Better Flash Access via Shape-shifting Virtual Memory Pages

Anirudh Badam, Microsoft Research
Vivek S. Pai, Princeton University
David W. Nellans, Nvidia Research

Abstract

Today, many system designers try to fit the entire data set of an application in RAM to avoid the cost of accessing magnetic disk. However, for many data-centric applications this is not an option due to the capacity and high $/GB constraints of RAM. As a result, system designers are relying on NAND-Flash to augment RAM. However, rewriting applications to efficiently tier data between memory and storage is a complicated process and may take months or years. In this paper, we present Chameleon, a system to transparently augment RAM with NAND-Flash. Chameleon is the first transparent tiering system to provide low-latency accesses to both RAM and NAND-Flash.

We show that applications using Chameleon outperform applications using state-of-the-art tiering mechanisms by providing more than two orders of magnitude improvement in latency for working sets that can fit in RAM. We also show that Chameleon provides up to 47% latency improvement for out-of-core applications. Finally, we show that Chameleon improves the flash device's lifetime by up to 8x.

1 Introduction

Driven by trends in social networking and cloud computing, many applications now require fast access to large amounts of data. One approach is to place all data in fast memory [22, 31], but power density and cost limitations make RAM-based system designs hard to scale as data sets increase in size [28]. Cost-effective servers typically limit RAM to 64GB per server, while high-end systems that can accommodate 1TB of RAM or more require denser RAM that is several times as expensive on a per-GB basis. So, while RAM access latency is in the hundreds of nanoseconds [8] it can require a minimum of 8 rack units (RU) to power and cool a system with 4TB of RAM. In comparison, PCIe-connected NAND-Flash can now scale to 20TB per RU, 6 GBPS sequential throughput, and 1+ million IOPS at a latency less than 100μsec/request [18], while costing 5-10 times less on a per-GB basis.

Flash is therefore a natural candidate to *augment* RAM at the point where cost or density issues make additional RAM unattractive. While flash is denser and cheaper than RAM, achieving good performance requires moving actively-accessed data to RAM, and using flash as a lower tier of the memory hierarchy. Leaving this tiering to the application developer is problematic, because obtaining good performance from flash requires understanding its idiosyncrasies [1]. Our previous approach to this problem was a system called SSDAlloc [5], which used fine-grained, library-level virtual memory management to provide applications with nearly-transparent tiering, at a cost of sacrificing some RAM capacity. In comparison, using flash as swap space did not sacrifice RAM capacity, but provided much worse performance. SSDAlloc formed the basis for commercial solutions [17] which also showed performance benefits against using the same flash as a disk cache [40].

In this paper, we present Chameleon, which transparently manages data spread across RAM and flash, but without sacrificing RAM capacity and without sacrificing flash performance. Chameleon uses *virtual address aliasing* with physical address allocation to track *sparsely-filled virtual pages*, but still *pack physical memory*. This allows multiple memory allocations to share the same physical page, while using different virtual pages. The virtual addresses never change, but the physical memory layout can change over the data's lifetime, providing the memory-packing benefits seen in managed runtimes, without breaking *traditional C pointer semantics*. This ability to pack multiple allocations per physical page also eliminates the RAM capacity overhead of SSDAlloc. More specifically, Chameleon is the first tiering system to provide the following properties:

- *Transparency*: No application modifications are needed to leverage flash via Chameleon's tiering system. Chameleon is a drop-in replacement for `malloc`.
- *Fine granularity*: Chameleon tracks data access/dirty information at a granularity smaller than

TRIOS'13, Nov. 03 2013, Farmington, PA, USA

http://dx.doi.org/10.1145/2524211.2524221

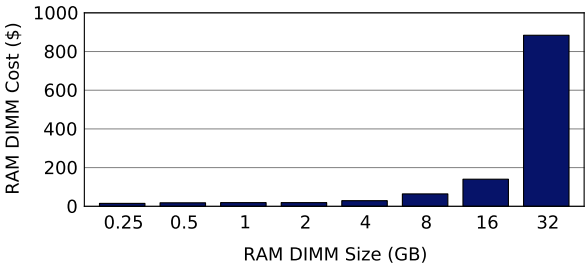


Figure 1: Cost of an ECC, DDR3 RAM DIMM

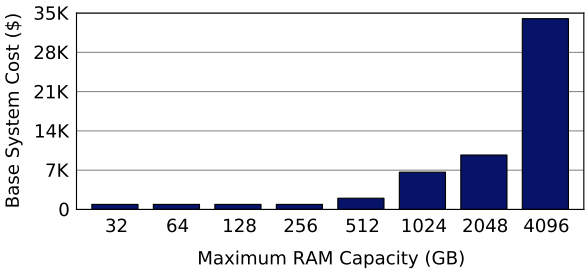


Figure 2: Base System Cost (without RAM) for a given maximum RAM Capacity

a hardware page. This tracking enables the finer access granularity and more efficient flash usage.

- *Low latency*: Chameleon packs as much frequently-used data in RAM as possible, with no latency penalty for RAM accesses. Additionally, Chameleon manages flash in a log-structured manner to reduce flash access overhead and wear.

The rest of this paper is organized as follows: We describe the motivation behind Chameleon in Section 2. Section 3 provides a detailed description of the design and implementation of the system. In Section 4, we present results from our experiments to demonstrate the usefulness of Chameleon. We present related work in Section 5 and we present our conclusions from this work in Section 6.

2 Motivation

Data centers must serve growing amounts of information more frequently and with lower latency, as people move more of their personal information into the cloud [15, 16]. Using magnetic disks for this workload is not attractive due to their high latency and low request rates [31]. Data-intensive computation with large memory footprints, like complex graph traversals, also impose tighter latency constraints on data accesses whether they are distributed [19] or not [27]. The ability to use flash as RAM augmentation will help these applications scale transparently.

One easy way to provide applications with more memory is to increase the RAM in each single system. There are three main reasons why such an approach would be difficult:

1. Squeezing more RAM into a machine requires the usage of high density RAM whose price per GB increases sharply with density, as shown in Figure 1. In comparison, flash prices recently fell below 1$/GB [38], a factor of 30 cheaper than the highest-density RAM.
2. Even the base system prices (without RAM) increase sharply when designed for higher RAM capacities. For reasons of cost and physics, low-end servers typically have only 3-4 DIMM slots, while the most we have seen in high-end servers is 24 DIMM slots per RU. Even using high-density 32GB DIMMs, a server would need 640 DIMMs to match the 20TB/RU density of the high-end flash memory. Today, systems supporting more than 512GB are NUMA machines with expensive interconnects, driving up costs. Figure 2 shows how the cost of a system increases as its maximum RAM capacity increases (not including cost of RAM).
3. Higher RAM usage increases power consumption, heat generation, and cooling requirements, all critical factors in data center design. High-density RAM has higher leakage currents, which generate heat and affect physical design considerations for servers. For example, the lowest specifications we have seen for systems capable of supporting 4TB of RAM requires 12 KW of power and 8RUs. In comparison, one can fit more than 20TB of flash (2x10TB units) in a single RU while requiring only 300 Watts [18] to power it.

Using flash-augmented RAM is also an attractive option for traditional datacenter scale-out techniques, where applications partition their workloads and run across machines to aggregate the RAM in all of the servers. Such systems try to parallelize applications to reduce network usage [3, 39], and we note that flash latencies have dropped to a point where they are now on the same order as end-to-end latency in large datacenters. Such techniques are complementary to the solution that we propose in this paper, and may dramatically help workloads where the CPU might not be the limiting factor [33, 44]. In such environments, adding more servers simply to obtain more RAM is a waste of CPU, energy, and performance, whereas augmenting RAM with flash more directly addresses the I/O problem, at much lower cost.

