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

Because most application data is dynamically allocated, the memory manager 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 information supplied by the programmer or obtained by profiling. We present a high-performance memory allocator that builds on previous allocator designs to achieve low fragmentation while transparently improving application locality. Our allocator, called *Vam*, improves 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 fine-grained size classes, and by allocating objects using a reap-based algorithm, Vam improves cache-level locality. 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. We show that synergy between Vam's layout algorithms and the Linux swap clustering algorithm increases its *swap prefetchability*, further improving its performance when paging.

1. Introduction

Explicit memory managers have traditionally focused on addressing the problem of fragmentation, discontiguous free chunks of memory. 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 [15, 16, 19].

This material is based upon work supported by the National Science Foundation under CAREER 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.

Submitted to MSP 2005 Chicago, IL 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 most applications, this means improving the locality of the heap. While applications typically exhibit temporal locality, 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.

In this paper, we present a new general-purpose memory allocator called *Vam* that improves data locality while providing low fragmentation. Vam improves 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 [9], 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) [17], 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. 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 [24]. Johnstone and Wilson in their subsequent studies evaluate a wide range of allocation policies using actual C/C++ programs and argue that fragmentation is near zero, given a good choice of allocation policy [15, 16]. 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 [13], and conclude that best-first search allocation 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 [12]. Truong et al. describe a memory allocator that separates the hot and cold fields of objects into different cache lines [23]. 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 which predicts object lifetime at allocation time and segregates these short-lived in the heap [6]. 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 [26, 22]. Their method also imposes some runtime overhead, which may have an adverse effect on the application performance. Vam's approach avoids the need for profiling and improves application performance both in the presence and absence of virtual memory paging.

3. General-Purpose Memory Allocators

Vam builds on previous allocator designs to achieve its goals of high performance and improved application-level locality. The most influential allocators in its design 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.

3.1 DLmalloc

DLmalloc is a widely-used malloc implementation written by Doug Lea [19]. 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 [9, 16]. The version we use in this study is the latest release, version 2.7.2.

DLmalloc is an approximate best-fit allocator with different behavior based on object size. *Small* objects (less than 64 bytes) are allocated from exact-size quicklists. 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 chunks of a particular size range. These size ranges are logarithmically spaced and DLmalloc sorts free chunks within each range by size, so that the first chunk that fits is the best fit. *Very large* objects (128KB or larger) are allocated and freed using `mmap`.

One notable implementation detail of DLmalloc common to 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. 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 [13]. 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.

3.2 PHKmalloc

The PHKmalloc allocator was designed for the FreeBSD operating system by Poul-Henning Kamp [17]. As far as we are aware, this memory allocator has not previously been described in the literature. We describe the current version here (1.89).

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 [17]. The heap is a contiguous space divided into 4K pages and a table stores the status of these pages (empty or occupied). Every object on 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 the object's address. The metadata field also contains a bitmap to record 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" [14]) and has been employed by many memory managers, including the Boehm-Demers-Weiser conservative garbage collector [11] and the Hoard multiprocessor memory allocator [7].

PHKmalloc distinguishes just two object size classes: *small* (less than 2KB) and *large* (2KB or more). Like the BSD 4.2 (Kingsley) allocator [24], 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*.

PHKmalloc's rounding-up of object sizes makes it susceptible to considerable *internal* fragmentation (unused space inside of each chunk) or *page-internal* fragmentation (unused space at the end of the last page of a large object) [4]. In practice, 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 chunks and reducing external fragmentation. In some situations, this can improve locality, as we show in Section 5.3 and Section 5.5.

A key advantage of PHKmalloc's page-oriented design is that it allows the allocator to discard empty pages via the `madvise` system call. In this case, although this page is still mapped from the kernel, the previously-allocated RAM space may be reclaimed by the kernel and the contents do not need to be written back to swap. The underlying physical page can thus be immediately reused. If the page is touched again, the virtual memory manager will materialize a demand-zero page.

3.3 Reaps

Reaps are a combination of regions and heaps that extend region semantics with individual object deletion [9]. 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. These headers contain metadata that allow the object to be subsequently placed on the heap. Reaps act like regions (performing pointer-bumping allocation) until a call to `reapFree` deletes an individual object. Reaps place freed objects 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.

