PALLOC: PARALLEL DYNAMIC MEMORY ALLOCATION

BY

PATRICK ALEXANDER SIMMONS

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Adviser:

Associate Professor Vikram Adve

Abstract

In this thesis, we describe two related memory allocators, each with novel properties. PALLOC1 contributes a unique strategy based on the traversal of a parallel tree data structure for allowing concurrent allocations and frees to proceed within a single thread's heap. PALLOC1 also provides a novel, provable guarantee limiting the allocator's requests for more memory from the operating system to only those situations where no contiguous block is available to satisfy the allocation request, and a pure bitmap allocation strategy speed-competitive even for sequential codes with the boundary-tag / binning strategy used in dlmalloc. We find that, for larger allocation patterns, our implementation exhibits competitive base performance relative to other parallel allocators, superior scaling, and better resistance in practice to fragmentation.

PALLOC2 contributes a second unique strategy for memory allocation based on bitmap allocation into variable-sized superpages. Our system provides the runtime with the useful ability to, given an arbitrary heap address, find both the start of the heap allocation and the size of the object allocated. Thus, PALLOC2 provide the capabilities of baggy bounds checking with no performance impact. In fact, we find that, for both sequential and parallel programs, PALLOC2's performance is superior to PALLOC1 and to other state-of-the-art allocators including Hoard, DLMalloc, and Streamflow for allocations of all sizes.

To my brother Sloane, whose life I value as my own.

Acknowledgments

First and foremost I would like to thank Stephen Kofsky and Alessandro Bellina, with whom I completed the initial class project that spawned this work, for their invaluable assistance in the development of PALLOC1. I would also like to thank my adviser, Vikram Adve, for his assistance in supervising and reviewing this thesis.

Table of Contents

List of Figures

1 Introduction

Many programs require dynamic memory allocation, traditionally provided by the malloc() and free() functions of the C standard library. Because a common method of making these functions thread-safe is the use of a single global lock [1], dynamic memory allocation often creates a serialization point. As the number of cores on modern machines continues to increase, this serialization point will represent a more and more significant bottleneck. There has been significant prior w[ork on parallel dynamic memory allocation [5] [14] [12] [3] [6] [13] [7]; this thesis builds on the lessons learned from this work.

PALLOC1 contributes a unique strategy based on the traversal of a parallel tree data structure for allowing concurrent allocations and frees to proceed within a single thread's heap. PALLOC1 also provides a novel, provable guarantee limiting the allocator's requests for more memory from the operating system to only those situations where no contiguous block is available to satisfy the allocation request, and a pure bitmap allocation strategy speed-competitive even for sequential codes with the boundary-tag / binning strategy used in dlmalloc. We find that, for larger allocation patterns, our implementation exhibits competitive base performance relative to other parallel allocators, superior scaling, and better resistance in practice to fragmentation.

PALLOC2 contributes a novel strategy for memory allocation based on bitmap allocation into variable-sized superpages. Our system provides the runtime with the useful ability to, given an arbitrary heap address, find both the start of the heap allocation and the size of the object allocated. Thus, PALLOC2 provide the capabilities of baggy bounds checking with no performance impact. In fact, we find that, for both sequential and parallel programs, PALLOC2's performance is superior to PALLOC1 and to other state-of-the-art allocators including Hoard, DLMalloc, and Streamflow for allocations of all sizes.

We compare against Hoard [14], the system allocator (often based on dlmalloc [9] or ptmalloc [5]), and ptmalloc [5]. When evaluating PALLOC2, we also compare against Streamflow [12] and TCMalloc [3]. dlmalloc [9] is a sequential memory allocator developed by Doug Lea, who has done extensive research into malloc implementations over the course of many years. It is currently used by several Unix systems as the system implementation for malloc(). dlmalloc uses a boundary-tag / binning strategy for allocation and is unique among the allocators discussed in that, were it to disable delayed coalescing, it could provide our guarantee regarding memory usage. Ptmalloc [5] is a variant of dlmal-

loc modified by Wolfram Gloger to support parallelism. A modified version of this malloc() is included in the GNU C Library as the standard allocator for Linux. [5] We find that, for larger allocation patterns, our implementation not only exhibits superior scaling relative to other parallel allocators but is also better resistant to fragmentation in practice. As both of the allocators described in this thesis implement the traditional malloc() C API call, PALLOC1 and PALLOC2 are applicable to all shared memory, cache-coherent architectures.

2 Other Allocation Work

This section briefly discusses other work in parallel memory allocation.

2.1 TCMalloc

TCMalloc [3], developed at Google, uses a size-class based approach, like PALLOC2, and uses a central heap with per-thread caching. This design decision contrasts with most other parallel allocators, which use full per-thread heaps. Deallocation is performed by inserting deallocated objects into per-thread free list caches.

2.2 Hoard

Hoard's implementation uses a single global heap and P sub-heaps, where P is the number of processors on the machine. Hoard uses fixed-length superpages and headers for allocation and deallocation.

2.3 Streamflow

Streamflow [12] focuses on reducing the overhead necessary for synchronization and remote frees and follows a lock-free design. Streamflow avoids locking during allocation and local frees by only allowing a thread to access its own data structures, except that it may access a queue owned by another thread in order to place a request that that thread perform a remote free. The Streamflow implementation uses fixed-length superpages and a BIBOP table to allow small allocations to be processed without headers.

2.4 CustoMalloc

CustoMalloc [6] profiles the behavior of applications and designs a customized malloc() implementation for those applications. One wishing to apply this strategy to PALLOC could generate statistics regarding the allocation sizes requested by the program and use this information to tune the PALLOC_WORD and PALLOC_PAGE_SIZE knobs. The behavior of these knobs is discussed elsewhere in this paper.

2.5 VMalloc

VMalloc [13] extends the traditional malloc() interface to allow programmers to designate allocations to occur in entities called *regions*. Each region has a specified allocation strategy, such as best fit or same-size-only allocation.

PALLOC1 is capable of being integrated with a memory pool allocator [8]. The compiler, rather than the programmer, will designate allocations to occur in *pools* with bounded lifetimes. Rather than one heap per thread, PALLOC1 will in this case create one heap per pool. The locking strategy used by PALLOC1 is already entirely compatible with this design. Special-casing PALLOC1's behavior to take advantage of properties of these pools is interesting potential future work.

3 PALLOC1 Design and Implementation

3.1 Parallel Tree Traversal

PALLOC1 allocates a distinct heap for each thread; however, other threads are allowed to access a thread's heap in order to perform a remote free. Each heap is organized into multiple "pages" of a fixed size; these are similar to the superpages of Streamflow [12]. In order to perform an allocation or a free, a *page record* structure describing the block in which the allocation or free will occur must be modified. This structure can be found through a search of a parallel tree data structure created for each heap, although optimizations described later are used to avoid this in the case of allocations. Since multiple threads can search this tree simultaneously, multiple allocations and frees can occur so long as they touch different pages.

In the current implementation, an AVL tree with a reader/writer lock is used for the parallel tree. The writer lock is only taken during insertion or deletion of a page record. Because pages are so large, this is an infrequent occurrence.

3.2 Bitmap Allocation within Pages

Within each page record is a bitmap containing information on whether each chunk of size PALLOC_WORD has been allocated. This bitmap is searched in order to find a chunk of memory large enough to satisfy the allocation request. Large allocations are special-cased as described later.

The decision to use a bitmap strategy inside each page, as opposed to binning or some other strategy, was made in order to minimize fragmentation without the need for coalescing. This implementation decision is entirely orthogonal to the parallel tree design described above. Another strategy could be substituted without changing the parallel tree aspect of the design.

3.3 The PALLOC1 Memory Usage Guarantee

PALLOC1 makes the following guarantee regarding memory usage: *If there exists a PALLOC_WORD-aligned contiguous chunk of memory not crossing a page boundary large enough to satisfy an allocation request among the currently allocated pages of a thread, PALLOC will satisfy that allocation request without*

allocating an additional page. Moreover, PALLOC will always assign the lowest-addressed such chunk available in a given page.

Due to delayed coalescing [9] in dlmalloc (and therefore ptmalloc) and the global-local heap design of Hoard, no other allocator we compare against makes a similar guarantee, and we believe this guarantee to be novel.

3.3.1 Proof of Guarantee

Assume the existence of such a contiguous chunk of memory in an allocated page. Then, such a page must be part of the AVL tree. If PALLOC1 does not satisfy the allocation from the recent-free or recent-success heuristic lists, PALLOC1 will search the entire AVL tree for a chunk large enough to satisfy the allocation request. Because the chunk is word-aligned, PALLOC1's bitmap searching algorithm will find this free chunk if no other chunk in the AVL tree is available. If another chunk is available, PALLOC1 will use it iff it comes before all other chunks in the page as PALLOC1's bitmap searching algorithm always searches a page from low memory addresses to high. Since PALLOC1 will never allocate another page unless its AVL tree search fails, PALLOC1 will never allocate an additional page when the conditions of the guarantee are met and will never allocate a chunk inside a page starting with a higher memory address when a chunk in the same page starting with a lower memory address is available.

3.3.2 Usefulness of Guarantee

In environments with hard constraints on memory usage, dynamic memory allocation is often disallowed entirely. The existence of this guarantee allows, in principle, the proof that a particular program will not violate the guarantee. By setting PALLOC_FREEBIRD to 1 (and thus keeping no excess pages allocated) and calculating the additional memory used by PALLOC1's auxiliary structures (which varies from system to system and with the values of PALLOC_WORD and PALLOC_PAGE_SIZE), a programmer can bound the total memory that may be consumed by dynamic allocations. Using the "lowest address first" guarantee, a programmer may be able to analyze his programmer's memory allocation pattern to prove a limit on the fragmentation his program will cause. After proving this limit, a programmer can make use of the "no excess page" guarantee to prove a total limit on his program's dynamic memory usage. This may allow a programmer to make use of the convenience of dynamic memory allocation in situations where the use of such functions would be inappropriate in the absence of memory usage bounds.

3.4 PALLOC1 Implementation

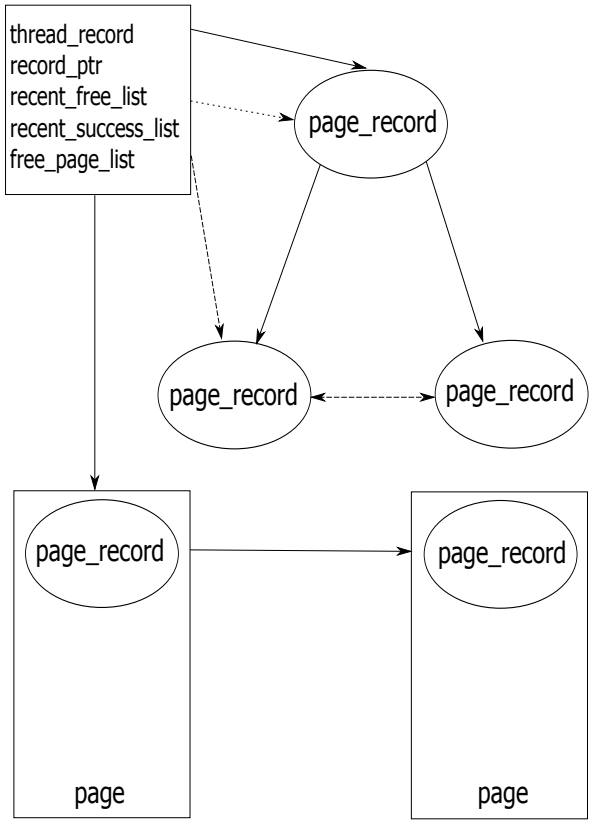

Figure 3.1: Diagram of PALLOC1, showing AVL tree, doubly linked lists, and free page list (with page records for freed pages contained at the top of the pages themselves). The dotted lines are pointers to lists inside of the AVL tree.

3.4.1 Interaction with Operating System

All memory used by PALLOC1 is either located in the data segment or stack or is an anonymous system page mapped into memory with the mmap() system call. The parallel malloc implementation never calls brk(), sbrk(), or the system malloc().

3.4.2 Data Structures

Internal Malloc for PALLOC1

PALLOC1 requires space to store its metadata. Its internal malloc implementation provides this space. It would be possible to instead use the beginning of each page mapped for user memory to store this metadata, but that would cause a significant slowdown for mallocs that are exactly the size of a page, and

we currently believe this to be an important use case. The internal malloc is only used for and can only provide memory in chunks exactly equal to the size of a page_record. PALLOC1's internal malloc uses a simple free list scheme to keep track of these chunks.

Thread Records

The API for PALLOC1 requires an initialization function be called with the maximum number of threads that will be used by the program. Using this information, PALLOC1 uses the mmap() system call to create a page where it stores an array of structures describing each thread. It gives this information to the internal malloc described above.

Each of these structures has a pointer to the root of a binary AVL tree of page records describing the pages malloc()ed by that thread and another pointer to a "free list" of pages, described later. This AVL tree is protected by a reader/writer lock scheme, described later, allowing multiple lookups to be done in parallel. A thread record also contains pointers to the heads of two doubly linked lists of page records. These lists are the "recent free list" and the "recent success list" and are described below. The lists are allowed to connect arbitrary nodes within the AVL tree but must not contain any nodes that do not exist in the AVL tree.