Flash's access characteristics are unlike RAM or even disk – block reads from flash (both random and sequential) are fast and efficient, while writes are slow and energy consuming. Moreover, writes can only be performed to empty blocks, and erasing blocks is a lengthy operation which can be performed only at a large granularity, typically 128KB or more. Additionally, each such block can be erased only a limited number of times be-

	Flash Aware	Fine-granular Flash Access	Low-latency RAM Access	Efficient RAM Caching	Transparency
Swap-like Systems			✓		✓
FlashVM [37]	✓		✓		✓
Log-structured Swap	✓		✓		✓
KV Stores (SILT [29] etc.,)	✓	✓	✓	✓	
SSDAlloc	✓	✓		✓	
Chameleon	✓	✓	✓	✓	✓

Table 1: Comparison of various RAM/Flash tiering systems

fore it stops storing data reliably [1, 7, 9, 23]. Therefore, flash systems often avoid in-place writes and utilize a log-structured design to wear the device uniformly.

2.1 State of the art

For several reasons, the assumptions built into modern operating systems about using swap space for virtual memory are a poor match for the characteristics of flash. Multiple studies have shown that using flash as swap space does not provide the raw performance of flash to the application [32, 37, 5]. Traditional virtual memory works at a minimum of 4KB granularity in all popular microarchitectures (x86, ARM, etc.), so swap is accessed at similar granularities. For workloads using objects smaller than the page size, pages may therefore contain a mix of objects, which may differ in important properties such as usage frequency or read/write behavior. As a result, when an object is read from swap, the other objects on the page are also loaded, wasting read bandwidth and increasing latency. Likewise, if an object has been modified and is written to swap, the other unmodified objects on the page are also written, wasting write bandwidth, increasing the amount of data written to the flash device, and therefore needlessly decreasing the device's lifetime. While the high access latency of hard disks made such extra traffic a trivial overhead, this extra traffic is wasteful on flash devices, which often can provide the highest performance when accessing 512-byte sectors.

The SSDAlloc [5] system takes a different approach, and manages virtual memory at the object level, rather than pages, but it operates transparently by leveraging virtual memory page-protection mechanisms. At the virtual page level, SSDAlloc aligns all objects to 4KB page boundaries and allocates only one object per virtual page. In this way, when a page is marked dirty due to an application write, the SSDAlloc library knows which object was modified. If each virtual page was backed by a physical page, most of RAM would be wasted when using small objects, so SSDAlloc avoids this problem by using most of RAM as an object cache that can contain multiple objects per physical page. This cache is not directly accessible to the application, but instead, when the application *first* page faults for a missing page, the SSDAlloc library copies the objects from cache or NAND-flash. Pages that have not been used for some time are eventually reclaimed, with their corresponding object placed into the object cache. This approach provides transparent access for C-style pointers, while providing performance much closer to raw flash rather than using flash as swap space.

The main tradeoff in SSDAlloc is the ratio of directly-accessible pages to object cache pages, which affects what fraction of RAM can be directly accessed by applications. Pages that are directly accessible may waste space, but have better performance due to lowering the page fault rate. Objects that have to be copied from the object cache to directly-accessible pages require page faults, copying, and page table manipulations, all of which are maintenance overhead. However, making too many pages directly-accessible reduces the amount of space available for the object cache, which reduces the number of objects kept in RAM and thereby increase the number of accesses to the NAND-flash.

An **ideal system** would, therefore, have the following properties: It must be able to detect data usage information at a finer granularity and cache data at that granularity in RAM, unlike OS swap. It should be able to provide applications with the access to the entire physical memory without page faults, unlike SSDAlloc. Additionally, it should be able to provide these benefits to the application without requiring any code modifications. New applications should be able to use it purely with existing system interfaces.

3 Design

Chameleon provides object-based virtual memory, with multiple virtual page sizes ranging from 512 bytes to 4 KB, and provides transparent access to these pages while still providing efficient usage of physical memory, and without requiring any major page faults to access data in RAM. As a result, it supports applications with full access to RAM, while at the same time ensuring that reads and writes to the flash memory do not waste bandwidth or create excess wear on the device. Chameleon does not require any additional hardware support for implementing and managing these small virtual pages. We believe

that Chameleon is the first transparent memory tiering system to provide the full power of flash and RAM to the application with the properties shown in Table 1.

Chameleon provides small page support for the heap of a process by redesigning the heap manager. It takes this approach because hardware support for efficiently detecting access happens at the page level, which is 4KB or larger on most systems. While every read or write access to a page could be trapped via hardware, using this process to precisely determine which parts of a page have been accessed/modified requires so many page faults as to be impractically slow. Instead, Chameleon redesigns the heap manager to be a better match for the hardware's preferred use of page access detection, once per page.

Chameleon's design stems from three central ideas: discontiguous virtual address allocation, using page-level information to detect object-level accesses, and aliasing multiple virtual pages to the the same physical page. These design principles affect the system in several ways:

- When allocating memory to the application, `malloc` need not use the virtual memory in a contiguous fashion. For example, a heap manager could use only the first 2KB portion of a 4KB page to allocate memory from and not use the remainder portion of the 4KB page for allocation. Likewise, it could also decide to use the *last* 2KB of the page, with a similar effect. For all practical purposes, such a sparsely used virtual memory page can be treated as containing only 2KB of application data.
- Hardware has a separate page table entry per virtual memory page. Therefore, sparsely allocating virtual memory pages provides finer granularity access information (referenced bit, dirty bit etc.), at the cost of increasing the number of VM structures. However, as VM latencies are much lower than flash latencies, this is a desirable tradeoff.
- The same physical page can be used to store the *active regions* of multiple virtual pages through aliasing, as long as the active regions of the virtual pages do not overlap when mapped to a physical page. The memory manager can do this by having the same physical page number in the page table entries of all the virtual pages. For example, if the heap manager uses only the first 2KB portion of virtual page # 1 and only the last 2KB portion of virtual page # 2 to allocate memory from, then pages 1 and 2 can share a physical memory page.
- Over an object's lifetime, the set of physical pages backing its virtual page may change, and the set of virtual objects mapped to the same physical page may change. Since none of these mappings overlap on the physical page,

Page Size	Allowed Shapes
0.5 KB	$S_0^{0.5KB}$, $S_{0.5K}^{0.5KB}$, $S_{1K}^{0.5KB}$, $S_{1.5K}^{0.5KB}$ $S_{2K}^{0.5KB}$, $S_{2.5K}^{0.5KB}$, $S_{3K}^{0.5KB}$, $S_{3.5K}^{0.5KB}$
1.0 KB	S_0^{1KB}, S_{1K}^{1KB}, S_{2K}^{1KB}, S_{3K}^{1KB}
2.0 KB	S_0^{2KB}, S_{2K}^{2KB}
4.0 KB	S_0^{4KB}

Table 2: Allowable shapes for pages in Chameleon

Since existing library functions like *malloc* make no guarantees about where allocations occur, applications should be to use Chameleon's allocation strategy without modification. Buggy applications that access memory outside their allocations are already not guaranteed any specified behavior, but even these can be somewhat accommodated by simply having Chameleon add some configurable padding to each application request.

3.1 Shape-shifting Virtual Memory Pages

Chameleon can choose how much space to allocate within each 4KB page and the offset within an actual 4KB virtual memory page where the allocation resides. We will represent each such page using the notation S_{Offset}^{Size} to represent the "shape" of a page in Chameleon for the rest of the paper. We call such pages shape-shifting virtual memory pages (SVMP). In a regular virtual memory system, all pages would be of the shape S_0^{4KB}. Other than this, regular virtual memory pages and SVMPs are similar in all other respects – each has its own page table entry, an underlying physical page when it is resident, and a TLB mapping when the SVMP is in use by a CPU.