4. Vam

The key design goal for Vam was to enhance application-level locality at the cache and page level while delivering high performance over a range of memory sizes. In particular, we wanted its performance to exceed that of other fast allocators both when there is enough physical memory to hold the entire heap and when physical memory is scarce.

We implemented Vam using Heap Layers, a C++-based infrastructure for building high-performance memory managers [8]. 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 size classes for objects up to 496 bytes in size, thus eliminating internal fragmentation.

Page-based: Vam uses a page-oriented heap layout similar to PHKmalloc, but uses a larger number of pages for large objects 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 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 improves locality and allows new objects to fill the free space in the front, increasing 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.

Approximate address-ordered first-fit at page-level: Vam maintains free pages in sorted order and preferentially allocates from the front, improving performance when paging by increasing swap prefetchability (Section 5.6).

4.1 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 than 32KB). These size boundaries are tunable parameters in the allocator. Each size class has two associated linked lists of blocks, groups of pages containing objects dedicated to that size class. The *available list* contains blocks with free space, while the *full list* contains blocks with no remaining space.

To improve cache line utilization, Vam uses much finer size classes than either DLmalloc or PHKmalloc. For small and medium objects, each size class is only 8 bytes apart. Fine-grained size classes eliminate internal fragmentation by providing exact fits for small and medium object allocation requests, 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 requested are small and 6.4% are medium-sized.

Nonetheless, using coarser size classes could improve locality of reference. A wider size range in each size class allows quicker reuse of free space across these sizes, which could result in improved cache locality and page-level locality. We have observed this phenomenon in the 253.perlbmk benchmark.

4.2 Large and Extremely Large Objects

Like small and medium objects, large object size classes are only 8 bytes apart and each size class has a dedicated free list. Vam uses a best-fit algorithm for large objects. It linearly searches the free list table for the first non-empty list containing a chunk large enough to satisfy the given size request. If the remaining space is large enough to hold the smallest large object (i.e., 504 bytes), Vam splits the chunk and places the remaining space onto the appropriate free list.

This use of fine-grained size classes for large objects also 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 chunks before it finds a best fit and these free chunks may be scattered in memory and have poor locality. 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 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. Vam collocates large objects in memory regions shared by all these sizes and aggressively coalesces free chunks in deallocation. This aggressive coalescing reduces fragmentation, and for these large objects, the per-byte cost for this coalescing is low.

Collocating large objects in large memory regions may have another beneficial impact because it tends to align them randomly. This alignment may reduce conflict misses. For example, if a program accesses some field of one type much more frequently than the other fields, and if the objects of that type are always regularly aligned (e.g., at the page boundary), some cache lines may suffer excessive conflict misses while others may be under-utilized. A more random alignment can map such hot fields in large objects more evenly.

Finally, Vam directly allocates extremely large objects from the kernel via the `mmap` system call and frees them using `munmap`.

4.3 Page-Based Heap Management

Vam allocates small and medium-sized objects from page-aligned blocks similarly to PHKmalloc. A block is one page for small objects. For medium objects, it is four pages; this reduces page-internal fragmentation at the end of the block [4]. Each block is divided into equal-sized chunks. This division makes it impossible to fragment memory inside a block.