Every thread record also contains an array of integers equal to the maximum number of threads in the system. This array ranks the likelihood that the next free() called from that thread will have that ID. The thread puts itself in slot 0 and fills the rest of the array with an arbitrary permutation of the remaining indices. An optimization could use some pattern to speculate about the relationship between thread ID and sharing topology, if any exists, but currently the initial pattern for thread `n` out of `N` total threads is simply `0,...,n-1,n+1,...N`. At every free(), the thread ID that owned the memory being freed will swap places in the array with its predecessor unless it is already in array slot 0, allowing for dynamic adaptation to runtime sharing topologies.

We free a thread record upon a thread's exit. The thread record is placed on a free list of thread records and is used on the next call to pthread_create.

Page Records

A page record stores the metadata for a page that a particular thread has allocated. The structure is used as a node in an AVL tree and as a node in at most one of two doubly linked lists and at most one singly linked list. The AVL tree is keyed on the memory address of the page being described.

The page record also contains a bitmap describing whether each chunk of PALLOC_WORD bytes (the smallest unit of space allocated by PALLOC1) is in use or not and a simple lock for this bitmap. The size of the allocation is stored as a 32-bit value directly before the address returned by the allocator.

In order to make sure this space is always free, the first PALLOC_WORD-byte chunk is never allocated (except in a large allocation, described below), and all allocations are treated as if they were 4 bytes longer than requested.

Also inside the page record is a 32-bit integer containing either zero, in which case it is ignored, or the smallest allocation size that has, through prior attempt and failure, been found not to be allocable within the page. In the event an allocation fails in the page, the size of that allocation is stored in this 32-bit integer in the page record. No allocations of this size or greater will be attempted until a free() occurs and this integer is reset to zero.

Synchronization Mechanisms

Because of the goals of this project, the synchronization mechanisms in our implementation were carefully designed in order to ensure that calls to malloc() and free() do not create sequential bottlenecks in parallel code. We built a library implementing simple and reader-writer locks, the high-level synchronization constructs needed for our purposes, for GCC/x86 and Sun Studio/SPARC. Spinlocks are used in both implementations.

Each thread record's AVL tree is protected by a reader-writer lock. The writer lock is taken only when adding or removing page records from the tree. A reader lock is taken whenever accessing the tree. Thread records also hold simple locks controlling access to each of the heuristic lists, the list of free page records, and the list of free pages.

The only lock held by a page record is a simple lock protecting its bitmap structure. This lock is taken whenever reading or updating a page record's bitmap.

3.4.3 Allocation

Small Allocations

It is expected that the majority of allocations will be "small", defined as less than a page in size. In a small allocation, a thread checks its list of recently freed pages (taking out a reader lock on its page record AVL tree) and searches its bitmap for a series of consecutive PALLOC_WORD-byte chunks large enough to hold the requested allocation. The efficiency of this search is increased through several methods, including storage of the index of the first zero in the bitmap in the page record, use of a variant of the Boyer-Moore string searching algorithm optimized for strings of the form "00*", and the use of bitmasks when testing the uint64_t variables in the bitmap.

If an allocation fails on a page in the recently freed list, that page is removed from the recently freed list. If it succeeds, it is removed from the recently freed list but placed as the head of the list of recent successes. If the list of recently freed pages is exhausted and space still has not been found for the allocation,

the list of pages where an allocation recently succeeded is traversed in the same manner. Our profiling results indicate that the vast majority of allocations are satisfied from one of these two heuristic lists, however, if this list is also exhausted, the entire tree is searched in a preorder traversal. Note that there is no inefficiency in attempting to allocate inside of the same page twice, for pages that were on the free list but failed, because their page records will already have set exactly the size of the attempted allocation in their "will not fit allocations of this size or greater" field by this point.

If the allocation still fails, a new page will be mmap()ed, or taken from the thread record's "list of freed pages" if this list is not empty, and a writer lock will be taken out on the AVL tree. After the lock is acquired, the page will be inserted into the AVL tree, and the tree will be rebalanced if necessary. For PALLOC1 configurations optimized for speed, we believe that exponentially increasing the number of pages requested per each mmap() may lead to less system call overhead. This optimization has not yet been implemented.

This could become a sequential bottleneck if multiple threads are frequently trying to free pages from the same thread's AVL tree, or if the owning thread is attempting to allocate a page while another thread is attempting to free one. However, since it is expected that most allocations and frees occur at below the granularity of a page, we do not expect this to be a problem in practice. Our profiling results indicated that the majority of time in our allocator is spent inside the code to find available chunks within a page, reinforcing our view.

If contention on the writer lock is found to be an issue by later implementors, a reader lock could instead be taken at first, with a writer lock taken later from within the AVL tree insertion code only if rebalancing is later found to be necessary. If contention were still a problem, a red-black tree could be used in place of the AVL tree, since it requires reblancing less often (at the cost of slower lookups), or the self-balancing tree could be replaced with a data structure explicitly designed to allow parallel access.

Large Allocations

A large allocation is an allocation of a page or more in size. For an allocation of exactly one page in size, a page is mmap()ed or taken from the thread record's list of freed pages; it is inserted into the AVL tree; the bitmap is put into a state such that the bitmap allocator knows it is unavailable; and, the address of the start of the page is returned. It is ensured that the singly linked list pointer – which is used to record pages necessarily contiguous in memory – is set to NULL. The 32-bit integer variable used to hold the maximal allocation size not possible in a small allocation is set to 1 since this variable is used to represent the size of the singly linked list in large allocations. Later, when a free() is called, it will be clear to PALLOC1 that this allocation was exactly one page in size, since it is not part of a list of contiguous pages, yet the requested

free address is equal to the address of the start of some page.[1] The page may then be freed or added to the thread's free list of pages.

For allocations larger than a page in size, the free list of pages is searched for a list of contiguous pages large enough. If one is found, those pages are used. Otherwise, as many pages as necessary are mmap()ed contiguously. Page records are created for each one and placed in an ordered linked list with the page with the lowest memory starting address as the head. The head of this linked list only is added to the AVL tree. In a free(), the head is removed from the AVL tree and placed on the list of freed pages. The other pages in the linked list are not touched, but their number remains known since it is stored in the head.

3.4.4 Freeing

In the previous section, we discussed how free() is implemented for large allocations. We will now discuss the implementation of free() for small allocations and how pages on the free list may eventually be munmap()ed and returned to the kernel.

Small Allocation Free

A free() differs from a malloc() in that the thread performing the free() is not necessarily the same as the thread performing the malloc().[2] free() must therefore first search potentially all threads' AVL trees in the order of the free()ing thread's dynamically generated and updated array of thread IDs described earlier, taking out a reader lock for each one and releasing the page of the free was not found in that AVL tree.

When the page is found in some thread's AVL tree, the two bytes immediately prior to the address to be free()d are read from memory. That value is the number of bytes to free(). free() takes out the simple lock on the page record's bitmap and zeroes the appropriate bits. The memory is now freed.

Before releasing the lock, however, free() checks whether the entire page is free. For 4K pages, this will only involve reading eight 64-bit words from memory and comparing each word to the constant zero. If the page is completely free, free() will upgrade its lock to a writer lock and perform an AVL tree deletion on the page, adding it to the owning thread's list of freed pages.

Deletion of Pages on Free List

Each thread keeps a saturating counter with saturation bounds of zero and FREEBIRD_COUNT, currently set to 10 for memory-optimized PALLOC1

[1] Note that it is impossible for free() to be called with the address of the start of a page used for a number of small allocations since the first PALLOC_WORD bytes of the page are reserved to store the size of the first allocation.

[2] In the future, when PALLOC1 is used inside of a pool allocation engine, it will be possible for multiple threads to allocate inside in the same heap.

and 16000 for speed-optimized PALLOC1. The counter is decremented when a page (or, for large allocations, a sequence of pages) is taken from the thread's free page list and incremented when a page (or contiguous sequence of pages) is added. If the counter ever reaches its FREEBIRD_COUNT saturation point, the top half of the free page list is munmap()ed (including all lists of contiguous pages), and the saturating counter is reset to zero. It is hoped that the strategy of deleting a constant fraction of all freed pages when more pages are being freed than malloc()ed will result in appropriate behavior no matter how drastic or shallow the drop in program memory usage.

In the event that a thread record is destroyed (because the thread exited after all its memory had been freed), any pages remaining on the free list are munmap()ed.

3.4.5 Status

PALLOC1 been implemented and tested on Solaris/SPARC, Linux/x86-64, and Linux/x86. There are no known bugs.

4 PALLOC2 Design and Implementation

4.1 Introduction

Unlike PALLOC1, PALLOC2 is designed to produce improvements in memory allocation performance for both parallel and sequential programs. PALLOC2 combines the first variable-length superpage allocator with a unique method for finding the start of a variable-length superblock and a novel approach to handling remote frees. PALLOC2 is useful both as a general-purpose high-performance sequential and parallel memory allocator and as a component in a larger runtime system that would otherwise require baggy bounds checking, as PALLOC2 can provide the capabilities of baggy bounds checking for free as a side effect of its superpage scheme.

4.2 Design of PALLOC2

PALLOC2 follows a per-thread subheap design with no global heap. Bitmap allocation takes place within variable-length superpages, with each superpage supporting allocation of a single power-of-two size class.

4.2.1 Superpages

Each PALLOC2 superpage contains in its header a 512-bit bitmap mapping the superpage into allocated and unallocated blocks. The blocks used by the header itself are always marked allocated; the rest of the superpage is initially marked free. Each superpage supports allocations of the size class such that one bit in the bitmap represents the allocation of a single object of that class.

Because superpages are not all the same size in PALLOC2, as they are in existing allocators, PALLOC2 uses a unique and useful scheme for finding the beginning of a superpage, where the header and bitmap are located, given an address pointing to an arbitrary memory location inside the superpage. First, superpages are always aligned on superpage boundaries[1]; thus, given the size of a superpage, it is possible to find the beginning of the superpage by masking off the lower bits of the address. The remaining problem is to find the size of a superpage given nothing but an arbitrary address inside the superpage boundaries.

[1] Consequently, allocations are always aligned to their size classes.

PALLOC2 solves this problem by zoning the address space. Specifically, six upper bits of the address of a PALLOC2 superpage are used to signify the size class of allocations supported by that superpage and, equivalently, the size of the superpage itself. PALLOC2 uses the `addr` field of the `mmap` system call to manage the virtual address space in a way that conforms to this zoning. A potentially useful side effect of PALLOC2's solution to this problem is that, given an address pointing into an object on a PALLOC2 heap, it is possible to find the start and end of the object; baggy bounds checking [4] or a similar technique would normally be necessary for the program runtime to have this capability.

4.2.2 Heap Layout

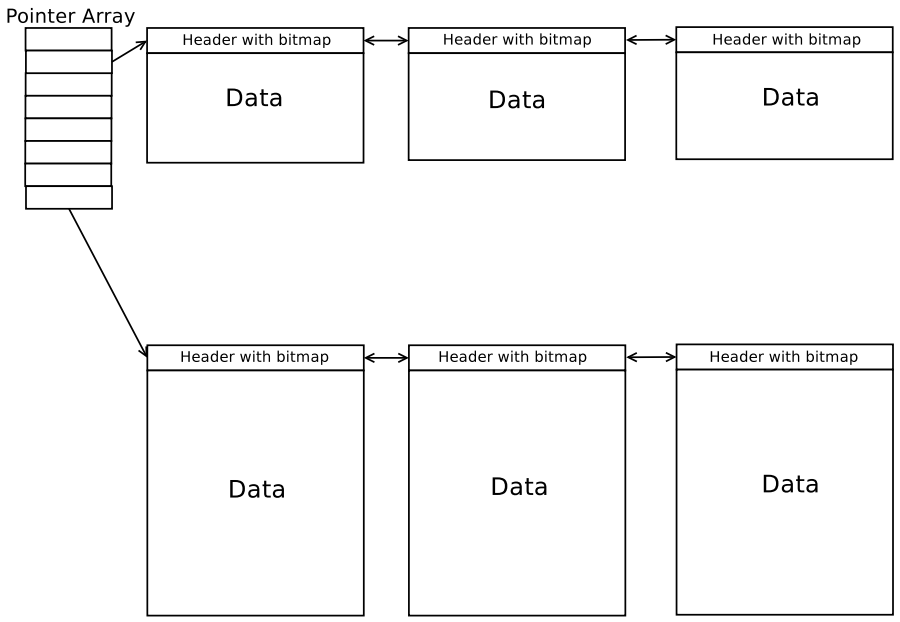

Figure 4.1: Diagram of PALLOC2 showing doubly linked superpage lists and variable-length superpages

Each PALLOC2 heap contains an array of linked lists of superpages. Each of these lists contains the partially free superpages of a particular size class. Allocations are done to the head of each list (the "active superpage"); when the active superpage is full, the next superpage in the list is made to be the head.