In our implementation, we restrict the sizes of SVMPs to 0.5KB, 1KB, 2KB and 4KB. For each size, we restrict the offsets to be multiples of that size starting from 0. For example, the allowed shapes for a 2KB SVMP are S_0^{2KB}, and S_{2K}^{2KB}. Table 2 presents the allowed shapes for all the SVMPs in our system. This policy simplifies the actual allocation process, which is described in more detail in Section 3.4.

Chameleon is a runtime system that has multiple threads to perform background tasks, as well as a page-fault handler to trap application accesses to SVMPs currently not in RAM. To provide transparent tiering to the application, it needs efficient sub-systems for managing physical memory, and flash memory. It also needs efficient methods to bring pages in from flash to RAM and vice versa:

Physical memory management: Physical memory must be managed such that allocation and deallocation of RAM should be quick. It must be able to allocate physical memory space for incoming SVMPs (from flash) and also reclaim space from outgoing SVMPs

(to flash) preferably in O(1) operations. Additionally, Chameleon should waste as little physical memory as possible even in the presence of sparse virtual memory usage. It must be able to compact as many active SVMPs as possible into physical memory, emulating LRU behavior with as little overhead as possible.

Flash memory management: Chameleon must mask flash's shortcomings from the application. It must transparently manage flash such that all the writes happen sequentially in the background to minimize the application-perceived latency. Chameleon organizes the data on flash in a log-structured fashion to achieve this goal. It must perform garbage collection and compaction in an efficient manner to provide the full benefits of flash to the application. Additionally, the SVMPs must be stored on flash so that each SVMP can be read in precisely one read operation, to maximize the application perceived IOPS.

Page-in process: When the application requests an SVMP that is currently not resident in RAM, Chameleon should be able to satisfy such a request with low latency. It must find a free physical memory location to map the SVMP to, and then it must read the SVMP from flash and populate the SVMP with data before returning the control back to the application. While Chameleon's physical memory manager provides the location of free physical memory or a location that has not been used recently, we still need a mechanism to efficiently find where the SVMP is located on Flash.

Page-out process: When the application requests an SVMP from flash and all of the physical memory is in active use, Chameleon needs to be able to evict an SVMP from RAM. Ideally, such an SVMP should not have been recently used. Therefore, Chameleon needs to keep track of the recency of usage of SVMPs that are in RAM, and to write dirty objects to flash in the background, so that space is available on demand without incurring the write latency.

Virtual memory management: In a traditional virtual memory system, a memory allocator like `malloc` allocates all the objects to the application using virtual memory pages of size 4KB. However, in Chameleon memory allocators have many choices for page sizes and shapes as shown in Table 2. Section 3.4 describes how we modify `malloc` to make the best use of SVMPs.

3.2 RAM Organization

Chameleon manages the lists of active and free SVMPs, using a two-FIFO Clock algorithm based on the one used by the Linux kernel, with the extension that free lists are maintained for sub-page allocation sizes, rather than only using full pages. Physical memory needs to be organized scalably so that unused data can be retired easily to the flash device to make space for SVMPs

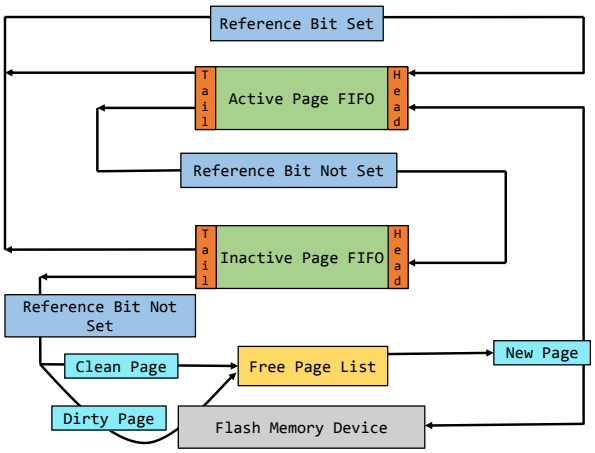


Figure 3: Chameleon uses two FIFOs to track activity of SVMPs instead of pages

that the application requires. Linux maintains two FIFO queues of pages to implement a version of the Clock algorithm [20]. One FIFO (active FIFO) contains pages that have been used in the recent past and another FIFO (inactive FIFO) contains pages that have not been used in the recent past. The OS uses the referenced bit (R-bit) in page table entries to detect if a page was referenced recently.

The main idea in the two-FIFO Clock scheme is to use one FIFO to track active pages, and then move inactive pages to a second FIFO, which is used to select victim pages. Every new page is added to the head of the active FIFO with its R-bit unset as shown in Figure 3. When the active FIFO fills up, the OS checks the R-bit of the page at the tail. If the R-bit is set, then the OS moves the page to the head of the active FIFO. If not, it is moved to the head of inactive FIFO. When the OS needs to evict a page, it inspects the R-bit of the page at the tail of the inactive FIFO. If the R-bit is set then the page is moved back to the head of the active FIFO with its R-bit unset as shown in Figure 3. If it is not set, then this page is a good candidate for replacement. If no such page is detected after many iterations, then the OS picks one at random from the tail for replacement. If the replaced page is dirty then the OS writes it to secondary storage. We refer to this two-FIFO clock scheme as TF-Clock. Additionally, the OS maintains a free list that contains a list of free physical pages that is always checked first before replacing an existing page.

Chameleon emulates the TF-Clock algorithm for each SVMP shape, so that allocations can not only get the appropriate size, but also the appropriate placement within a page, which is important when an previously-allocated object needs physical memory, and its alignment on the page already exists. Therefore, when an application requests a new SVMP or an SVMP that currently resides

on flash, the OS needs to find a physical memory page that can accommodate an SVMP of a given shape at a given location. If the OS cannot find such a free range that matches, then it needs to find a physical memory page where that range has not been recently used. Therefore, the OS needs to track free and TF-Clock information of physical memory at a much finer granularity.

Fine granularity free list: We implement a free list for each of the permitted shapes, keeping track of the allocation status of every physical page with regard to all the known shapes. For example, if a physical page is free in the first 2KB portion, then it is free to accommodate pages of the following shapes: $S_0^{0.5KB}$, $S_{0.5K}^{0.5KB}$, $S_{1K}^{0.5KB}$, $S_{1.5K}^{0.5KB}$, S_0^{1KB}, S_{1K}^{1KB}, and S_0^{2KB}. However, such a free range is added only to the free list corresponding to the shape S_0^{2KB} to reduce metadata overhead.

Contiguous free ranges within a single page are always coalesced when the combined free range can be moved to a free list that contains larger ranges. We store enough metadata for every physical memory page to be able to do the coalescing efficiently. For example, a physical page range of shape S_{1K}^{1KB} will be in the free list corresponding to its shape if and only if the same physical page has some SVMP of shape S_0^{1KB} mapped to it and is currently in core. When that SVMP is freed by the application, the free list manager coalesces these two free ranges and adds the combined free range to the free list corresponding to the shape S_0^{2KB}.

Allocation requests can also split a larger free range, so if the application requests a physical page of the shape $S_{1K}^{0.5KB}$, the free list manager would split this page of shape S_0^{1KB}, and allocate the portion with shape $S_{1K}^{0.5KB}$ to the application and add the portion with shape $S_{1.5K}^{0.5KB}$ to its corresponding free list. The free list manager then appropriately modifies the metadata for the underlying physical page that would later be needed for coalescing.

