A Locality-Improving Dynamic Memory Allocator

Yi Feng and Emery D. Berger
Department of Computer Science
University of Massachusetts Amherst
140 Governors Drive
Amherst, MA 01002
{yifeng, emery}@cs.umass.edu

ABSTRACT

In general-purpose applications, most data is dynamically allocated. The memory manager therefore plays a crucial role in application performance by determining the spatial locality of heap objects. Previous general-purpose allocators have focused on reducing fragmentation, while most locality-improving allocators have either focused on improving the locality of the allocator (not the application), or required programmer hints or profiling to guide object placement. We present a high-performance memory allocator called *Vam* that transparently improves both cache-level and page-level locality of the application while achieving low fragmentation. Over a range of large-footprint benchmarks, Vam improves application performance by an average of 4%–8% versus the Lea (Linux) and FreeBSD allocators. When memory is scarce, Vam improves application performance by up to 2X compared to the FreeBSD allocator, and by over 10X compared to the Lea allocator.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Memory management (garbage collection); D.4.2 [**Storage Management**]: Main memory, Virtual memory

General Terms

Algorithms, Languages, Performance

Keywords

Vam, memory management, allocator, fragmentation, cache locality, paging, virtual memory

1. Introduction

Explicit memory managers have traditionally focused on reducing the number of discontiguous free chunks of memory, or *fragmentation*. Reducing fragmentation improves space efficiency and understandably has received considerable attention by memory manager designers. For example, the widely-used Lea allocator that forms the basis of the Linux `malloc` (DLmalloc) was designed specifically for high performance and low fragmentation [12, 13, 16].

This material is based upon work supported by the National Science Foundation under Award CNS-0347339. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

MSP'05, Chicago, USA.

However, the widely-acknowledged increasing latency gap between the CPU and the various levels of the memory hierarchy (caches, RAM, and disk) makes improving data locality a first-level concern. For many applications, this means improving the locality of the heap. While applications typically exhibit temporal locality, their spatial locality is dictated by the memory allocator, which determines where and how to lay out the application's dynamic data. This allocator-controlled locality can have a significant impact on the application's overall performance.

We present a new general-purpose memory allocator called *Vam* that improves data locality while providing low fragmentation. Vam increases page-level locality by managing the heap in page-sized chunks and aggressively giving up free pages to the virtual memory manager. By eliminating object headers, using a judicious selection of size classes, and by allocating objects using a reap-based algorithm [5], Vam improves cache-level locality.

We compare Vam to the low-fragmentation Linux allocator (DLmalloc) and to the page-level locality-improving FreeBSD allocator (PHKmalloc) [14], both of which we describe in detail. To our knowledge, PHKmalloc has not been discussed previously in the memory management literature. We build on these algorithms, incorporating their best features while removing most of their disadvantages.

Our experiments on a suite of memory-intensive benchmarks show that Vam consistently achieves the best performance. Vam performs on average 8% faster than DLmalloc and 4% faster than PHKmalloc when there is sufficient physical memory to avoid paging. When physical memory is scarce, Vam outperforms these allocators by over 10X and up to 2X, respectively. We show that part of this improvement is due to an unintended but fortunate synergy between Vam and the way Linux manages swap space, which holds evicted pages on disk. We call this phenomenon *swap prefetchability* and show that it leads to improved performance when paging.

2. Previous General-Purpose Memory Allocators

Before discussing Vam, we describe in detail the most influential allocators in its design. These are DLmalloc, which focuses on reducing fragmentation; PHKmalloc, which focuses on improving page-level locality; and reaps, which provide high-speed allocation and cache-level locality.

2.1 DLmalloc

DLmalloc is a widely-used `malloc` implementation written by Doug Lea [16]. It forms the basis of the Linux memory allocator included in the GNU C library. DLmalloc has been tuned over many years and is widely considered to be both among the fastest and most space-efficient allocators [5, 13]. The version we use in this study is the latest release, version 2.7.2.

DLmalloc is an approximate best-fit allocator with different be-

havior based on object size. *Small* objects (less than 64 bytes) are allocated from exact-size *quicklists*, linked lists of free objects. Requests for a *medium*-sized object (between 72 and 504 bytes) and certain other events trigger DLmalloc to *coalesce* the objects in these quicklists (combining adjacent free objects) in the hope that this reclaimed space can be reused for the medium-sized object. For medium-sized objects, DLmalloc performs immediate coalescing and *splitting* (breaking objects into smaller ones) and approximates best-fit. DLmalloc manages *large* objects (between 512 and 128K bytes) similarly, but places these in a group of free lists containing free objects of a particular size range. These size ranges are logarithmically spaced and DLmalloc sorts free objects within each range by size, so that the first object that fits is the best fit. *Very large* objects (128KB or larger) are allocated using `mmap`.

One notable implementation detail of DLmalloc shared by other allocators is that each object has a *header* that stores metadata containing the object's size and status. This metadata is also referred as *boundary tags* and simplifies coalescing. In DLmalloc, each object header is an 8-byte chunk placed before the object. This space overhead can become significant if an application allocates a large number of small objects. Placing the header next to the object itself also degrades data locality, because the header is only accessed by the allocator and not by the application accessing the object [9]. In other words, the header and the object have different access patterns and frequencies and, if put in the same cache line, may lower cache line utilization.

2.2 PHKmalloc

The PHKmalloc allocator was designed for the FreeBSD operating system by Poul-Henning Kamp [14]. As far as we are aware, this memory allocator has not previously been described in the literature. We describe the latest release (version 1.89) here.

Unlike DLmalloc, which disregards page boundaries, PHKmalloc's design is page-oriented. The central design goal of PHKmalloc was to minimize the number of pages accessed by both the application and the allocator [14]. The heap is a contiguous space divided into 4K pages and a table stores the status of these pages (empty or occupied).

In PHKmalloc, every object in a page is the same size. This organization allows PHKmalloc to avoid individual object headers by storing metadata such as object size at the start of the page, which can be located by bitmasking an object's address. The metadata field also contains a bitmap that records the status of each object (free or allocated). This technique of avoiding per-object headers is sometimes referred to as a BIBOP-style organization ("Big Bag of Pages" [10]) and has been employed by many memory managers, including the Boehm-Demers-Weiser conservative garbage collector [7] and the Hoard multiprocessor memory allocator [3].

PHKmalloc distinguishes just two object size classes: *small* (less than 2KB) and *large* (2KB or more). Like the BSD 4.2 allocator by Chris Kingsley [22], PHKmalloc rounds up small object requests to the nearest power of two and rounds large object requests up to the nearest multiple of the page size; the remainder in the last page is not reused. PHKmalloc keeps pages containing free space in a doubly-linked list sorted by address order, implementing the policy known as *address-ordered first-fit* [22].