Because frequently allocating from different superpages is deleterious to locality, we never change the active superpage unless it is completely full. We further organize the tail of each list (all pages other than the head) such that the least used superpage is always the successor to the head. Thus, PALLOC2's design aims to minimize the need to change the active superpage.

4.2.3 Allocation

A request to PALLOC2 for allocation is first rounded up to the nearest size class. If there is no active superpage, one is obtained from the operating system. The bitmap of the active superpage is then searched for a free block; this block is then marked used in the bitmap and returned to the program for its use. In the case that this allocation completely fills the active superpage, it is deleted from the list and the next page (if any exists) becomes the active superpage.

Because an entire superpage is allocated for a single allocation, it is possible for PALLOC2 to operate with a dramatically higher virtual memory usage compared with other allocators. For instance, a single allocation of 4MB will result in an allocation of 2GB of virtual address space usage. This is not typically a problem, however, as the operating system will not allocate physical memory for an allocation of virtual address space unless and until it is used by the program. For extremely large allocations, PALLOC2 dispenses with the superpage scheme and uses `mmap` directly to fulfill the allocation request.

4.2.4 Freeing

This section describes the freeing process for allocations that are not extremely large. Extremely large allocations are freed directly with `munmap`.

For normal-sized allocations, PALLOC2 first finds the thread which "owns" the superpage in which the allocation takes place. Thread ownership information is recorded in the superpage header. PALLOC2's subsequent behavior depends on whether the thread performing the free is the same as the thread owning the superpage (and thus performed the allocation).

Local Frees

If the thread freeing the block is the same as the one to allocate it, the free is called "local." In a local free, PALLOC2 modifies the bitmap of the superpage to mark the freed block as available and either adds the superpage to the tail of the appropriate list or, if this free has made it so that this superpage has more free blocks than its predecessor, swaps its position with its predecessor to maintain the most-free-to-least-free ordering of the list.

Remote Frees

Remote frees are fundamentally the most difficult problem a parallel malloc implementation must handle, and PALLOC2 uses a novel approach. In PALLOC2, the first time a thread wishes to perform a remote free, it uses an atomic swap to fill in a pointer in the superpage header to a remote free bitmap. Then, this and all threads use atomic add instructions to mark remote frees in this bitmap. When the owner thread believes a superpage to be full, it checks to see

if the remote free bitmap exists; if so, the owner uses atomic swap instructions to clear the bitmap and processes the remote frees into the primary bitmap.

4.3 PALLOC2 Implementation

4.3.1 Architecture

Because PALLOC2 zones memory, effectively removing 6 or more bits from the address space, its design works best on architectures with an abundance of address space. The PALLOC2 implementation is for x86-64.

4.3.2 Operating System Support

PALLOC2 is written for Linux. Porting PALLOC2 to other UNIX-like operating systems should not be difficult as long as the operating system does not ignore the `addr` argument to `mmap`.

4.3.3 Internal Malloc for PALLOC2

Unlike PALLOC1, PALLOC2 uses superblock headers for its metadata. PALLOC2's need for internal dynamic memory allocation is therefore limited to the 64-byte chunks necessary for remote free bitmaps. PALLOC2 uses the same simple free list scheme as PALLOC1 for the internal malloc implementation used to support these allocations.

4.3.4 Interaction with Operating System

PALLOC2 uses `mmap` to allocate virtual pages from the operating system. Like PALLOC1, PALLOC2 never calls `brk()`, `sbrk()`, or the system malloc implementation. Unlike PALLOC1, PALLOC2 makes extensive use of the ability to request specific virtual addresses from `mmap`.

4.3.5 Testing

There are no known bugs in PALLOC2. Emacs does not work with PALLOC2 as it attempts to use headers present in the GLibC implementation of `malloc` to discover the size of objects at runtime.

5 Benchmarking and Results

5.1 Benchmarking Methodology

5.1.1 Testing Environment

For benchmarking, we used an 8-core Sun Microsystems UltraSPARC T1 machine capable of supporting 32 hardware executions threads and a 24-core Intel Xeon system. PALLOC2, TCMalloc, and Streamflow do not support SPARC, so our results for SPARC include numbers for only the system allocator, PALLOC1, Hoard, and Ptmalloc. We ran each benchmark ten times and averaged the results. In order to measure memory usage, we repeatedly polled the virtual memory usage using the UNIX `ps` command and recorded the maximum resident memory observed.

For testing PALLOC1 with large-allocation microbenchmarks, we compiled each of PALLOC1, Ptmalloc, and Hoard using the Sun Studio compiler. On the Linux machine, we used GCC and also tested PALLOC2 and Streamflow. We used full optimization parameters (`-O3`) for both compilers for all allocators.

5.1.2 PALLOC1 Implementation Parameters

Our PALLOC1 implementation supports three tweakable compile-time knobs. These knobs are implemented as the macros PALLOC_PAGE_SIZE, PALLOC_WORD, and PALLOC_FREEBIRD.

We have tested page size parameters from 4K to 4MB and have set PALLOC_PAGE_SIZE to 4MB for both speed- and memory-optimized PALLOC1 on SPARC and to 1MB on x86-64. We set PALLOC_FREEBIRD to 16000 for speed-optimized PALLOC1 and 10 for memory-optimized PALLOC. We used a word size of 128 bytes for speed-optimized PALLOC and 16 bytes for memory-optimized PALLOC.

5.2 Synthetic Microbenchmarks

We report these results to validate the potential of PALLOC1's design, however, these results should not be taken as a complete performance assessment. Future tuning of PALLOC1 may significantly change these numbers. Moreover, radically different results are possible using different parameters for the synthetic benchmarks (heaptest, threadtest, consume, and larson), as shown later. We

illustrate the significance of the parameters given to a synthetic benchmark with a graph of PALLOC1's performance versus Hoard for different parameters to larson. For most applicatins, we report two sets of results: we used parameters indicating fairly large values for allocation sizes since PALLOC1 is optimized for large allocations, and we also report results for more typical allocation sizes.

The execution times are the sums of 10 trials. The memory usage sizes are the averages of the maximum virtual memory allocated over 3 trials. Execution times are given in seconds. Memory usage is given in kilobytes of allocated virtual memory.

