Memory Allocation for Long-Running Server Applications

Per-Åke Larson and Murali Krishnan

Microsoft

palarson@microsoft.com, muralik@microsoft.com

1. ABSTRACT

Prior work on dynamic memory allocation has largely neglected long-running server applications, for example, web servers and mail servers. Their requirements differ from those of one-shot applications like compilers or text editors. We investigated how to build an allocator that is not only fast and memory efficient but also scales well on SMP machines. We found that it is not sufficient to focus on reducing lock contention - higher speedups require a reduction in cache misses and bus traffic. We then designed and prototyped a new allocator, called LKmalloc, targeted for both traditional applications and server applications. LKmalloc uses several subheaps, each one with a separate set of free lists and memory arena. A thread always allocates from the same subheap but can free a block belonging to any subheap. A thread is assigned to a subheap by hashing on its thread ID. We compared its performance with several other allocators on a server-like, simulated workload and found that it indeed scales well and is quite fast but memory more efficiently.

1.1 Key words

Dynamic memory allocation, server applications, concurrency, multiprocessor scalability, reducing lock contention, cache-conscious algorithms.

Authors' address: Microsoft, One Microsoft Way, Redmond, WA 98052-6399, USA.

2. INTRODUCTION

Long-running server application like web servers, mail servers, and database servers are widely used but there has been little research on dynamic memory allocation and garbage collection for this class of applications. Server applications have different allocation patterns and different requirements than traditional one-shot applications like compilers or text editors. They are usually multithreaded and frequently run on large SMP systems, which implies that allocators targeted for this class of applications must be able to handle high levels of concurrency.

This paper describes our progress in developing a dynamic memory allocator targeted both for traditional applications and server applications. In addition to the traditional objectives of speed and efficient memory usage, our design emphasizes scalability on SMP systems.

The rest of the paper is organized as follows. Section 3 sets the stage by describing typical server applications, their workload, and how they are architected. Section 4 summarizes our view of the requirements on dynamic memory allocators for server applications. Section 5 provides a brief summary of prior work in this area. The current design of our allocator is described in section 6. Experimental results, using a simulated workload, are reported in section 7. Section 8 summarizes our findings and offers some conclusions.

ISMM '98 10/98 Vancouver, B.C.

3. BACKGROUND

In this section we explain the background surrounding the problem space. First we talk in general about the server applications and next we describe generically how requests are processed in typical server applications.

3.1 Server applications

It is impossible to come up with an exact definition of "server application" but, for the purpose of this paper, we define a server application as follows. A server application provides some service; its purpose is to accept requests from clients and process them. Typically a server application runs for a long time (indefinitely), processing requests received from a variety of client applications. Most commonly server applications are used for networked services. A server application is expected to process many requests per unit of time and do so with minimum delay.

A request is the unit of work given to a server application. The requests can be and usually are generated by a wide range of client applications that can be running either

locally or on wide area networks. Requests arrive at random intervals at the server. The number of requests and the rate of requests are usually not within the control of the server. A typical request is "small", that is, the processing and resources required are usually small in comparison to the resources available on the server systems.

Here are a few examples of widely used servers that fit the above description of servers and request types.

- Database servers.
- Web servers.
- Directory/name servers.
- Authentication servers.
- Mail servers.

3.2 Anatomy of server applications

A server application can be implemented in many different ways depending on the needs of the service and the type of environment in which it is used. Commonly we can categorize servers based on state involved and the manner in which requests are processed.

A server may have to accumulate and maintain state about a client for processing future requests from the same client. Such a server is said to be *stateful*.. A stateful server typically requires the client to first open a connection to the server before making requests. The connection is closed after all the requests are handled. To the server, the connection *is* the client and the server usually stores the state information about the client along with the connection object.

A server that does not maintain any state about the client is said to be *stateless*. Some server applications function this way (eg: time servers or DNS servers).

The architecture of many servers is based on the thread-per-connection model[1]. The basic version of this architecture creates a thread for each open connection, which maintains the state information and also processes all requests arriving on its connection. Two refinements of this model are common: worker threads and service threads.

It may be quite expensive to create a thread for each connection and keep the thread alive between requests. This is the case, for example, on some Unix systems where a "thread" must actually be implemented as a separate OS process. In that case, usually a fixed number of worker threads (a pool of workers) are created and used. When a request arrives, it is put on a queue of requests waiting to be processed, from where it will eventually be picked up by a worker thread for processing. In this scenario, a connection is a passive data structure that survives between requests and is handed to the worker thread together with a request. Worker threads not only avoid the overhead of creating/destroying a thread for each connection but the level of concurrency can be adjusted simply by increasing or decreasing the number of worker threads.

It is not desirable to tie up the main worker threads by waiting for I/O operations or requests to other servers to finish. With a limited supply of threads, this may result in all threads being blocked even though there are requests that could be processed. This can be handled by having specialized service threads, for example, one I/O thread for each disk. When a worker thread needs to read or write from a disk, it puts one or more I/O request on the queue for the appropriate disk, suspends processing of the current request and resumes processing of another request. The same idea can be applied when a server needs to request services from another server, for example, a database server sending a request to an authentication server.