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-

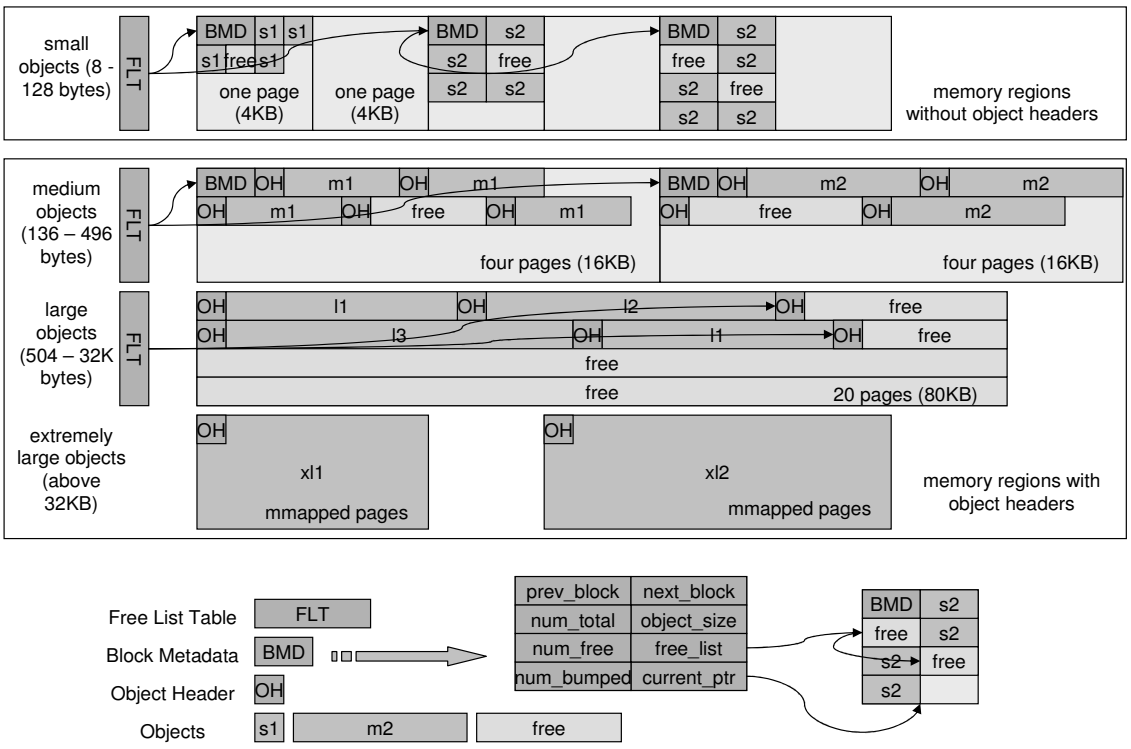


Figure 1: An example of Vam's heap layout (see Section 4).

local objects will be of the same size [15, 24].

4.4 Elimination of Object Headers for Small Objects

Like PHKmalloc, Vam uses the BIBOP technique to eliminate individual object headers and locates metadata at the beginning of each block. Vam uses this approach only for small objects, because the resulting space savings and locality improvement are most significant for small objects. Larger objects each have per-object headers, simplifying coalescing. To distinguish the two cases, Vam partitions the entire address space into 16MB regions and uses a table to record the type of objects in each region. Because a one-byte flag is enough to hold the information for each region, this table is only 256 bytes for 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. Since most objects do not have headers, this branch is also highly predictable.

4.5 Reap Allocation

Unlike PHKmalloc, Vam does not use per-block bitmaps to track which objects are free or allocated. Instead, it uses a cheaper pointer-bumping allocation until the end of the block is reached. It then reuses objects from a free list for that block. This technique is a variant of reap allocation [9]. 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. Vam's use of a single size class per block ensures that this approach does not lead to external fragmentation. Pointer-bumping also improves cache locality by maintaining allocation order.

4.6 Ordered Per-Size Allocation

To minimize misses, Vam preferentially allocates objects from recently-accessed 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 block, Vam moves the block from the full list to the front of the available list. This page-level ordering ensures that new objects always fill free space on the page in the front of the available list and increases the chance that pages near the end become entirely free.

PHKmalloc uses a similar approach, but sorts non-full pages in increasing address order. We do not use address order because the sorting operation is costly.

4.7 Aggressive Discarding of Pages

Blocks of small-to-medium objects and regions of large objects are all multiples of pages. In Vam, a *page manager* manages these pages, by recording status information for each page in a page descriptor table and keeping consecutive free pages in a set of free lists.

Vam uses the `madvise` call to discard blocks of small and medium objects whenever they become empty. For large objects, Vam discards empty pages inside the large object memory region. As described above, Vam releases all extremely large objects upon free by a call to `munmap`.

This strategy reduces application footprint and can greatly reduce paging when under memory pressure. However, aggressive discarding of pages does add some runtime overhead. Each page discard requires one 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). In fact, these overheads are very low in practice, thanks to the efficient implementation of system calls and soft page handling in the Linux kernel. We would prefer to discard pages only in response to memory

	176.gcc	197.parser	253.perlbmk	255.vortex
Execution Time	24s	280s	42s	50s
Instructions	40G	424G	114G	101G
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 CPU2000 benchmarks, run with DLmalloc.

scarcity, but this feature is not supported by the current kernel.

