SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy

Anirudh Badam and Vivek S. Pai
Princeton University

Abstract

We introduce SSDAlloc, a hybrid main memory management system that allows developers to treat solid-state disk (SSD) as an extension of the RAM in a system. SSDAlloc moves the SSD upward in the memory hierarchy, usable as a larger, slower form of RAM instead of just a cache for the hard drive. Using SSDAlloc, applications can nearly transparently extend their memory footprints to hundreds of gigabytes and beyond without restructuring, well beyond the RAM capacities of most servers. Additionally, SSDAlloc can extract 90% of the SSD's raw performance while increasing the lifetime of the SSD by up to 32 times. Other approaches either require intrusive application changes or deliver only 6–30% of the SSD's raw performance.

1 Introduction

An increasing number of networked systems today rely on in-memory (DRAM) indexes, hashtables, caches and key-value storage systems for scaling the performance and reducing the pressure on their secondary storage devices. Unfortunately, the cost of DRAM increases dramatically beyond 64GB per server, jumping from a few thousand dollars to tens of thousands of dollars fairly quickly; power requirements scale similarly, restricting applications with large workloads from obtaining high in-memory hit-rates that are vital for high-performance.

Flash memory can be leveraged (by **augmenting** DRAM with flash backed memory) to scale the performance of such applications. Flash memory has a larger capacity, lower cost and lower power requirement when compared to DRAM and a great random read performance, which makes it well suited for building such applications. Solid State Disks (SSD) in the form of NAND flash have become increasingly popular due to pricing. 256GB SSDs are currently around $700, and multiple SSDs can be placed in one server. As a result, high-end systems could easily augment their 64–128GB RAM with 1–2TB of SSD.

Flash is currently being used as program memory via two methods – by using flash as an operating system (OS) swap layer or by building a custom object store on top of flash. Swap layer, which works at a page granularity, reduces the performance and also undermines the lifetime of flash for applications with many random accesses (typical of the applications mentioned). For every application object that is read/written (however small) an entire page of flash is read/dirtied leading to an unnecessary increase in the read bandwidth and the number of flash writes (which reduce the lifetime of flash memory). Applications are often modified to obtain high performance and good lifetime from flash memory by addressing these issues. Such modifications not only need a deep application knowledge but also require an expertise with flash memory, hindering a wide-scale adoption of flash. It is, therefore, necessary to expose flash via a swap like interface (via virtual memory) while being able to provide performance comparable to that of applications redesigned to be flash-aware.

In this paper, we present SSDAlloc, a **hybrid DRAM/flash memory manager** and a **runtime library** that allows applications to fully utilize the potential of flash (large capacity, low cost, fast random reads and non-volatility) in a transparent manner. SSDAlloc exposes flash memory via the familiar page-based virtual memory manager interface, but internally, it works at an object granularity for obtaining high performance and for maximizing the lifetime of flash memory. SSDAlloc's memory manager is compatible with the standard C programming paradigms and it works entirely via the virtual memory system. Unlike object databases, applications do not have to declare their intention to use data, nor do they have to perform indirections through custom handles. All data maintains its virtual memory address for its lifetime and can be accessed using standard pointers. Pointer swizzling or other fix-ups are not required.

SSDAlloc's memory allocator looks and feels much like the `malloc` memory manager. When `malloc` is directly replaced with SSDAlloc's memory manager, flash is used as a fully log-structured page store. However, when SSDAlloc is provided with the additional information of the size of the application object being allocated, flash is managed as a log-structured object store. It utilizes the object size information to provide the applications with benefits that are otherwise unavailable via existing transparent programming techniques.

Using SSDAlloc, we have modified four systems built originally using `malloc`: memcached [4] (a key-value store), a Boost [1] based B+Tree index, a packet cache

Application	Original LOC	Edited LOC	Throughput Gain vs	
			SSD Swap Unmodified	SSD Swap Write Log
Memcached	11,193	21	5.5 - 17.4x	1.4 - 3.5x
B+Tree Index	477	15	4.3 - 12.7x	1.4 - 3.2x
Packet Cache	1,540	9	4.8 - 10.1x	1.3 - 2.3x
HashCache	20,096	36	5.3 - 17.1x	1.3 - 3.3x

Table 1: SSDAlloc requires changing only the memory allocation code, typically only tens of lines of code (LOC). Depending on the SSD used, throughput gains can be as high as 17 times greater than using the SSD as swap. Even if the swap is optimized for SSD usage, gains can be as high as 3.5x.

backend (for accelerating network links using packet level caching), and the HashCache [9] cache index. As shown in Table 1, all four systems show great benefits when using SSDAlloc with object size information –

- **4.1–17.4** times faster than when using the SSD as a swap space.
- **1.2–3.5** times faster than when using the SSD as a log-structured swap space.
- Only **9–36** lines of code are modified (`malloc` replaced by SSDAlloc).
- Up to **31.2** times less data written to the SSD for the same workload (SSDAlloc works at an object granularity).

The rest of this paper is organized as follows: We describe related work and the motivation in Section 2. The design is described in Section 3, and we discuss our implementation in Section 4. Section 5 provides the evaluation results, and we conclude in Section 6.

2 Motivation and Related Work

While alternative memory technologies have been championed for more than a decade [10, 25], their attractiveness has increased recently as the gap between the processor speed and the disk widened, and as their costs dropped. Our goal in this paper is to provide a transparent interface to using flash memory (unlike the application redesign strategy) while acting in a flash-aware manner to obtain better performance and lifetime from the flash device (unlike the operating system swap).

Existing transparent approaches to using flash memory [18, 20, 23] cannot fully exploit flash's performance for two reasons – 1) Accesses to flash happen at a page granularity (4KB), leading to a full page read/write to flash for every access within that page. The write/erase behavior of flash memory often has different expectations on usage, leading to a poor performance. Full pages containing dirty objects have to be written to flash. This behavior leads to write escalation which is bad not only for performance but also for the durability of the flash device. 2) If the application objects are small compared to the page size, only a small fraction of RAM contains

useful objects because of caching at a page granularity. Integrating flash as a filesystem cache can increase performance, but the cost/benefit tradeoff of this approach has been questioned before [21].

FlashVM [23] is a system that proposes using flash as a dedicated swap device, that provides hints to the SSD for better garbage collection by batching writes, erases and discards. We propose using 16–32 times more flash than DRAM and in those settings, FlashVM style heuristic batching/aggregating of in-place writes might be of little use purely because of the high write randomness that our targeted applications have. A fully log-structured system would be needed for minimizing erases in such cases. We have built a fully log-structured swap that we use as a comparison point, along with native linux swap, against the SSDAlloc system that works at an object granularity.

Others have proposed redesigning applications to use flash-aware data structures to explicitly handle the asymmetric read/write behavior of flash. Redesigned applications range from databases (BTrees) [19, 24] and Web servers [17] to indexes [6, 8] and key-value stores [7]. Working set objects are cached in RAM more efficiently and the application aggregates objects when writing to flash. While the benefits of this approach can be significant, the costs involved and the extra development effort (requires expertise with the application and flash behavior) are high enough that it may deter most application developers from going this route.

Our goal in this paper is to provide the right set of interfaces (via memory allocators), so that both existing applications and new applications can be easily adapted to use flash. Our approach focuses on exposing flash only via a page based virtual memory interface while internally working at an object level. Similar approach was used in distributed object systems [12], which switched between pages and objects when convenient using custom object handlers. We want to avoid using any custom pointer/handler mechanisms to eliminate intrusive application changes.

Additionally, our approach can improve the cost/benefit ratio of flash-based approaches. If only a few lines of memory allocation code need to be modified to migrate an existing application to a flash-enabled one with performance comparable to that of flash-aware application redesign, this one-time development cost is low compared to the cost of high-density memory. For example, the cost of 1TB of high-density RAM adds roughly $100K USD to the $14K base price of the system (e.g., the Dell PowerEdge R910). In comparison, a high-end 320GB SSD sells for $3200 USD, so roughly 4 servers with 5TB of flash memory cost the same as 1 server with 1 TB of RAM.