We predominantly focus on stateful multi-threaded servers. There are two notable trends. First, servers are no longer fully monolithic pieces of code. Many server applications load and use shared modules for processing a request. A second trend is that the server applications and shared modules are increasingly implemented in C++ or other object oriented languages that rely more heavily on dynamic memory allocation than traditional languages.

For instance let us look at what happens inside a Microsoft's web server (IIS) when it processes a request. The web servers maintain state for connections and clients. Besides the web server's connection state, several additional code modules, may be loaded into the server application and called upon to process the request. For example, a request for legacy data access on a web server is processed by the web server code, by database connector, and by page formatter modules. To avoid tying up the I/O threads for long duration, the server implements a pool of worker threads and delegates work from the I/O service thread to the worker thread. We found that request processing can impose loads as high as 1000 allocation/frees per request. For most part objects allocated were small – 80% of allocations were for blocks of 40 bytes or smaller.

A server application may have to handle very high request rates, often in the range of thousands of requests per second. To support a high request rate, it is often necessary

[1] We use "thread" as a generic term meaning an independent thread of control. A "thread" can be implemented in several ways: as an OS thread, an OS process, or as a user-level (lightweight) thread. The specific implementation choice depends most often on what the underlying OS provides.

to run the application on an SMP systems. The most common configuration today is four to eight processors but some systems go much higher.

4. REQUIREMENTS

Does this mean that server applications impose different requirements on memory allocators than traditional single-shot applications, like compilers, word processors, or mail clients? We claim that they do and attempt to explain the requirements in this section.

Consider a web server running on an SMP system with, say, eight processors. Assume that its basic architecture is a simple thread-per-request with a worker pool, that is, when a request arrives it is put on a queue, a worker thread eventually picks it up for processing, and the worker thread does all the processing required to complete the request. Also, assume that the server receives say 1000 requests per second at peak time and that the processing of a request involves 1000 allocations. The peak allocation rate is then 10^6 allocate and 10^6 free operations per second. A good memory allocator must be designed to handle millions of malloc-free operations per second on large SMP systems.

Here is our list of requirements for dynamic memory allocators targeted at any application augmented with requirements specific for server environment.

1. **Fast** – Speed is essential. Allocate and free are the most common operations so they must be very fast. Less frequently used operations like realloc, compaction, or gathering statistics need not be particularly fast.
2. **High memory utilization** – As always, reducing overhead and wasted space is important. However, server applications tend to run on systems with massive amounts of memory so this requirement is not as critical as it might be on smaller systems.
3. **Size independence** – Speed should not be greatly affected by the size distribution and ordering of allocation requests. Given that object sizes can be very varied on server systems, some dependent on the nature of incoming requests, it is important to have size independence.
4. **Maximize locality** – Allocating chunks of memory that are typically used together near each other. This reduces page and cache misses during execution and thus improves performance. This also has the potential of reducing unused space.

These four are all quite standard requirements that apply to all memory allocators. However, the subsequent ones are more specific to server applications.

5. **Scalable** – The allocator must support highly concurrent operations and execution in SMP systems, ideally scaling linearly with the number of processors.
6. **Thread independence** – A block must not be tied to the thread that allocated it. A block should be free to migrate among threads, that is, it should be possible for one thread to allocate a block, a second thread to use it, and a third thread to free it (even if the original thread has already died). This makes it possible to pass objects from thread to thread in a server system. This is a strict requirement for server applications.
7. **Predictable speed** – Servers are (soft) real-time applications and need to exhibit not only low but also predictable response times. This means that the time it takes to allocate or free a block should, ideally, be constant and independent of its size, the number of allocated or free blocks, amount of memory in use, allocation history, etc. In particular, occasional long pauses for garbage collection or large-scale coalescing are unacceptable.
8. **Stability** – For long running systems it is very important that the memory allocator's performance remains stable over time. In other words, memory utilization should not decrease or allocation times increase over time if the load on the system remains stable.

We do not claim that each one of points 5 to 7 is unique to server applications. For example, animation software needs highly predictable speed (but not necessarily scalability) and video conferencing clients are long running and care greatly about stability. We do, however, believe that the combination of requirements is unique to server applications.

5. PRIOR WORK

Many dynamic memory allocators have been designed over the years. Wilson, Johnstone, Neely, and Boles [15]have written an excellent survey of this work. The main focus appears to have been on sequential allocators, with emphasis on speed and memory utilization. Benjamin Zorn maintains a site [16] with links to dynamic memory allocators that are publicly available on the net.

The allocator designed by Doug Lea [10], here called DLmalloc, has been found to be both the fastest and most memory efficient on several applications 9. However, the original version is neither thread-safe nor scalable. Wolfram Gloger created a thread-safe and scalable version, called Ptmalloc [4].

Vmalloc by K-P Vo [14] is more of an allocator framework. Memory is divided into regions and each

region can be managed by different policies. The specific allocators to be used are chosen at link time. This makes it easy to experiment with different allocators for a given application. Grunwald and Zorn [5] also investigated how to tailor allocators to specific programs, mainly to improve speed but without too high a cost in memory space.