5.2.1 ThreadTest

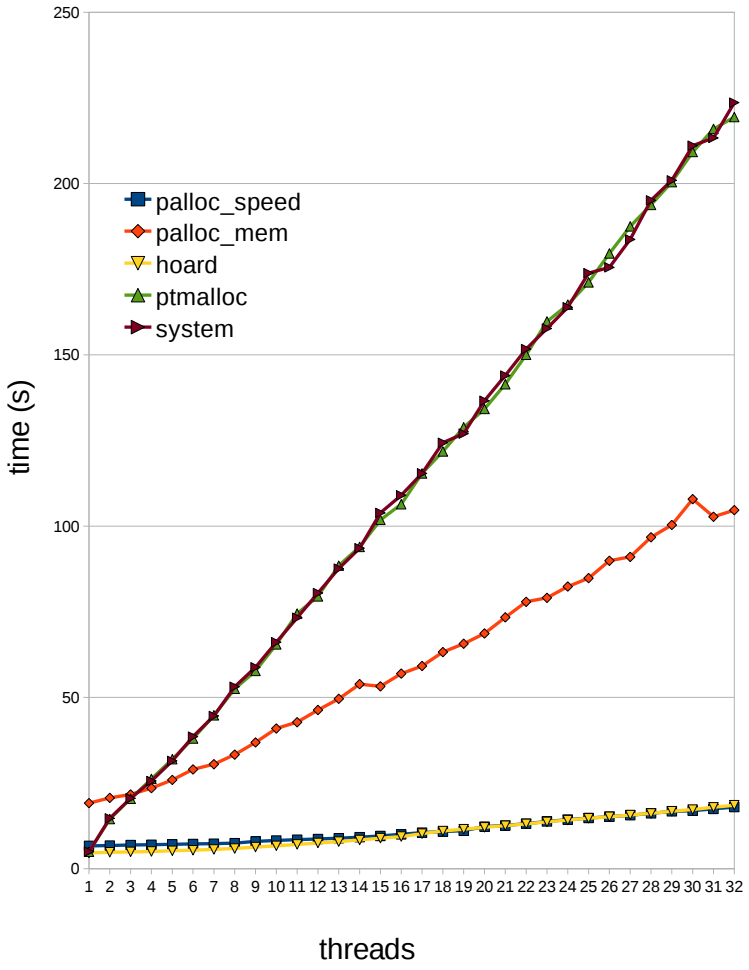

Figure 5.1: Execution time for ThreadTest on SPARC 32 Bit

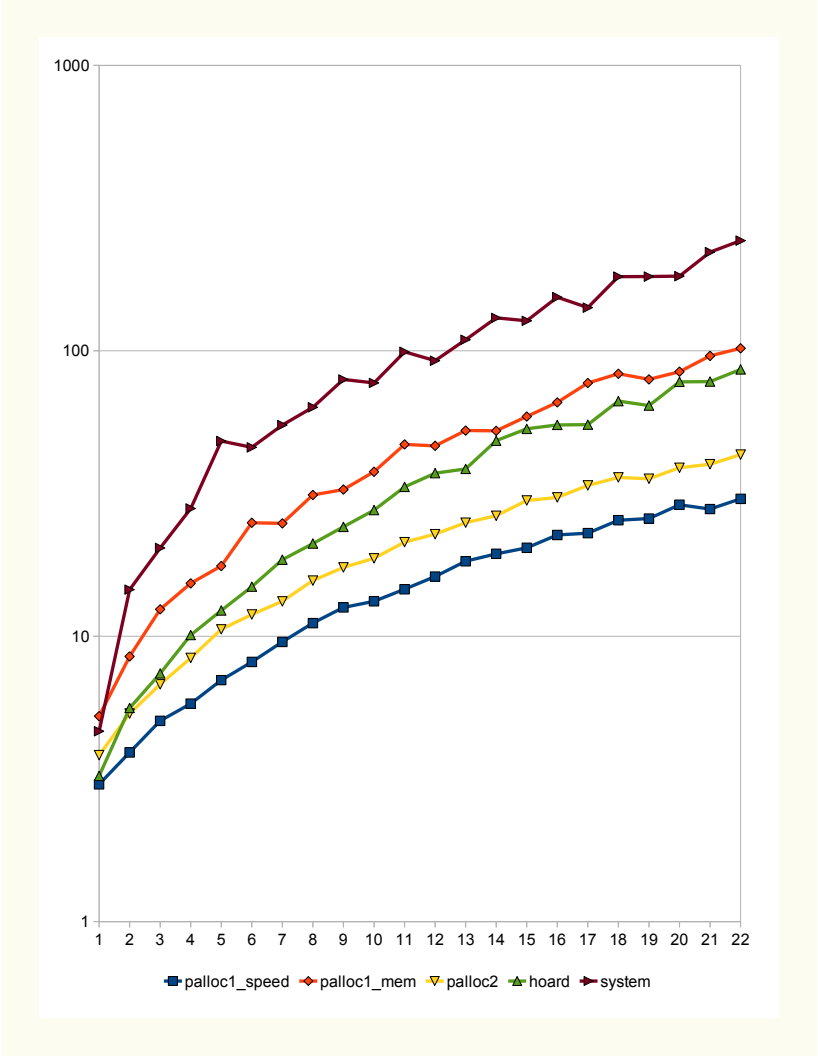

Figure 5.2: Large Allocation Execution Time for ThreadTest on x86_64

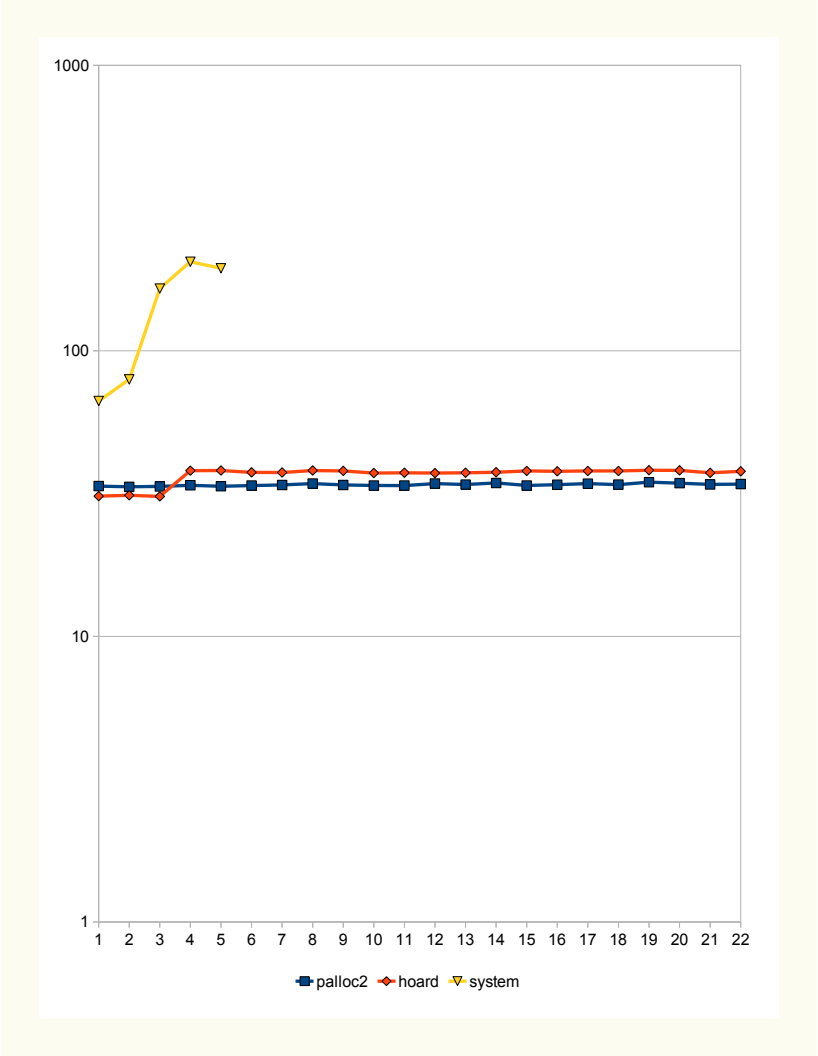

Figure 5.3: Typical Allocation Execution Time for ThreadTest on x86_64

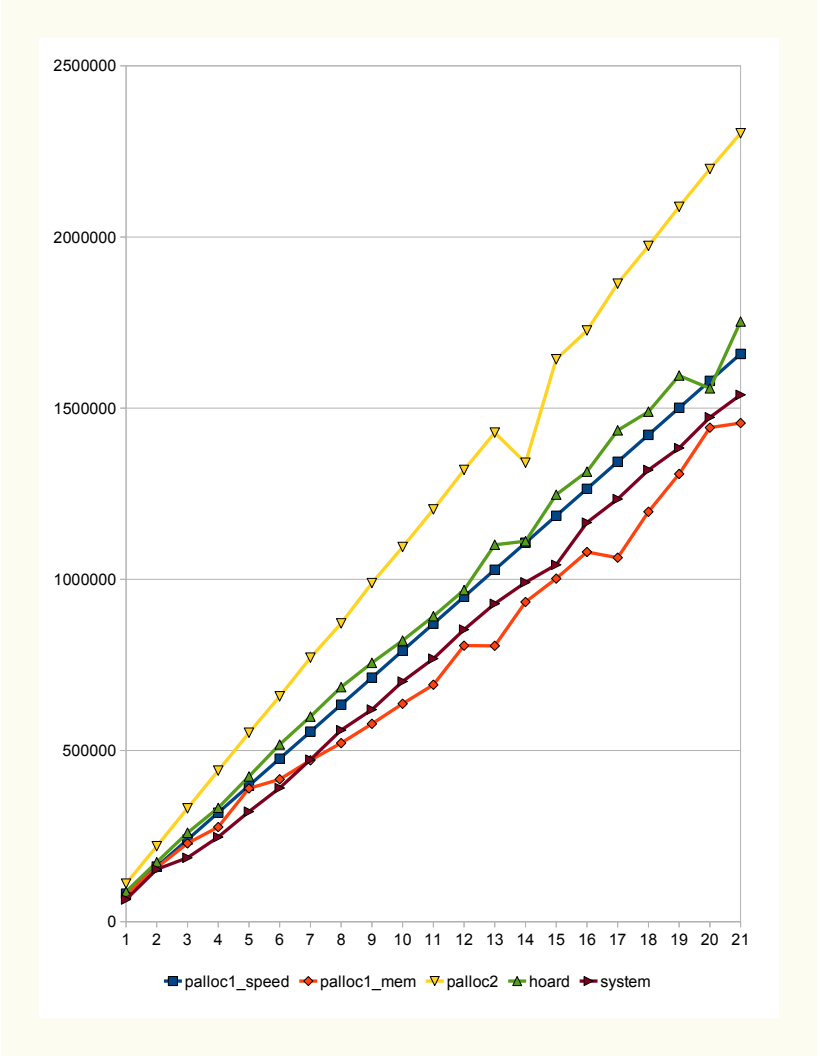

Figure 5.4: Memory Usage for ThreadTest on x86_64

ThreadTest is packaged with Hoard and is a natural benchmark for us to use. It generates one or more threads that allocate and free a large number of chunks of memory of the same size. Each thread is independent: they don't synchronize or share objects [14].

The ThreadTest benchmark contains a loop that repeatedly allocates then frees a memory allocation. For our large allocation test, 100 Kbyte allocations were used on x86_64 and 50Kbyte allocations were used on SPARC (our SPARC machine was slower). For our typical allocation test, 100 byte allocations were used. We find that the run times for all the memory allocators increase with the number of threads as the benchmark simply does more work when run with more threads. We do not report results for Streamflow as the allocator crashed during the test. We limit reporting of PALLOC1 and system allocator results for normal allocation sizes as performance was very poor for many threads. On

x86-64, PALLOC2 performs well with both large and typical allocation sizes, and performs best for typical allocation sizes.