SSD Usage Technique	Write Logging	Read/Write < a page	Garbage Collects Dead pages/data	Avoids DRAM Pollution	Persistent Data	High Performance	Programming Ease
SSD Swap							✔
SSD Swap (Write Logged)	✔						✔
SSD `mmap`					✔		✔
Application Rewrite	✔	✔	✔	✔	✔	✔	
SSDAlloc	✔	✔	✔	✔	✔	✔	✔

Table 2: While using SSDs via swap/mmap is simple, they achieve only a fraction of the SSD's performance. Rewriting applications can achieve greater performance but at a high developer cost. SSDAlloc provides simplicity while providing high performance.

SSD Make	reads / sec		writes / sec	
	4KB	0.5KB	4KB	0.5KB
RiDATA (32GB)	3,200	3,700	500	675
Kingston (64GB)	3,300	4,200	1,800	2,000
Intel X25-E (32GB)	26,000	44,000	2,200	2,700
Intel X25-V (40GB)	27,000	46,000	2,400	2,600
Intel X25-M G2 (80GB)	29,000	49,000	2,300	2,500

Table 3: SSDAlloc can take full advantage of object-sized accesses to the SSD, which can often provide significant performance gains over page-sized operations.

3 SSDAlloc's Design

In this section we describe the design of SSDAlloc. We first start with describing the networked systems' requirements from a hybrid DRAM/SSD setting for high-performance and ease of programming. Our high level goals for integrating SSDs into these applications are:

- To present a simple interface such that the applications can be run mostly unmodified – Applications should use the same programming style and interfaces as before (via virtual memory managers), which means that objects, once allocated, always appear to the application at the same locations in the virtual memory.
- To utilize the DRAM in the system as efficiently as possible – Since most of the applications that we focus on allocate large number of objects and operate over them with little locality of reference, the system should be no worse at using DRAM than a custom DRAM based object cache that efficiently packs as many hot objects in DRAM as possible.
- To maximize the SSD's utility – Since the SSD's read performance and especially the write performance suffer with the amount of data transferred, the system should minimize data transfers and (most importantly) avoid random writes.

SSDAlloc employs many clever design decisions and policies to meet our high level goals. In Sections 3.1 and 3.4, we describe our page-based virtual memory system using a modified heap manager in combination with a user-space on-demand page materialization runtime that appears to be a normal virtual memory system to the application. In reality, the virtual memory pages are materialized in an on-demand fashion from the SSD by intercepting page faults. To make this interception as precise as possible, our allocator aligns the application level objects to always start at page boundaries. Such a fine grained interception allows our system to act at an application object granularity and thereby increases the efficiency of reads, writes and garbage collection on the SSD. It also helps in the design of a system that can easily serialize the application's objects to the persistent storage for a subsequent usage.

In Section 3.2, we describe how we use the DRAM efficiently. Since most of the application's objects are smaller than a page, it makes no sense to use all of the DRAM as a page cache. Instead, most of DRAM is filled with an object cache, which packs multiple useful objects per page, and one which is not directly accessible to the application. When the application needs a page, it is dynamically materialized, either from the object cache or from the SSD.

In Sections 3.3 and 3.5 we describe how we manage the SSD as an efficient log-structured object store. In order to reduce the amount of data read/written to the SSD, the system uses the object size information, given to the memory allocator by the application, to transfer only the objects, and not whole pages containing them. Since the objects can be of arbitrary sizes, packing them together and writing them in a log not only reduces the write volume, but also increase the SSD's lifetime.

Table 2 presents an overview of various techniques by which SSDs are used as program memory today and provides a comparison to SSDAlloc by enumerating the high-level goals that each technique satisfies. We now describe our design in detail starting with our virtual address allocation policies.

3.1 SSDAlloc's Virtual Memory Structure

SSDAlloc ideally wants to non-intrusively observe what objects the application reads and writes. The virtual memory (VM) system provides an easy way to detect what pages have been read or written, but there is no easy way to detect at a finer granularity. Performing copy-on-write and comparing the copy with the original can be used for detecting changes, but no easy mechanism de-

Application

Interacts with the SSD only via virtual memory

Virtual Memory Allocation

Heap Manager

An OPP based pool allocator for individual allocations

An OPP based coalescing allocator for OPP arrays

An MP based coalescing allocator similar to malloc

Virtual Memory Usage

DRAM & Runtime Level

Page Buffer

A FIFO cache of actual pages in core occupying only a small amount of RAM

Pages Flushed in LIFO Order

On-demand Paging

RAM Object Cache

An LRU cache of objects materialized from the SSD occupying all available RAM

Notify Objects' State

Fetch Needed Objects

Flush Dirty Objects

SSD Level

Address Translation Module

A datastructure that translates virtual memory addresses to the locations on the SSD

Translate Address for Object Read

Add/Modify Addresses of New/Moved Objects

SSD Manager

SSD Reader

SSD Writer: A copy-and-compact garbage collector

Read/ Write

SSD

A log- structured object store

Figure 1: SSDAlloc uses most of RAM as an object-level cache, and materializes/dematerializes pages as needed to satisfy the application's page usage. This approach improves RAM utilization, even though many objects will be spread across a greater range of virtual address space.

termines what parts of a page were read. Instead, SSDAlloc uses the observation that virtual address space is relatively inexpensive compared to actual DRAM, and reorganizes the behavior of memory allocation to use the VM system to observe object behavior. Servers typically expose 48 bit address spaces (256TB) while supporting less than 1TB of physical RAM, so virtual addresses are at least 256x more plentiful.

We propose the Object Per Page (OPP) model, using which, if an application requests memory for an object, the object is placed on its own page of virtual memory, yielding a single page for small objects, or more (contiguous) when the object exceeds the page size. The object is always placed at the start of the page and the rest of the page is not utilized for memory allocation. In reality, however, we employ various optimizations (described in Section 3.2) to eliminate the physical memory

wastage that can occur because of such a lavish virtual memory usage. An OPP memory manager can be implemented just by maintaining a pool of pages (details of the actual memory manager used are given in Section 3.4). OPP is suitable for individual object allocations, typical of the applications we focus on. OPP objects are stored on the SSD in a log-structured manner (details are explained in Section 3.5). Additionally, using virtual memory based page-usage information, we can accurately determine which objects are being read and written (since there is only one object per page). However, it is not straightforward to use arrays of objects in this manner. In an OPP array, each object is separated by the page's size as opposed to the object's size. While it is possible to allocate OPP arrays in such a manner, it would require some code modifications to be able to use arrays in which objects separated by page boundaries as opposed being separated by object boundaries. We describe later in Section 3.4 how an OPP based coalescing allocator can be used to allocate OPP based arrays.

3.1.1 Contiguous Array Allocations

In the C programming language, array allocations via `malloc`/`calloc` expect array elements to be contiguous. We present an option called Memory Pages (MP) which can do this. In MP, when the application asks for a certain amount of memory, SSDAlloc returns a pointer to a region of virtual address space with the size requested. We use a `ptmalloc` [5] style coalescing memory manager (further explained in Section 3.4) built on top of bulk allocated virtual memory pages (via `brk`) to obtain a system which can allocate C style arrays. Internally, however, the pages in this space are treated like page sized OPP objects. For the rest of the paper, we treat MP pages as page sized OPP objects.

While the design of OPP efficiently leverages the virtual memory system's page level usage information to determine application object behavior, it could lead to DRAM space wastage because the rest of the page beyond the object would not be used. To eliminate this wastage, we organize the physical memory such that only a small portion of DRAM contains actual materializations of OPP pages (Page Buffer) while the rest of the available DRAM is used as a compact hot object cache.