We have not found much work on parallel memory allocators. The latest appears to be Arun Iyengar's work [7] [8], which is somewhat difficult to assess because the experimental results are for such an unusual machine (a dataflow machine). His most scalable allocator uses multiple free lists, with a lock on each free list. Operating system kernels for parallel machines also need scalable memory allocators. A paper by McKenney and Slingwine [12] describes the kernel allocator used in Sequent's version of Unix. Somewhat simplified, it has one subheap per processor. No locks are needed on these subheaps because all allocation and deallocation is restricted to one processor. Unfortunately, this idea cannot be applied at the user level because an application program typically has no control over which processor it runs on at any given time.

Extensive research has been done on garbage collection techniques, especially incremental and concurrent garbage collection [18]. Appel *et al* [1] describe concurrent garbage collection technique based on a virtual memory marking technique. This allows concurrency between garbage collection and the mutator (application program). While this and other refinements make garbage collection more efficient, it does not address issues with traditional heap based allocation.

6. LKmalloc

This section describes our design of a memory allocator, called LKmalloc, targeted for both traditional applications and for server applications. We have concentrated on scalability and speed. We describe not only the final design but also some of the not-so-successful attempts along the way.

Doug Lea's allocator [10] has been found to be both fast and memory efficient. We adopted three of its key design features with little change.

A: Binning. Free blocks are kept in 128 bins, grouped by size. All blocks are aligned on 8-byte boundaries. There are 64 bins for blocks of size 512 or less, space 8 bytes apart, each holding blocks of exactly the same size. The remaining (large-block) bins have coarser spacing and a bin may contain blocks of different size. The bins are implemented as doubly linked lists. DLmalloc keeps blocks in large-block bins sorted by size but we did not retain this feature.

B: Approximate best fit. Searching for a free block starts from the first list containing free blocks of sufficient size and proceeds one list at a time. The first free block found is taken. This is a combination of best fit and first fit. If the free block found is on a small-block list, it is guaranteed to be the best-fitting block but not if it is found on a large-block list because large-block list are not kept sorted.

C: Immediate coalescing. When a block is freed, we immediately try to coalesce it with its left and/or right neighbor. This reduces fragmentation and improves memory utilization. It also avoids postponing work, providing more predictable speed, and improves stability. This policy requires that checking whether two blocks can be coalesced and performing the actual coalescing must be very fast. To this end, we use two standard techniques, namely boundary tags and doubly-linked free list. Each block, whether free or allocated, carries size and type information in a 4-byte field at the beginning and at the end of the block. These techniques make it possible to check and perform coalescing in constant time (and very fast).

LKmalloc does not maintain a cache of free blocks, often called quick lists or look-aside lists. The basic operations, including coalescing, are sufficiently fast that there was little to be gained from quick lists and delaying coalescing tends to reduce memory utilization.

DLmalloc organizes each small-block list as a queue (or FIFO list). Large-blocks lists are kept sorted on block size to make best-fit allocation faster. We changed the policy for small-block lists to LIFO, that is, blocks are added to and deleted from the front of the list. The idea is to improve cache locality (both for the allocator and the application) by reusing blocks as quickly as possible, hopefully, before the block has been purged from the processor cache. Keeping large-block lists unsorted and applying first-fit were also adopted so as to reduce cache misses both during searching (fewer free blocks touched) and in the application.

Normally one expects searching to slow down as the number of items searched increases. Here we see the opposite effect. As the number of free blocks increases, there will be fewer empty small-block bins, which reduces the number of such lists that have to be checked.

The remaining key design decisions were driven primarily by the need to support a high level of concurrency.

D: Lock on each free list. A spin lock protects each free list; there are no other locks. The lock on a free list protects additions to and deletions from the free list, nothing more. No locks are held when checking whether blocks can be coalesced, nor when coalescing the blocks.

We use our own spin lock implementation, shown below. It is designed to minimize load on the memory bus by performing the lock testing in the cache during a busy wait. When the lock becomes free, the thread tries to acquire it. (InterlockedExchange is a WIN32 funtion that atomically sets a variable to a new value and returns the old value.) If it fails to acquire the lock, it continues spinning. Busy waits are bounded, i.e. a thread yields after it has tested the lock a maximum number of times. The bound was set to 4000 times in all our tests.

```
_inline  static int S_LOCK( long *laddr )
{
  int cnt, sleeps=0 ;
  do {
    cnt = MAXSPIN ;
    /* check max MAXSPIN times then yield */
    /* spinning in cache until lock changes */
    while( *laddr == LOCKED ) {
      cnt-- ;
      if( cnt < 0 ){
        sleeps++ ;
        Sleep(0) ; /* yield */
        cnt = MAXSPINS ;
      }
    }
  } while( InterlockedExchange(laddr, LOCKED)
           == LOCKED ) ;
  return(sleeps) ;
}
```

E: Multiple subheaps. Having a lock on each free list and minimizing lock time reduced contention on the free lists to virtually nothing. Even so, it wasn't enough – we found that the speedup was still less than two on a 4-way or 8-way processor.

The problem is caused by "cache sloshing", that is, the current value of a cache line rapidly migrating from cache to cache. When different processors read and modify the same cache line frequently, the current value is almost never available in the cache when needed by a processor, resulting in a cache miss and a memory read. Cache sloshing occurs not only on free list locks and headers but also on boundary tags.