PHKmalloc's rounding-up of object sizes makes it susceptible to considerable *internal* fragmentation (unused space inside of each object) or *page-internal* fragmentation (unused space at the end of the last page of a large object) [1]. In practice, however, the space saved by eliminating individual object headers is largely offset by this internal fragmentation.

On the other hand, using coarse size classes dramatically reduces the number of free lists, allowing the quick reuse of freed objects and reducing external fragmentation. In some situations, this can improve locality, as we show in Sections 4.3 and 4.5.

A key advantage of PHKmalloc's page-oriented design is that it allows the allocator to return to the kernel any free page via the `madvise` system call. After this call, although the page is still mapped from the kernel, its contents will not be written back to disk and so its physical space may be immediately reclaimed by the kernel. If the page is touched again, the virtual memory manager will materialize a demand-zero page.

2.3 Reaps

Reaps are a combination of regions and heaps that extend region semantics with individual object deletion [5]. A reap consists of a chunk of memory, a "bump" pointer set to the start of the chunk, and an associated heap. Allocation in a reap initially consists of bumping its pointer through the chunk of memory. Reaps add object headers to every allocated object (our adaptation of reaps does not; see Section 3.4). These headers contain metadata that allows the object to be subsequently placed on a heap. Reaps act like regions (performing pointer-bumping allocation) until a call to `reapFree` deletes an individual object. Freed objects are placed onto an associated heap. Subsequent allocations from that reap use memory from the heap until it is exhausted, at which point it reverts to region mode. Experimental results show that reaps capture most of the performance of region allocators [5].

3. Vam

Vam builds on previous allocator designs to achieve its goals of high performance and improved application-level locality both at the cache and page level. We implemented Vam using Heap Layers, a C++-based infrastructure for building high-performance memory managers [4]. Figure 1 presents an example of Vam's heap layout. The following is an overview of Vam's design, which we explore in detail in the rest of this section.

Fine-grained size classes: Vam improves cache utilization by using exact-fit size classes for objects up to 496 bytes in size, thus eliminating internal fragmentation. For larger objects, Vam also uses fine-grained size classes to provide efficient best-fit searches.

Page-based: Vam uses a page-oriented heap layout similar to PHKmalloc, but collocates large objects in contiguous regions to minimize page-internal fragmentation.

No object headers for small objects: Vam reduces cache pollution by eliminating object headers for all objects under 128 bytes.

Reap allocation: Vam uses a variant of reap allocation in each page both to improve throughput and to enhance cache locality.

Ordered per-size allocation: Vam maintains non-full pages for each small or medium size sorted in the order in which the pages become non-full. This ordering allows new objects to fill the free space in the front, improving locality. It also increases the likelihood that empty pages emerge from the end.

Aggressive discarding of empty pages: Whenever a page is made empty, Vam gives it back to the virtual memory manager.

3.1 Fine-Grained Size Classes

Like DLmalloc, Vam classifies object sizes into four categories: *small* (below 128 bytes), *medium* (between 128 and 496 bytes), *large* (between 504 bytes and 32KB), and *extremely large* (more

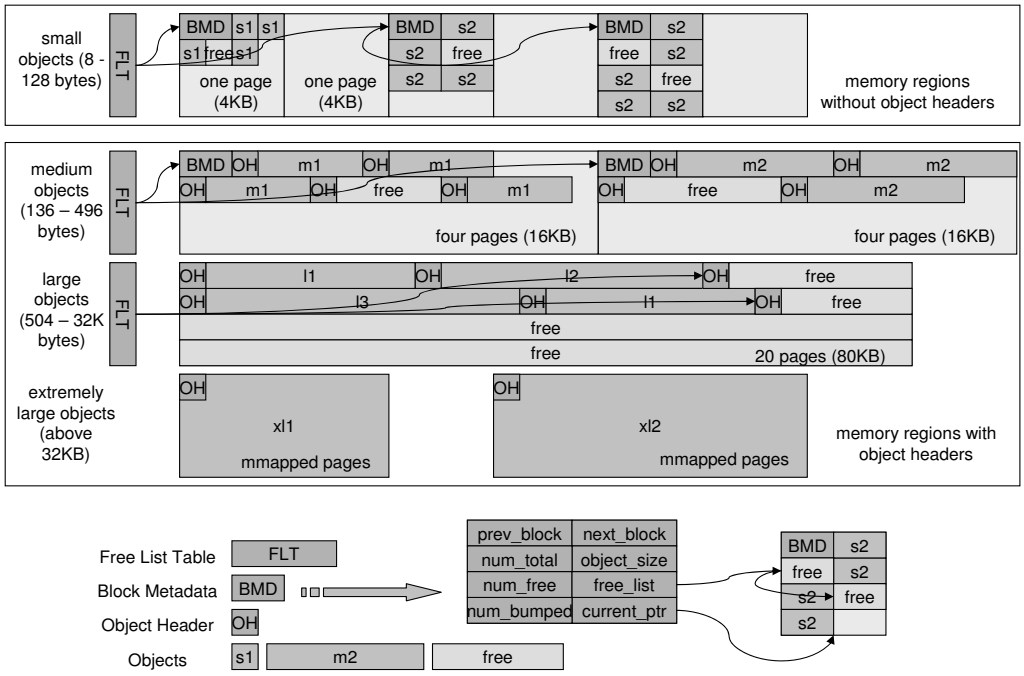


Figure 1: An example of Vam's heap layout (see Section 3).

than 32KB). These size boundaries are tunable parameters in the allocator.

Vam uses very fine-grained size classes for all small, medium and large objects – each size class is only 8 bytes apart. A *free list table* stores pointers to linked lists of the free space for each size class. Since Vam manages objects of each size category rather differently, we discuss the algorithms in detail below. Vam directly allocates extremely large objects from the kernel via the `mmap` system call and frees them using `munmap`.

Small and Medium Objects

Small and medium objects are allocated from segregated *page blocks*, groups of pages dedicated to each size class. Each page block is divided into equal-sized objects. The size of the page block is one page for small objects. For medium objects, it is four pages. This increased number of pages reduces page-internal fragmentation at the end of the block [1].

Page blocks of each size class are maintained in two linked lists. The *available list* contains page blocks with free space, while the *full list* contains page blocks with no remaining space. If the available list for a particular size is empty and a new allocation request arrives, Vam creates a new page block for that size which it adds to the available list.