5. Experimental Evaluation

To evaluate the efficacy of Vam's design, we sought to answer the following questions:

- Does Vam reduce total application execution time?
- Does Vam increase cache-level locality?
- What is the effect of Vam's policies on fragmentation?
- Under memory pressure, does Vam reduce paging?

To answer these questions, we use the four memory-intensive applications from the SPEC CPU2000 benchmark suite [2]: 176.gcc, 197.parser, 253.perlbmk, and 255.vortex. The other benchmarks in the suite either use very little memory or only allocate memory at the start of execution [3]. 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 benchmark CPU and memory allocation statistics.

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 [9]. The use of custom allocation means that the original applications make only occasional calls to `malloc`. We create versions of these applications that use general-purpose memory allocators. We can replace 197.parser's custom allocator directly because xalloc has the same interface and semantics as `malloc`and `free`. This replacement decreases the maximum virtual memory requirements of 197.parser from 30MB to 15MB. The obstack allocator 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 [9]. 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 start using `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 [20] to set the performance counters according to the manufacturer's manual [1].

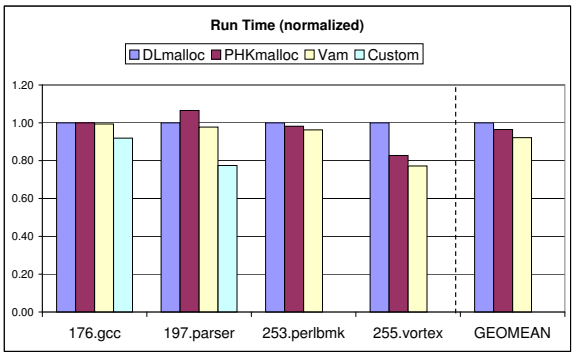


Figure 2: Total execution time, normalized to DLmalloc.

We run each experiment five times and report the median. To minimize variance, we perform all experiments with the machine in single-user mode.

5.1 Total Execution Time

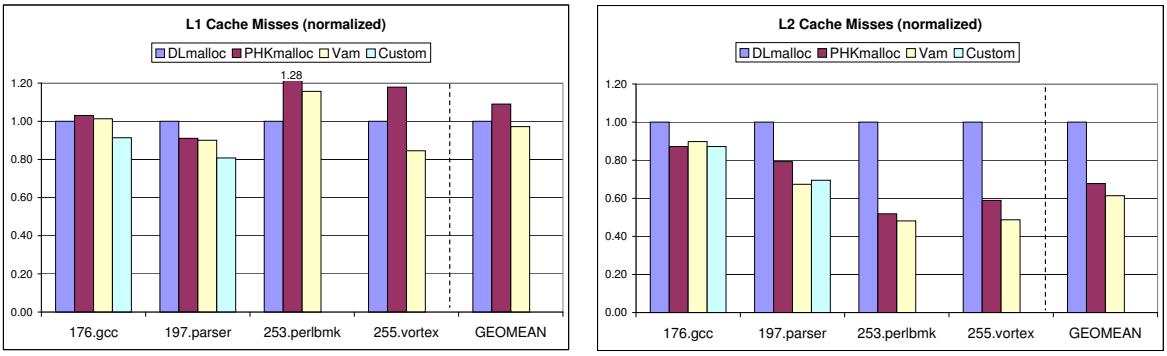
Figure 2 presents total execution time results. Vam consistently improves application performance over both PHKmalloc and DLmalloc. Vam's improvement over PHKmalloc ranges from 1–8%, and improves over DLmalloc by 1–23%. On average, Vam is 4% and 8% faster than PHKmalloc and DLmalloc, respectively. The 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. These allocators improve performance because both applications 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. [9] (showing a smaller gap between DLmalloc and xalloc) to our use of shared objects for the allocators, which precludes link-time optimizations.

5.2 Cache Locality

L1 Locality

We measure both L1 and L2 cache locality for the different allocators. Figure 3(a) shows L1 cache misses using different allocators normalized to DLmalloc. Vam reduces L1 cache misses for two of the four benchmarks. We attribute this result to Vam's reduction of internal fragmentation and elimination of object headers. 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. However, this result is somewhat misleading: 253.perlbmk's L1 cache miss rate is very low for all allocators, and so has very little impact on total execution time.