5.2.2 Consume

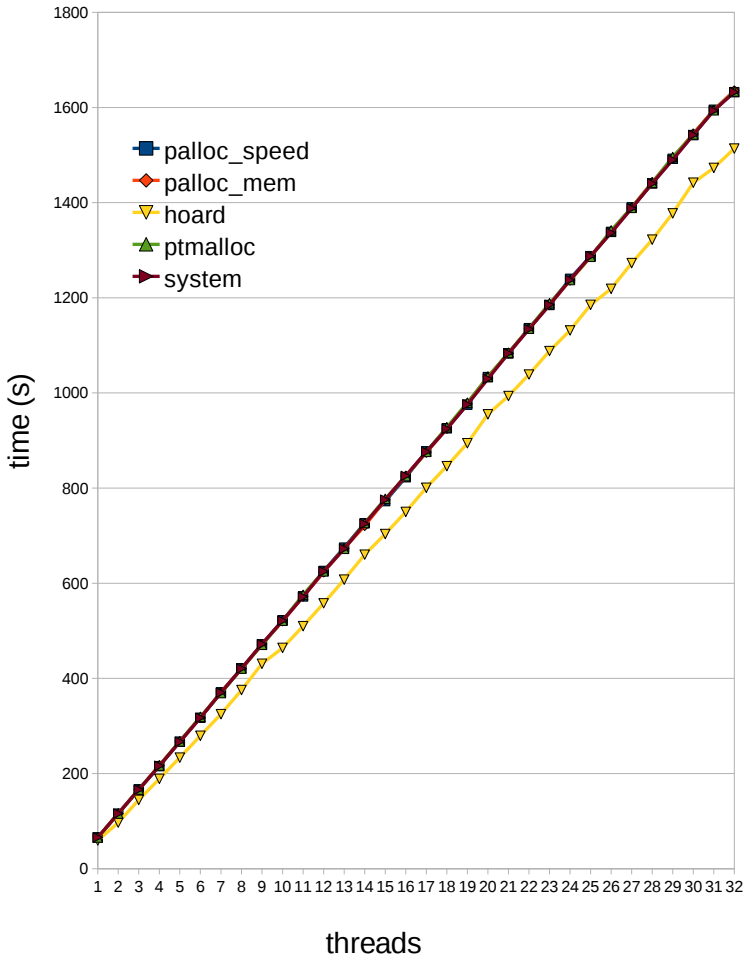

Figure 5.5: Execution Time for Consume on SPARC 32 Bit

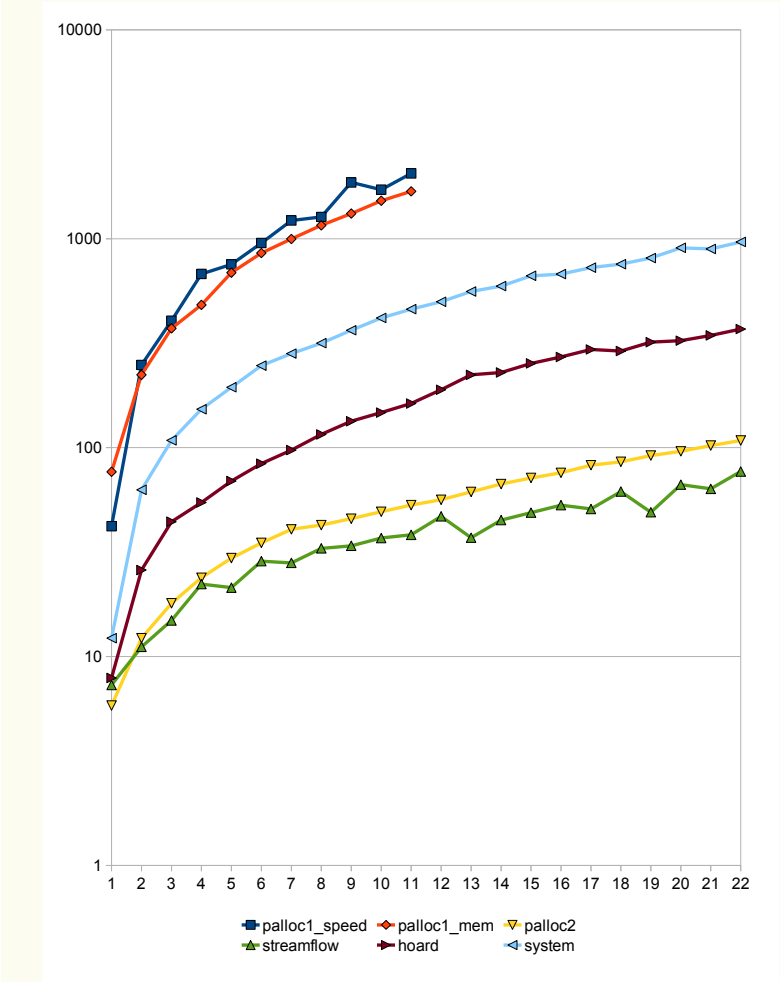

Figure 5.6: Large Allocation Execution Time for Consume on x86_64

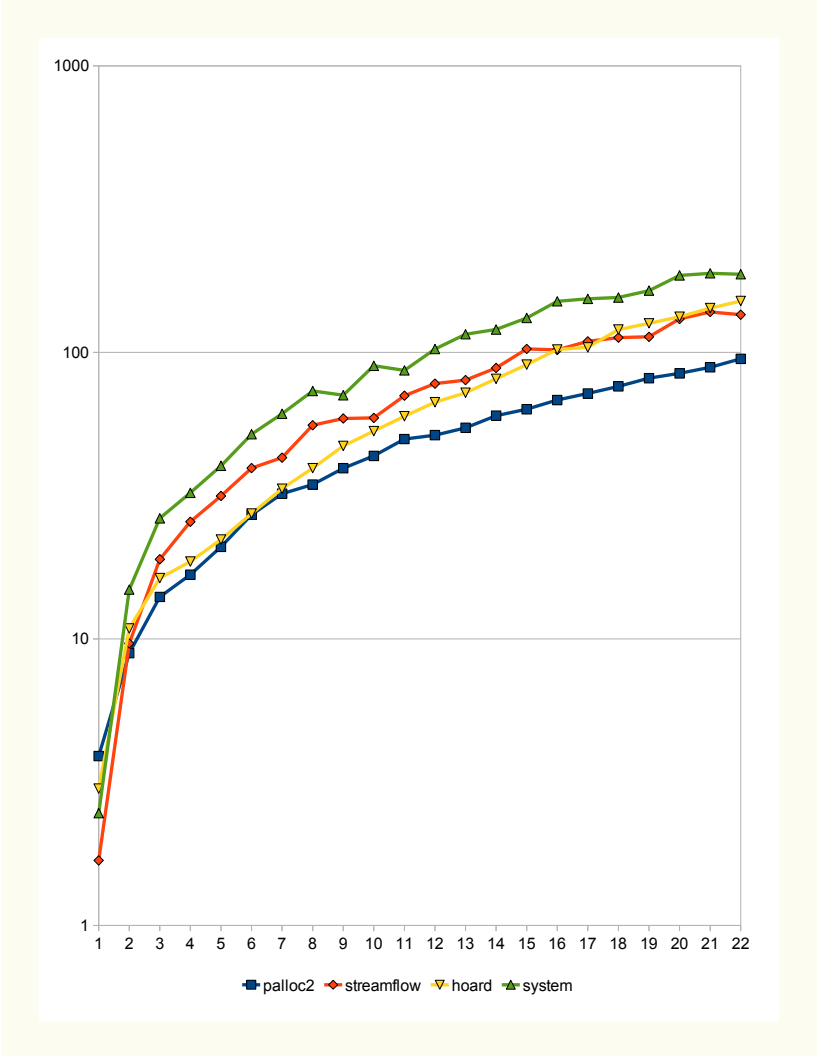

Figure 5.7: Typical Allocation Execution Time for Consume on x86_64

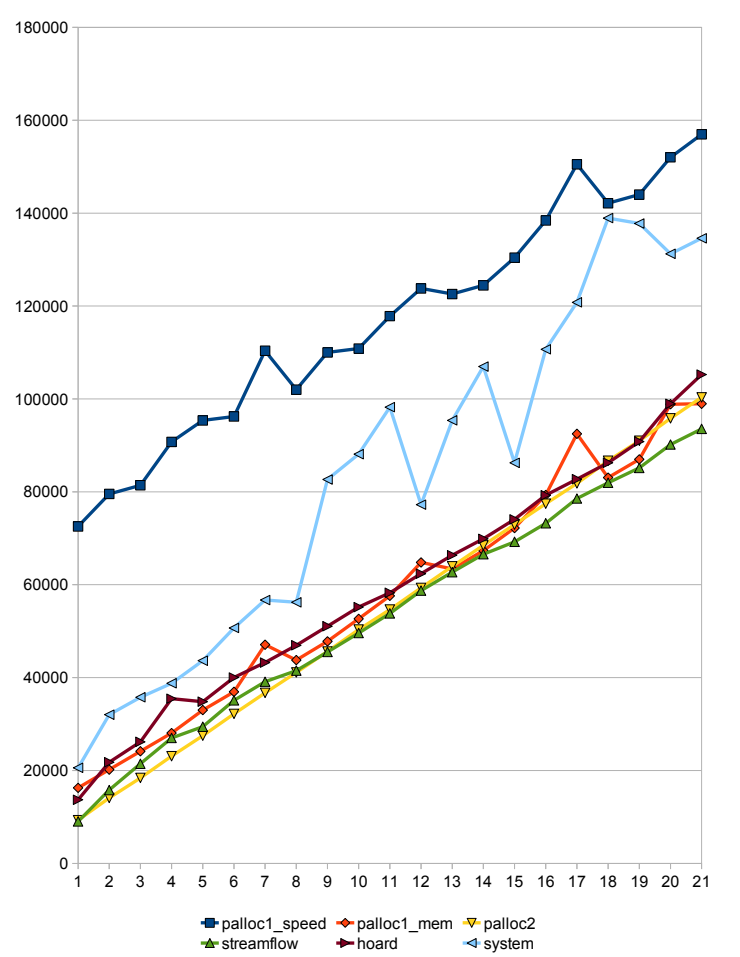

Figure 5.8: Memory Usage for Consume on x86_64

The Producer-Consumer benchmark (consume) is also packaged with Hoard. It simulates heavy communication among threads, where one thread allocates memory and the others free it.

For the large allocation consume benchmark, we used a buffer of 500K and an iteration count of 10 on both architectures. We used a buffer of 100 bytes and iteration count of 10000 for our typical allocation run. We limit reporting of PALLOC1 results on x86-64 as performance was very poor. On x86-64, Streamflow performs best for large allocation sizes, but PALLOC2 performs best for typical allocation sizes.

5.2.3 Larson

This Hoard benchmark attempts to simulate the behavior of a multithreaded server. Threads are repeatedly created and destroyed during each run of it. The

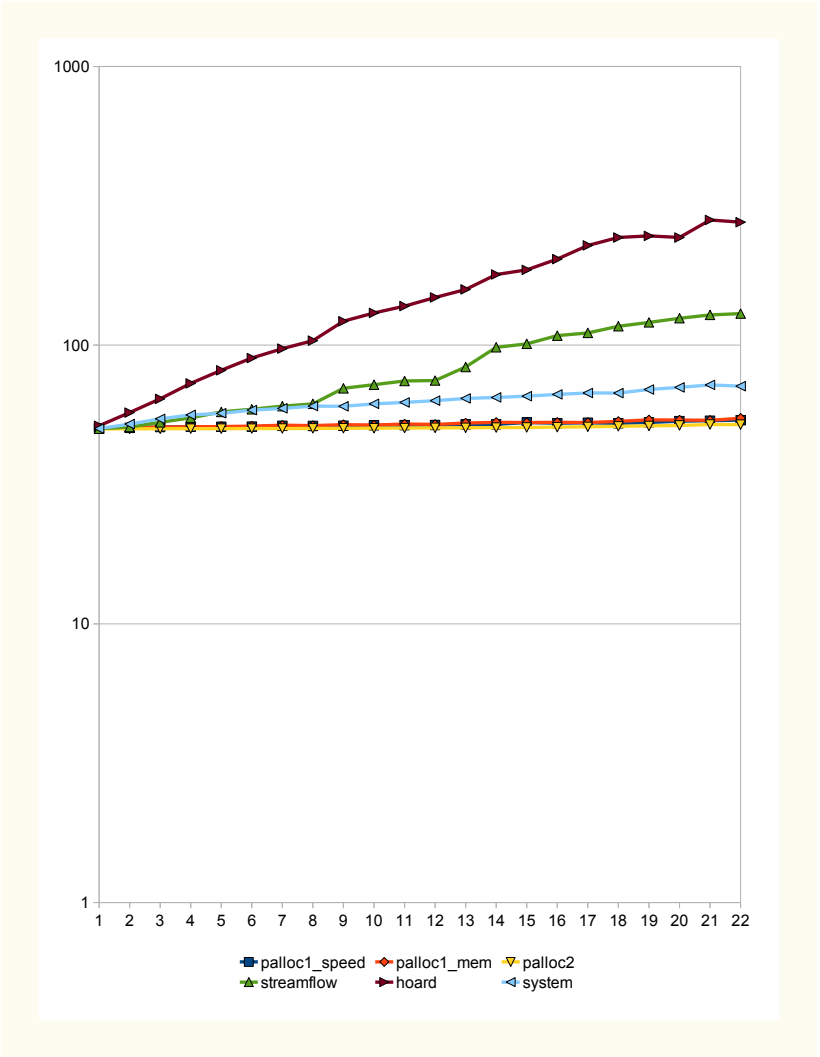

Figure 5.9: Large Allocation Execution Time for Larson benchmark on x86_64

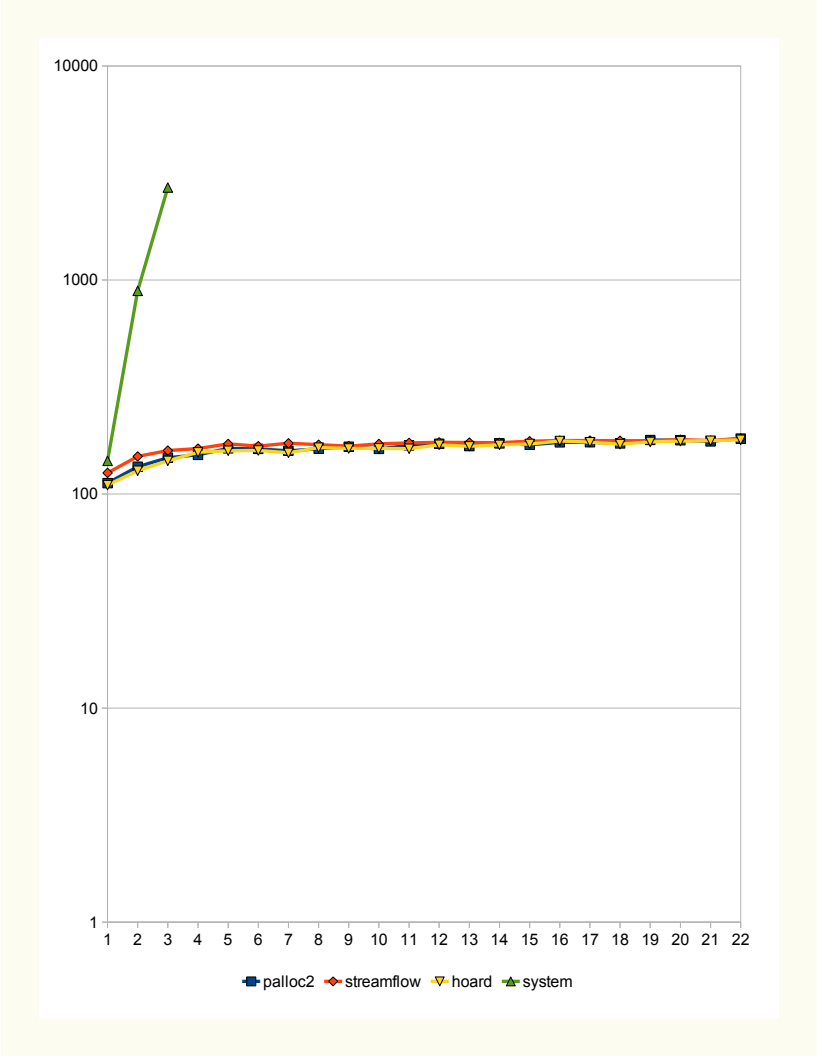

Figure 5.10: Typical Allocation Execution Time for Larson benchmark on x86_64

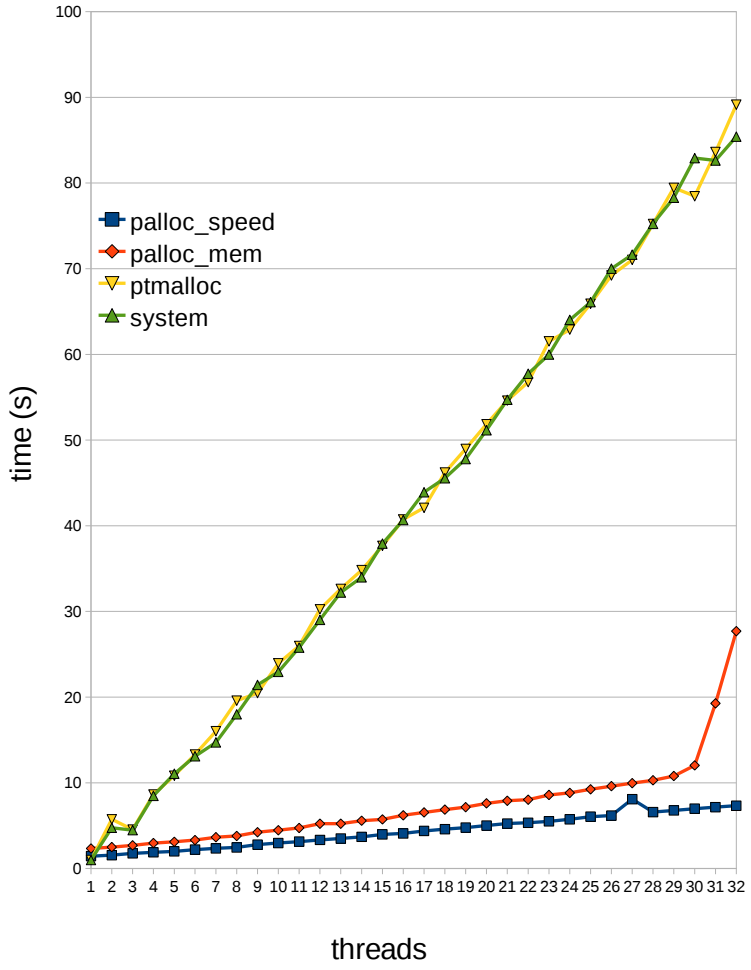

Figure 5.11: Execution Time for Larson benchmark on SPARC 32 Bit

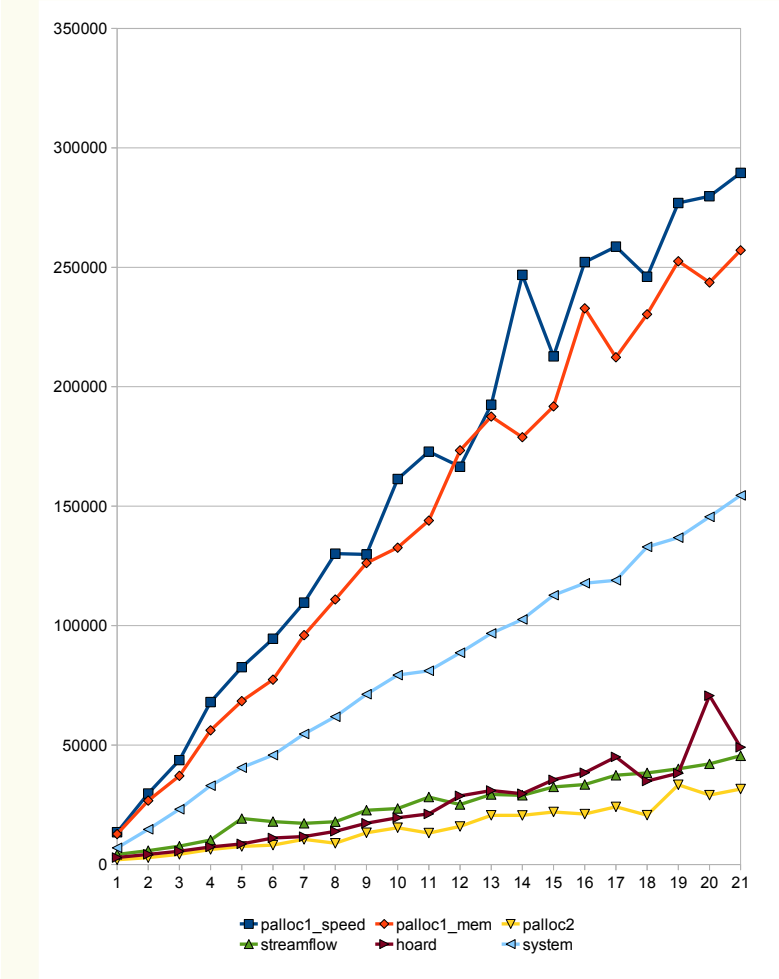

Figure 5.12: Memory Usage for Larson benchmark on x86_64

benchmark sleeps for 5 seconds at the end of every run; this sleep time has been removed from the reported results for all allocators. For our large allocation run, we configured both x86_64 and SPARC, we configured larson to run with allocations ranging from 1K to 100K, with 250 chunks per thread, for 1 round. We used allocations ranging from 10 to 100 bytes with 5000 chunks per thread for our typical allocation run.

