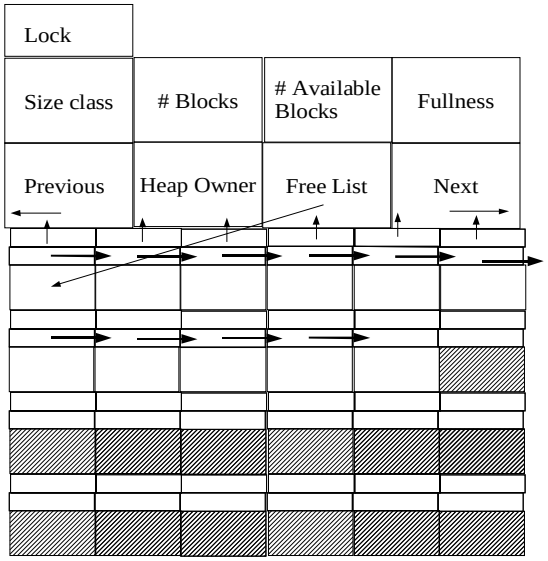

Figure 6: A Hoard superblock.

The pseudo-code for the Hoard allocator is:

```
malloc(sz)
1. if sz > half a superblock, allocate from the OS directly
2. hash the thread id
3. lock the appropriate thread heap
4. find the most full superblock that satisfies the request
5. if there is no such superblock
6.       check the global heap for a superblock
7.       if there is none
8.                allocate a new superblock,assign owner
9.       else
```

```
10.              transfer superblock from global thread heap
11.update heap statistics and adjust ordering by fullness
12.unlock the heap
13.return a block from the selected superblock
```

free(ptr)

```
1. skip back from the ptr address to find the block header
2. extract the superblock pointer from the header
3. lock the thread heap which owns this superblock
4. deallocate the block from the superblock
5. if there is too much free memory in the thread heap
6.       lock and transfer empty superblock to the global heap,
        unlock the superblock
7. update heap statistics and adjust ordering by fullness
8. unlock the thread heap
```

The allocation routine for the Hoard allocator is more complex than the uC++ allocator. Large memory requests are satisfied from the underlying storage allocator. Otherwise a block from a superblock of appropriate size is allocated to the user. If no such superblock is available, one must be created, either by recycling empty superblocks from the thread heap or global heap, or by allocating 8 Kb of memory from the OS and constructing a new superblock. After the block has been allocated, the statistics for the thread heap are updated.

The deallocation routine is also more complex than the uC++ deallocator. The block header contains a pointer to the superblock that owns the storage. This pointer is dereferenced and the block is returned from where it originated. Next, the heap statistics are updated, and if the upper bound on the amount of free space in the thread heap is exceeded, an empty superblock is returned to the global heap.

The approached used by the Hoard allocator is complex and more difficult to understand. However, the approach reduces false sharing and contention by satisfying memory requests from separate thread heaps. The superblock data structure is wasteful if memory requests only use a small portion of blocks while a majority of the blocks are never utilized.

7.0 Benchmarks:

In order to verify the claims of the Hoard paper, a group of students, of which I was one, recreated the results of some of the experiments presented in the paper. The platform used to test these claims was an SGI Origin 2000 cc/NUMA architecture with 8 processors, IRIX64 version 6.4, and 2GB of RAM . The multi-threaded benchmarks tested were the following:

shbench: each thread randomly allocates and frees randomly sized objects 100000/N times, where N = the number of processors. This benchmark tests how well the allocator services a fixed range of block sizes. The randomness of the allocations ensures that multiple free-lists are accessed.

threadtest: each thread repeatedly allocates and then deallocates 100000 / N objects, where N = the number of processors. This benchmark tests how well the allocator services a fixed block-size. The fixed block-size ensures that contention is maximized for a single free-list.

Both benchmarks are saturation tests involving a high number of allocations and deallocations over a short duration. Saturation tests do not usually reflect normal dynamic memory usage, and hence, are examples of worst case scenarios.