Fine granularity TF-Clock: If the free list manager cannot satisfy a request, then Chameleon needs to know a physical page that has not been used recently in a required range. For example, if the free lists corresponding to the shapes S_0^{2KB} and S_0^{4KB} are empty, and the application wants to bring in a new page of shape S_0^{2KB}, then Chameleon needs to find a physical page that has not been recently used in the range corresponding to the shape S_0^{2KB}.

Chameleon maintains separate active and inactive FIFOs of physical page ranges for each permitted shape by using the the R-bit in a slightly different manner when managing the active and inactive FIFOs. When a physical page has been detected as referenced in the range corresponding to the shape S_0^{1KB}, it is also treated as referenced in the ranges corresponding to the shapes $S_0^{0.5KB}$, and $S_{0.5K}^{0.5KB}$. This overloading of information ensures that an SVMP with shape $S_0^{0.5KB}$ page would not replace the page with shape S_0^{1KB}. Similarly, when a physical page has been detected as referenced in the range corresponding to the shape $S_0^{0.5KB}$, it is also treated as referenced in the ranges corresponding to the shapes S_0^{1KB}, S_0^{2KB} and S_0^{4KB}.

Such a tracking mechanism allows Chameleon to retire cold data from RAM to flash at a much finer granularity – only the smallest cold SVMP mapped to a physical page is written to flash. Also, it helps pack every physical page with hot data – multiple small hot SVMPs could be mapped to each physical page. Therefore, each physical page in the system is potentially much more useful to the application compared to the traditional virtual memory system where cold and hot data could be co-located on a page.

Our replacement policy considers larger SVMPs as candidates for replacement by a smaller SVMP. Similarly, a larger SVMP can also replace multiple smaller contiguously mapped SVMPs that have not been recently used. Our free list can choose to split free ranges for mapping multiple SVMPs and it always coalesces free ranges within a single physical page.

3.3 Flash Organization

The aim of our flash organization is to minimize latency of reads and writes. We minimize the latency of reads by ensuring that there is enough metadata in RAM to be able to perform all page fetches in a single read operation. We minimize the latency of writes by ensuring that all writes are performed in the background in a sequential manner.

Structuring the flash as a log [36, 1, 4, 6] not only provides wear-leveling, but also maximizes write bandwidth by performing as few erases as possible. Chameleon organizes flash as a log of SVMPs. This design allows Chameleon to be flash aware, and to maximize performance on low-end flash devices that do not internally perform write remapping. On flash devices that remap writes at page level, this approach still provides benefits by packing multiple small SVMP writes into a page. For devices that can perform remapping at a sector level, where Chameleon's log-structuring is redundant, this functionality can be disabled in Chameleon.

Similar to the system page tables, Chameleon uses a hierarchical page table that is indexed by virtual memory address to store the location of SVMPs on flash. We refer to this index as the SVMP table and we store it in RAM. In the future, to reduce RAM overhead, we could overload the system page tables to store the location of SVMPs that reside on flash.

Any live objects not in memory reside only in the log, and any object re-written to the log potentially creates garbage because any older image of that object is

not needed, so Chameleon needs a log cleaner to ensure space in log. Chameleon operates over the log in rounds, where each round involves four phases – read, modify, write, and commit. In the read phase, Chameleon reads into memory a 256KB block containing multiple SVMPs from the head of the log. In the modify phase, it discards any of these SVMPs that are garbage, potentially creating gaps within the blocks' in-memory copy. It then fills these gaps with as many dirty SVMPs as it can. In the write phase, it writes the block to its location on flash. Finally, in the commit phase, it modifies the SVMP table to reflect the new location on flash of the dirty SVMPs, making their old locations garbage. Additionally, Chameleon marks the system page tables corresponding to these SVMPs as non-resident before the modify phase. The next time the application accesses these pages, it incurs a page fault that is served by Chameleon. We further describe this process in Section 3.5.

In the modify phase, Chameleon needs to know if a certain SVMP on flash is garbage. To find this information, Chameleon stores a back-pointer for each SVMP on flash pointing to its SVMP table entry. If the location of the SVMP agrees with the location stored inside its SVMP table entry, then the SVMP is live, or else it is garbage. We store these reverse pointers in the header of each 256KB block on flash. While this information could be stored in RAM, flash is substantially cheaper, so we chose to store these reverse pointers on flash along with the SVMPs. The decision to store these reverse pointers in the header of a block was made so that SVMPs can be properly aligned to 512 byte boundaries on flash. Such an alignment leads to quicker IO operations – each SVMP access reads exactly the size of the SVMP from the flash device.

While the writes happen in the background, reads from flash are triggered on-demand by application page faults. In the page fault handler, Chameleon obtains a free physical page range or victimizes one to accommodate the desired SVMP. It then consults the SVMP table, finds the SVMP on flash, and reads it into the target physical memory location. Finally, it marks the page as resident and returns control to the application.

3.4 Memory Allocator

To achieve full transparency, Chameleon also implements its own version of the `malloc` memory allocator so that even memory-allocation calls do not need to be modified, as they were in SSDAlloc. In traditional `malloc` implementations, the library interacts with the operating system at the level of pages, and any subdivision of pages is tracked only within the library itself.

In Chameleon's memory allocator, the allocation policy is tied to the request size as well as the permitted page shapes. Requests that span multiple pages are allocated via large contiguous regions obtained from the operating system. Requests that are within the range of the SVMP sizes, starting with requests for more than half the smallest SVMP size, are rounded to the higher size, and then allocated as an SVMP. Requests that are smaller than half the smallest SVMP size are first rounded to the next power of two, and then packed into a minimum-sized SVMP.

When picking which offset to use for a given sized SVMP, our `malloc` simply cycles through all possible offsets for that size. For example, for creating 64 byte chunks, our `malloc` would begin by dividing a page of shape $S_0^{0.5KB}$ into 64 byte regions. In the next round of creation of 64 byte chunks, our `malloc` uses a page of shape $S_{0.5K}^{0.5KB}$. It would iterate this process all the way through to the shape $S_{3.5K}^{0.5KB}$ and circle back to the shape $S_0^{0.5KB}$. The rationale behind such cycling is that, at steady state the shapes are uniformly distributed such that they can be fit into physical pages compactly.

Reducing the size of SVMPs helps reduce the read and write sizes for flash device operations, leading to better utilization of the flash device and potentially higher performance. At the same time, reducing the minimum SVMP size has other less desirable ramifications. For example, we do not set the minimum size as low as 64 bytes, because creating a large number of extremely small sparse pages would bloat the virtual address space consumption of the application and increase TLB pressure. Additionally, each sparse page has its own page table entry (8 bytes), an additional SVMP table entry, and RAM overhead from the free list and also from the recency tracker. Limiting the smallest SVMP to be 0.5KB keeps the metadata overhead less than 5% of RAM.

The policies described above are largely design choices, and one can imagine alternatives that optimize for different properties. For example, there are other rounding strategies that waste less space [25] at a cost of higher management overhead. It may be possible to use these selectively for applications where the internal fragmentation caused by powers of two is excessive. Also, some optimization could be performed for larger regions. Requests for memory allocations of size more than 4KB are allocated using contiguous pages of shape S_0^{4KB}. In our current implementation, Chameleon creates SVMPs using pages from anonymously mapped memory regions. This choice creates more VM objects inside the OS, and if their management becomes an issue, it may be possible to have Chameleon request a smaller number of larger ranges and handle subdivision itself.