PHKmalloc increases L1 cache misses in three of the four benchmarks. We attribute this to the internal fragmentation from PHKmalloc's coarse size classes. The only benchmark for which PHKmalloc reduces L1 cache misses is 197.parser, which primarily allocates small objects, and the dominant object sizes are 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.



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

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

Figure 3: Cache-level locality results.

Variant	Description
PHK_sc	size classes every 8 bytes instead of 2^x
PHK_reap	replaces bitmap operations with reap allocation [9]
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 5.3).

L2 Locality

Both Vam and PHKmalloc significantly reduce L2 cache misses over DLmalloc, 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 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, and 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.

5.3 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 and reap-based allocation. Figures 4 and 5 present the L2 cache misses, instruction counts and run time performance of these PHKmalloc and Vam variants. Note that the results are normalized to their respective original versions, i.e., PHKmalloc variants are normalized to PHKmalloc and Vam variants are normalized to Vam.

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 much more different sizes than the other benchmarks. The coarser size classes in the original PHKmalloc allows quicker reuse of freed space within each size class, yielding better cache locality. Although this PHKmalloc variant'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 L2 cache misses for most of the benchmarks (except 255.vortex). This increase is the greatest for 253.perlbmk. However, because the absolute number of misses is quite 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, this variant improves run time performance by 4%. It notably reduces cache misses in 197.parser and 255.vortex and instructions in 176.gcc and 197.parser.

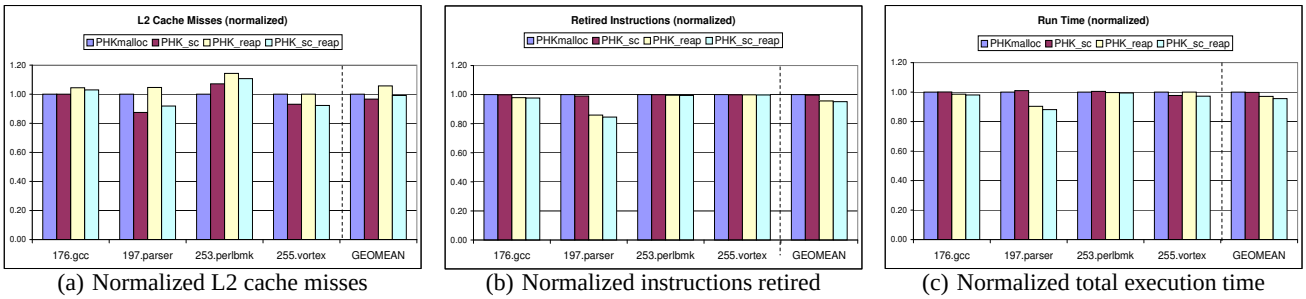
Impact of Headers and Bitmaps: Vam

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 the Vam_bitmap variant significantly increases the number of instructions executed in 197.parser. On average, this Vam 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.

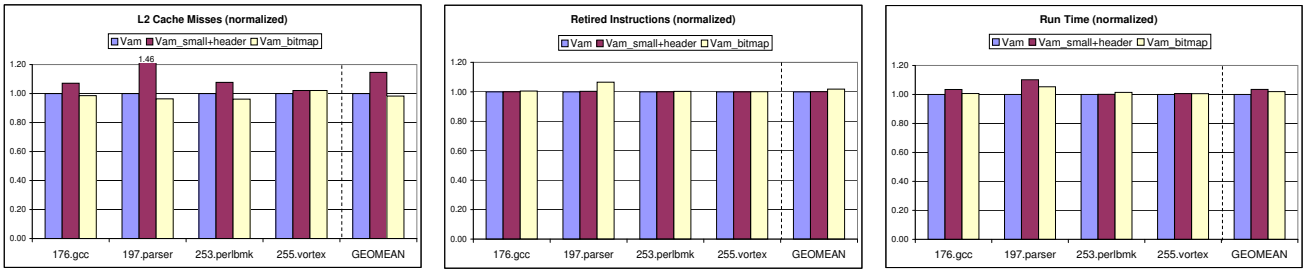
5.4 Fragmentation

We evaluate the effect of allocator design on memory fragmentation. We define fragmentation as the maximum number of pages in use divided by the maximum amount of memory (in pages) requested by the application. 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. This view of appli-



(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 the Vam variants, normalized to the original.