Hoard is not included in the execution time graph for SPARC in order to increase the graphs' legibility; the allocator performed pathologically poorly in this benchmark, taking almost 700 seconds on x86_64 to complete 10 trials and over 2600 seconds on SPARC for 32 threads. Performance was almost identical for PALLOC2, Streamflow, and Hoard for both large and typical allocation sizes on x86-64.

5.2.4 HeapTest

This benchmark, also packaged with Hoard, simply spawns multiple threads which repeatedly allocate and free chunks of data of a fixed size. For our large allocation run, an allocation size of 1MB and allocation count of 50000 was used for both x86_64 and SPARC. For our typical allocation run, the allocation size was 100 bytes and the allocation count was 100 million. For measuring memory, the allocation size was increased to 100MB. We do not report results for PALLOC1 and limit reporting of results for the system allocator as performance was very poor.

PALLOC2's superior performance in this benchmark may due to its lack of headers: as memory is allocated and freed without being touched, PALLOC2 is able to avoid the allocation of a physical page for it.

5.3 Real Applications

In this section, we test real, heap-intensive applications in an attempt to validate the PALLOC designs. Because these applications tend to use small, not large, allocation sizes, PALLOC1 is outperformed by other allocators. However, PALLOC2 performs admirably.

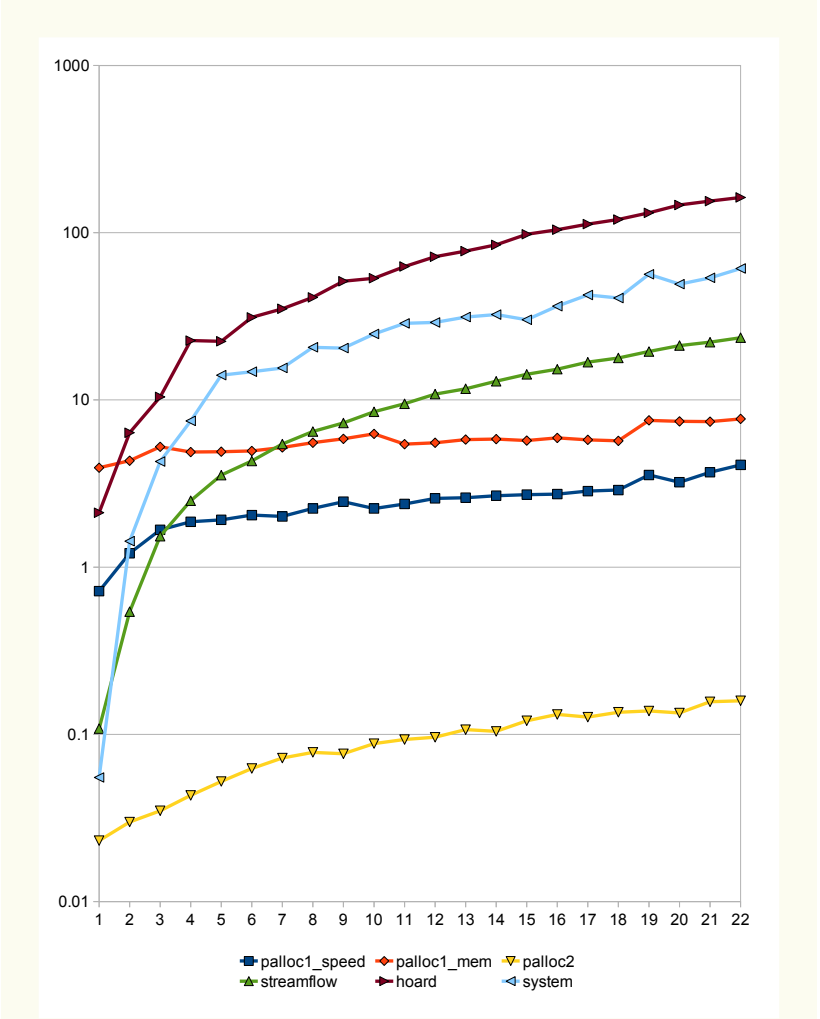

Figure 5.13: Large Allocation Execution Time for HeapTest benchmark on x86_64

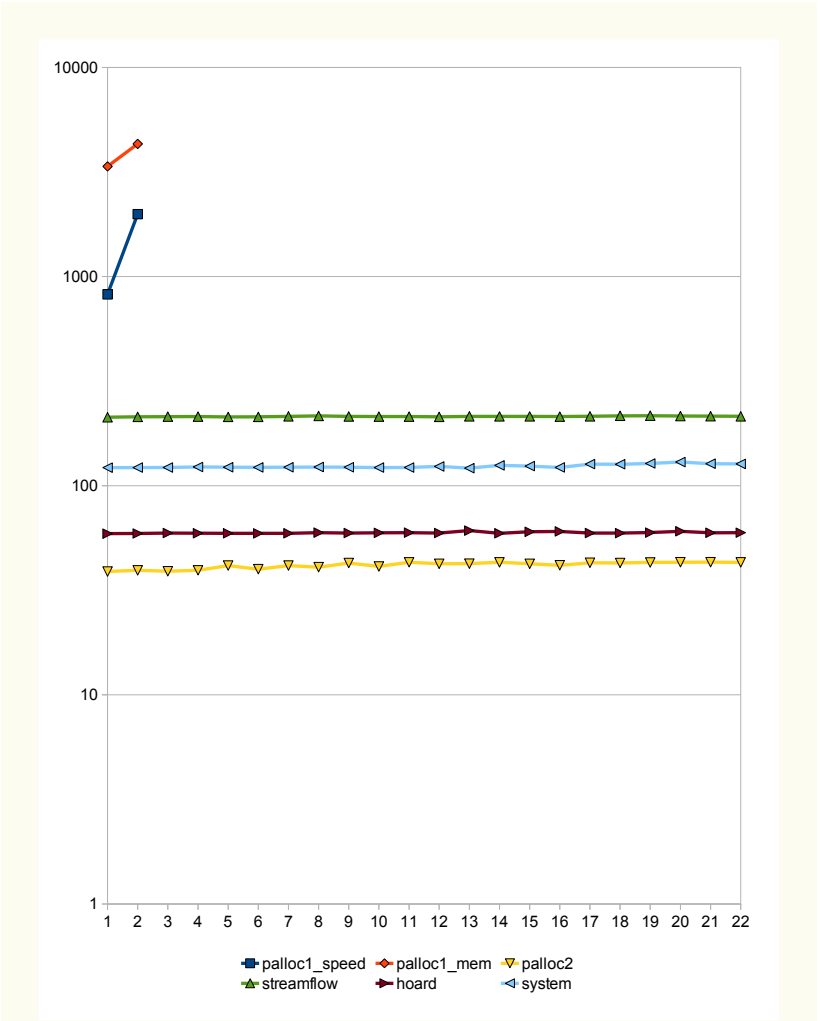

Figure 5.14: Typical Allocation Execution Time for HeapTest benchmark on x86_64

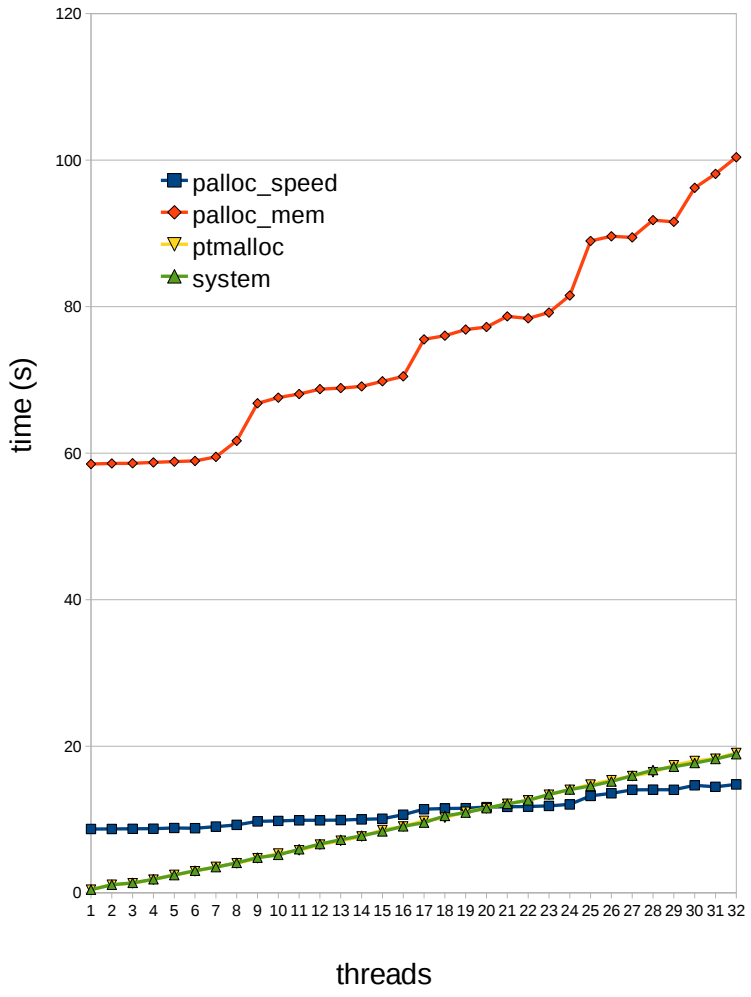

Figure 5.15: Execution Time for HeapTest benchmark on SPARC 32 Bit

5.3.1 Parallel Applications

Barnes-Hut

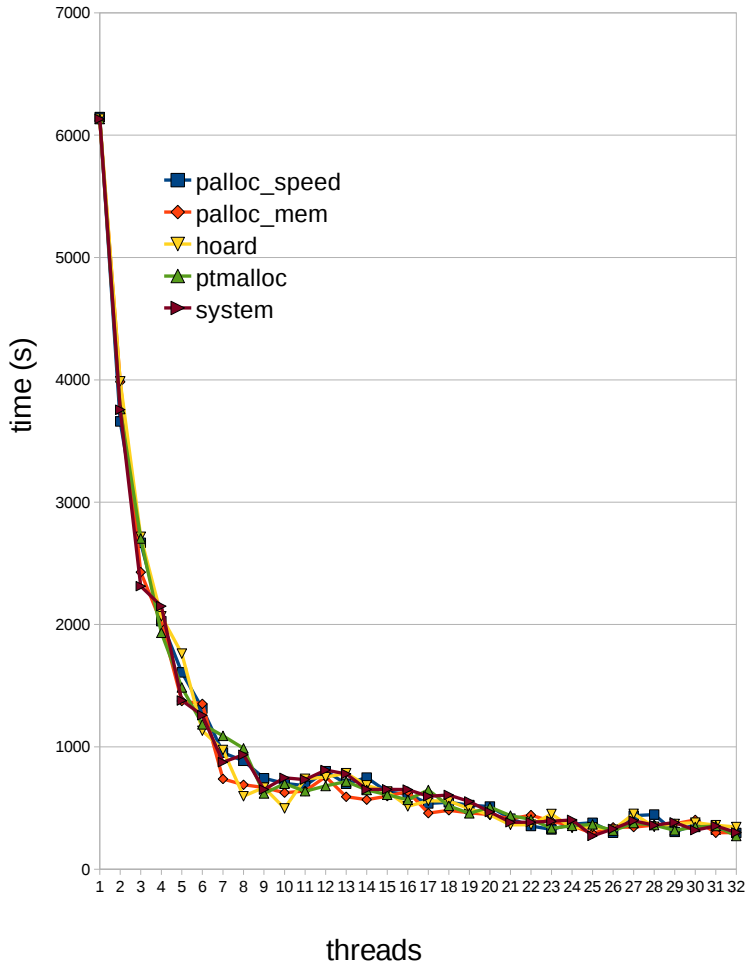

Figure 5.16: Execution Time for Barnes-Hut Benchmark on SPARC 32 Bit

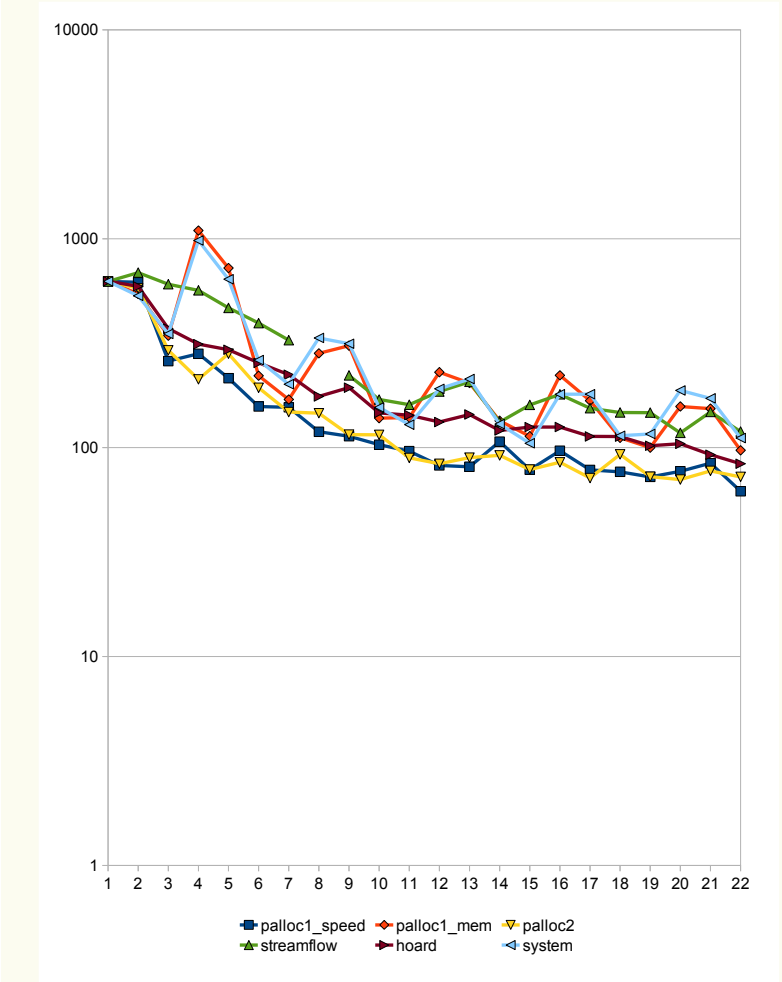

Figure 5.17: Execution Time for Barnes-Hut Benchmark on x86_64

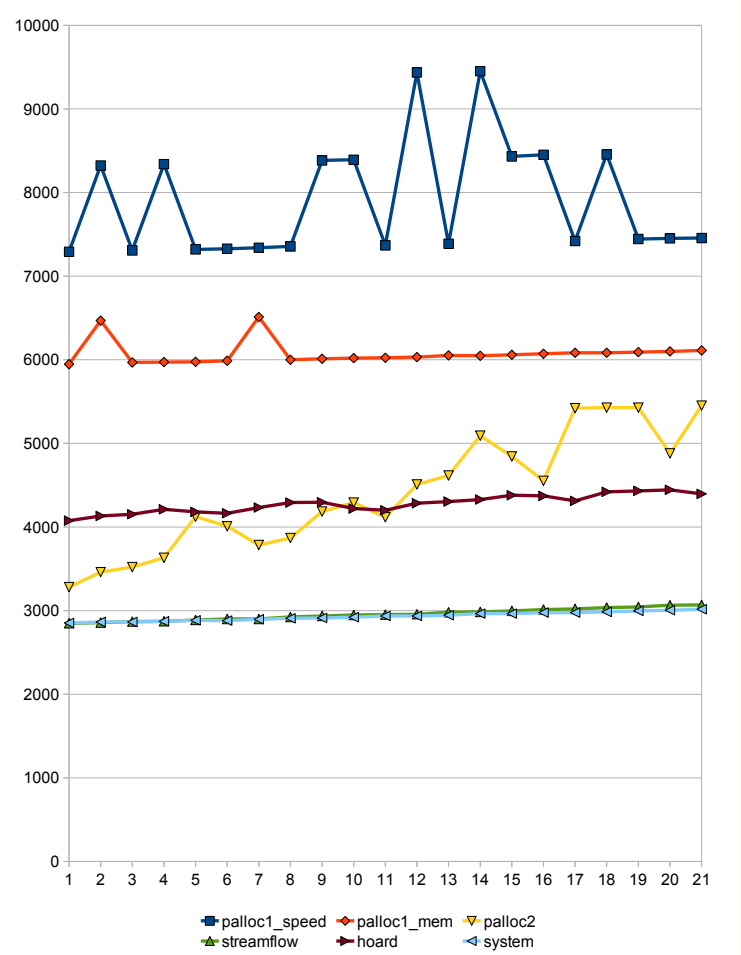

Figure 5.18: Memory Usage for Barnes-Hut Benchmark on x86_64

The Barnes-Hut N-Body force simulation is an implementation of the Barnes-Hut n-body algorithm which solves for the interaction (i.e. mechanic, electric force) that exists between bodies in space.