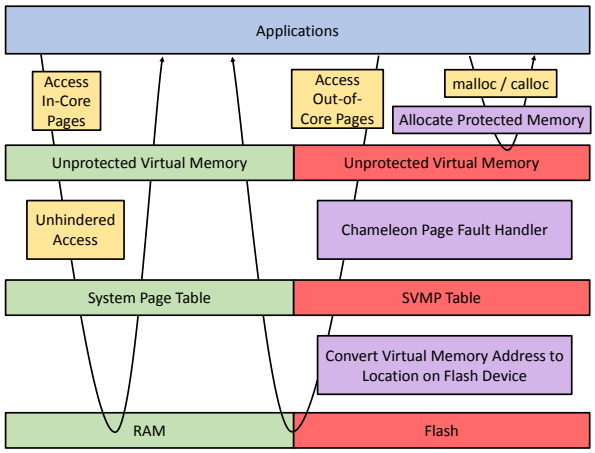


Figure 4: Overview of how Chameleon services a page fault triggered when application accesses a page on flash. Pages in RAM are freely available.

3.5 Implementation Details

We implement most of Chameleon in roughly 8000 lines of code in the C programming language as a library that can be plugged into any application as a runtime system. The library exports interfaces for `malloc`, `calloc`, `realloc` and `free`, taking over the corresponding functionality from the standard library. Chameleon tracks page accesses by allocating read/write protected virtual memory pages to the application. When the application accesses any protected virtual memory regions, it takes a page fault. Chameleon receives these page faults from the OS and services them appropriately. Figure 4 presents a high level overview of how Chameleon's page fault handler works.

Chameleon also includes a kernel component necessary to interface to the virtual memory system in ways not provided to user applications. Chameleon requires the ability to map multiple virtual memory pages to the same physical memory page. It uses a loadable kernel module that can modify page tables to provide this functionality. It uses the same kernel component to obtain read/write access to the dirty bit and recency bit in the page table entries. The dirty bit is used to detect if a page is dirty and the recency bit is used to implement the two FIFO SVMP replacement algorithm of Chameleon. Additionally, this kernel component allows Chameleon to map a physical page to a virtual memory page that is protected at the user level. This enables Chameleon to provide thread-safe operations for concurrent applications. Without such support, Chameleon would have to unprotect the page to service the page fault at the user level, read data from flash and write to its virtual memory page location. While the page is being populated in the unprotected state, another thread could potentially freely read garbage data.

In our current implementation, we reserve a large pool of physical memory pages (spanning most of the available RAM) in the system. For security purposes, these are the only pages that are used to implement Chameleon's RAM organization – additionally, physical pages are shared (via multiple SVMPs mapped to it) only within the process. Chameleon manages the flash device as a raw disk operating directly without any caching or write-back support enabled. This provides low-latency reads and writes to the device and avoids other unnecessary filesystem overheads.

3.6 Chameleon's Overhead

Chameleon has memory usage overhead from the way it organizes RAM and flash, but many of these are implementation artifacts and as such can be reduced using optimizations that we propose in this section. We quantify computational overheads from using more virtual memory using experiments as described in the next section. In this section, we focus on the storage overhead.

The SVMP table tracks the location of pages on flash. It is implemented as a three level page table, with an overhead of roughly 8 bytes in RAM per page. This overhead can be minimized simply by overloading the system page tables to store the location of pages on flash. We chose a separate SVMP table in our current implementation because we needed additional bits to represent the size (two bits for representing four sizes) and shape of an SVMP (three bits for representing eight shapes). If the flash location information is moved to system page tables, the size of SVMP page tables, now necessary only for maintaining size and shape information, would be reduced to 1 byte per SVMP. Since we primarily focus on large-memory workloads, we currently keep the SVMP tables in RAM to avoid having to read them from flash.

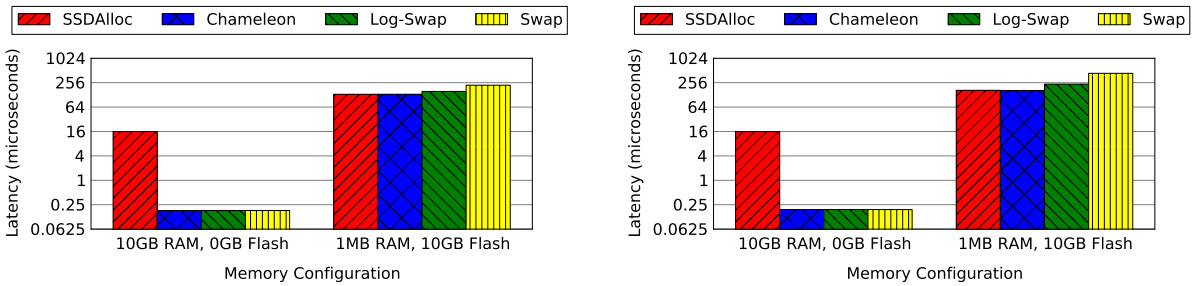
On-flash metadata that stores the back-pointers for SVMPs has an overhead of roughly 10 bytes per SVMP on flash. Chameleon stores additional information for each SVMP that is in core. It needs to store the information necessary for implementing the TF-Clock algorithm and the free list. This overhead is 128 bytes per physical page.

4 Evaluation

We compare Chameleon with three other state of the art tiering systems, across various workloads including microbenchmarks, sorting, graph search and bloomfilters.

4.1 Experimental Setup

In all our experiments, we use a server with a quadcore Intel Xeon 3GHz CPU with 16GB RAM. We use two SSDs for evaluation – the OCZ-VERTEX4, and the Intel 520. In each experiment, we compare Chameleon with



(a) 90% Reads & 10% Writes

(b) 10% Reads & 90% Writes

Figure 5: Microbenchmarks with random reads/writes on 0.5KB objects. Chameleon matches the lowest latency for random accesses under all settings when compared to SSDAlloc and Log-Swap.

these three tiering systems:

- **Swap**: We use the in-built swap mechanism in Linux as the base system against which we compare the rest of the systems. We will refer to this system simply as the "Swap" in the rest of this section. Swap was originally designed as the worst-case protection mechanism for applications that may run out of memory. In addition, it was originally designed for the magnetic disk and therefore, was not optimized to exploit the low-latency and high-concurrency characteristics of modern SSDs. To provide a fair comparison, we use our memory allocator (Section 3.4), as opposed to using the default `malloc`.
- **Log-structured Swap**: We built a log-structured swap system. We simply configured our `malloc` to use pages of only 4KB in size – pages of shape S_0^{4KB}. This will let the application use only 4KB pages in Chameleon. The performance of this system will give us a close approximation of the characteristics of a page based system. We will refer to this system simply as the "Log-Swap" in the rest of this section. We also compared against OS-swap. As reported in previous work [5, 32, 37], we find that OS-swap performed consistently worse compared to log-structured swap because of synchronized random writes. Therefore, we report comparisons only against log-structured swap for the rest of the paper.
- **SSDAlloc**: We compare Chameleon against our previous work on SSDAlloc [5]. While SSDAlloc is extremely useful network bottlenecked, memory-intensive applications like HTTP caches, query caches like Memcache, it was not designed to explicitly support computationally intensive tasks like graph traversals.

The purpose of our evaluation is to demonstrate the capabilities of Chameleon when compared to the other systems under various circumstances. We use the Microbenchmarks in Section 4.2 to show the limiting performance of these systems. We use the sorting benchmarks in Section 4.3 to demonstrate the performance of Chameleon when application's hot objects are spread across the flash device, but can be compactly cached in RAM. We use the graph search benchmarks in Section 4.4 to evaluate the computational overhead from excessive usage of virtual memory in Chameleon. Finally, we use the bloomfilter benchmarks in Section 4.5 to demonstrate the benefits for Chameleon for write-heavy workloads.

4.2 Microbenchmarks

We present some experimental results using microbenchmarks to demonstrate the limits of each system. We allocate 10GB worth of 0.5KB objects and perform random read/write operations on them. Specifically, we test read:write ratios of 10:90 and 90:10. Additionally, for each setting, we present RAM-driven and flash-driven performance. In the RAM-driven setting, we ensure that all the objects fit in RAM. In the flash-driven setting we ensure that only a handful of the objects can reside in RAM (10MB worth) at any point of time.