Each benchmark was run with the Hoard allocator and compared with a run using the standard IRIX system allocator. Speedup was measured by dividing the runtime of a multi-processor run by the runtime of a single processor run. The expected result for each benchmark test was that the runtime would be inversely proportional to the number of active threads. To eliminate anomalies, each test was run three times and the resulting times averaged. The speedup graphs show that as the number of processors increased, there is an increase in speedup for Hoard. The rate at which the speedup occurred is different for each benchmark. Figure 7 shows the speedup graphs of the benchmarks.

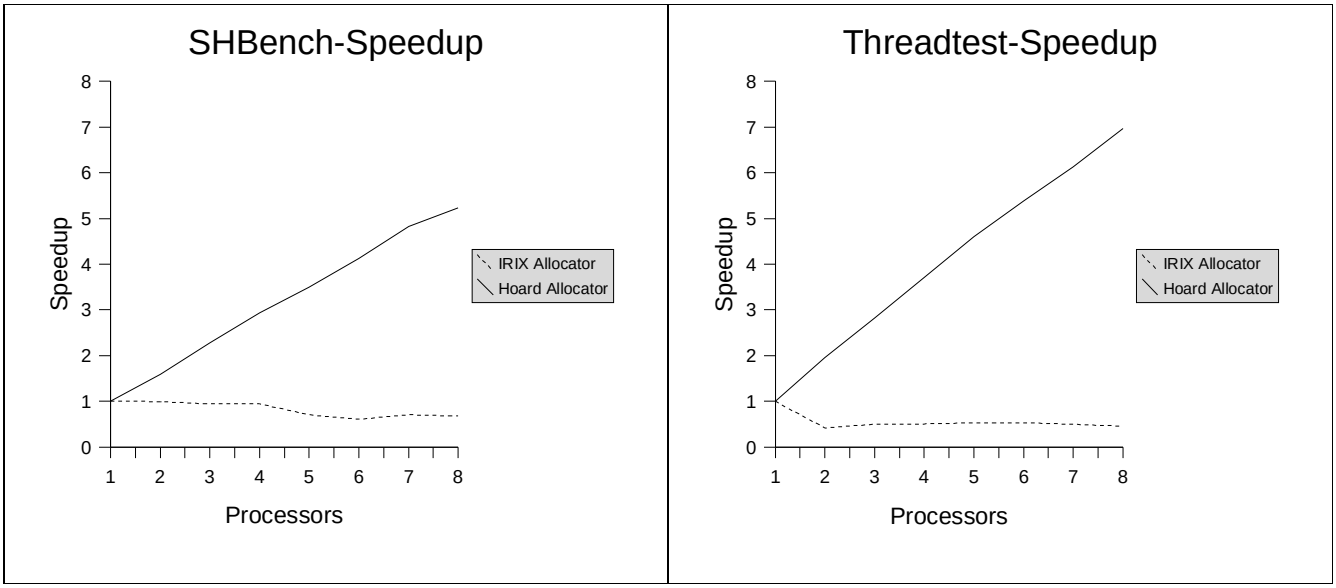

Figure 7: Benchmark speedup graphs.

8.0 Benchmark analysis:

A brief analysis of the graphs shows that the Hoard paper was accurate in its claims that it scales linearly as compared to the standard system allocator. [1]

These results are not perfect though as there is a near-linear speedup with the threadtest benchmark but the speedup graph for the shbench benchmark is less than optimal, which is evident when comparing the slope of both speedup lines. This difference is explained by the design of the shbench benchmark. Interestingly, it is an example of a sequence of allocations that stresses the Hoard allocator. The random sequence of allocations fall in a wide range of size classes. Each time a new size class is requested it results in the creation of an 8 Kilobyte superblock. This creation overhead is what limits the scalability of this particular benchmark.[1].

8.1 uC++ benchmarks:

The second phase of testing involved porting the two benchmarks to uC++. Fortunately, this task was straightforward as there is a simulator library included with uC++ for the POSIX thread calls used in the benchmark programs. After porting was completed, the students re-ran the ported shbench and threadtest benchmarks for uC++.