3.2 SSDAlloc's Physical Memory Structure

The SSDAlloc runtime system eases application transparency by allowing objects to maintain the same virtual address over their lifetimes, while their physical location may be in a temporarily-materialized physical page mapped to its virtual memory page in the Page Buffer, the RAM Object Cache, or the SSD. Not only does the runtime materialize physical pages as needed, but it also reclaims them when their usage drops. We first describe

how objects are cached compactly in DRAM.

RAM Object Cache – Objects are cached in *RAM object cache* in a compact manner. RAM object cache occupies available portion of DRAM while only a small part of DRAM is use for pages that are currently in use (shown in Figure 1). This decision provides several benefits – 1) Objects cached in RAM can be accessed much faster than the SSD, 2) By performing usage-based caching of objects instead of pages, the relatively small RAM can cache more useful objects when using OPP, and 3) Given the density trends of SSD and RAM, object caching is likely to continue being a useful optimization going forward.

RAM object cache is maintained in LRU fashion. It indexes objects using their virtual memory page address as the key. An OPP object in RAM object cache is indexed by its OPP page address, while an MP page (a 4KB OPP object) is indexed with its MP page address. In our implementation, we used a hashtable with the page address as the key for this purpose. Clean objects being evicted from the RAM object cache are deallocated while dirty objects being evicted are enqueued to the SSD writer mechanism (shown in Figure 1).

Page Buffer – Temporarily materialized pages (in physical memory) are are collectively known as the Page Buffer. These pages are materialized in an on-demand fashion (described below). Page Buffer size is application configurable, but in most of the applications we tested, we found that a Page Buffer of size less than 25MB was sufficient to bring down the rate of page materializations per second to the throughput of the application. However, regardless of the size of the Page Buffer, physical memory wastage from using OPP has to be minimized. To minimize this wastage we make the rest of the active OPP physical page (portion beyond the object) a part of the RAM object cache. RAM object cache is implemented such that the shards of pages that materialize into physical memory are used for caching objects.

SSDAlloc's Paging – For a simple user space implementation we implement the Page Buffer via memory protection. All virtual memory allocated using SSDAlloc is protected (via `mprotect`). A page usage is detected when the protection mechanism triggers a fault. The required page is then unprotected (only read or write access is given depending on the type of fault to be able to detect writes separately) and its data is then populated in the seg-fault handler – an OPP page is populated by fetching the object from RAM object cache or the SSD and placing it at the front of the page. An MP page is populated with a copy of the page (a page sized object) from RAM object cache or the SSD.

Pages dematerialized from Page Buffer are converted to objects. Those objects are pushed into the RAM object cache, the page is then `madvise`d to be not needed and finally, the page is reprotected (via `mprotect`) – in case of OPP/MP the object/page is marked as dirty if the page faults on a write.

Page Buffer can be managed in many ways, with the simplest way being FIFO. Page Buffer pages are unprotected, so our user space implementation based runtime would have no information about how a page would be used while it remains in the Page Buffer, making LRU difficult to implement. For simplicity, we used FIFO in our current implementation. The only penalty is that if a dematerialized page is needed again then the page has to be rematerialized from RAM.

OPP can have more virtual memory usage than `malloc` for the same amount of data allocated. While MP will round each virtual address allocation to the next highest page size, the OPP model allocates one object per page. For 48-bit address spaces, the total number of pages is 2^{36} ($\approx$ 64 Billion objects via OPP). For 32-bit systems, the corresponding number is 2^{20} ($\approx$ 1 million objects). Programs that need to allocate more objects on 32-bit systems can use MP instead of OPP. Furthermore, SSDAlloc can coexist with standard `malloc`, so address space usage can be tuned by moving only necessary allocations to OPP.

While the separation between virtual memory and physical memory presents many avenues for DRAM optimization, it does not directly optimize SSD usage. We next present our SSD organization.

3.3 SSDAlloc's SSD Maintenance

To overcome the limitations on random write behavior with SSDs, SSDAlloc writes the dirty objects when flushing the RAM object cache to the SSD in a log-structured [22] manner. This means that the objects have no fixed storage location on the SSD – similar to flash-based filesystems [11]. We first describe how we manage the mapping between fixed virtual address spaces to ever-changing log-structured SSD locations. Our SSD writer/garbage-collector is described later.

To locate objects on the SSD, SSDAlloc uses a data structure called the **Object Table**. While the virtual memory addresses of the objects are their fixed locations, Object Tables store their ever-changing SSD locations. Object Tables are similar to page tables in traditional virtual memory systems. Each Object Table has a unique identifier called the OTID and it contains an array of integers representing the SSD locations of the objects it indexes. An object's Object Table Offset (OTO) is the offset in this array where its SSD location is stored. The 2-tuple <OTID, OTO> is the object's internal persistent pointer.

To efficiently fetch the objects from the SSD when they are not cached in RAM, we keep a mapping between each virtual address range (as allocated by the OPP or the

MP memory manager) in use by the application and its corresponding Object Table, called an **Address Translation Module** (ATM). When the object of a page that is requested for materialization is not present in the RAM object cache, <OTID,OTO> of that object is determined from the page's address via an ATM lookup (shown in Figure 1). Once the <OTID,OTO> is known, the object is fetched from the SSD, inserted into RAM object cache and the page is then materialized. The ATM is only used when the RAM object cache does not have the required objects. A successful lookup results in a materialized physical page that can be used without runtime system intervention for as long as the page resides in the Page Buffer. If the page that is requested does not belong to any allocated range, then the segmentation fault is a program error. In that case the control is returned to the originally installed seg-fault handler.

The ATM indexes and stores the 2-tuples <Virtual Memory Range, OTID> such that when it is queried with a virtual memory page address, it responds with the <OTID,OTO> of the object belonging to the page. In our implementation, we chose a balanced binary search tree for various reasons – 1) virtual memory range can be used as a key while the OTID can be used as a value. The search tree can be queried using an arbitrary page address and by using a binary search, one can determine the virtual memory range it belongs to. Using the queried page's offset into this range, the relevant object's OTO is determined, 2) it allows the virtual memory ranges to be of any size and 3) it provides a simple mechanism by which we can improve the lookup performance – by reducing the number of Object Tables, there by reducing the number of entries in the binary search tree. Our heap manager which allocates virtual memory (in OPP or MP style) always tries to keep the number of virtual memory ranges in use to a minimum to reduce the number of Object Tables in use. Before we describe our heap manager design, we present a few simple optimizations to reduce the size of Object Tables.

We try to store the Object Tables fully in DRAM to minimize multiple SSD accesses to read an object. We perform two important optimizations to reduce the size overhead from the Object Tables. First – to be able to index large SSDs for arbitrarily sized objects, one would need a 64 bit offset that would increase the DRAM overhead for storing Object Tables. Instead, we store a 32 bit offset to an aligned 512 byte SSD sector that contains the start of the object. While objects may cross the 512 byte sector boundaries, the first two bytes in each sector are used to store the offset to the start of the first object starting in that sector. Each object's on-SSD metadata contains its size, using which, we can then find the rest of the object boundaries in that sector. We can index 2TB of SSD this way. 40 bit offsets can be used for larger SSDs.

Our second optimization addresses Object Table overhead from small objects. For example, four byte objects can create 100% DRAM overhead from their Object Table offsets. To reduce this overhead, we introduce object batching – small objects are batched into larger contiguous objects. We batch enough objects together such that the size of the larger object is at least 128 bytes (restricting the Object Table overhead to a small fraction – $\frac{1}{32}$). Pages, however, are materialized in regular OPP style – one small object per page. However, batched objects are internally maintained as a single object.

3.4 SSDAlloc's Heap Manager

Internally, SSDAlloc's virtual memory allocation mechanism works like a memory manager over large Object Table allocations (shown in Figure 1). This ensures that a new Object Table is not created for every memory allocation. The Object Tables and their corresponding virtual memory ranges are created in bulk and memory managers allocate from these regions to increase ATM lookup efficiency. We provide two kinds of memory managers – An object pool allocator which is used for individual allocations and a `ptmalloc` style coalescing memory manager. We keep the pool allocator separate from the coalescing allocator for the following reasons: 1) Many of our focus applications prefer pool allocators, so providing a pool allocator further eases their development, 2) Pool allocators reduce the number of page reads/writes by not requiring coalescing, and 3) Pool allocators can export simpler memory usage information, increasing garbage collector efficiency.