To combat this, we decided to use multiple, independent subheaps and assign each thread to a subheap. A subheap is a complete heap with its own set of 128 free lists and memory arena. The address space is divided into "stripes" of fixed size (currently 4MB). An arena grows and shrinks one stripe at a time.

A thread always allocates blocks from its assigned subheap but can free blocks in any subheap. To which subheap a block belongs can be determined from its address by first determining which stripe (explained below) it belongs to and then looking up to which subheap that stripe is assigned.

In the current implementation, the number of subheaps must be determined when the library is initialized and remains fixed thereafter. Choosing the number of subheaps is an open question but setting it slightly higher than the number of processors is a reasonable first heuristic.

F: Select subheap by hashing. Given a call to allocate a block for a particular thread, how do we quickly decide which subheap to use? LKmalloc does it by hashing on the ID of the thread. This has the virtue of being fast and simple because the library maintains no information about existing threads. The drawback is the pseudo-random assignment of threads to subheaps: multiple concurrent threads may be assigned to the same subheap even though there are currently unused subheaps. However, any other scheme requires some explicit bookkeeping and assignment of threads to subheaps. It is not clear that an explicit scheme would do much better than random assignment.

G: Memory striping. We must be able to grow and shrink the memory area assigned to a subheap. A fixed division of the address space won't do; a subheap must, if necessary, be allowed to grow arbitrarily large. LKmalloc divides the address space into fixed-size stripes (default size 4 MB). When a subheap needs more space it is assigned another stripe. A subheap can also return a stripe. A small array keeps track of which stripes have been assigned to which subheaps.

Initially, no physical memory is committed to a stripe. Physical memory is requested from (committed) and returned to (decommitted) the operating system one page at a time. The amount of memory committed to a stripe and to which pages within as stripe varies over time depending on demand. When to decommit memory is a policy decision. The current version decommits pages whenever a free block larger than two pages is created.

6.1 Discussion

As far as we know, the idea of using a fixed number of subheaps and assigning threads to subheaps by hashing is new. Here is a brief explanation of why we adopted this solution.

Let's first consider the two extremes: a single subheap used by all threads and a completely separate subheap for each thread. We tried using a single subheap and found that it scaled poorly even if when lock contention was reduced to virtually nothing. As mentioned above, the problem is probably bus saturation caused by many processors accessing the same fast-changing data items (locks, free list headers, block headers and footers).

Using a subheap per thread is not viable for server applications because the overhead of keeping track of threads, creating/destroying subheaps and mapping threads

to subheaps could be substantial. Furthermore, a block might survive longer than the thread that created it by being transferred to another thread. If so, it is not clear when to destroy a subheap and, most likely, memory utilization would suffer.

Using a small number of subheaps and assigning threads to subheaps by hashing improves scalability without having to explicitly keep track of the mapping of threads to subheaps.

7. EXPERIMENTAL RESULTS

This section reports on the observed performance of five allocators on a simulated server-like workload. We compare the performance of two versions of LKmalloc with four other allocators.

1. **Lkmalloc with 10 subheaps**. This version is identified by "Lkmalloc 10" in subsequent figures.
2. **LKmalloc with a single heap**. Results for this version are labeled "Lkmalloc 1" in the figures.
3. **DLmalloc with a global lock**. This is Dlmalloc (Doug Lea's malloc), which we made thread-safe with the addition of a single, global lock. The lock is implemented as a bounded spin lock. The label "dlm, global" in the figures refers to this allocator.
4. **DLmalloc with local locks**. This is another thread-safe version of DLmalloc with no global lock but instead using a lock on each of its 128 free lists. We had to slightly reorder the code and eliminate one speed-enhancing device (a bit array used for quickly finding a non-empty free list), which slowed down the allocator somewhat. The label "dlm, local" is used for this allocator in the figures.
5. **Ptmalloc**. This is a third thread-safe version of DLmalloc, designed by Wolfram Gloger [4], and intended to be scalable. It uses a linked list of subheaps where each subheap has a lock, 128 free lists, and some memory to manage. When a thread needs to allocate a block, it scans the linked list of subheaps and grabs the first unlocked one, allocates the required block, and returns. If it can't find an unlocked subheap, it creates a new one and adds it to the list. In this way, a thread never waits on a locked subheap. Ptmalloc has no provisions for reducing the number of subheaps so memory utilization is likely to suffer, especially for long running server applications.
6. **Libc malloc**. This is the allocator distributed in the standard C library of Microsoft Visual C++ 5.0. It is thread-safe, which is accomplished by means of a single, global lock (actually a WIN32 critical section). It was not designed for scalability on SMP systems.

Workload. The workload generated is intended to model a server responding to client requests. It is based on a thread-per-request model, creating a new thread for each request. The number of threads running concurrently is an input parameter.