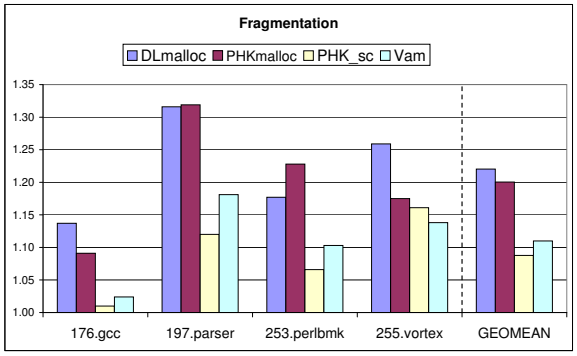


Figure 6: Fragmentation results.

cation memory usage is from the VM manager's perspective and, we believe, better reflects the actual resource consumption.

We compare four allocators here: DLmalloc, PHKmalloc, Vam and the PHK_sc variant of PHKmalloc. Figure 6 shows the results. We were surprised to see that DLmalloc, an allocator known for low fragmentation, in fact leads to the highest fragmentation on average. The first reason for this is the space overhead of per-object headers. More importantly, DLmalloc is unable to distinguish and discard any free pages it may have. 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. Vam combines these fragmentation-reducing features and nearly matches PHK_sc's low fragmentation.

5.5 Performance While Paging

To evaluate the effect of limited physical memory, we launch a process that pins down a specified amount of RAM, leaving the desired amount of available RAM for the benchmark applications.

Figures 7(a) through 7(d) show 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. For all other benchmark applications, 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 the 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 80MB for 253.perlbmk, while PHKmalloc and Vam suffer the same slowdown only after just 30MB RAM remains. 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 avail-