Object Pool Allocator: SSDAlloc provides an object pool allocator for allocating objects individually via OPP. Unlike traditional pool allocators, we do not create pools for each object type, but instead create pools of different size ranges. For example, all objects of size less than 0.5KB are allocated from one pool, while objects with sizes between 0.5KB and 1KB are allocated from another pool. Such pools exist for every 0.5KB size range, since OPP performs virtual memory operations at page granularity. Despite the pools using size ranges, we avoid wasting space by obtaining the actual object size from the application at allocation time, and using this size both when the object is stored in the RAM object cache, and when the object is written to the SSD. When reading an object from the SSD, the read is rounded to the pool size to avoid multiple small reads.

SSDAlloc maintains each pool as a free list – a pool starts with a single allocation of 128 objects (one Object Table, with pages contiguous in virtual address space) initially and doubles in size when it runs out of space (with a single Object Table and a contiguous virtual memory range). No space in the RAM object cache or the SSD is actually used when the size of pool is in-

creased, since only virtual address space is allocated. The pool stops doubling in size when it reaches a size of 10,000 (configurable) and starts linearly increasing in steps of 10,000 from then on. The free-list state of an object can be used to determine if an object on the SSD is garbage, enabling object-granularity garbage collection. This type of a separation of the heap-manager state from where the data is actually stored is similar to the "frame-heap" implementation of Xerox Parc's Mesa and Cedar languages [15].

Like Object Tables, we try to maintain free-lists in DRAM, so the free list size is tied to the number of free objects, instead of the total number of objects. To reduce the size of the free list we do the following: the free list actively indexes the state of only one Object Table of each pool at any point of time, while the allocation state for the rest of the Object Tables in each pool is managed using a compact bitmap notation along with a count of free objects in each Object Table. When the heap manager cannot allocate from the current one, it simply changes the current Object Table's free list representation to a bitmap and moves on to the Object Table with the largest number of free objects, or it increases the size of the pool.

Coalescing Allocator: SSDAlloc's coalescing memory manager works by using memory managers like pt-malloc [5] over large address spaces that have been reserved. In our implementation we use a simple *best-first with coalescing* memory manager [5] over large pre-allocated address spaces, in steps of 10,000 (configurable) pages; no DRAM or SSD space is used for these pre-allocations, since only virtual address space is reserved. Each object/page allocated as part of the coalescing memory manager is given extra metadata space in the header of a page to hold the memory manager information (objects are then appropriately offset). OPP arrays of any size can be allocated by performing coalescing at the page granularity, since OPP arrays are simply arrays of pages. MP pages are treated like pages in the traditional virtual memory system. The memory manager works exactly like traditional `malloc`, coalescing freely at byte granularity. Thus, MP with our *Coalescing Allocator* can be used as a drop-in replacement for log-structured swap.

A dirty object evicted by RAM object cache needs to be written to the SSD's log and the new location has to be entered at its OTO. This means that the older location of the object has to be garbage collected. An OPP object on the SSD which is in a free-list also needs to be garbage-collected. Since SSDs do not have the mechanical delays associated with a moving disk head, we can use a simpler garbage collector than the seek-optimized ones developed for disk-based log-structured file systems [22]. Our cleaner performs a "read-modify-write" operation over the SSD sequentially – it reads any live objects at the head of the log, packs them together, and writes them along with flushed dirty objects from RAM.

3.5   SSDAlloc's Garbage Collector

The SSDAlloc Garbage Collector (GC) activates whenever the RAM object cache has evicted enough number of dirty objects (as shown in Figure 1) to amortize the cost of writing to the SSD. We use a simple read-modify-write garbage collector, which reads enough partially-filled blocks (of configurable size, preferably large) at the head of the log to make space for the new writes. Each object on the SSD has its 2-tuple $<$OTID,OTO$>$ and its size as the metadata, used to update the Object Table. This back pointer is also used to figure out if the object is garbage, by matching the location in the Object Table with the actual offset. To minimize the number of reads per iteration of the GC on the SSD, we maintain in RAM the amount of free space per 128KB block. These numbers can be updated whenever an object in an erase block is moved elsewhere (live object migration for compaction), when a new object is written to it (for writing out dirty objects) or when the object is moved to a free-list (object is "free").

While the design so far focused on obtaining high performance from DRAM and flash in a hybrid setting, memory allocated via SSDAlloc is not non-volatile. We now present our durability framework to preserve application memory and state on the SSD.

3.6   SSDAlloc's Durability Framework

SSDAlloc helps applications make their data persistent across reboots. Since SSDAlloc is designed to use much more SSD-backed memory than the RAM in the system, the runtime is expected to maintain the data persistent across reboots to avoid the loss of work.

SSDAlloc's checkpointing is a way to cleanly shutdown an SSDAlloc based application while making objects and metadata persistent to be used across reboots. Objects can be made persistent by simply flushing all the dirty objects from RAM object cache to the SSD. The state of the heap-manager, however, needs more support to be made persistent. The bitmap style free list representation of the OPP pool allocator makes the heap-manager representation of individually allocated OPP objects easy to be serialized to the SSD. However, the heap-manager information as stored by a coalescing memory manager used by the OPP based array allocator and the MP based memory allocator would need a full scan of the data on the SSD to be regenerated after a reboot. Our current implementation provides durability only for the individually allocated OPP objects and we wish to provide durability for other types of SSDAlloc data in the future.

We provide durability for the heap-manager state of the individually allocated OPP objects by reserving a known portion of the SSD for storing the correspond-

ing Object Tables and the free list state (a bitmap). Since the maximum Object Table space to object size overhead ratio is $\frac{1}{32}$, we reserve slightly more than $\frac{1}{32}$ of the total SSD space (by using a file that occupies that much space) where the Object Tables and the free list state can be serialized for later use.

It should be possible to garbage collect dead objects across reboots. This is handled by making sure that our copy-and-compact garbage collector is always aware of all the OTIDs that are currently active within the SSDAlloc system. Any object with an unknown OTID is garbage collected. Additionally, any object with an OTID that is active is garbage collected only according to the criteria discussed in Section 3.5.

Virtual memory address ranges of each Object Table must be maintained across reboots, because checkpointed data might contain pointers to other checkpointed data. We store the virtual memory address range of each Object Table in the first object that this Object Table indexes. This object is written once at the time of creation of the Object Table and is not made available to the heap manager for allocation.

3.7 SSDAlloc's Overhead

We observe that the overhead introduced by the SSDAlloc's runtime mechanism is minor compared to the performance limits of today's high-end SSDs. On a test machine with a 2.4 GHz quad-core processor, we benchmark the SSDAlloc's runtime mechanism to arrive at that conclusion. To benchmark the latency overhead of the signal handling mechanism, we protect 200 Million pages and then measure the maximum seg-fault generation rate that can be attained. For measuring the the ATM lookup latency, we build an ATM with a million entries and then measure the maximum lookup throughput that can be obtained. To benchmark the latency of an on-demand page materialization of an object from the RAM object cache to a page within the Page Buffer, we populate a page with random data and measure the latency. To benchmark the page dematerialization of a page from the Page Buffer to an object in the RAM object cache, we copy the contents of the page elsewhere, `madvise` the page as not needed and reprotect the page using `mprotect` and measure the total latency. To benchmark the latency of TLB misses (through L3) we use a CPU benchmarking tool, the Calibrator [2], by allocating 15GB of memory per core. Table 4 presents the results. Latencies of all the overheads clearly indicate that they would not be a bottleneck even for the high-end SSDs like the FusionIO IOXtreme drives, which can provide up to 250,000 IOPS. In fact, one would need 5 such SSDs for the SSDAlloc runtime to saturate the CPU.