The results from each of these benchmarks were collected and added to the graphs from the original benchmark tests. Since the uC++ concurrent allocator is a concurrent single-heap allocator it was expected to performed poorly in scalability testing. What was not expected was how poorly the uC++ allocator actually performed. The analysis of the speedup graphs indicate that the uC++ allocator has significant problems in comparison to the Hoard and IRIX allocator during a saturation test. This fact is evident in figure 8 as both the shbench and threadtest graphs show the uC++ line below the IRIX allocator, indicating that it has a linear slowdown as the number of processors are increased.

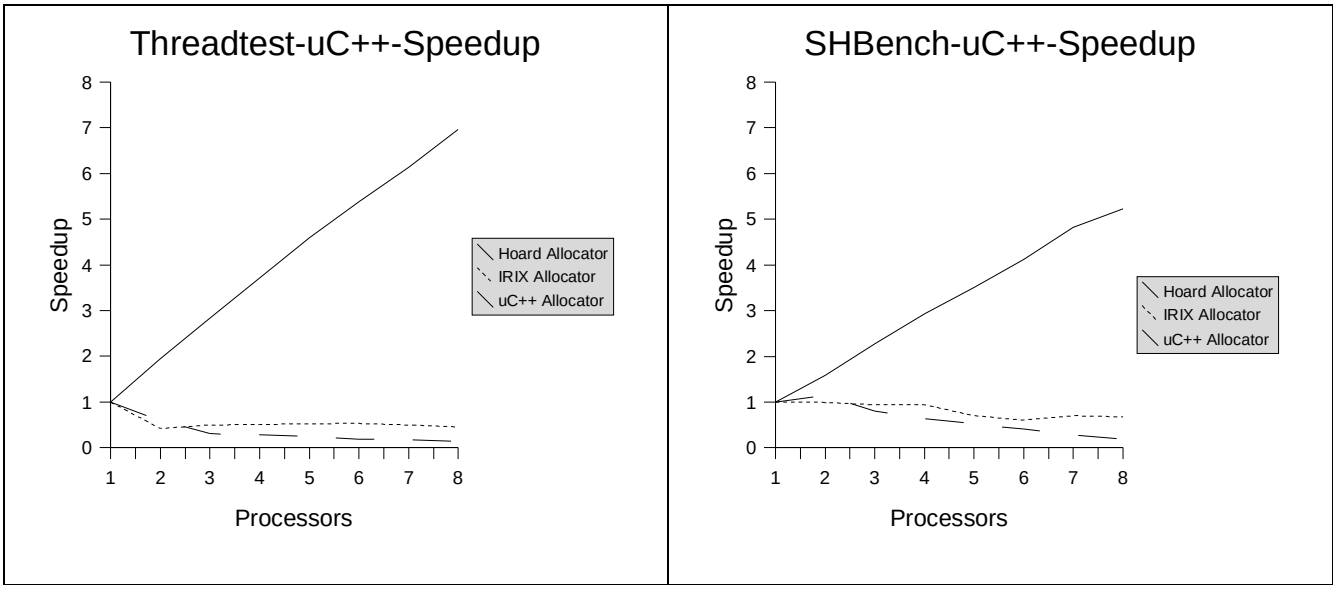

Figure 8: uC++ speedup graphs.

8.2 uC++ benchmark analysis:

Time was spent by the student group to understand why the uC++ linear slowdown occurred. Some of

the slowdown occurred because of the design of the uC++ allocator and some occurred because of the particular implementation of the uC++ allocator.

To further investigate contention in the uC++ free lists, two additional tests were created.
The first test involved multiple threads allocating random block sizes between 8 and 1024 bytes. This test was similar to shbench but used a random number generator to generate block sizes. The random number generator ensured that a wider range of free-lists were being accessed.
The second test involved multiple threads allocating unique block sizes between 8 and 1024 bytes. This test was designed so that no two threads would ever contend for a free-list lock. The runtimes of both tests were recorded and analysed. Surprisingly, both tests showed a linear slowdown.