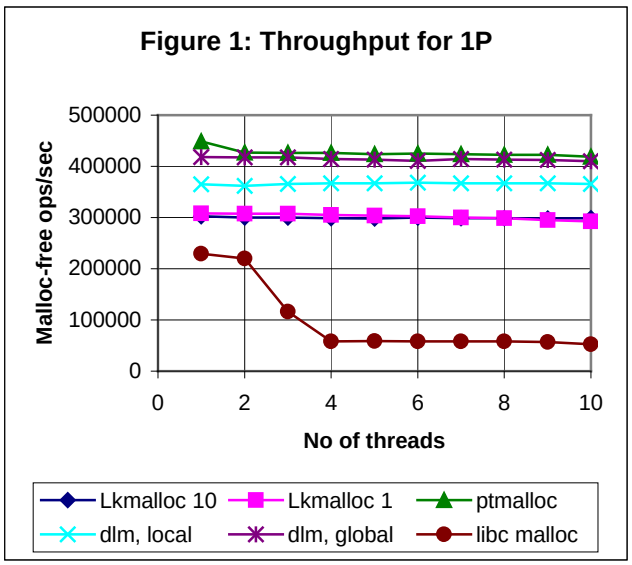
When a worker thread is created, it receives a set of blocks already allocated. It then performs a sequence of random replacements, that is, one of the existing blocks is randomly selected and freed and a new block of random size is allocated. We call this a malloc-free operation. How many such operations a thread performs is another input parameter. When the thread has completed its sequence of malloc-free operations, it creates a new thread, passes its currently allocated blocks to the new thread, and terminates. After 30 seconds, all activity was stopped and performance data collected. Threads do nothing else so even the slow allocators perform several million malloc-free operations during 30 seconds.

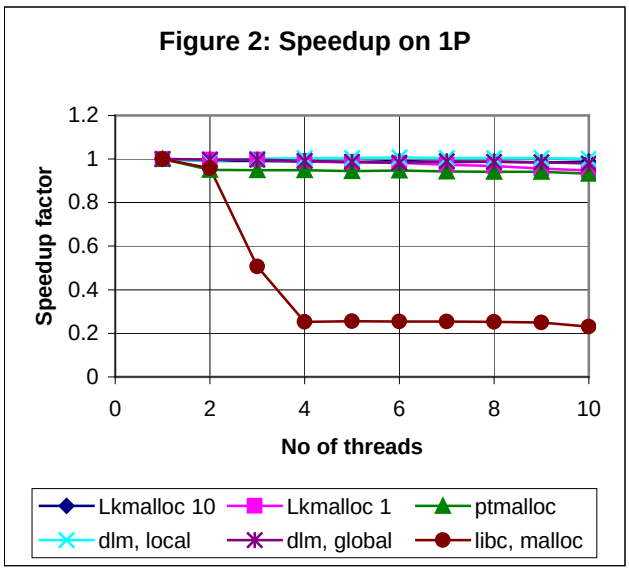
Why pass allocated blocks between threads? In server applications, a small fraction of the blocks allocated by one thread is typically passed to other threads, used in some way, and finally freed by some thread other than the creating thread. We refer to this as blocks "bleeding" between threads. On a large web server, we observed bleeding in the 2-3% range.

For the series of experiments reported here, the input parameters were set as follows. Each thread received 1000 blocks and performed 50,000 malloc-free cycles (2% bleeding). Block size was randomly drawn from a uniform distribution with range 10 to 1000 bytes. Experiments were run on three different machines: a uniprocessor system with a 300 MHz Pentium II processor, on a 4-processor SMP with 200 MHz Pentium Pro processors, and an 8-processor SMP also with 200 MHz Pentium Pro processors. All systems were running Windows NT 4.0. The number of concurrently executing threads was varied from one to ten.

Metrics. We are mainly interested in how fast the allocators are, how well they stand up to increased levels of concurrency, and how effectively they use memory, which we assess by the following metrics:

1. *Throughput* in total malloc-free operations per second (malloc-free pairs per second).
2. *Speedup*, defined as throughput when running *n* threads in parallel, divided by throughput when running only one thread at a time (using the same allocator).
3. *Memory utilization*, defined as the fraction of the total arena size (memory requested from the OS) occupied by application data. This metric takes into account both external and internal fragmentation.