Vam's fine-grained size classes (8 bytes apart) and exact-fit allocation eliminate internal fragmentation for small and medium objects, since the C standard requires that all objects returned by `malloc` be double-word (8-byte) aligned. Reducing fragmentation for these objects is important for improving overall cache utilization because most objects are small or medium-sized. In our benchmarks, 89.6% of all objects allocated are small (33.1% of space requested) and 6.4% are medium-sized (3.2% of space requested).

Nonetheless, using coarser size classes can also improve locality of reference. A wider size range in each size class allows quicker reuse of free space across these sizes, which may result in improved cache locality and page-level locality. In fact, we observe this phenomenon in the 253.perlbmk benchmark (see Sections 4.3 and 4.5).

Large Objects

As with small and medium objects, large object size classes are also only 8 bytes apart and each size class has a dedicated free list. Pointers to these free lists are also put into the free list table indexed by the sizes. Vam uses a best-fit algorithm for large objects. It linearly searches the free list table for the first non-empty list containing a free object large enough to satisfy the requested size. If the remaining space of the object is large enough to hold the smallest large object (i.e., 504 bytes), Vam splits the object and places the remaining space onto an appropriate free list.

This use of fine-grained size classes for large objects improves allocator-level locality. Because each size class provides an exact fit, no search within the size class is needed for a best fit. This can improve locality because such a search (as in DLmalloc) may visit several free objects before it finds a best fit and these free objects may be scattered in memory and have poor locality [9]. Vam only scans the free list table, which is a contiguous space and has good locality. However, this linear scan may occasionally visit a large number of table entries and flush caches. It is possible to solve this problem by hierarchically indexing into the table, but we have not implemented this optimization.

Allocation requests for large objects are rare and often have poor size locality. For example, applications may allocate large buffers of varied lengths corresponding to file inputs. Our use of a large number of size classes, however, does not lead to excessive exter-

nal fragmentation because Vam collocates large objects in *memory regions* (by default, 20 pages) shared by all these sizes and immediately coalesces freed objects. This aggressive coalescing greatly reduces fragmentation, and for these large objects, the per-byte cost for this coalescing is low.

Collocating large objects in shared memory regions may have another beneficial impact because it tends to align them randomly, thus reducing conflict misses. For example, a program may frequently access a particular field of a number of large objects. When objects are always aligned similarly, as in PHKmalloc, accesses to this field will always conflict in the cache. By avoiding fixed object alignment, Vam reduces the likelihood of this sort of conflict misses.

3.2 Page-Based Heap Management

Vam allocates small and medium-sized objects from segregated page blocks and large objects from memory regions, which are also multiples of pages. A *page manager* manages all these pages by recording the status information for each page in a *page descriptor table* and keeping consecutive free pages in a set of free lists.

We note that, in principle, segregating objects of different sizes could harm locality by preventing adjacent allocation of temporally-local objects of different sizes. This potential cost must be weighed against the locality and space benefit of eliminating external fragmentation. Wilson and others have observed a strong skew towards a small number of size classes, increasing the odds that temporally-local objects will be of the same size [12, 22].

3.3 Elimination of Object Headers for Small Objects

Vam uses the BIBOP technique to eliminate individual object headers only for small objects, where the resulting space savings and locality improvement are the most significant. All larger objects have per-object headers, simplifying deallocation and coalescing. To distinguish these two cases, Vam partitions the entire address space into 16MB areas and only allows homogeneous objects to be allocated in the same area. Vam uses a table that records whether objects in each area have headers. Because a one-byte flag is enough to hold the information for each area, only 256 bytes are needed to manage a 4GB address space. Although every object deallocation needs to perform a conditional check on the corresponding entry in this table, these checks have very good locality and the branch is highly predictable since most objects are small and do not have headers.

3.4 Reap Allocation

Unlike PHKmalloc, Vam does not use per-block bitmaps to track which objects are allocated or free in a page block. Instead, it uses a cheaper pointer-bumping allocation until the end of the page block is reached. It then reuses objects from a free list for that page block. This technique is a variant of the reap allocation as discussed in Section 2.3. The original reap algorithm adds per-object headers and employs a full-blown heap implementation to manage freed objects. Vam instead manages its (headerless) free objects by threading a linked list through them; the current release of Hoard (version 3.2.2) uses a similar strategy. Vam's use of a single size class per page block ensures that this approach does not lead to external fragmentation. Pointer-bumping allocation also improves cache locality by preserving allocation order.

3.5 Ordered Per-Size Allocation

To minimize page faults, Vam preferentially allocates small and medium objects from recently-accessed page blocks. It allocates from the first block in the available list until the block becomes full. It then moves the block to the full list and uses the next block in the available list, creating one if none exists. Vam places freed objects onto the appropriate per-block free list for reuse. When an object is freed to a previously-full page block, Vam moves it from the full list to the front of the available list. PHKmalloc uses a similar approach, but sorts non-full pages in an increasing address order. We do not sort the page blocks because it can be costly.

Page-level ordering ensures that new objects always fill free space in the page block in the front of the available list. Objects in an allocation sequence are then likely to be placed in a small number of pages, improving page-level locality. This ordered allocation also increases the chance that pages near the end become entirely free.

	176.gcc	197.parser	253.perlbmk	255.vortex
Execution Time	24s	275s	43s	62s
Instructions (billion)	40	424	114	102
VM Size	130MB	15MB	120MB	65MB
Max Live Size	110MB	10MB	90MB	45MB
Total Allocations	9M	788M	5.4M	1.5M
Alloc. Rate (#/sec)	373K	2813K	129K	30K
Avg. Size (bytes)	52	21	285	471

Table 1: CPU and memory allocation statistics of memory-intensive SPECint2000 benchmarks, run with DLmalloc.

3.6 Aggressive Discarding of Pages

When a page becomes empty, Vam uses the `madvise` call to discard it, thus reducing the application's virtual memory footprint. For regions of large objects, Vam discards those pages entirely within a freed object, that is, excluding per-object metadata.

Discarding empty pages from individual applications to the kernel can reduce system-wide memory pressure, benefiting either the application itself or other applications in the system. However, aggressive page discarding does add some runtime overhead. Each discard requires a system call. When the page is later reused, there is a cost in reassigning a physical page to the free page in the kernel (soft page fault handling and page zeroing). When weighed against the penalty caused by paging, these overheads are low. Ideally, we would discard pages only in response to memory scarcity, but we have not yet explored this possibility.

4. Experimental Evaluation

To evaluate the efficacy of Vam's design, we sought to answer the following questions: can Vam reduce total application execution time, increase cache-level locality, maintain low fragmentation, and reduce paging when under memory pressure?