We find that Chameleon, SSDAlloc and Log-Swap have 15.8 μsec of software overhead to service page faults on average – this includes `memcpy`'s required to bring in the new page and move the old page to a buffer, the overhead from managing the free list and TF-Clock, and the system calls needed to read/modify page table entries. This does not include the read latency of flash (typically 100–150μsec), however. The results with the combined latency are presented in Figure 5.

Figure 5(a) presents the results for the case when 90% of the random operations on these objects are reads. When all the objects can fit in memory, Chameleon matches the performance of Log-Swap and Swap. Essentially, when enough memory is available, Chameleon, even with higher virtual memory usage, can match the random access performance of Log-Swap, and Swap – because random accesses do not provide any lo-

cality in the TLB. SSDAlloc, on the other hand, needs page faults to access pages that are cached in RAM non-transparently. When the amount of RAM is meager compared to the data set size, Chameleon matches the out-of-core performance of SSDAlloc. Unlike Log-Swap and Swap, Chameleon and SSDAlloc access only 0.5KB per operation. Log-Swap, and Swap access a full 4KB page every time and incur 15.7%, and 67.4% higher latency respectively.

Figure 5(b) shows the comparisons when 90% of the operations are writes. In this case, Chameleon and SSDAlloc perform 43.23% better than Log-Swap, and 2.65X better than Swap. Even though the writes happen in the background, they influence the latency of reads necessary for the page-in process by keeping the device occupied. Chameleon and SSDAlloc write substantially less data to flash compared to Log-Swap, and Swap because of fine-grained access. Chameleon, like SSDAlloc, operates over flash at a granularity of 0.5KB while Log-Swap, and Swap operate at 4KB. This decreases the total data written to flash by 8x. Such savings are especially useful considering the fact that flash's blocks can be written only a limited number of times. The poorer performance of Swap when compared to Log-Swap is primarily because of the random write behavior of Swap [5, 37].

4.3 Sorting Workload

We use a sorting workload to demonstrate the benefit of Chameleon's free list and TF-Clock implementation. SVMPs allow the application to transparently cache data in RAM at a finer granularity than 4KB. When an application's hot objects (smaller than 4KB) are spread across the flash device, the smaller pages provided by Chameleon make more effective use of RAM. Each physical page contains more hot objects in Chameleon when compared to Log-Swap because multiple 0.5KB SVMPs can be packed into the same physical page.

To demonstrate this benefit, we perform the following experiment. We create 100GB worth of 0.5KB records. Chameleon uses all the allowed shapes for 0.5KB SVMPs shown in Table 2 uniformly to create them. SSDAlloc uses only the shape $S_0^{0.5KB}$ to create them to emulate one object per page. Likewise, Log-Swap uses only the shape S_0^{4KB} for allocating them – however, each such page is used to create eight 0.5KB records. We select a small number of these records at random and sort them in place, using quicksort, according to randomly-generated 8 byte keys. We vary the number of records sorted and we also vary the amount of RAM in the system.

We begin with sorting 10GB worth of randomly-selected 0.5KB records in two configurations. In one configuration, the system has 10GB of RAM. We use this configuration to demonstrate the benefits in Chameleon of being able to map more than one 0.5KB SVMPs on to the same 4KB page so that each physical page in RAM contains eight 0.5KB records that are hot. Log-Swap, and Swap on the other hand can cache only at 4KB granularity and have to cache many cold 0.5KB records in RAM along with the hot ones. Pages of Log-Swap, and Swap are static and therefore cache a lot of records in RAM that are not even part of the data being sorted – roughly speaking, only $\frac{1}{8}^{th}$ of each page is of interest to the sorting workload. The rest of the page simply wastes RAM space by storing only cold data that is colocated with hot data. This effectively, brings down the RAM utilization of Log-Swap down to a mere $\frac{1}{8}^{th}$ of ideal.

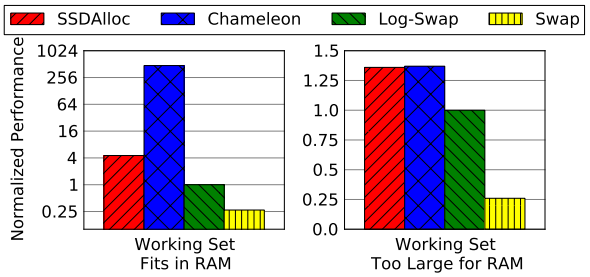


Figure 6: Sorting workload finds in-memory and out-of-core sorting performance of the three systems.

Figure 6 shows how Chameleon outperforms Log-Swap by 475x for this reason. While SSDAlloc can cache all the hot records in RAM in a compact fashion, it cannot provide page-fault free access to all of them. More specifically, for 0.5KB records, only $\frac{1}{8}^{th}$ of the entire RAM is available to the application without page faults – each physical page has one 0.5KB record mapped to it and the rest of the 3.5KB of the page is used for caching records non-transparently. A page fault is needed to access them and page faults have a high overhead as shown in Section 4.2. Figure 6 shows how Chameleon outperforms SSDAlloc by a factor of 105x for this reason. Log-Swap performs 3.81 times better than Swap because it reduces the number of random writes by using the SSD sequentially.

In another configuration, the system has 10MB of RAM. We use this configuration to demonstrate the benefit of fine-granularity dirty information. When quicksort pivots, Chameleon only needs to swap 0.5KB records on flash, while Log-Swap on the other hand would have to swap entire 4KB pages containing the 0.5KB records being pivoted. Figure 6 presents the results. SSDAlloc and Chameleon perform 36.3% and 37.8% better when compared to Log-Swap because of their fine granularity flash access. We believe that the slight advantage of Chameleon over SSDAlloc is be-

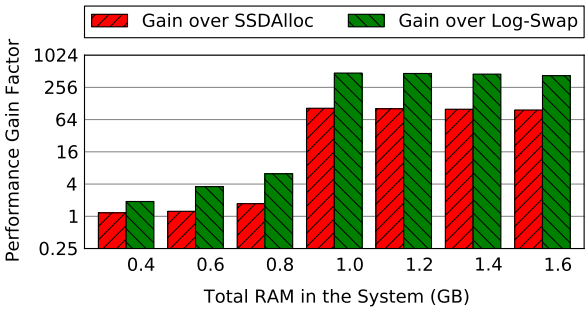


Figure 7: Performance gain of Chameleon over SSDAlloc and Log-Swap when the amount of RAM in the system is varied. Chameleon's efficient RAM usage enables it to obtain these benefits.

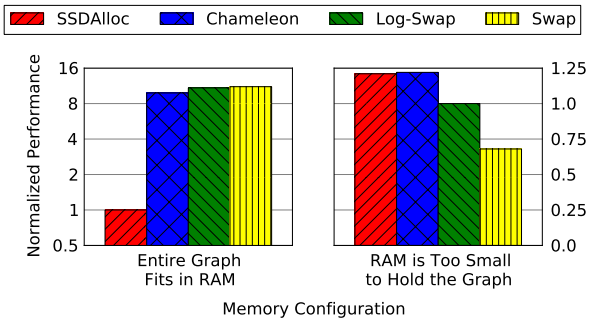


Figure 8: Searching workload – testing in-memory and out-of-core graph traversal performance of the three systems.

cause of its more efficient usage of the meager, but useful, RAM in the system. More importantly, Chameleon writes 8x less data when compared to Log-Swap for the same in-place sorting workload improving average flash access latency. This is primarily because, like SSDAlloc, Chameleon can operate on flash at a 0.5KB granularity for managing the 0.5KB records.