Due to contention for free-list locks in the basic design, it was understandable why there was a linear slowdown with the first benchmark because there was a sufficient number of common requests at each free list. However, it was difficult to understand why there was a linear slowdown in the second benchmark. Each thread was accessing its own free-list, and hence, there should be no contention whatsoever. The problem was traced to false sharing in the uC++ allocator data structures. The elements of the array containing the free list information were small enough that multiple elements fit in a single cache line. Hence, false sharing is induced when tasks using adjacent elements attempt to acquire the free-list locks.

To prove the false-sharing hypothesis, the elements of the array were padded to the size of a cache line, and both tests were rerun. The results were a small linear speedup for the first test, since there was still contention for free-lists, and a larger linear speedup for the test where there was no contention for the free-lists.

Without the padding, the uC++ memory allocator data structures were too close together. The cache lines on the SGI are large enough to store multiple free-list headers. Hence, the linear slowdown in the original tests was caused by false sharing of the uC++ data structures. However, it was not enough to pad between the array elements since the uC++ allocator still suffered from allocator-induced false sharing. This was evident when the students retested the original shbench and threadtest benchmarks with the padded uC++ memory allocator. The results in figure 9 are slightly different than the results in figure 8. There is a slight improvement in the performance of the padded uC++ allocator in the shbench test but no significant improvement in the threadtest benchmark. However, the padded uC++ allocator still failed to outperform the IRIX allocator. Therefore, it was concluded that the uC++ memory allocator needed to be redesigned to solve the contention and false-sharing problem.

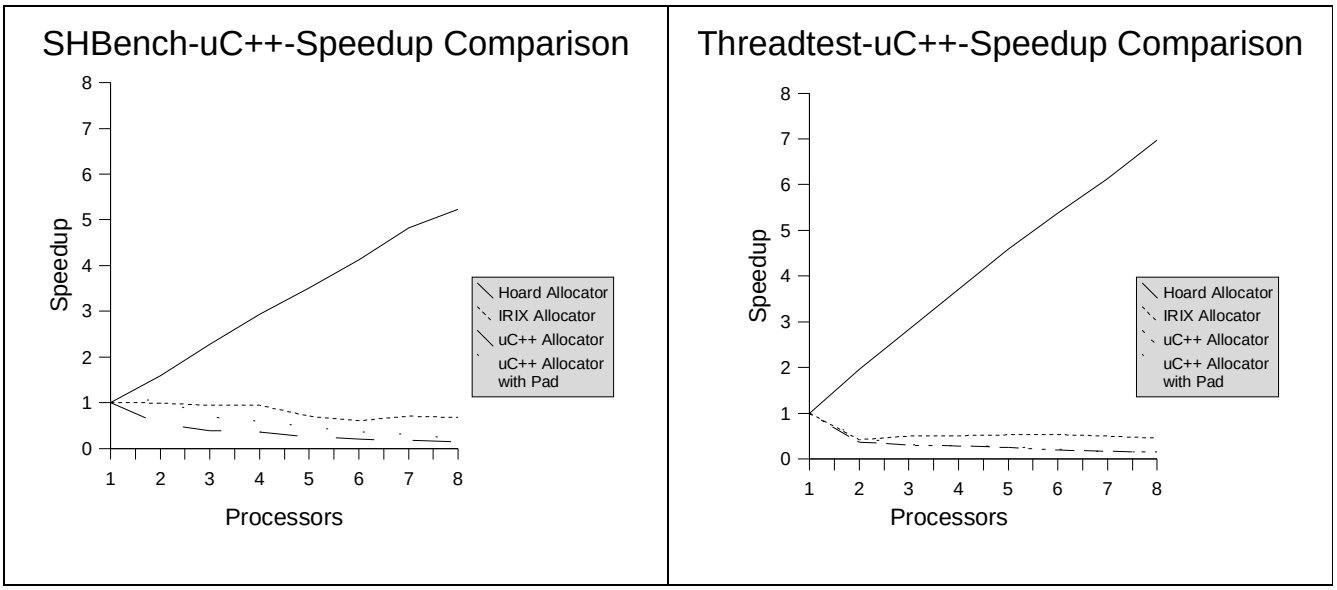

Figure 9: uC++ speedup graphs with padding.