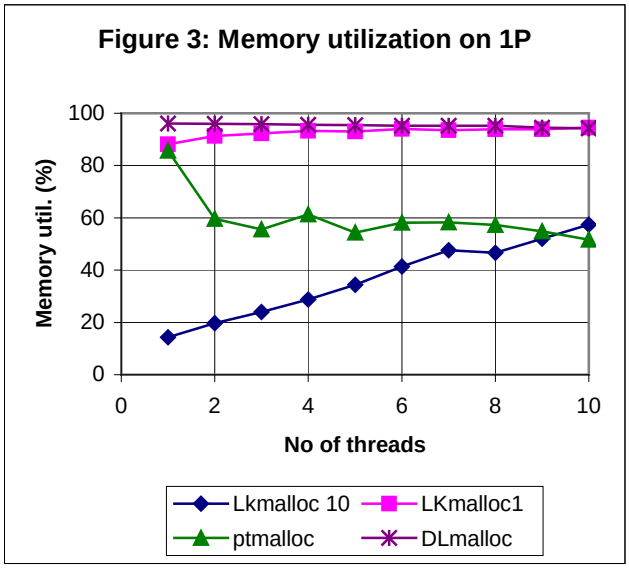


7.1 Results for 1P

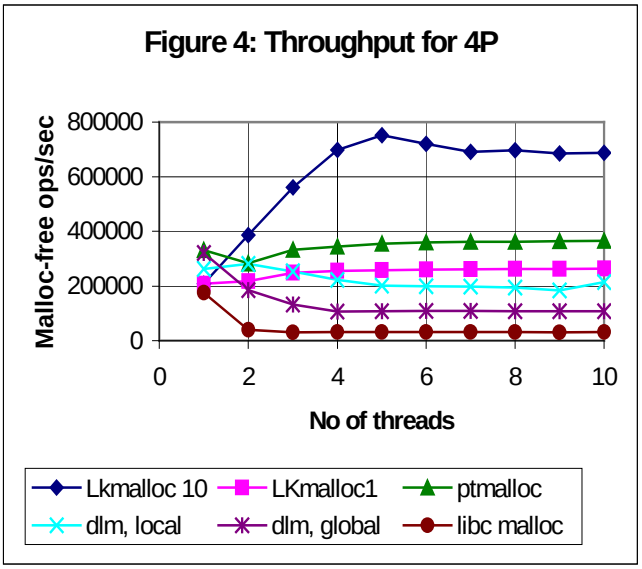
Figures 1 to 3 show the results obtained on a uniprocessor system. Not surprisingly, DLmalloc with a single global lock and Ptmalloc have the highest throughput because they retain all the performance optimizations of the serial version of the code. DLmalloc with local locks is slower because some of these optimizations had to be eliminated when adding the locks. LKmalloc is slower yet because of the overhead caused by retrieving and hashing the thread ID for every malloc call and locating the appropriate subheap for every free call. Libc malloc is the slowest allocator, especially when the number of concurrent threads is high.

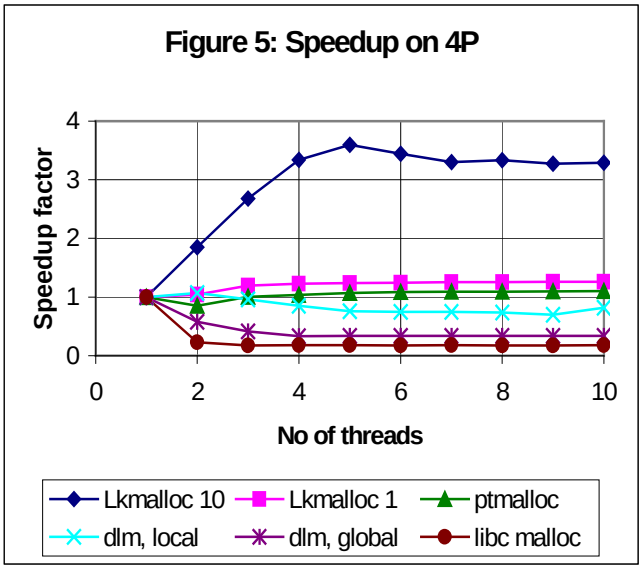
Figure 2 show the speedup. In the uniprocessor case, the objective is to achieve the same throughput regardless of the number of threads. Libc malloc clearly does not achieve this. Relying on a single critical section for synchronization causes too many context switches, thereby reducing throughput. Interestingly enough, using a single (bounded) spin lock for synchronization does not have this effect, witness the line for “dlm, global”.

Figure 3 plots the memory utilization, except for libc malloc for which we couldn’t get the data easily. DLmalloc, regardless of lock type, and LKmalloc with a single heap have high and stable memory utilization for all levels of concurrency. Ptmalloc and LKmalloc with multiple subheaps are not as memory efficient. Ptmalloc tends to waste memory because it creates a new subheap and arena as soon as there is a lock conflict. Also, it has no policies or mechanisms for eliminating subheaps.



LKmalloc 10 spreads the allocated blocks over ten subheaps, regardless of the number of concurrently executing threads. The low memory utilization for low levels of concurrency may be caused by the design of the experiments but not for higher levels of concurrency. A highest memory utlization of less then 60% is clearly a problem. We run some experiments (not shown) increasing the number of concurrently executing thread beyond ten but memory utlization did not improve.





7.2 Results on 4P

Figures 4 and 5 show interesting scalability effects: we have allocators with negative scaling, allocators with flat performance and one allocator with significant positive scaling.

The two allocators relying on a single global lock (libc malloc and DLmalloc with a global spin lock) have clear negative scaling, that is, as the number of threads increases, the total throughput *decreases*. Three allocators (DLmalloc with local locks, Ptmalloc and LKmalloc with a single subheap) gain virtually nothing from the extra processors. Their throughput appears to be limited by bus saturation caused by cache sloshing. LKmalloc with multiple subheaps is the only one gaining signficantly from the additional processors. A maximum speedup of 3.5 on a 4 processor system is outstanding.

Memory utilization is plotted in Figure 6. As one would expect, Ptmalloc's memory utilization is lower than for the uniprocessor case. With four processors, there is a higher probability of finding no unlocked subheaps and creating a new subheap. This results in more subheaps and a lower memory utilization.