Figure 7 shows the performance gains that Chameleon obtains for different amounts of RAM in the system when sorting 1GB worth of randomly selected 0.5KB records. We vary the RAM in the system from 0.4GB to 1.6GB in steps of 0.2GB. When the RAM in the system is larger than the working set ($\geq$ 1GB), Chameleon is the only system which is able to utilize most of the RAM to not only cache the entire working set but also provide page-fault free access to it. Chameleon is able to avoid using flash for this workload and provides the impressive gains over Log-Swap. Chameleon is also able to minimize page faults unlike SSDAlloc and provides gains over SSDAlloc. When the working set is larger than the RAM in the system ($<$ 1GB), Chameleon outperforms Log-Swap and SSDAlloc by upto 632% and 72%, respectively. However, the margins are lower because the benefits are masked by flash access latency that is more than 150μsec as shown in Section 4.2.

4.4 Graph-Search Workload

We use a graph traversal benchmark to demonstrate the performance implications of increased virtual memory usage because of the SVMP model. We also use this workload to demonstrate benefits of fine-grained access for read-heavy applications similar to the Graph500 [21] workload. We generate a directed acyclic graph on which we perform depth first search (DFS). We use the Graph500 reference code to perform these experiments. We modify the reference code to generate graphs in memory using the four systems that we wish to compare to manage the tiering.

We generate a graph using Graph500's graph generator with 2^{28} vertices and 2^{32} edges. The size of the graph generated is 127GB when generated using our memory allocator (Section 3.4). We also generate another graph with 2^{24} vertices and 2^{28} edges. The size of the smaller graph generated is 7.9GB.

We modify the reference code to use SSDAlloc, Chameleon, Swap, and Log-Swap for managing the tiering of data between DRAM and the SSD. Similar to the sorting scenario, Chameleon uses 0.5KB SVMPs to allocate the vertices; SSDAlloc uses 0.5KB pages for allocations; while Chameleon uses pages with the shape S_0^{4KB} to allocate them. On these graphs, we then perform DFS on 30 different 64 byte keys (search for all instances of the key) using the Graph500 reference benchmark.

There are two settings in which we perform the experiment. In the first setting, we benchmark the performance of each system when searching the smaller graph that fits entirely in DRAM. Figure 8 shows that Chameleon outperforms SSDAlloc by a factor of 9.85x. This is primarily because Chameleon provides unhindered access to RAM to the application while SSDAlloc provides only a portion of RAM in such a manner. Similarly, Log-Swap outperforms SSDAlloc by a factor of 10.88x. Log-Swap obtains higher performance benefits because it uses significantly fewer virtual memory pages (8x less) to perform the graph search. Therefore, it incurs fewer TLB misses. Swap performs 9.8% better compared to Log-Swap because it does not have the additional overhead from the internal data structures that Log-Swap (Chameleon based system) has to maintain internally.

In the second setting, we configure the system to have 1MB of RAM and perform the searches on the larger graph to demonstrate out-of-SSD performance of each system. Figure 8 shows that Chameleon and SSDAlloc perform 21.2% and 22.4% better than Log-Swap. They obtain these performance benefits by reading the device at a granularity of 0.5KB. The performance improve-

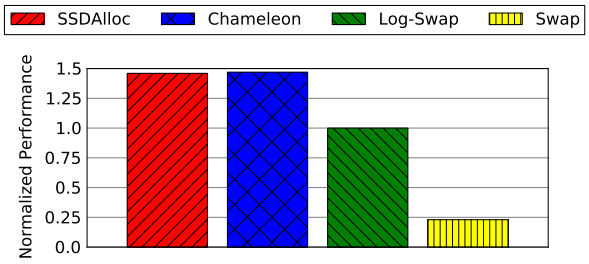


Figure 9: Out-of-core performance for bloomfilter insertions. Chameleon and SSDAlloc outperform Log-Swap by a margin of 46.7%

ments are slightly lower compared to the flash-driven sorting due to the fact that this is a read-only workload.

4.5 Bloomfilter Workload

We use a bloomfilter workload to demonstrate the performance benefits of SVMPs for write-heavy workloads. In this experiment, we build a bloomfilter using the three systems. In each system, we implement the bloomfilter over an array of pages. For Chameleon and SSDAlloc, we pick a page of size 512 bytes. For this experiment, we present only the flash-driven setting by restricting the RAM size to 10MB. The total size of the bloomfilter we use is 80 billion bits(10GB). We perform several insertions of 20 byte keys in the bloomfilter using eight different hash functions that would touch eight different locations in the bloomfilter. We do not perform any look-ups so as to study a write-heavy workload.

Figure 9 presents the results. Chameleon outperforms Log-Swap by 46.7% for write intensive workloads – the background writes increase the latency of the page-in process. Chameleon matches the performance of SSDAlloc for such workloads. Additionally, Chameleon writes 8x less data to flash when compared to Log-Swap. Due to excessive amount of random writes, Swap performance 4.12X poorer compared to Log-Swap.

5 Related Work

FlashVM [37] optimizes the Linux swap subsystem [20] to make flash aware decisions like batching writes. It prioritizes dirty data over clean data when evicting pages to improve batching efficiency. However, similar to other flash based swap systems [26, 30, 32], FlashVM works at a 4KB page granularity and cannot provide the full benefits of RAM and flash to the application.

SSDAlloc [5] addresses this problem by aligning application objects to page boundaries and using the virtual memory page level access information to deign object level access information. However, SSDAlloc does not make an effective use of RAM in the system because most of the RAM in SSDAlloc is trapped behind a page fault.

Other systems that use flash as part of the program's memory usually build non-transparent object stores [2, 4, 12, 13, 29]. These systems can fully exploit the benefits of flash and RAM. However, using these object stores within applications will need intrusive modifications to the code where objects are allocated and used. We believe that packaging and tiering flash under RAM transparently via virtual memory will lead to a wider and quicker adoption of flash as program memory.

The aim of Chameleon is to expose flash as a slower, but denser form of memory rather than providing support for durability of data by leveraging the non-volatility of flash. Many existing systems provide these properties for various non-volatile memory technologies [10, 11, 34, 32, 42], and can be implemented inside Chameleon. However, obtaining transactional support may require source code modifications.

Many modern scale-out systems are proposing the use of RAM only to build systems [22, 31, 33, 44] to improve latency. Chameleon can help these systems scale the limitations of RAM and help them quickly adopt flash without any application modifications.

Multiview [24] uses virtual address aliasing with physical allocation for fine-granular distributed shared memory. Chameleon uses SVMPs for paging to flash at a fine granularity. Additionally, Chameleon implements an efficient replacement mechanism for these SVMPs and transparently manages RAM and flash for the application so that the RAM is fully packed with hot data. Additionally, our `malloc` is co-designed with the paging system.

Mondrian memory protection (MMP) [43] is a fine-grained hardware-level protection for arbitrarily sized virtual memory regions. Such fine-grained protection can allow fine-grained flash access by producing page faults at a fine-granularity. However, MMP requires hardware modifications. The aim of Chameleon is to provide the benefits of flash and RAM without any application or hardware modifications.

Hardware solutions have been proposed to use RAM at a finer granularity than a page to obtain benefits similar to Chameleon's SVMPs [14, 28, 35, 41]. These systems architect RAM the way CPU caches are architected. The aim of Chameleon is to provide most of the benefits of such systems without any hardware modifications.