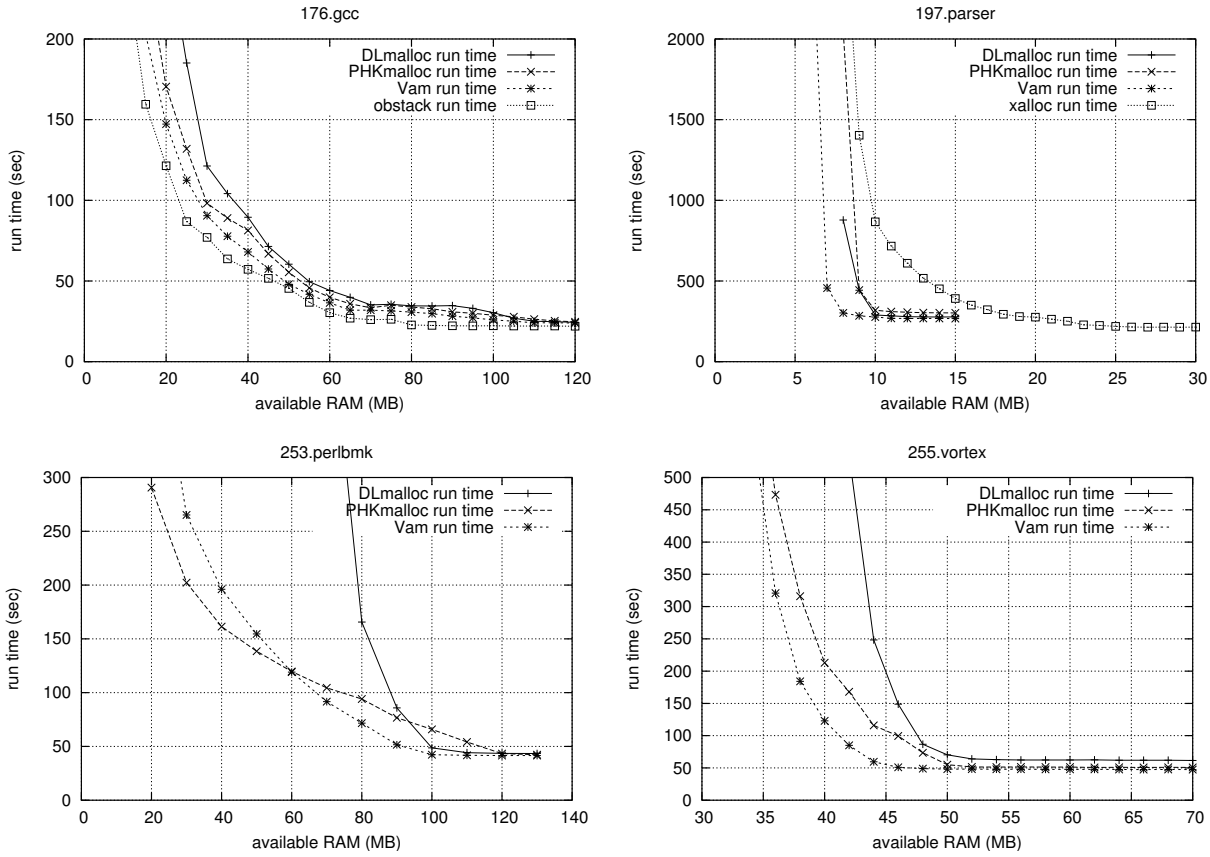


Figure 7: Performance using different memory allocators over a range of available RAM sizes.

able 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.

We note that, for 253.perlbmk, PHKmalloc degrades performance slightly less than Vam when available RAM is less than 60MB. This is, again, because PHKmalloc's coarse size classes result in locality improvement for this particular benchmark in some situations. We also run 253.perlbmk with the PHK_sc variant and the performance degradation curve is then very close to that of Vam across all memory sizes.

5.6 Page-Level Locality

In this section, we explore the effect of allocator choice on application page-level locality in more detail by using an LRU simulator and page-level reference traces. 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 [18].

We then use run these traces through an LRU simulator to generate page miss curves that indicate the number of misses (page faults) that would arise for every possible size of available memory. While no real system implements LRU, many systems closely approximate it, including the Linux kernel we use here. Our LRU simulator is similar to that described by Yang et al. [25]. We use placeholders in the LRU queue for pages discarded by `madvise`, in addition to pages unmapped by `munmap`/`sbrk`. These placeholders allow us to more accurately approximate a real VM system.

We compare the miss curves generated from the simulator with the actual page faults. The actual page faults are the major (hard) page faults measured in the experiments we described in Section 5.5. 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. This inconsistency is due to the swap prefetching used by the Linux VM manager but not in our simulator. In addition to swapping in non-resident pages into RAM whenever they are accessed, the Linux virtual memory manager also speculatively prefetches adjacent pages on the swap disk. To verify this, we turn off prefetching in the kernel, and re-run the paging experiments. The actual number of page faults then closely matches the simulated results for all benchmarks and allocators.

Swap Prefetchability

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 will be effective. Page allocation on the swap disk is managed by the virtual memory manager. . The Linux virtual manager attempts to cluster pages that are adjacent in virtual address space to store them con-

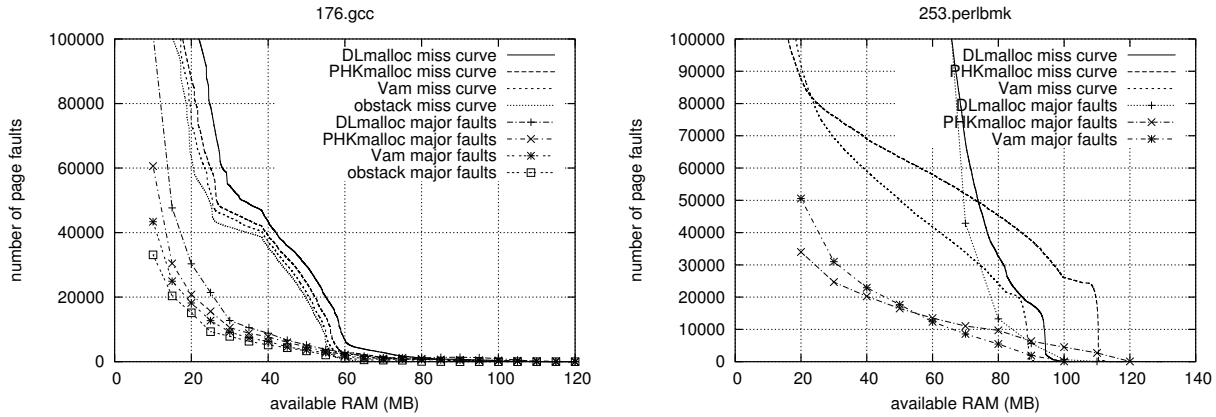


Figure 8: Predicted page miss curves versus actual major (page) faults in a real system with prefetching.

tiguously on disk [10]. For this reason, the application's locality of reference affects the effectiveness of prefetching in the kernel when the system is paging.