We evaluate Vam's with four memory-intensive applications from the SPEC CPU2000 benchmark suite [19]: 176.gcc, 197.parser, 253.perlbmk, and 255.vortex. The other benchmarks in the suite either use very little memory or only allocate a few memory buffers at the start of execution [20], so for those applications, the choice of allocator has essentially no impact. Whenever multiple inputs were available, we use the reference input that consumes the most memory. These are *scilab.i*, *ref.in*, *splitmail.pl 850 5 19 18 1500*, and *lendian1.raw*, respectively. Table 1 summarizes the CPU and memory allocation statistics of the benchmarks.

The original 176.gcc and 197.parser applications use custom memory allocators. 176.gcc uses *obstacks* and 197.parser uses a custom allocator called *xalloc* [5]. The use of custom allocation means that the original applications make only occasional calls to `malloc`. We modified these applications to use general-purpose memory allocators. This modification is trivial for 197.parser because its custom allocator has the same interface and semantics as `malloc` and `free`. Replacing *xalloc* actually decreases the max-

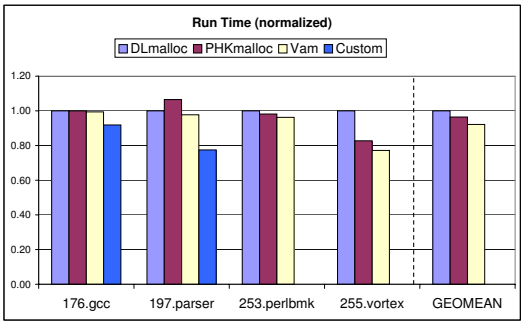


Figure 2: Total execution time, normalized to DLmalloc.

imum virtual memory usage of 197.parser from 30MB to 15MB. Replacing the obstack allocator in 176.gcc is more complicated since it has a different interface and semantics than the general-purpose memory allocator. To replace it, we use an obstack layer that directly invokes `malloc` for individual objects [5]. This layer requires additional metadata and thus increases 176.gcc's peak memory usage from 85MB to about 130MB.

We use a Dell Optiplex SX270 as our experimental platform (3.0GHz Pentium 4, 1GB RAM, 40GB 5400RPM hard drive, Linux version 2.4.24). The Pentium 4 has an 8KB L1 data cache (64-byte cache lines, 4-way set-associative) and a 512KB L2 cache (64-byte cache lines, 8-way set-associative). All memory allocators are compiled into shared libraries at the highest optimization level with gcc version 3.2.2 and preloaded into memory before the applications via `LD_PRELOAD`.

We measure total execution time using `/usr/bin/time`, and measure instructions retired, L1/L2 cache misses, and data TLB misses using the Pentium 4's on-chip performance counters. We use the *perfctr* patch for Linux and the *perfex* tool [17] to set the performance counters according to the manufacturer's manual [11]. We run each experiment five times with the machine in single-user mode and report the mean. The mean deviations in the results are below 1%.

4.1 Total Execution Time

Figure 2 presents the total execution times normalized to DLmalloc. Vam consistently improves application performance over both PHKmalloc and DLmalloc. Vam's improvement over PHKmalloc ranges from 1–8%, and improves over DLmalloc by 1–23%. On average, Vam is 4% and 8% faster than PHKmalloc and DLmalloc, respectively. The custom memory allocators in 176.gcc and 197.parser are faster than the general-purpose ones: the obstack allocator in 176.gcc is 8% faster than DLmalloc and the xalloc allocator in 197.parser is 23% faster. The custom allocators improve performance of these two benchmarks because both benchmarks are very allocation intensive (see Table 1). In fact, 197.parser is so allocation-intensive that the number of cycles executed by the allocator dictates its performance. We attribute the difference between this result and that obtained by Berger et al. [5] (showing a smaller gap between DLmalloc and xalloc) to the different compiler infrastructure and to our use of shared objects for the allocators, which precludes link-time optimizations.

4.2 Cache Locality

We measure both L1 and L2 cache locality for the different allocators. Figure 3(a) shows L1 cache misses normalized to DLmalloc. Vam reduces L1 cache misses for two of the four benchmarks. This result is due to Vam's reduction of internal fragmentation and elimination of object headers (see Section 4.3). However, Vam significantly increases L1 cache misses for one benchmark, 253.perlbmk. This benchmark allocates from a wide range of sizes, and Vam's use of fine-grained size classes causes more cache traffic than DLmalloc. This result is somewhat misleading: 253.perlbmk's L1 cache miss rate is low for all allocators, and so has little impact on total execution time.

PHKmalloc increases L1 cache misses in three of the four benchmarks. This increase is due to internal fragmentation caused by PHKmalloc's coarse size classes (also see Section 4.3). PHKmalloc reduces L1 cache misses for only one benchmark, 197.parser. It primarily allocates small objects (8, 16 and 24 bytes). These objects fit into PHKmalloc's power-of-two size classes with little fragmentation, and the lack of object headers leads to efficient cache line utilization both for PHKmalloc and for Vam.

Unlike their effect on L1 cache misses, both Vam and PHKmalloc significantly reduce L2 cache misses, as Figure 3(b) shows. On average, Vam reduces L2 cache misses by 39% over DLmalloc. This cache-level locality improvement is more significant in 253.perlbmk and 255.vortex than in 176.gcc and 197.parser. For 176.gcc, the obstack allocator produces the fewest cache misses. This result is partially due to the extra metadata required to simulate obstack semantics. Unlike L1 locality, the L2 cache performance here is strongly correlated to application run time performance. However, PHKmalloc's locality improvement is offset by its excessive number of instructions, particularly in 197.parser. We also measured data TLB misses; these exhibit nearly identical trends, so we do not report them here.

Summary: Vam generally provides better L1 cache locality than the other allocators. The use of a page-oriented heap layout improves L2 cache locality for both PHKmalloc and Vam, although Vam's improvement is somewhat greater.