The largest CPU overhead is from the signal handling mechanism, which is present only because of a

Overhead Source	Avg. Latency (μsec)
TLB Miss (DRAM read)	0.014
ATM Lookups	0.046
Page Materialization	0.138
Page Dematerialization	0.172
Signal Handling	0.666
Combined Overhead	0.833

Table 4: SSDAlloc's overheads are quite low, and place an upper limit of over 1 million operations per second using low-end server hardware. This request rate is much higher than even the higher-performance SSDs available today, and is higher than even what most server applications need from RAM.

user space implementation. With an in kernel implementation, the VM pager can be used to manage the Page Buffer, which would further reduce the CPU usage. We designed OPP for applications with high read randomness without much locality, because of which, using OPP will not greatly increase the number of TLB (through L3) misses. Hence, applications that are not bottlenecked by DRAM (but by CPU, network, storage capacity, power consumption or magnetic disk) can replace DRAM with high-end SSDs via SSDAlloc and reduce hardware expenditure and power costs. For example, Facebook's memcache servers are bottlenecked by network parameters [3]; their peak performance of 200,000 tps per server can be easily obtained by using today's high-end SSDs as RAM extension via SSDAlloc.

DRAM overhead created from the Object Tables is compensated by the performance gains. For example, a 300GB SSD would need 10GB and 300MB of space for Object Tables when using OPP and MP respectively for creating 128 byte objects. However, SSDAlloc's random read/write performance when using OPP is 3.5 times better than when using MP (shown in Section 5). Additionally, for the same random write workload OPP generates 32 times less write traffic to the SSD when compared to MP and thereby increases the lifetime of the SSD. Additionally, with an in kernel implementation, either the page tables or the Object Tables will be used as they both serve the same purpose, further reducing the overhead of having the Object Tables in DRAM.

4 Implementation and the API

We have implemented our SSDAlloc prototype as a C++ library in roughly 10,000 lines of code. It currently supports SSD as the only form of flash memory, though it could later be expanded, if necessary, to support other forms of flash memory. In our current implementation, applications can coexist by creating multiple files on the SSD. Alternatively, an application can use the entire SSD, as a raw disk device for high performance. While the current implementation uses flash memory via an I/O controller such an overhead may be avoided in the fu-
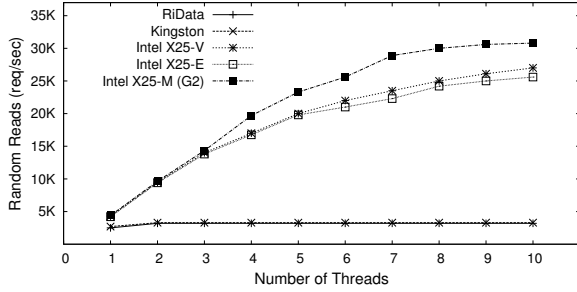
Figure 2: SSDAlloc's thread-safe memory allocators allow applications to exploit the full parallelism of many SSDs, which can yield significant performance advantages. Shown here is the performance for 4KB reads.

ture [13]. We present an overview of the implementation via a description of the API.

ssd_oalloc: *void* ssd_oalloc(int numObjects, int object-Size)*: is used for OPP allocations – both individual and array allocations. If *numObjects* is 1 then the object is allocated from the in-built OPP pool allocator. If it is more than 1, it is allocated from the OPP coalescing memory manager.

ssd_malloc: *void* ssd_malloc(size_t size)*: allocates *size* bytes of memory using the heap manager (described in Section 3.4) on MP pages. Similar calls exist for **ssd_calloc** and **ssd_realloc**.

ssd_free: *void ssd_free(void* va_address)*: deallocates the objects whose virtual allocation address is *va_address*. If the allocation was via the pool allocator then the <OTID,OTO> of the object is added to the appropriate free list. In case of array allocations, the in-built memory manager frees the data according to our heap manager. SSDAlloc is designed to work with low level programming languages like 'C'. Hence, the onus of avoiding memory leaks and of freeing the data appropriately is on the application.

checkpoint: *int checkpoint(char* filename)*: flushes all dirty objects to the SSD and writes all the Object Tables and free-lists of the application to the file *filename*. This call is used to make the objects of an application durable.

restore: *int restore(char* filename)*: It restores the SSDAlloc state for the calling application. It reads the file (*filename*) containing the Object Tables and the free list state needed by the application and mmaps the necessary address for each Object Table (using the first object entry) and then inserts the mappings into the ATM as described in Section 3.6.

SSDs scale performance with parallelism. Figure 2 shows how some high-end SSDs have internal parallelism (for 0.5KB reads, other read sizes also have parallelism). Additionally, multiple SSDs could be used within an application. All SSDAlloc functions, including the heap manager, are implemented in a thread safe manner

to be able to exploit the parallelism.

4.1 Migration to SSDAlloc

We believe that SSDAlloc is suited to the memory-intensive portions of server applications with minimal to no locality of reference, and that migration should not be difficult in most cases – our experience suggests that only a small number of data types are responsible for most of the memory usage in these applications. The following scenarios of migration are possible for such applications to embrace SSDAlloc:

- Replace all calls to `malloc` with `ssd_malloc`: Application would then use the SSD as a log-structured page store and use the DRAM as a page cache. Application's performance would be better than when using the SSD via unmodified Linux swap because it would avoid random writes and circumvent other legacy swap system overheads that are more clearly quantified in FlashVM [23].
- Replace all `malloc` calls made to allocate memory intensive datastructures of the application with `ssd_malloc`: Application can then avoid SSDAlloc's runtime intervention (copying data between Page Buffer and RAM object cache) for non-memory intensive datastructures and can thereby slightly reduce its CPU utilization.
- Replace all `malloc` calls made to allocate memory intensive datastructures of the application with `ssd_oalloc`: Application would then use the SSD as a log-structured object store only for memory intensive objects. Application's performance would be better than when using the SSD as a log-structured swap because now the DRAM and the SSD would be managed at an object granularity.

In our evaluation of SSDAlloc, we tested all the above migration scenarios to estimate the methodology that provides the maximum benefit for applications in a hybrid DRAM/SSD setting.

5 Evaluation Results

In this section we evaluate SSDAlloc using microbenchmarks and applications built or modified to use SSDAlloc. We first present microbenchmarks to test the limits of benefits from using SSDAlloc versus SSD-swap. We also examine the performance of memcached (with SSDAlloc and SSD-swap), a popular key-value store used in datacenters, where SSDs have been shown to minimize energy consumption [7]. Later, we benchmark a B+Tree index for SSDs, where we replace all calls to `malloc` with `ssd_malloc` to see the benefits and impact of an automated migration to SSDAlloc.

After that, we compare the performance of systems designed to use SSDAlloc to the same system specifically customized to use the SSD directly, to evaluate the

(a) Throughput Vs. Allocation Style (X25-E)   (b) Throughput Vs. Allocation Style (50% Reads)   (c) Write Efficiency Vs. Allocation Style
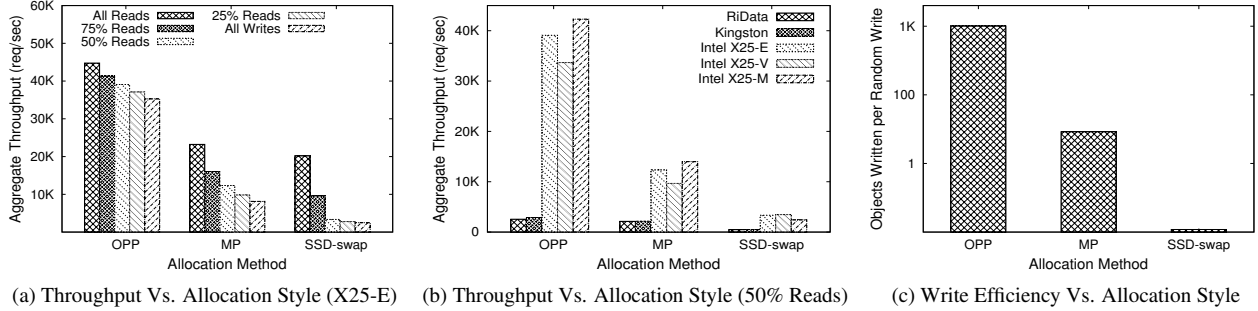
Figure 3: Microbenchmark results on 32GB object (128 byte each) array. In (a), OPP works best (1.8–3.5 times over MP and 2.2–14.5 times over swap), MP and swap take a huge performance hit when write traffic increases. In (b), OPP, on all SSDs, trumps all other methods by reducing read and write traffic. In (c), OPP has the maximum write efficiency (31.5 times over MP and 1013 times over swap) by writing only dirty objects as opposed to writing full pages containing them.