Barnes-Hut is based on an octree in 3-D, or a quadtree in 2-D. This type of tree provides a natural hirearchy of the domain, where leaves represent the objects, and subtrees are groups of objects. After the creation of this data structure, subtrees can be cut and distributed amongst different processors [11] This parallelization scheme is aided by a simplifying assumption that is common in this type of physical problem: the contribution of the forces from a an object that is far away (between galaxies) can be approximated as the cumulative force of a group of nearby galaxies (i.e. a whole subtree).

The algorithm works as follows [10]:

1. Initialize list of bodies with starting location and velocity.

2. Iterate over time steps:

    (a) Create an octree

    (b) Insert bodies (galaxies) into octree

    (c) Compute center of mass for each object, and cumulative mass.

    (d) Compute force acting on each body by traversing octree.

        i. If current node is a leaf (body) or internal node (cell) and is far enough, terminate.

    (e) Update bodies' position and velocity.

This benchmark has been used to study the performance of other parallel memory allocators, including Hoard [14].

Barnes-Hut is a not a microbenchmark but a real, multithreaded application that relies heavily on dynamic memory allocation. We used the Lonestar implementation. On x86_64, we used the "run B" input data. We used "run A" for SPARC. PALLOC1 optimized for speed and PALLOC2 exhibit superior performance for this application.

Swaptions

Swaptions is a financial analysis application forming part of the PARSEC [2] parallel benchmark suite. It was chosen for benchmarking as it makes extensive use of dynamic memory allocation. PALLOC2 and Streamflow performed best on this application, with Streamflow scaling slightly better.

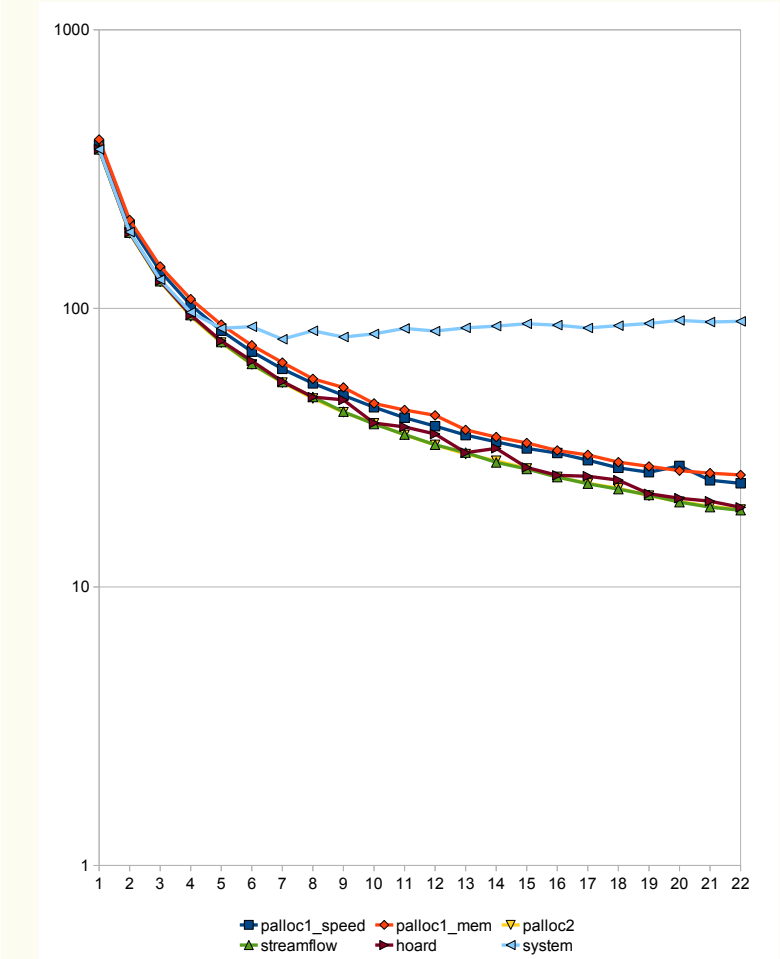

Figure 5.19: Execution Time for Barnes-Hut Benchmark on x86_64

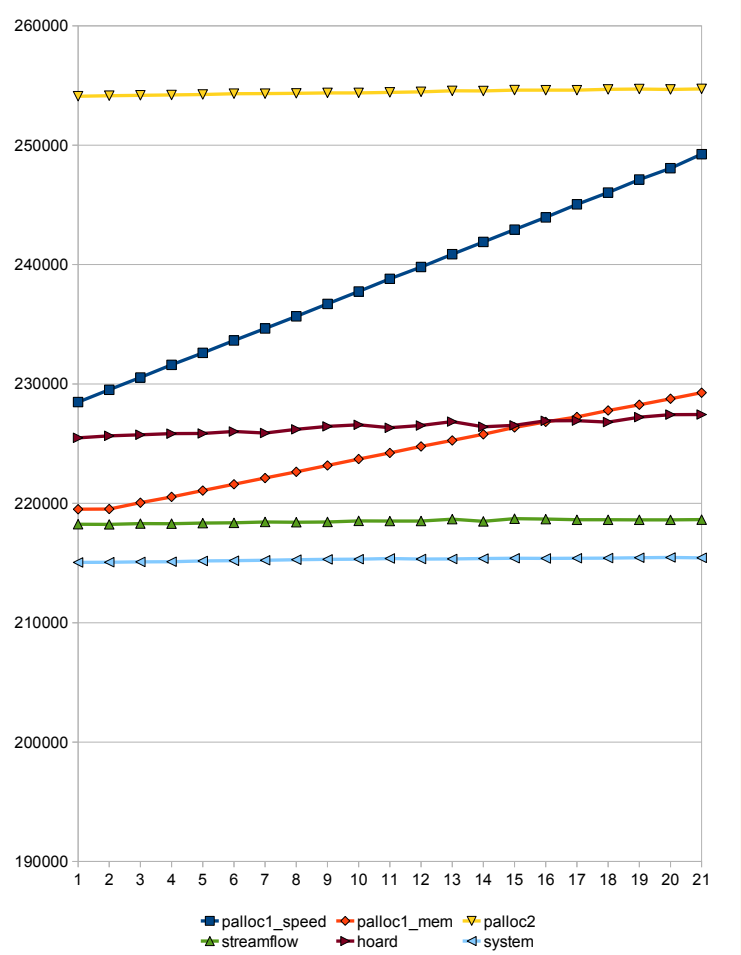

Figure 5.20: Memory Usage for Barnes-Hut Benchmark on x86_64

5.3.2 Sequential Applications

As PALLOC2 is intended to provide benefits for sequential as well as parallel programs, we also evaluate its performance with sequential programs heavily reliant on dynamic memory.

GCC

This benchmark consisted of using GCC to compile a stripped-down version of itself packaged a single source file. Only PALLOC2, Hoard, and the system allocator were able to successfully complete this test due to bugs in Streamflow and functionality limitations in PALLOC1. PALLOC2 exhibited superior performance in this test.