Variant	Description
PHK_sc	size classes every 8 bytes instead of 2^x
PHK_reap	replaces bitmap operations with reap allocation [5]
PHK_sc_reap	combines PHK_sc and PHK_reap
Vam_small+header	adds 8-byte headers to small objects
Vam_bitmap	replaces reap allocation with bitmap operations for small and medium objects

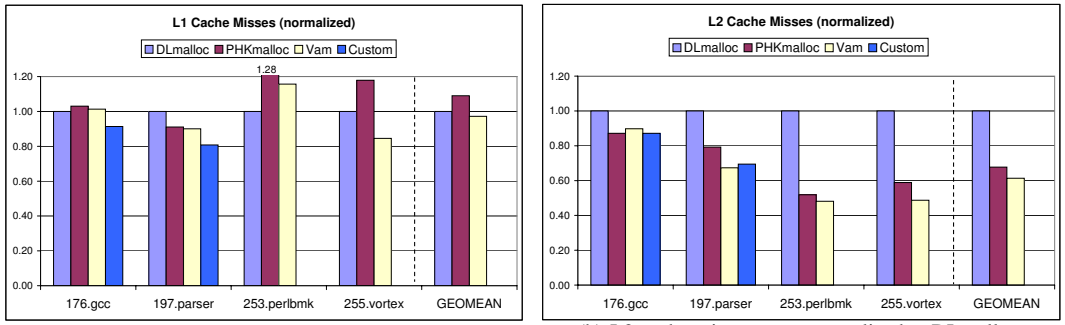
Table 2: Variants of PHKmalloc and Vam (see Section 4.3).

4.3 Performance of Allocator Variants

To evaluate the effects of Vam's design decisions, we developed several variants of both PHKmalloc and Vam, summarized in Table 2. These variants let us quantify the impact of the choice of fine-grained size classes, reap-based allocation and object header elimination. Figures 4 and 5 present the L2 cache misses, instruction counts and run time performance of these PHKmalloc and Vam variants. We do not show the L1 cache misses as they exhibit nearly identical trends as the L2 results. Note that all results are normalized to their respective original versions, e.g., PHKmalloc variants are normalized to PHKmalloc.

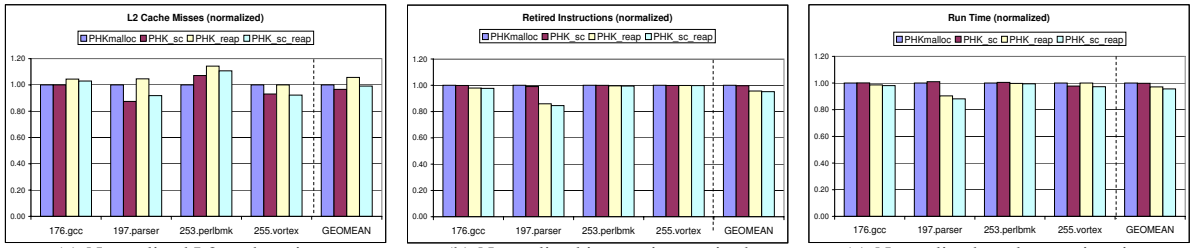
Impact of Size Classes and Reaps: PHKmalloc

As Figure 4(a) shows, PHK_sc (fine-grained size classes) reduces cache misses in three of the four benchmarks. The exception is 253.perlbmk, which uses more object sizes than the other benchmarks. The coarser size classes in the original PHKmalloc allow quicker reuse of freed space within each size class, yielding bet-



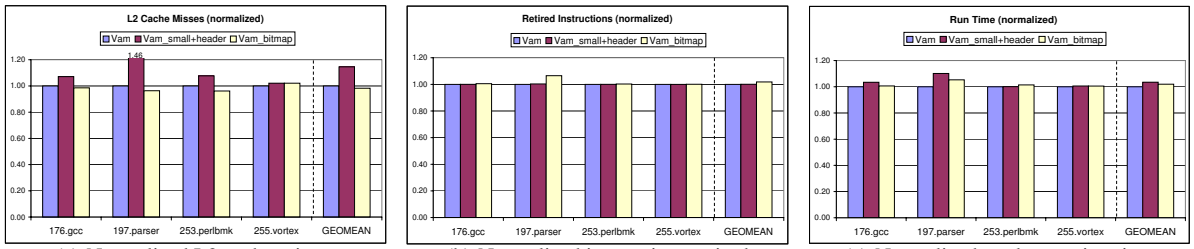
(a) L1 cache miss counts, normalized to DLmalloc. (b) L2 cache miss counts, normalized to DLmalloc.

Figure 3: Cache-level locality results.



(a) Normalized L2 cache misses (b) Normalized instructions retired (c) Normalized total execution time

Figure 4: Comparison of PHKmalloc variants, normalized to the original.



(a) Normalized L2 cache misses (b) Normalized instructions retired (c) Normalized total execution time

Figure 5: Comparison of Vam variants, normalized to the original.

ter cache locality. Although PHK_sc's changes in cache misses do not notably affect the overall run times shown in Figure 4(c), it greatly improves the space efficiency over the original allocator and achieves better VM performance when under memory pressure.

PHK_reap (replacing bitmap operations with reap allocation) reduces instructions executed by 14% for 197.parser and runs 10% faster than the original PHKmalloc. On average, this variant improves application performance by 3%. However, because this modification adds extra memory accesses, it also increases cache misses for most of the benchmarks except 255.vortex. This increase is the greatest for 253.perlbmk. However, because the absolute number of cache misses is small for 253.perlbmk, these extra misses do not affect run time.

The PHK_sc_reap variant, combining the changes in PHK_sc and PHK_reap, shows that these improvements are generally complementary. On average, PHK_sc_reap improves run time performance by 4%. It noticeably reduces cache misses in 197.parser and 255.vortex and instructions in 176.gcc and 197.parser.

Impact of Headers and Bitmaps: Vam

Figures 5(a) and 5(c) show that adding headers to the small objects in Vam results in an average increase in L2 cache misses of 15% and a 3% increase in run times. The impact of adding headers is the greatest for 197.parser, increasing run time by 10%. The average object size in 197.parser is only 21 bytes and the extra headers substantially increase its working set.

Figure 5(b) shows that Vam_bitmap significantly increases the number of instructions executed in 197.parser. On average, this variant reduces L2 cache misses by 2% and increases the instructions by 2%, resulting in a 2% increase in run time.

Summary: The use of fine-grained size classes and elimination of object headers generally improve cache locality and reduce total runtime. The choice between bitmap operations and reap-like allocation is a trade-off. Vam currently uses reaps, but trading CPU instructions for fewer memory accesses during allocation may eventually prove more beneficial.

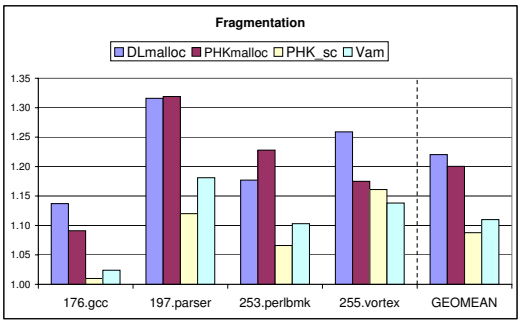


Figure 6: Fragmentation results.