overhead from SSDAlloc's runtime. We examine a network packet cache backend that was built using transparent SSDAlloc techniques described in this paper and also the non-transparent mechanism described in our workshop paper [8]. We also evaluate the performance of a web proxy/WAN accelerator cache index for SSDs introduced in prior work [9, 8] and similar to the problems addressed more recently [6, 14]. Here, we demonstrate how using OPP makes efficient use of DRAM while providing high performance.

In all these experiments we evaluate applications using three different allocation methods: **SSD-swap** (via `malloc`), **MP** or log-structured SSD-swap (via `ssd_malloc`), **OPP** (via `ssd_oalloc`). Our evaluations use five kinds of SSDs and two types of servers. The SSDs and some of their performance characteristics are shown in Table 3. The two servers we use have a single core 2GHz CPU with 4GB of RAM and a quad-core 2.4GHz CPU with 16GB of RAM respectively.

5.1 Microbenchmarks

We examine the performance of random reads and writes in an SSD-augmented memory by accessing a large array of 128 byte objects – an array of total size of 32GB using various SSDs. We further restrict the accessible RAM in the system to 1.5GB to test out-of-DRAM performance. We access objects randomly (read or write) 2 million times per test. The array is allocated using four different methods – SSD-swap (via `malloc`), MP (via `ssd_malloc`), OPP (via `ssd_oalloc`). Object Tables for each of OPP, and MP occupy 1.1GB and 34MB respectively. Page Buffers are restricted to a size of 25 MB (it was sufficient to pin a page down while it was being accessed in an iteration). Remaining memory was used by the RAM object cache. To exploit the SSD's parallelism, we run 8–10 threads that perform the random accesses in parallel.

The results of this microbenchmark are shown in Figure 3. Figure 3(a) shows how (for the Intel X25-E SSD)

	OPP	MP	SSD-swap
Average (μsec)	257	468	624
Std Dev (μsec)	66	98	287

Table 5: Response times show that OPP performs best, since it can make the best use of the block-level performance of the SSD whereas MP provides page-level performance. SSD-swap performs poorly due to worse write behavior.

allocating objects via OPP achieves much higher performance. OPP beats MP by a factor of **1.8–3.5** times depending on the write percentage and it beats SSD-swap by a factor of **2.2–14.5** times. As the write traffic increases, MP and SSD-swap fare poorly due to reading/writing at a page granularity. OPP reads only 512 byte sector per object access as opposed to reading a 4KB page; it dirties only 128 bytes as opposed to dirtying 4KB per random write.

Figure 3(b) demonstrates how OPP performs better than all the allocation methods across all the SSDs when 50% of the operations are writes. OPP beats MP by a factor of **1.4–3.5** times and it beats SSD-swap by a factor of **5.5–17.4** times. Table 5 presents response time statistics when using the Intel X25-E SSD. OPP has the lowest averages and standard deviations. SSD-swap has a high average response time compared to OPP and MP. This is mainly because of storage sub-system inefficiencies and random writes (quantified more clearly in [23]).

Figure 3(c) quantifies the write optimization obtained by using OPP in log scale. OPP writes at an object granularity, which means that it can fit more number of dirty objects in a given write buffer when compared to MP. When a 128KB write buffer is used, OPP can fit nearly 1024 dirty objects in the write buffer while MP can fit only around 32 pages containing dirty objects. Hence, OPP writes more number of dirty objects to the SSD per random write when compared to both MP and SSD-swap (which makes a random write for every dirty object). OPP writes **1013** times more efficiently compared to SSD-swap and **31.5** times compared to MP (factors

(a) Throughput Vs. Num Instances (50% get) (b) Throughput Vs. Object Size (50% get) (c) Gain over SSD-Swap (50% get)
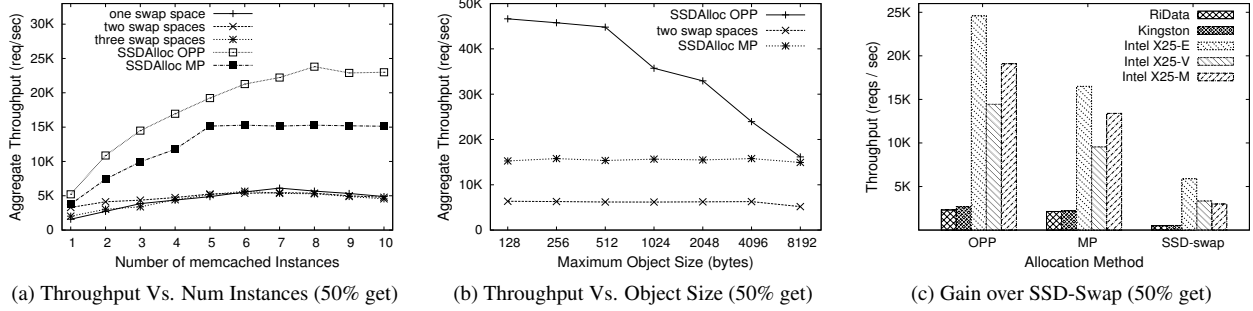
Figure 4: Memcached results. In (a), OPP outperforms MP and SSD-swap by factors of 1.6 and 5.1 respectively (mix of 4byte to 4KB objects). In (b), SSDAlloc's use of objects internally can yield dramatic benefits, especially for smaller memcached objects. In (c), SSDAlloc beats SSD-Swap by a factor of 4.1 to 6.4 for memcached tests (mix of 4byte to 4KB objects).

independent of SSD make). Additionally, OPP not only increases write efficiency but also writes **31.5** times less data compared to MP and SSD-swap for the same workload by working at an object granularity and thereby increases the SSD lifetime by the same factor.

Overall, OPP trumps SSD-swap by huge gain factors. It also outperforms MP by large factors providing a good insight into the benefits that OPP would provide over log-structured swaps. Such benefits scale inversely with the size of the object. For example with 1KB objects OPP beats MP by a factor of **1.6–2.8** and with 2KB objects the factor is **1.4–2.3**.

5.2 Memcached Benchmarks

To demonstrate the simplicity of SSDAlloc and its performance benefits for existing applications, we modify memcached. Memcached uses a custom slab allocator to allocate values and regular `malloc`s for keys. We replaced memcache's slabs with OPP (`ssd_oalloc`) and with MP(`ssd_malloc`) to obtain two different versions. These changes require modifying 21 lines of code out of over 11,000 lines in the program. When using MP, we replaced `malloc` with `ssd_malloc` inside memcache's slab allocator (used only for allocating values).

We compare these versions with an unmodified memcached using SSD-swap. For SSDs with parallelism we create multiple swap partitions on the same SSD. We also run multiple instances of memcached to exploit CPU and SSD parallelism. Figure 4 shows the results.

Figure 4(a) shows the aggregate throughput obtained using a 32GB Intel X25-E SSD (2.5GB RAM), while varying the number of memcached instances used. We compare five different configurations – memcached with OPP and MP, memcached with one, two and three swap partitions on the same SSD. For this experiment we populate memcached instances with object sizes distributed uniformly randomly from 4 bytes to 4KB such that the total size of objects inserted is 30GB. For benchmarking, we generate 1 million memcached *get* and *set* requests (100% hitrate) each using four client machines that stati-

cally partition the keys and distribute their requests to all running memcached instances.