7.3 Results for 8P

The 8P results shown in figures 7 to 9 (on next page) amplify the findings for 4P. The allocators relying on global locks continue to scale negatively and the same three allocators gain virtually nothing from the seven additional processors. LKmalloc with 10 subheaps continues to scale well, reaching a maximum speedup 5.3 out of 8. We had set a goal of a million malloc-free operations per second and reached 1.1 million.

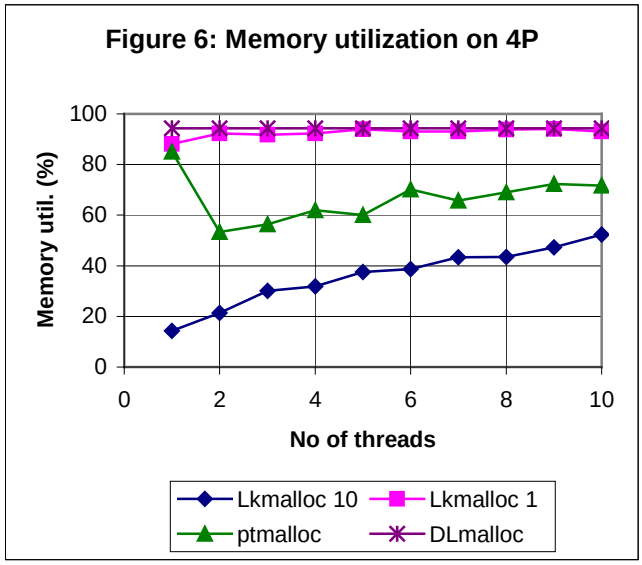
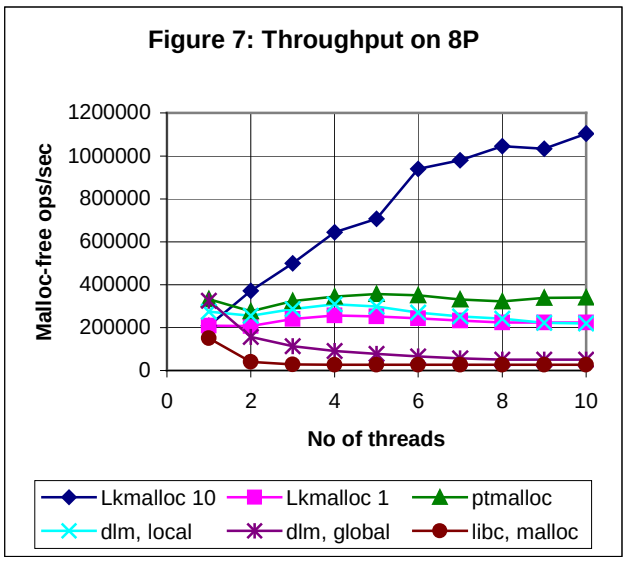
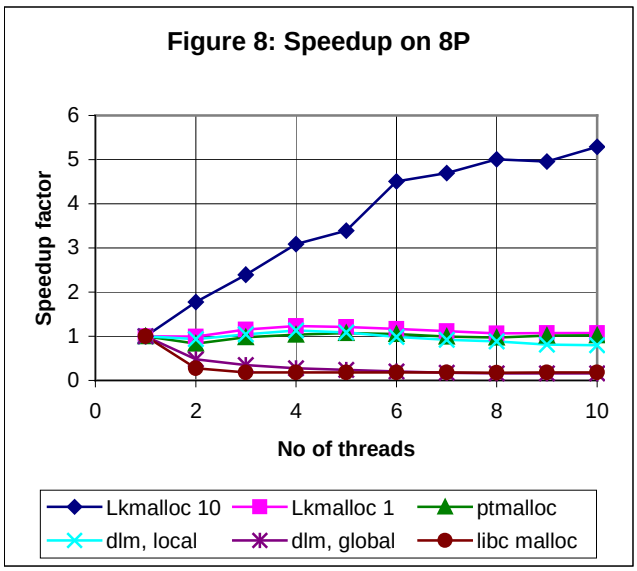


Figure 9 shows the same pattern for memory utilization as figures 3 and 6. The dip in the Lkmalloc 10 curve for 9 and 10 does not indicate a trend – it did not show up in other runs. Again, Lkmalloc shows a high memory utilization of only 60%.





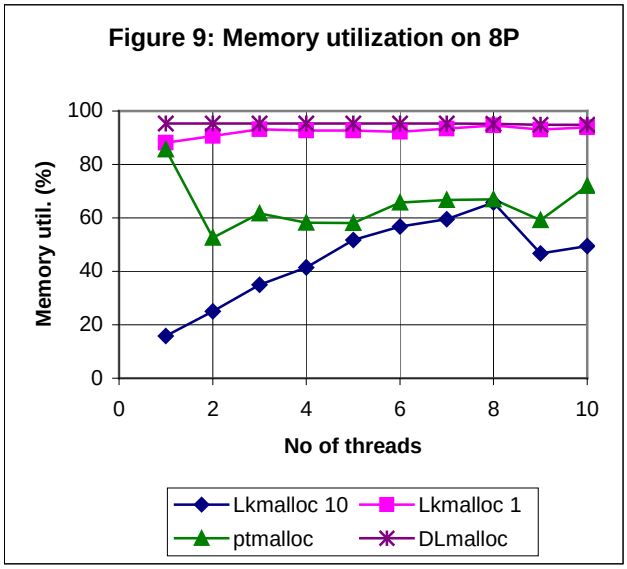
8. Summary and conclusion

Long-running server applications behave differently than traditional client applications and impose different requirements on dynamic memory allocators. We first described what server applications typically do and how they are architected. We identified several additional requirements on memory allocators intended for server applications: good scalability on SMP systems, thread independence, stability, and predictable performance.