4.4 Fragmentation

We next evaluate the effect of allocator design on memory fragmentation. Our measurement of fragmentation is derived from Wilson and Johnstone [13], who measure the amount of memory used by the allocator relative to the live data of the application. However, to better evaluate the space efficency of memory allocators, we refine the measurements of the application's live data and memory usage. Unlike Wilson and Johnstone, we measure the amount of live data as only that requested by the application, not including any space overhead imposed by the allocator implementation (e.g., object headers). We then measure the application's memory usage by tracking the number of pages in use. In use pages are those mapped from the kernel and touched, but not discarded. Pages mapped but never touched do not have physical space allocated; discarded pages have their previously-allocated memory reclaimed, thus do not consume any physical space. This view of application memory usage is from the VM manager's perspective and more accurately reflects the actual resource consumption.

We compare four allocators here: DLmalloc, PHKmalloc, Vam and the PHK_sc variant of PHKmalloc. Figure 6 shows the application's maximum number of pages in use divided by its maximum live data (in pages). We were surprised to see that DLmalloc, an allocator known for low fragmentation, in fact leads to the highest fragmentation on average. One reason for this is the space overhead of the per-object headers. More importantly, DLmalloc is unable to distinguish and discard any free pages it may have. This limitation of DLmalloc makes it possible for its free pages to occupy physical memory even when the system is under heavy memory pressure. PHKmalloc overcomes both of these shortcomings. However, its coarse size classes lead to internal fragmentation that negates its other advantages. Our PHK_sc variant uses fine-grained size classes and on average, yields the lowest fragmentation (under 10%). Vam combines these fragmentation-reducing features and nearly matches PHK_sc's low fragmentation.

4.5 Performance While Paging

The memory allocator's impact on space consumption and data locality is especially significant when the system is under memory pressure. If the application's footprint can not fit into available RAM, its performance will start to degrade because of paging. This performance degradation can be severe if the application's page-level locality is poor. To evaluate the application performance while paging, we launch a process that pins down a specified amount of RAM, leaving the desired amount available for the application.

Figure 7 shows the run times of the four SPEC benchmarks under a range of available RAM sizes, using different memory allocators. The rightmost point of each line shows the run time of the application with sufficient RAM to run without paging. As available memory is reduced (moving left), application performance degrades. This performance degradation is markedly different with different memory allocators, except for 176.gcc, where all the allocators degrade similarly with reduced RAM (Figure 7(a)). Of the general-purpose allocators, Vam delivers the best performance across a wide range of available RAM.

Recall that for 176.gcc, we needed to add extra metadata to simulate the obstack semantics with general-purpose allocators. The original obstack allocator thus performs better than the general-purpose allocators when RAM is scarce. Nonetheless, all of the general-purpose allocators similarly preserve application locality because of the clustered allocations and deallocations in 176.gcc. The slight difference between these allocators is largely due to their respective space efficiency, for which the original obstack custom allocator is the best.

The story is different for the other custom allocator. As Figure 7(b) shows, 197.parser's custom allocator (xalloc) requires substantially more RAM to avoid paging and performs much worse than the general-purpose allocators as available RAM is reduced. This poor performance is due to a limitation in xalloc. Unlike the general-purpose allocators, xalloc can not reuse heap space immediately after objects are freed. Instead, it must wait until consecutive objects at the end of the heap are all free, at which point it reuses memory from after the last object in use. While this strategy is effective when physical memory is ample, under memory pressure, it degrades performance dramatically.

Figure 7(c) and Figure 7(d) highlight the effectiveness of both PHKmalloc's and Vam's page discarding algorithms. DLmalloc suffers a 5x slowdown when available physical memory is reduced to just below 80MB for 253.perlbmk, while PHKmalloc and Vam run at the same speed with just 30 to 40MB. With both of those allocators, 253.perlbmk exhibits a more graceful performance degradation than when using DLmalloc. For 255.vortex, Vam performs better than the other two allocators over all available RAM sizes we tested. DLmalloc required about 6MB more available RAM to achieve Vam's performance. Only the page discarding algorithms play a role here: 255.vortex's average object size is 471 bytes, so DLmalloc's 8-byte object headers have little impact.

Note that, for 253.perlbmk, PHKmalloc degrades performance slightly less than Vam when available RAM is less than 60MB. This relative improvement for PHKmalloc is because its coarse size classes improve the locality of this particular benchmark. When running 253.perlbmk with PHK_sc (with fine-grained size classes), we find that the performance degradation curve is then very close to that of Vam across all memory sizes.

4.6 Page-Level Locality

In this section, we explore the effect of the allocator's page-level locality on the application's virtual memory performance in more detail by using an LRU simulator and a prefetching simulator. Approximate LRU page replacement and swap disk prefetching are two commonly used components in many VM managers, including the one in Linux.

LRU Simulation

We first gather application page-level references into the heap using a tool that intercepts system memory calls (`brk`, `sbrk`, `mmap`, `munmap`, and `madvise`) to keep track of heap pages currently mapped from the kernel and traps memory references by page protection. We use the SAD (Safely-Allowed-Drop) algorithm to reduce the trace to a manageable size [15].

We then run these traces through an LRU simulator to generate page miss curves that indicate the number of misses (page faults)