Results indicate that SSDAlloc's write aggregation is able to exploit the device's parallelism, while SSD-swap based memcached is restricted in performance, mainly due to the swap's random write behavior. OPP (at 8 instances of memcached) beats MP (at 6 instances of memcached) and SSD-swap (at 6 instances of memcached on two swap partitions) by factors of 1.6 and 5.1 respectively by working at an object granularity, for a mix of object sizes from 4bytes to 4KB. While using SSD-Swap with two partitions lowers the standard deviation of the response time, SSD-Swap had much higher variance in general. For SSD-Swap, the average response time was 667 microseconds and the standard deviation was 398 microseconds, as opposed to OPP's response times of 287 microseconds with a 112 microsecond standard deviation (high variance due to synchronous GC).

Figure 4(b) shows how object size determines memcached performance with and without OPP (Intel X25-E SSD). Here, we generate requests over the entire workload without much locality. We compare the aggregate throughput obtained while varying the maximum object size (actual sizes are distributed uniformly from 128 bytes to limit). We perform this experiment for three settings – 1) Eight memcached instances with OPP, 2) Six memcached instances with MP and 3) Six memcached instances with two swap partitions. We picked the number of instances from the best performing numbers obtained from the previous experiment. We notice that as the object size decreases, memcached with OPP performs much better than when compared to memcached with SSD-swap and MP. This is due to the fact that using OPP moves objects to/from the SSD, instead of pages, resulting in smaller reads and writes. The slight drop in performance in case of MP and SSD-swap when moving from 4KB object size limit to 8KB is because the runtime sometimes issues two reads for objects larger than 4KB. When the Object Table indicates that they are contiguous on SSD, we can fetch them together. In comparison,

(a) Response Time Vs. Packet Size   (b) Response Time Vs. Key Size
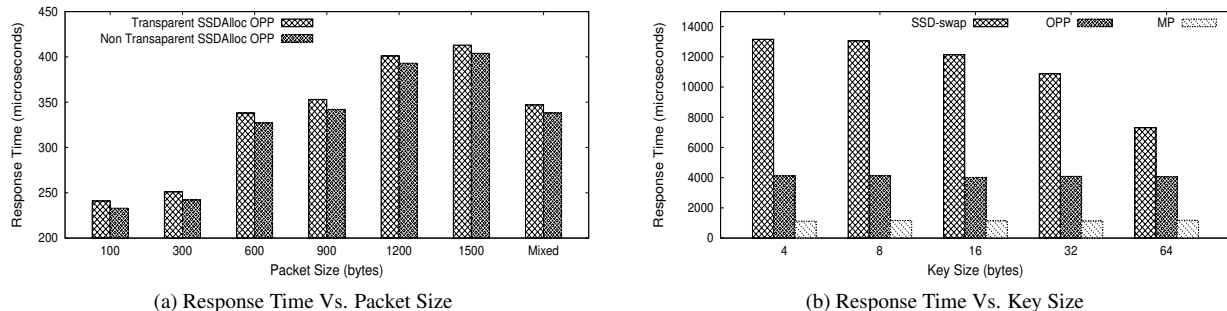
Figure 5: Packet Cache Benchmarks: In (a) we see that SSDAlloc's runtime mechanism adds only up to 20 microseconds of latency overhead, while there was no significant difference in throughput. B+Tree Benchmarks: In (b), we see that SSDAlloc's ability to internally use objects beats page-sized operations of MP or SSD-swap.

SSD-swap prefetches when possible.

Figure 4(c) quantifies these gains for various SSDs (objects between 4byte and 4KB) at a high insert rate of 50%. The benefits of OPP can be anywhere between **4.1–6.4** times higher than SSD-swap and **1.2–1.5** times higher than MP (log-structured swap). For smaller objects (each 0.5KB) the gains are **1.3–3.2** and **4.9–16.4** times respectively over MP and SSD-swap (16.4 factor improvement is achieved on the Intel X25-V SSD). Also, depending on object size distribution, OPP writes anywhere between **3.88–31.6** times more efficiently when compared to MP and **24.71–1007** times compared to SSD-swap (objects written per SSD write). The total write traffic of OPP is also between **3.88–31.6** times less when compared to MP and SSD-swap, increasing the lifetime and reliability of the SSD.

5.3 Packet Cache Benchmarks

Packet caches (and chunk caches) built using SSDs scale the performance of network accelerators [6] and inline data deduplicators [14] by exploiting good random read performance and large capacity of flash. Similar capacity DRAM-only systems will cost much more and also consume more power. We built a packet cache backend that indexes a packet with the SHA1 hash of its contents (using a hash table). We built it via two methods – 1) packets are allocated via OPP (`ssd_oalloc`), and 2) packets are allocated via the non-transparent object get/put based SSDAlloc that we describe in our workshop paper [8] – where the SSD is used directly without any runtime intervention. Remaining data structures in both the systems are allocated via `malloc`. We compare these two implementations to estimate the overhead from SSDAlloc's runtime mechanism for each packet accessed.

For the comparison, we test the response times of packet get/put operations into the backend. We consider many settings – we vary the size of the packet from 100 to 1500 bytes and in another setting we consider a mix of packet sizes (uniformly, from 100 to 1500 bytes). We use a 20 byte SHA1 hash of the packet as the key that is

stored in the hashtable (in DRAM) against the packet as the value (on SSD) – the cache is managed in LRU fashion. We generate random packet content from "/dev/random". We use the Intel X25-M SSD and the high-end CPU machine for these experiments, with eight threads for exploiting device parallelism. We first fill the SSD with 32GB worth of packets and then perform 2 million lookups and inserts (after evicting older packets in LRU fashion). In this benchmark, we configured the Page Buffer to hold only a handful of packets such that every page get/put request leads to a signal raise, and an ATM lookup followed by an OPP page materialization.

Figure 5(a) compares the response times of OPP method using the transparent techniques described in this paper and the non-transparent calls described in the workshop paper [8]. The results indicate that the overhead from SSDAlloc's runtime mechanism is only on the order of ten microseconds, there is no significant difference in throughput. Highest overhead observed was for 100 byte packets, where transparent SSDAlloc consumed 6.5% more CPU than the custom SSD usage approach when running at 38K 100 byte packets per second (30.4 Mbps). We believe this overhead is acceptable given the ease of development. We also built the packet cache by allocating packets via MP (`ssd_malloc`) and SSD-swap (`malloc`). We find that OPP based packet cache performed **1.3–2.3** times better than an MP based one and **4.8–10.1** times better than SSD-swap for mixed packets (from 100 to 1500 bytes) across all SSDs. Write efficiency of OPP scaled according to the packet size as opposed to MP and SSD-swap which always write a full page (either for writing a new packet or for editing the heap manager data by calling `ssd_free` or `free`). Using an OPP packet cache, three Intel SSDs can accelerate a 1Gbps link (1500 byte packets at 100% hit rate). Whereas, MP and SSD-swap would need 5 and 12 SSDs respectively.

12
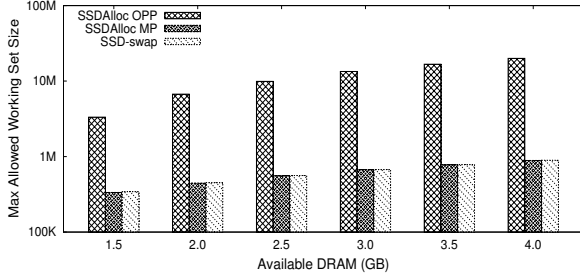
Figure 6: HashCache benchmarks: SSDAlloc OPP option can beat MP and SSD-Swap on RAM requirements due to caching objects instead of pages. The maximum size of a completely random working set of index entries each allocation method can cache in DRAM is shown (in log scale).