9.0 Avoiding False Sharing and Lock Contention in Hoard:

The reason the Hoard allocator outperforms the IRIX and uC++ allocator is that it reduces the problems

of actively and passively induced false sharing and minimizes lock contention.

Hoard uses multiple-heaps to avoid most active and passive false sharing. Only one thread may allocate from a superblock at a time, because a superblock is owned by exactly one thread heap at a time. When multiple threads make simultaneous memory requests for memory, the requests will generally be satisfied from different superblocks in different thread heaps, avoiding actively induced false sharing.

When a program deallocates a block of memory, Hoard returns the block to the superblock it originated from. This prevents multiple threads from reusing pieces of cache lines that were passed to these threads by a user program, thus avoiding passively-induced false sharing.
Lock contention occurs when multiple threads race to grab the same lock for memory list for a given size class. In Hoard, different threads use different thread heaps, each with their own memory list for a size class. This signifantly reduces the problem and improves the scalability of Hoard. Lock contention can only occur under two scenarios: when multiple threads use the same thread heap (when the number of threads exceeds the maximum number of thread heaps), and when memory is deallocated by a different thread heap than the one that allocated it. The latter can happen in producer/consumer problems, where a producer thread creates and item and passes a reference to that item to a consumer thread for processing and deallocation.

10.0 Problems with the Hoard allocator:

The Hoard allocator suffers from two problems, which may inhibit its performance in certain situations. The first problem is internal fragmentation, e.g., a single memory request of 8 bytes may actually result in the creation of an 8K superblock. Hence, applications with memory requests similar to shbench result in a great deal of wasted memory. The uC++ allocator would fare much better in this test if the

false sharing problem is solved. The second problem is the true scalability of the allocator. In reality, the Hoard allocator statically creates only 64 thread heap objects from which superblocks can be accessed. A thread ID is extracted from a task and is used to determine which thread-heap to access. However, if the number of active threads is scaled up to and approaches 64, the hashing strategy fails to avoid the false sharing problem since multiple threads may use the same thread heap.

11.0 uC++/Hoard hybrid:

I built a hybrid uC++/Hoard allocator, which uses the multiple heaps with thresholds approach in uC++. I eliminated the scalability problem of Hoard by making each thread responsible for managing its own heap instead of having the allocator manage the thread-heap. The objective of this hybrid allocator is to separate out the design elements of the Hoard allocator to learn precisely what each element adds to the total memory allocator package. That is, it allows modification and testing of each element separately. It will be used by future students for experimentation and analysis.

12.0 Future work:

There is still a great deal of analysis that can be done using the uC++/Hoard hybrid allocator. Future students can experiment with the effects of modifications to the allocator such as maintaining a minimum allocation size of a cache line, or developing superblocks with multiple size classes. An interesting experiment would be to see the effects of keeping the multiple heap model of the Hoard allocator while using the uC++ free-list data structures to manage storage.

13.0 Acknowledgements

I would like to thank Dr. Peter Buhr, Ashif Harji, and Dr. Steve MacDonald for their time and effort in helping improve this paper.

14.0 Bibliography:

[1] E. Berger, K. McKinley, R. Blumofe, P. Wilson Hoard: A Scalable Memory Allocator for Multi-threaded Application, Proceeding of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems pages 117-128, 2000

[2] D. Lea dlmalloc: http://gee.cs.oswego.edu/dl/html/malloc.htm

[3] P. Buhr: uC++ : http://plg.uwaterloo.ca/~usystem/uC++.html