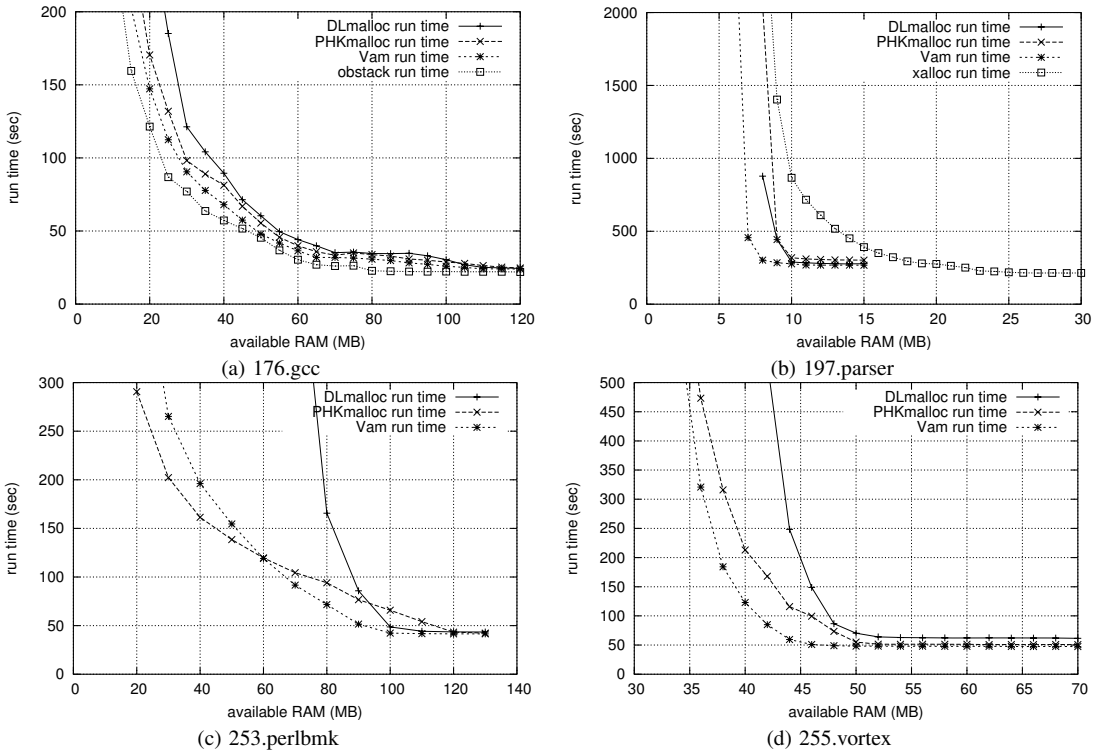


Figure 7: Performance using different memory allocators over a range of available RAM sizes.

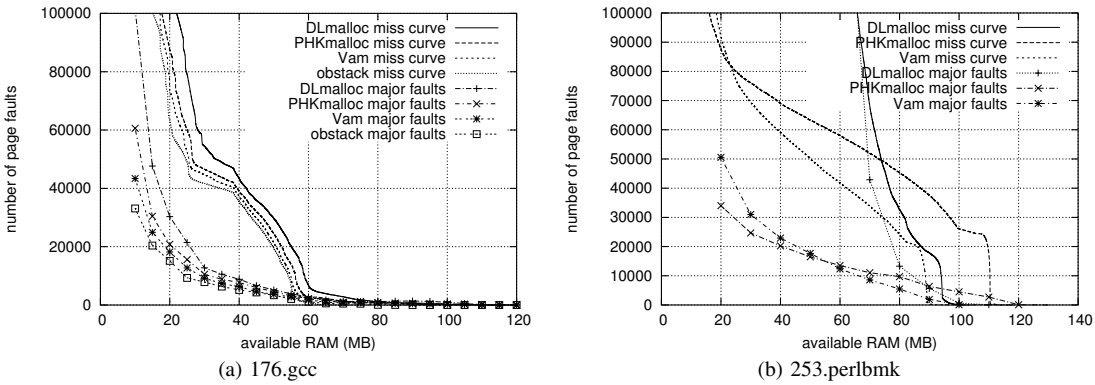


Figure 8: Simulated page miss curves ignoring prefetching versus actual major page faults in a real system with prefetching.

that would arise for every possible size of available memory. Our LRU simulator is similar to that described by Yang et al. [23]. We use placeholders in the LRU queue for pages discarded by `madvise` or unmapped by `munmap`/`sbrk`. These placeholders allow us to more accurately approximate a real VM manager.

We compare the miss curves generated from the simulator with the actual number of page faults. Actual page faults are the major (hard) page faults measured in the experiments we describe in Section 4.5. For two of our benchmarks, 197.parser and 255.vortex, the simulated miss curves are nearly the same as the actual page faults (except for the xalloc custom allocator in 197.parser).

However, for 176.gcc and 253.perlbmk, the actual page faults are far fewer than the simulated ones, as Figure 8 shows. For example, for 176.gcc with 40MB of RAM, the simulated faults are around 40,000 while the actual page faults measured are under 10,000. The is due to the swap prefetching employed by the Linux VM manager but not in our LRU simulator. To verify this, we turn off prefetching in Linux, and re-run the paging experiments. The actual number of page faults then closely matches the simulated results for all benchmarks and allocators.

Prefetching Simulation

As Figure 8 shows, swap disk prefetching can significantly reduce page faults. The effectiveness of prefetching is determined by the locality of page misses on the swap disk. If page misses require contiguous pages on the swap disk to be swapped in, prefetching

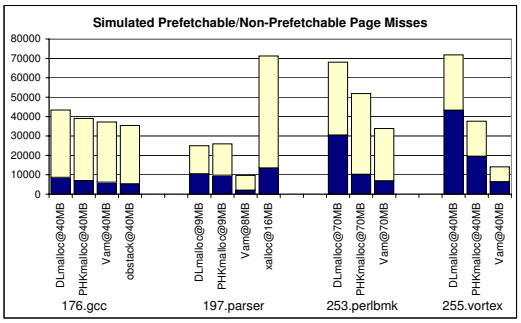


Figure 9: Swap prefetchability: each bar shows simulated prefetchable misses (top) and non-prefetchable misses (bottom).

will be effective. Page allocation on the swap disk is managed by the VM manager. The Linux VM manager (particularly in the 2.4 kernels) attempts to cluster pages that are adjacent in virtual address space to store them contiguously on disk [6]. For this reason, the application's page-level locality also affects the effectiveness of prefetching in the kernel when the system is paging.

We measure the application's *swap prefetchability* by simulating the Linux swap prefetching, assuming a strict address-preserving mapping from the application's address space to the swap disk. We gather the application's page references that would result in a miss for a given memory size in the LRU simulation and feed this page miss trace to our prefetching simulator. The simulator maintains a buffer of the last 256 pages prefetched from disk. A page miss is *prefetchable* if it is in the buffer; otherwise, it is *non-prefetchable*. A prefetchable miss causes the page to be removed from the buffer while a non-prefetchable miss causes adjacent pages in the group of 8 to be prefetched from swap disk into the buffer, replacing the oldest pages if the buffer is full. Subsequent page misses on these still in the buffer are therefore prefetchable.

Figure 9 presents our swap prefetchability results for different allocators with specific memory sizes noted on the figure. For 176.gcc and across all allocators, more than 80% of the misses are prefetchable. This prefetchability is due to 176.gcc's strong locality in its obstack-style memory allocation. The original version of 197.parser (using xalloc) also exhibits great prefetchability (81% prefetchable). Vam nearly matches this prefetchability (79% prefetchable) while the other two general-purpose allocators result in less prefetchability. With PHKmalloc and Vam, 253.perlbmk has few non-prefetchable misses – as many as 80% of the misses are prefetchable. However, its prefetchability is much less with DLmalloc (only 55% prefetchable). This result demonstrates that 253.perlbmk's data locality is better preserved by PHKmalloc and Vam than by DLmalloc.