We then described the design of a new allocator called LKmalloc targeted for both traditional applications and server applications. LKmalloc uses several subheaps, each one with a separate set of free list and memory arena. A thread always allocates from the same subheap but can free a block belonging to any subheap. A thread is assigned to a subheap by hashing on its thread ID. This slightly complex scheme was necessary to achieve any significant scalability – all simpler schemes failed.

On uniprocessor systems, LKmalloc was about 25% slower than the fastest allocator we tested. On 4-way and 8-way SMP systems, LKmalloc scaled far better than any of the other allocators tested. Two allocators relying on a single global lock showed negative scaling on SMP systems, that is, throughput decreased when the number of processors increased. Allocators with multiple locks but a single common arena showed positive but limited scaling (a speedup of less than 1.5 on an 8-way SMP).

The main lesson learned from our experiments is as follows. *To build a highly scalable allocator, it is not sufficient to minimize lock contention. One must also reduce bus traffic by reducing the frequency of access to shared, fast-changing data items like list heads, counters, block headers and block footers.*



LKmalloc's memory utilization is definitely too low when multiple subheaps are used. For ten subheaps and five or more concurrent threads, it needed almost twice as much memory as the most memory efficient allocators tested. We are looking into ways of automatically adjusting the number of subheaps based on the level of concurrency exhibited by the application. The problem is how to improve memory utilization without losing too much in scalability.

We believe that LKmalloc results in fewer cache misses not only within the allocator itself but also in application code. However, additional experiments are needed to determine whether this is true.

9. References

[1] Andrew W. Appel, John R. Ellis, and Kai Li, Real-time concurrent collection on stock multi-processors, *ACM SIGPLAN Notices*, 23(7): 11-20, 1988.

[2] David Detlefs, Al Dosser, and Benjamin Zorn, Memory Allocation Costs in Large C and C++ Programs, *Software Practice and Experience* 24(6): 527--542, June 1994.

[3] J. S. Fenton and D. W. Payne. Dynamic storage allocations of arbitrary sized segments. In Proc. IFIPS, pages 344--348, 1974.

[4] Wolfram Gloger, Dynamic memory allocator implementations in Linux system libraries, http://www.dent.med.uni-muenchen.de/~wmglo/malloc-slides.html (site visited May 11, 1998)

[5] Dirk Grunwald and Benjamin Zorn, CustoMalloc: Efficient Synthesized Memory Allocators, *Software: Practice and Experience*. 23(8): 851--869, August 1993.

[6] Dirk Grunwald and Benjamin Zorn and Rob Henderson, Improving the Cache Locality of Memory Allocation, *ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pp 177--186. Albuquerque, NM. June 1993.

[7] Arun K. Iyengar, Parallel dynamic storage allocation algorithms, In *Fifth IEEE Symposium on Parallel and Distributed Processing*. IEEE Press, 1993.

[8] Arun Iyengar, Scalability of Dynamic Storage Allocation Algorithms, In *Frontiers '96 - The 6th Symposium on Frontiers of Massively Parallel Computing*, IEEE Computer Society Press. Pages: 223-232.

[9] Donald E. Knuth. Fundamental Algorithms, Vol. 1 of The Art of Computer Programming, chapter 2, pages 435--451. Addison Wesley, Reading, MA, 2nd edition, 1973.

[10] Doug Lea, A memory allocator, http://g.oswego.edu/dl/html/malloc.html (site visited May 11, 1998).

[11] B. W. Leverett and P. G. Hibbard. An adaptive system for dynamic storage allocation. *Software Practice and Experience*, 12(6): 543--556, June 1982.

[12] Paul E. McKenney and Jack Slingwine. Efficient kernel memory allocation on shared-memory multiprocessors. In *USENIX Conference Proceedings*, Berkeley CA, February 1993.

[13] C. J. Stephenson. Fast fits: New methods for dynamic storage allocation. In Proceedings of the Ninth ACM Symposium on Operating System Principles, pages 30--32, Bretton Woods, NH, October 1983.

[14] Kiem-Phong Vo, Vmalloc: A general and efficient memory allocator. *Software Practice and Experience*, 26(3), 357-374, 1996.

[15] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In 1995 International Workshop on Memory Management, Kinross, Scotland, UK, 1995. Springer Verlag LNCS.

[16] Benjamin Zorn, Malloc/free and GC implementations, http://www.cs.colorado.edu/~zorn/Malloc.html (site visited May 11, 1988).

[17] Benjamin Zorn and Dirk Grunwald, Evaluating Models of Memory Allocation, *ACM Transactions on Modeling and Computer Simulation*. 4(1): 107--131, January 1994.

[18] Richard Jones, and Rafael Lins, *Garbage Collection: Algorithms for automatic dynamic memory management*, John Wiley & Sons, 1998