6 Conclusion and Future Work

Flash has caused a disruption in the storage and memory hierarchy by providing a distinct third tier in what has long been, conceptually, a two-tier system. Compared to RAM, flash is higher density, lower cost, and more power efficient, but these advantages comes with a hundred-fold latency penalty. To leverage the advan-

tages of flash at the application, most systems have chosen to simply use flash as a disk replacement. While applications can access their stored data more quickly without need to modify the application, unfortunately, for a growing number data-intensive applications flash memory cannot compete with the low price per GB of magnetic media. In this work, we take an alternative approach and tier flash-memory *behind* RAM rather than *in front* or *in place of* of magnetic disk.

Chameleon uses flash to expand the application memory space into the multi-terabyte range and then, transparently migrates application memory between DRAM and flash. Chameleon uses sparsely-allocated virtual memory pages to gain fine-grained data usage information to operate efficiently on flash. It has the ability to hold important application data compactly in RAM. We show that applications using Chameleon outperform applications using state-of-the-art tiering mechanism by providing more than two orders of magnitude improvement in latency for working sets that can fit in RAM. We also show that Chameleon provides up to 47% latency improvement for out-of-core applications. Additionally, we show that Chameleon can decrease the data written to flash by 8x for the same write workload.

In the future, we wish to use the system page tables to store the location of SVMPs on flash to save RAM space required for the SVMP table. We also wish to further investigate into the problem of increased TLB pressure. Further, we wish to investigate better strategies to maintain the free list and also to find better SVMP packing algorithms to further improve RAM locality.

7 Acknowledgments

We would like to thank our shepherd, Brad Chen, as well as the anonymous ACM SOSP'13, and ACM TRIOS'13 reviewers. This research was funded in part by the NSF Award CNS-0916204, and was performed while the lead author was finishing his PhD at Princeton University.

References

[1] N. Agarwal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proc. USENIX ATC*, Boston, MA, June 2008.

[2] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and Large CAMs for High Performance Data-Intensive Networked Systems. In *Proc. 7th USENIX NSDI*, San Jose, CA, Apr. 2010.

[3] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *Proc. 9th USENIX NSDI*, San Jose, CA, Apr. 2012.

[4] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proc. 22nd ACM SOSP*, Big Sky, MT, Oct. 2009.

[5] A. Badam and V. S. Pai. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. In *Proc. 8th USENIX NSDI*, Boston, MA, Mar. 2011.

[6] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A Design for High-Performance Flash Disks. *SIGOPS OSR*, 41(2):88–93, Apr. 2007.

[7] S. Boboila and P. Desnoyers. Write Endurance in Flash Drives: Measurements and Analysis. In *Proc. 8th USENIX FAST*, San Jose, CA, Feb. 2010.

[8] J. A. Brown and D. M. Tullsen. The Shared-Thread Multiprocessor. In *Proc. 22nd ACM ICS*, Island of Kos, Greece, June 2008.

[9] F. Chen, D. A. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proc. ACM SIGMETRICS*, Seattle, WA, June 2009.

[10] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making Persistent Objects Fast and Safe With Next-Generation, Non-Volatile Memories. In *Proc. ACM ASPLOS*, Newport Beach, CA, Mar. 2011.

[11] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, D. Burger, B. Lee, and D. Coetzee. Better I/O Through Byte-Addressable, Persistent Memory. In *Proc. 22nd ACM SOSP*, Big Sky, MT, Oct. 2009.

[12] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash. In *Proc 30th ACM SIGMOD*, Athens, Greece, June 2011.

[13] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson. Turbocharging DBMS Buffer Pool Using SSDs. In *Proc 30th ACM SIGMOD*, Athens, Greece, June 2011.

[14] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi. Simple but Effective Heterogenous Main Memory with On-Chip Memory Controller Support. In *Proc. SC*, New Orleands, LA, Nov. 2010.

[15] Dropbox TPS. `http://www.dropbox.com/press`.

[16] Facebook Message Workload. `https://www.facebook.com/note.php?note_id=454991608919`.

[17] Fusion-io: Extended Memory API. `http://www.fusionio.com/products/iomemorysdk/`.

[18] Fusion-io: ioDrive Octal. http://www.fusionio.com/platforms/iodrive-octal/.

[19] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proc. 10th USENIX OSDI*, Hollywood, CA, Oct. 2012.

[20] M. Gormen. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004.

[21] Graph500: Graph Search Benchmarks. http://www.graph500.org/index.html.

[22] M. Grund, J. Krueger, H. Plattner, A. Zeier, and P. C.-M. S. Madden. HYRISE–A Main Memory Hybrid Storage Engine. In *Proc. 36th VLDB*, Singapore, Singapore, Sept. 2010.

[23] L. M. Grupp, J. D. Davis, and S. Swanson. The Bleak Future of NAND Flash Memory. In *Proc. 10th USENIX FAST*, San Jose, CA, Feb. 2012.

[24] A. Itzkovitz and A. Schuster. Multiview and Millipage – Fine-Grain Sharing in Page-Based DSMs. In *Proc. 3rd USNIX OSDI*, New Orleans, LA, Feb. 1999.

[25] Jemalloc Memory Allocator. http://http://www.canonware.com/jemalloc/.

[26] S. Ko, S. Jun, Y. Ryu, O. Kwon, and K. Koh. A New Linux Swap System for Flash Memory Storage Devices. In *Proc. ICCSA*, Perugia, Italy, June 2008.

[27] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proc. 10th USENIX OSDI*, Hollywood, CA, Oct. 2012.

[28] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *Proc. 36th ACM SIGARCH*, New York, NY, June 2009.

[29] H. Lim, B. Fan, D. Andersen, and M. Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proc. 23rd ACM SOSP*, Cascais, Portugal, Oct. 2011.

[30] M. Lin, S. Chen, G. Lv, and Z. Zhou. Optimizing Linux Swap System for Flash Memory. *In IEEE Electronic Letters*, 2011.

[31] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *SIGOPS OSR*, 43(4):92–105, Jan. 2010.

[32] R. Pearce, M. Ghokale, and N. M. Amato. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *Proc. SC*, New Orleands, LA, Nov. 2010.

[33] R. Power and J. Li. Piccolo: Building Fast, Distributed Programs with Patitioned Tables. In *Proc. 9th USENIX OSDI*, Vancouver, Canada, Oct. 2010.

[34] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional Flash. In *Proc. 8th USENIX OSDI*, San Diego, CA, Dec. 2008.

[35] M. K. Qureshi and G. Loh. Fundamental Latency Trade-offs in Architecting DRAM Caches. In *Proc. 45nd IEEE MICRO*, Vancouver, Canada, Dec. 2012.

[36] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Trans. on Computer Systems*, 10(1):26–52, Feb. 1992.

[37] M. Saxena and M. M. .Swift. FlashVM: Virtual Memory Management on Flash. In *Proc. USENIX ATC*, Boston, MA, June 2010.

[38] Slashdot: Most SSDs Now Under a $ Per GB. http://hardware.slashdot.org/story/12/10/05/2156209/most-ssds-now-under-a-dollar-per-gigabyte.

[39] M. Stonebreaker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *Proc. VLDB*, Vienna, Austria, 2007.

[40] K. Sudan, A. Badam, and D. W. Nellans. NAND-Flash: Fast Disk or Slow Memory? In *Proc. 3rd NVMW*, San Diego, CA, Mar. 2012.

[41] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis. Micro-Pages: Increasing DRAM Efficiency with Locality-Aware Data Placement. In *Proc. ACM ASPLOS*, Pittsburgh, PA, Mar. 2010.

[42] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proc. ACM ASPLOS*, Newport Beach, CA, Mar. 2011.

[43] E. Witchel and K. Asanovic. Mondrian Memory Protection. In *Proc. 10th ACM ASPLOS*, San Jose, CA, Oct. 2002.

[44] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstractions for In-Memory Cluster Computing. In *Proc. 9th USENIX NSDI*, San Jose, CA, Apr. 2012.