255.vortex has much less prefetchability than the other applications: about 45% of the misses are non-prefetchable with VAM, 52% with PHKmalloc, and 60% with DLmalloc. In fact, 255.vortex's poor page-level locality is also reflected in the steep VM performance degradation curves in Figure 7(d) and simulated miss curves. This occurs either because 255.vortex's data locality is instrinsically poor or because it is not well preserved by any of the allocators.

Note that this prefetchability model assumes ideal swap page placement. The real VM manager may not be able to prefetch all the prefetchable misses. Nevertheless, our model appears to reflect observed application VM performance on a real system. We attribute the improved prefetchability in PHKmalloc and Vam to their page-oriented design and ordered allocation at the page level.

5. Related Work

There has been extensive research on dynamic memory allocation. In their well-known survey paper, Wilson et al. devote most of their attention to the question of fragmentation, which they identify as the most important metric for evaluating memory allocators [22]. Johnstone and Wilson in their subsequent studies evaluate a wide range of allocation policies using actual C/C++ programs and claim that fragmentation is near zero, given a good choice of allocation policy [12, 13]. While they argue that reducing fragmentation generally improves locality, we show that Vam's approach is more effective.

Most previous researchers have attacked the problem of locality in memory allocation either by improving the locality of the allocator itself or by using extra information such as programmer hints or profiles to guide placement decisions. Grunwald and Zorn investigate the locality impact of allocation algorithms by simulating caches using reference traces [9], and conclude that sequential-fit search schemes are the primary culprit for poor allocator locality. Their benchmark suite is highly allocation-intensive, causing locality effects in the allocator to dominate. Vam's algorithms focus instead on the effect of allocator data layout decisions on the application's overall locality, rather than on locality within the allocator. Our benchmark suite of memory-intensive programs is also less allocation-intensive, emphasizing the impact of allocator layout policies.

Chilimbi et al. describe `ccmalloc`, a memory allocator that allows the programmer to help the allocator group objects with temporal locality [8]. Truong et al. describe a memory allocator that separates the hot and cold fields of objects into different cache lines [21]. Both of these approaches improve cache-level locality but require programmer intervention. Vam's approach is largely orthogonal. Its use of the standard `malloc` interface allows it to be used to improve the locality of unaltered programs. It should be possible to build custom locality-improving allocators like `ccmalloc` on top of Vam, but we do not investigate that possibility here.

Barrett and Zorn use a profile-based approach that predicts object lifetime at allocation time and segregates short-lived objects from long-lived objects in the heap [2]. Their system improves locality and space efficiency while reducing allocation cost, but requires profiling and imposes runtime overhead. Zorn and Seidl extend this approach by incorporating the reference behavior and lifetime prediction gathered during profiling to guide memory allocation and improve virtual memory performance [18, 24]. Their method also imposes modest runtime overhead. Vam's approach avoids the need for profiling and improves application performance both in the presence and absence of virtual memory paging.

6. Conclusions

Because of its determining role in creating spatial locality, dynamic memory allocation can have a significant impact on performance. We present Vam, a memory allocator that improves application locality and performance both at the cache level and at the virtual memory level. We explore the impact of Vam's design decisions and find that its fine-grained size classes, reap-like allocation, object header elimination, and page-oriented design all contribute to its effectiveness. Our experiments show that Vam can improve the performance even of relatively non-allocation-intensive applications.

7. References

[1] D. F. Bacon, P. Cheng, and V. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'2003)*, San Diego, CA, June 2003. ACM Press.

[2] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, pages 187–196, Albuquerque, NM, June 1993. ACM Press.

[3] E. Berger, K. McKinley, R. Blumofe, and P. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, Cambridge, MA, Nov. 2000.

[4] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Snowbird, Utah, June 2001. ACM Press.

[5] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *OOPSLA'02 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Seattle, WA, Nov. 2002. ACM Press.

[6] D. Black, J. Carter, G. Feinberg, R. MacDonald, S. Mangalat, E. Sheinbrood, J. Sciver, and P. Wang. OSF/1 virtual memory improvements. In *Proceedings of the USENIX Mac Symposium*, pages 87–103, November 1991.

[7] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[8] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 1–12, Atlanta, May 1999. ACM Press.

[9] D. Grunwald, B. Zorn, and R. Henderson. Improving the cache locality of memory allocation. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, pages 177–186, Albuquerque, NM, June 1993. ACM Press.

[10] D. R. Hanson. A portable storage management system for the Icon programming language. *Software Practice and Experience*, 10(6):489–500, 1980.

[11] IA-32 Intel architecture software developer's manual, volume 3: System programming guide. `ftp://download.intel.com/design/Pentium4/manuals/25366814.pdf`.

[12] M. S. Johnstone. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, University of Texas at Austin, Dec. 1997.

[13] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: Solved? In R. Jones, editor, *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 26–36, Vancouver, Oct. 1998. ACM Press.

[14] P.-H. Kamp. Malloc(3) revisited. `http://phk.freebsd.dk/pubs/malloc.pdf`.

[15] S. F. Kaplan, Y. Smaragdakis, and P. R. Wilson. Flexible reference trace reduction for VM simulations. *ACM Trans. Model. Comput. Simul.*, 13(1):1–38, 2003.

[16] D. Lea. A memory allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html`.

[17] M. Pettersson. The perfctr patch for Linux and the perfex tool. `http://user.it.uu.se/~mikpe/linux/perfctr/`.

[18] M. L. Seidl and B. G. Zorn. Implementing heap-object behavior prediction efficiently and effectively. *Software Practice and Experience*, 31(9):869–892, 2001.

[19] SPEC CPU2000. `http://www.spec.org/osg/cpu2000/`.

[20] SPEC CPU2000 memory footprint. `http://www.spec.org/osg/cpu2000/analysis/memory/`.

[21] D. N. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. In *IEEE PACT*, pages 322–329, 1998.

[22] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In H. Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, Sept. 1995. Springer-Verlag.

[23] T. Yang, E. D. Berger, M. Hertz, S. F. Kaplan, and J. E. B. Moss. Automatic heap sizing: Taking real memory into account. In A. Diwan, editor, *ISMM'04 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, Vancouver, Oct. 2004. ACM Press.

[24] B. Zorn and M. Seidl. Segregating heap objects by reference behavior and lifetime. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.