We investigate the effect of different allocators on application locality by measuring this *swap prefetchability*. We measure this by quantifying the locality of page misses. We gather those application's page references that would result in a miss for a given memory size in the LRU simulation. We then feed this page miss trace to a page miss adjacency calculator. This calculator measures the minimum distance (in pages) between the current miss and the previous N misses. The N parameter roughly models the memory buffer size for prefetching in the VM manager. We set N to 32, meaning that the last 32 prefetches can be buffered. We denote page misses that have a minimum distance to the previous 32 misses of no more than 8 pages (the Linux default prefetch size) as *prefetchable misses*. The remaining misses are *non-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, as many as 90% of the misses are prefetchable. This prefetchability is due to 176.gcc's strong locality in obstack-style memory allocation. The original version of 197.parser (using xalloc) also exhibits this strong locality. However, this locality is less well preserved in the general-purpose allocators, although among these, Vam leads to the greatest prefetchability. With PHKmalloc and Vam, 253.perlbmk has very few non-prefetchable misses – over 90% of the misses are prefetchable. However, it has a large number of non-prefetchable misses with DLmalloc and only 64% of the misses are 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 50% of the misses are non-prefetchable with PHKmalloc and Vam, and 66% with DLmalloc. In fact, 255.vortex's poor page-level locality is also reflected in the very 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 preserved by any of the allocators.

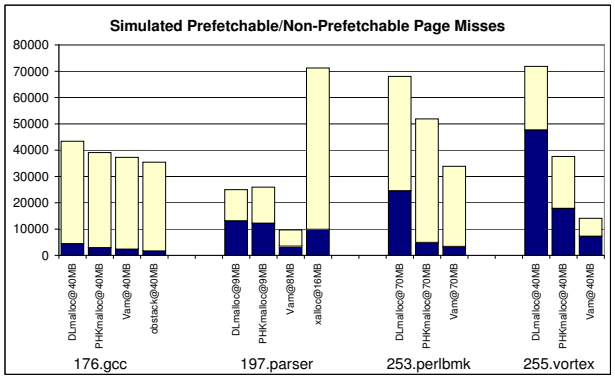


Figure 9: Swap prefetchability: each bar shows simulated prefetchable misses (top) and non-prefetchable misses (bottom).

Note that this prefetchability calculation assumes an ideal prefetching scenario. The real VM manager may not actually be able to prefetch all the prefetchable misses. Nevertheless, they appear to reflect observed application performance on a real system. We attribute the improved prefetchability in PHKmalloc and Vam to their page-oriented design and address-ordered first-fit allocation at the page level.

6. Conclusions

In this paper, we present Vam, a memory allocator that builds on previous allocator designs to improve data locality and provide high performance while reducing fragmentation. We show that, compared to the Linux and FreeBSD allocators and over a suite of memory-intensive benchmarks, Vam improves application performance by an average of 4–8% when memory is plentiful, and by factors ranging from 2X to over 10X when memory is scarce. Vam's performance degrades gracefully as physical memory becomes scarce and paging begins. We explore the impact of Vam's design decisions and find that its fine-grained size classes, reap-like allocation, and page-oriented design all contribute to its effectiveness. We also find that a synergy between Vam's design and the Linux swap space clustering algorithm leads to improved disk prefetching when paging.

7. References

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

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

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

[4] 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.

[5] D. F. Bacon, P. Cheng, and V. Rajan. A real-time garbage collecor with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, New Orleans, LA, Jan. 2003. ACM Press.

[6] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. In PLDI [21], pages 187–196.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] D. Grunwald, B. Zorn, and R. Henderson. Improving the cache locality of memory allocation. In PLDI [21], pages 177–186.

[14] D. R. Hanson. A portable storage management system for the Icon programming language. *j-SPE*, 10(6):489–500, June 1980.

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

[16] 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.

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

[18] 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.

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

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

[21] *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, Albuquerque, NM, June 1993. ACM Press.

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

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

[24] 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.

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

[26] 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.