5.4 B+Tree Benchmarks

We built a B+Tree data structure via Boost framework [1] using the in-built Boost *object_pool* allocator (which uses `malloc` internally). We then ported it to SSDAlloc OPP (in 15 lines of code) by replacing calls to `object_pool` with `ssd_oalloc`. We also ported it to MP by replacing all calls to `malloc` (inside `object_pool`) with `ssd_malloc` (in 6 lines of code). Hence, in the MP version, every access to memory happens via the SSDAlloc's runtime mechanism.

We use the Intel X25-V SSD (40GB) for the experiments and restrict the amount of memory in the system to 256MB for both the systems to test out-of-DRAM behavior. We allow up to 25 keys stored per inner node and 25 values stored in the leaf node, and we vary the key size. We first populate the B+Tree such that it has 200 million keys, to make sure that the height of the B+Tree is at least 5. We vary the size of the key, so that the size of the inner object and leaf node object vary. We perform 2 million *updates* (values are updated) and *lookups*.

Figure 5(b) shows that MP and OPP provide much higher performance than using SSD-swap. As the key size increases from 4 to 64 bytes, the size of the nodes increases from 216 bytes to 1812 bytes. The performance of SSD-swap and MP is constant in all cases (with MP performing **3.8** times better than SSD-swap with log-structured writes) because they access a full page for almost every node access, regardless of node size, increasing the size of the total dirty data, thereby performing more erasures on the SSD. OPP, in comparison, makes smaller reads when the node size is small and its performance scales with the key size in the B+Tree. We also report that across SSDs, B+Tree operations via OPP were **1.4–3.2** times faster when compared to MP and **4.3–12.7** times faster than when compared to SSD-swap (for a 64 byte key). In the next evaluation setting, we demonstrate how OPP makes the best use of DRAM transparently.

5.5 HashCache Benchmarks

Our final application benchmark is the efficient Web cache/WAN accelerator index based on HashCache [9]. HashCache is an efficient hash table representation that is devoid of pointers; it is a set-associative cache index with an array of sets, each containing the membership information of a certain (usually 8–16) number of elements currently residing in the cache. We wish to use an SSD backed index for performing HTTP caching and WAN Acceleration for developing regions. SSD backed indexes for WAN accelerators and data deduplicators are interesting because only flash can provide the necessary capacity and performance to store indexes for large workloads. A netbook with multiple external USB hard drives (upto a terabyte) can act as a caching server [8]. The inbuilt DRAM of 1–2 GB would not be enough to index a terabyte hard drive in memory, hence, we propose using SSDAlloc in those settings – the internal SSD can be used as a RAM supplement which can provide the necessary index lookup bandwidth needed for WAN Accelerators [16] which make many index lookups per HTTP object.

We create an SSD based HashCache index for 3 billion entries using 32GB SSD space. For creating the index, HashCache creates a large contiguous array of 128 byte sets. Each set can hold information for sixteen elements – hashes for testing membership, LRU usage information for cache maintenance and a four byte location of the cached object. We test three configurations of HashCache: with OPP (via `ssd_oalloc`), MP (via `ssd_malloc`) and SSD-swap (via `malloc`) to create the sets. In total, we had to modify 28 lines of code for these modifications. While using OPP we made use of *Checkpointing*. This is because we want to be able to quickly reboot the cache in case of power outages (netbooks have batteries and a graceful shutdown is possible in case of power outages).

Figure 6(a) shows, in log scale, the maximum number of useful index entries of a web workload (highly random) that can reside in RAM for each allocation method. With available DRAM varying from 2GB to 4.5GB, we show how OPP uses DRAM more efficiently than MP and SSD-swap. Even though OPP's Object Table uses almost 1GB more DRAM than MP's Object Table, OPP still is able to hold much larger working set of index entries. This is because OPP caches at set granularity while MP caches at a page granularity, and HashCache has almost no locality. Being able to hold the entire working set in memory is very important for the performance of a cache, since it not only saves write traffic but also improves the index response time.

We now present some reboot and recovery time measurements. Rebooting the version of HashCache built with OPP Checkpointing for a 32GB index (1.1GB Ob-

ject Table) took **17.66 sec** for the Kingston SSD (which has a sequential read speed of 70 MBPS).

We also report performance improvements from using OPP over MP and SSD-swap across SSDs. For SSDs with parallelism, we partition the index horizontally across multiple threads. The main observation is that using MP or SSD-swap would not only reduce performance but also undermine reliability by writing more number of times and more data to the SSD. OPP's performance is **5.3–17.1** times higher than when using SSD-Swap, and **1.3-3.3** times higher than when using MP across SSDs (50% insert rate).

6 Conclusion

SSDAlloc provides a hybrid memory management system that allows new and existing applications to easily use SSDs to extend the RAM in a system, while performing up to 17 times better than SSD-swap, up to 3.5 times better than log-structured SSD-swap and increasing the SSD's lifetime by a factor of up to 30 with minimal code changes, limited to memory allocation. The performance of SSDAlloc applications is close to that of custom-developed SSD applications. We demonstrate benefits of SSDAlloc in a variety of contexts – a data center application (memcached), a B+Tree index, a packet cache backend and an efficient hashtable representation (Hash-Cache), which required only minimal code changes, little application knowledge, and no expertise in the inner workings of SSDs.

7 Acknowledgments

References

[1] Boost, . http://www.boost.org/.

[2] Calibrator, . http://homepages.cwi.nl/~manegold/Calibrator/#6.

[3] Scaling Memcaced at Facebook, . http://www.facebook.com/note.php?note_id=39391378919.

[4] Memcached, . http://www.danga.com/memcached/.

[5] ptmalloc, . http://www.malloc.de/en/.

[6] 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.

[7] 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 Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[8] A. Badam and V. S. Pai. Beating Netbooks into Servers: Making Some Computers More Equal Than Others. In *Proc. 3rd ACM Workshop on Networked Systems for Developing Regions (NSDR)*, BigSky, MO, 2009.

[9] A. Badam, K. Park, V. S. Pai, and L. L. Peterson. Hashcache: Cache storage for the next billion. In *Proc. 6th USENIX NSDI*, Boston, MA, Apr. 2009.

[10] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. In *Proc. AS-PLOS'92*, 1992.

[11] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. *Operating Systems Review*, 42(2): 88–93, 2007.

[12] M. Castro, A. Adya, B. Liskov, and A. C. Myers. Hac: Hybrid adaptive caching for distributed storage systems. In *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malô, France, Oct. 1997.

[13] 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 Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[14] B. Debnath, S. Sengupta, and J. Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2010.

[15] P. V. der Linder. *Expert C Programming: Deep C Secrets*. Prentice Hall, Englewood Cliffs, N.J, 1994.

[16] S. Ihm, K. Park, and V. S. Pai. Wide-area Network Acceleration for the Developing World. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2010.

[17] T. Kgil and T. N. Mudge. Flashcache: A NAND flash memory file cache for low power web servers. In *Proc. of CASES'06*, 2006.

[18] S. Ko, S. Jun, Y. Ryu, O. Kwon, and K. Koh. A New Linux Swap System for Flash Memory Storage Devices. In *In ICCSA'09*, 2008.

[19] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory SSD in enterprise database applications. In *Proc. ACM SIGMOD*, Vancouver, BC, Canada, June 2008.

[20] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating system support for NVM+DRAM hybrind main memory. In *Proc. HotOS XII*, Monte Verita, Switzerland, May 2009.

[21] D. Narayanan, E. Thereska, A. Donelly, S. Elnikety, and A. Rowstron. Migrating server storage to ssds, analysis of tradeoffs. In *Proceedings of EuroSys'09*, 2009.

[22] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[23] M. Saxena and M. M. Swift. Flashvm: Virtual memory management on flash. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2010.

[24] C.-H. Wu, L.-P. Chang, and T.-W. Kuo. An efficient b-tree layer for flash-memory storage systems. In *Proccedings of RTCSA'04*, 2004.

[25] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *Proc. 6th International Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, Oct. 1994.