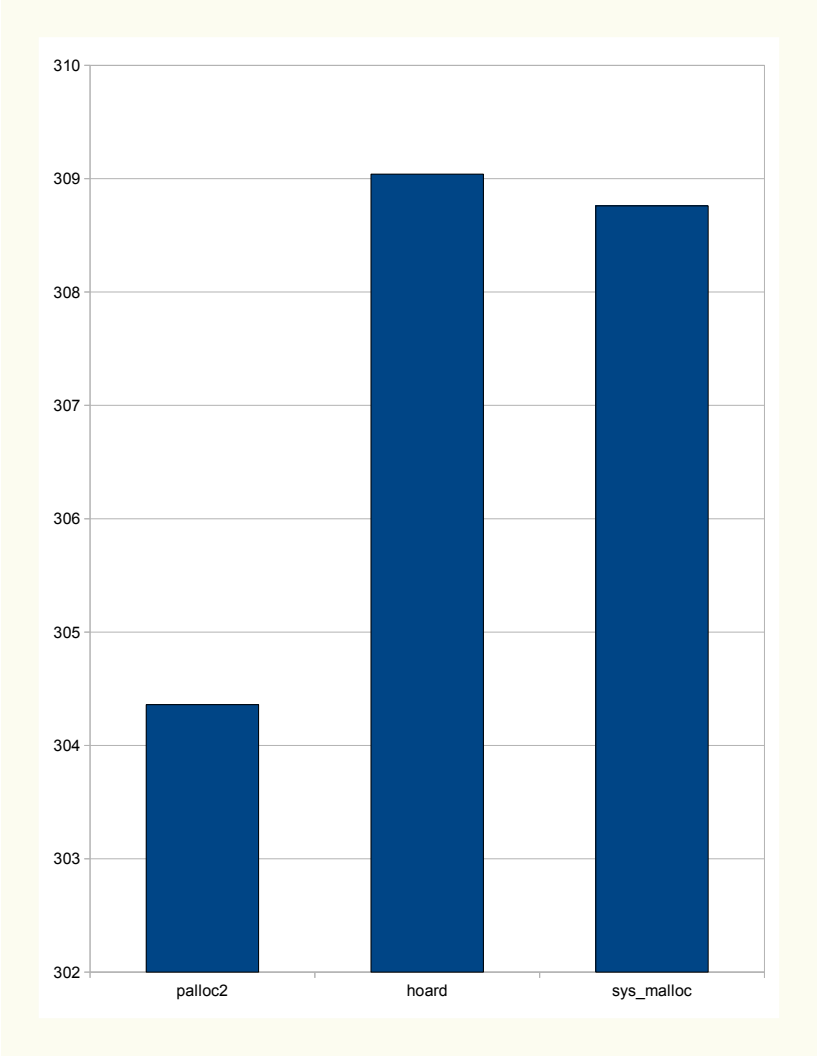

Figure 5.21: Execution Time for Barnes-Hut Benchmark on x86_64

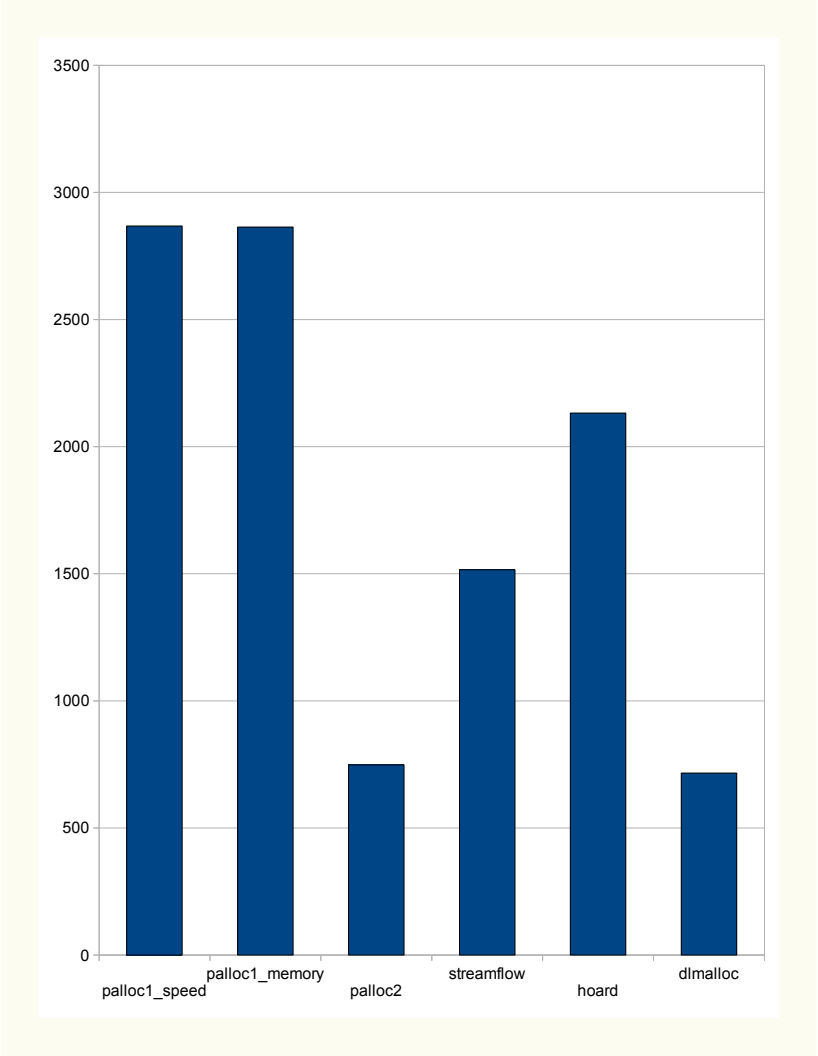

Figure 5.22: Memory Usage for Barnes-Hut Benchmark on x86_64

197.parser

197.parser forms a part of the SPEC integer benchmark suite. This program was chosen out of SPEC for its extensive use of dynamic memory allocation. PALLOC2 exhibited superior performance in this test.

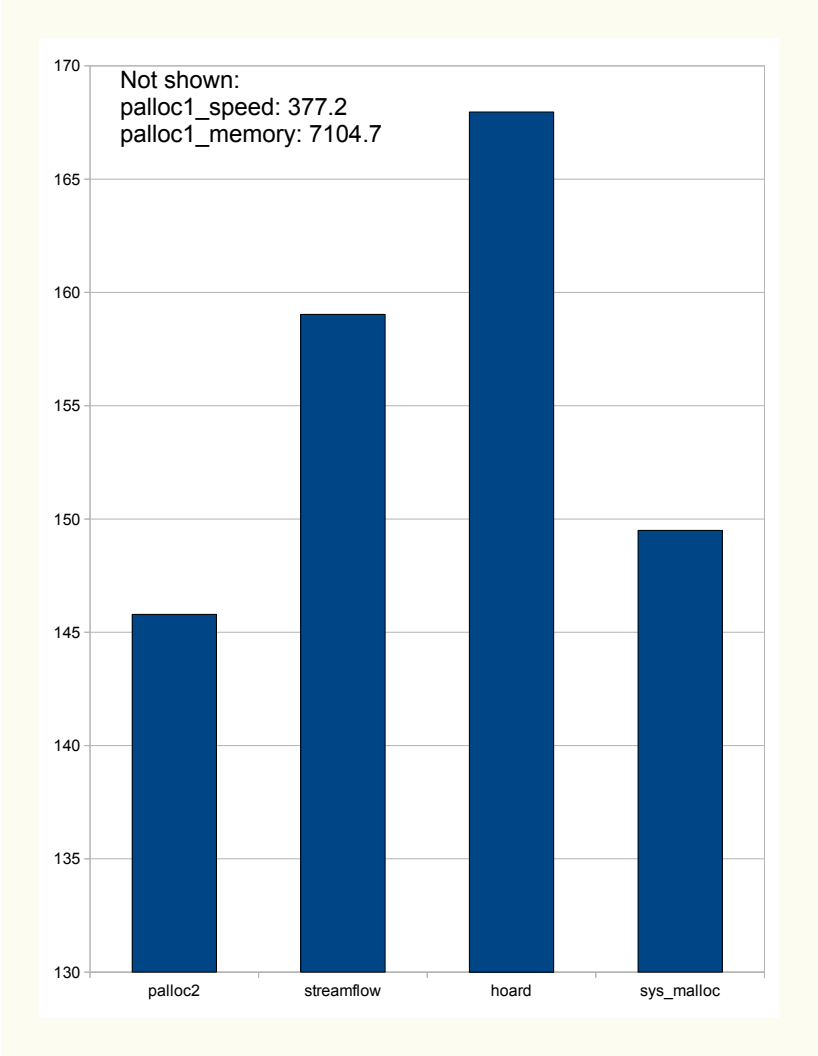

Figure 5.23: Execution Time for Barnes-Hut Benchmark on x86_64

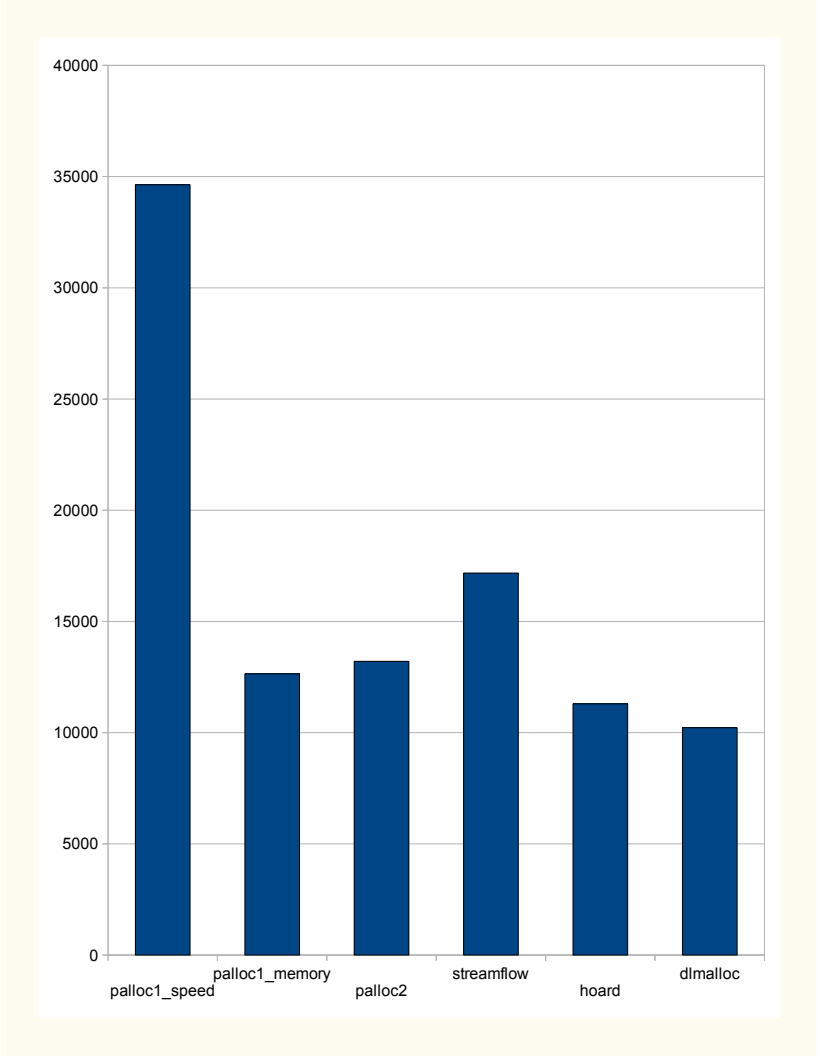

Figure 5.24: Memory Usage for Barnes-Hut Benchmark on x86_64

6 Conclusion

This thesis discusses two novel designs for parallel implementations of the C memory allocator. PALLOC1 provides a strong, novel guarantee limiting this allocator's memory usage, and PALLOC2 provides the capabilities of baggy bounds checking for free as a side effect of its memory management strategy. These allocators are evaluated with microbenchmarks as well as real applications, and the results show that the performance of PALLOC2 is always competitve and sometimes superior to other malloc() implementations which do not provide its baggy bounds checking capability. For PALLOC1, in addition to the previously proposed modifications for lock-free and region-based allocation, interesting future research related to this allocator design may include more fully exploring the implementation space defined by the PALLOC_PAGE_SIZE, PALLOC_WORD, and PALLOC_FREEBIRD parameters as well as examining the performance of alternatives to the AVL tree reader-writer lock design. Evaluating or modifying this design for NUMA architectures may also prove useful. The most promising avenue for future PALLOC2 research is to examine whether runtime techniques previously limited by the performance overhead of baggy bounds checking would are more practical in light of PALLOC2's capabilities.

References

[1] A comparison of memory allocation in multiprocessors - http://developers.sun.com/solaris/articles/multiproc/multiproc.html.

[2] The parsec benchmark suite - http://parsec.cs.princeton.edu.

[3] Tcmalloc: Thread-caching malloc - http://google-perftools.googlecode.com/svn/trunk/doc/tcmalloc.html.

[4] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Canada, August 2009.

[5] Wolfram Gloger. Wolfram gloger's malloc homepage - http://www.malloc.de/en/.

[6] Dirk Grunwald and Benjamin Zorn. Customalloc: efficient synthesized memory allocators. *Softw. Pract. Exper.*, 23(8):851–869, 1993.

[7] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. Mcrt-malloc: a scalable transactional memory allocator. In *ISMM '06: Proceedings of the 5th international symposium on Memory management*, pages 74–83, New York, NY, USA, 2006. ACM.

[8] Chris Lattner and Vikram Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, Chigago, Illinois, June 2005.

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

[10] University of Texas at Austin. Barnes-hut n-body simulation - http://iss.ices.utexas.edu/lonestar/barneshut.html.

[11] S.N. Pangfeng Liu; Bhatt. Experiences with parallel n-body simulation. *Parallel and Distributed Systems, IEEE Transactions on Parallel and Distributed Systems*, 1994.

[12] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *ISMM '06: Proceedings of the 5th international symposium on Memory management*, pages 84–94, New York, NY, USA, 2006. ACM.

[13] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator, 1996.

[14] Emery D. Berger; Kathryn S. McKinley; Robert D. Blumofe; Paul R. Wilson. 4. hoard: A scalable memory allocator for multithreaded applications. 2000.

Author's Biography

Patrick Simmons was born in New Orleans, Louisiana, to Erny and Karen Simmons in 1987. His family moved to Texas in 1993. Patrick attended public school in Plano, Texas, graduating from Plano Senior High in 2005, and completed a BS in Computer Sciences at the University of Texas in Austin in 2008. He then enrolled in the University of Illinois at Urbana-Champaign graduate computer science program and is currently completing a computer science doctorate.