

Beating Netbooks Into Servers: Making Some Computers More Equal Than Others

Anirudh Badam and Vivek S. Pai
Department of Computer Science
Princeton University
{abadam,vivek}@cs.princeton.edu

Abstract

With the advent of cheap netbooks, expanding Internet access to the developing world is becoming more and more feasible. However, classroom computers benefit from applications typically deployed on servers, such as proxy caches, WAN accelerators, backup, etc. Server infrastructure, unfortunately, is not as well suited for developing-world use, since it tends to be less ruggedized, more power-hungry, and more expensive, making it difficult for small deployments to adequately maintain, stock and replace.

To address these issues, we consider the notion of turning a netbook in to a low-end server via software support. Many netbooks ship with multi-gigabyte solid state disks (SSD) that are intended for local storage. By attaching an external disk and using the SSD to augment the typically-meager RAM memory, we can instead use a netbook as a server. Unfortunately, SSD behaves very differently from RAM, and requires special consideration to be used effectively. To this end, we introduce SSDAlloc, a hybrid memory manager that provides simple programming constructs that developers can use to build server software with small RAM footprints and good performance. With SSDAlloc and an external hard disk, a netbook can act as a server for the entire school, dramatically reducing deployment cost and complexity.

1 Introduction

While the gap between the “haves” and “have-nots” in Internet access is wide, the gap between the “haves” and the “almost-haves” may not be much better. As the first world moves toward user-generated content, social networking, blogs, comment-driven sites, and more participation, having anything less than full access to the Internet will degrade the Internet user experience. While many people believe that read-only offline access to the Internet is “good enough”, we believe that this approach will hinder developing-world users from sharing information not only with the developed world, but also with each other, which appears to be a largely unaddressed desire [13]. In the longer term, we also believe that offline-only access will fail to spur the kind of Internet growth seen in the first world.

While being a second-class Internet citizen is no doubt better than being excluded completely, a number of technological advances may soon render the choice between first-class and second-class a false dichotomy. Low-cost laptops can bring personal computing to large numbers of people [4, 9]. Long-range wireless can bring connectivity where no connectivity existed [10]. Large-capacity low-cost disks can provide bulk storage that transforms how developers think

of data retention. Solid-state disks can boost the performance of out-of-core applications. Current low-cost laptops combined with USB-attached hard drives can provide this level of hardware for \$400 USD per unit, and this cost may drop over time.

Our focus, at a high level, is to use these technologies to narrow the gap between the usage experiences of the developed world and the developing world. This combination will likely mean that our focus will be on the growing urban middle class and the upper-middle class in particular, but even this target audience is sizable – if we assume that one-quarter of India’s and China’s population falls into this category, the number of users exceeds the total population of the United States. We target this audience based on the observation that even if \$400 USD is a large value in local currency, many middle-class parents in these countries view it as an investment in their children’s education. If such technology can provide a usage experience similar to that of the developed world, it also provides self-empowerment rather than charity, with a family laptop seen as an aspirational item, akin to a television, scooter, or car.

We also view targeting the urban middle class as a means of helping build local ecosystems. Online access can more easily drive advertising and advertising-based purchases, both of which subsidize the cost of developing and delivering content. As more people use the online Internet, the fixed costs of traffic delivery are spread across more users, lowering the cost of delivery, which can then generate more demand from more users. Commercialization of the Internet in the US has generated so much volume that the researchers who originally used the Internet can now buy bandwidth and access much more cheaply than prior to commercialization. We hope that lowering the cost of online access in the developing world can generate a similar effect.

Low-cost netbooks make an attractive platform for such deployments in the developing world, given their cost, portability, and power consumption. They can be used in many different scenarios including schools, small clinics, government offices, bank branches etc. However, without coordination, laptops make poor use of network resources, such as redundantly fetching cacheable content over the network. To address this problem, the practical approach is to deploy servers that provide centralized services such as Web caching, WAN acceleration, disk storage, local e-mail, etc. These servers could be deployed in schools, in apartment complexes, in Web cafes, or even by the ISPs themselves.

While servers solve the coordination-related problems, they introduce new management problems for the deployment. Servers tend to cost more than laptops, due to fewer economies of scale and the desire to use higher-quality components. For smaller deployments, the cost of the server may constitute a significant portion of the over-

all deployment cost. Servers tend to be less ruggedized than laptops, so they are more sensitive to environmental conditions and other problems [14]. To avoid downtime from failure, either a secondary server needs to be available on site, or easy replacements need to be available locally. Deploying secondary servers costs extra, while ensuring a ready supply of backups complicates the supply chain and inventory management. Servers typically also use more energy than netbooks, often by a factor of 10 or so, and without their own batteries, must depend on a uninterruptible power supply to handle short outages.

Given their downsides, it is worth considering whether netbooks can be used to replace servers in such deployments. The netbooks already have several advantages over servers – they are lower cost, they are ruggedized, and when every student already has a netbook, the supply chain and replacement system problems become much simpler. In the worst case, failure of the “server” netbook can be handled simply by taking one from a student and using it instead.

One barrier to using netbooks in this capacity is the limited amount of RAM typically found in these systems, compared to the amount of RAM needed in servers that are running multiple server-style applications. These applications often access or index large volumes of data, and the RAM requirements of these applications often scale in proportion to their activity or the amount of storage/data they are handling. So when a netbook is serving a classroom full of students, the demands placed on it are higher than when it is being used as a student’s personal machine. Unfortunately, netbooks are not typically designed to be very expandable, and do not provide multiple available connectors to expand the RAM capacity of the machine.

To address this problem, we can exploit the fact that many netbooks ship with multiple gigabytes of solid state disk (SSD), which is normally used for local storage, but which has speed properties that fall closer to RAM than disk. When a netbook is converted to a server, this small SSD is insufficient for storage, so an external USB-attached hard disk will be used for the primary storage, freeing the SSD to be used for other purposes. In particular, we want to use the SSD to augment the meager RAM of these systems, such that the programs that run on the netbook-turned-server can use the SSD as program memory, rather than storage.

We present SSDAlloc, a hybrid main memory management mechanism, to aid application developers in writing high performance applications for hybrid RAM-SSD systems. It provides a mechanism to tag data items according to their usage characteristics, providing the information needed to determining when to store it on stable storage, when to cache it, and when to replicate it for non-volatility.

1.1 Rationale

The problems with using SSD as a RAM substitute relate to the behavioral disparity between the two. While SSD has fast read speeds, writes tend to be slower, and writes can only be performed on regions that have been bulk-erased, a very slow process that is performed at coarse granularity. Other researchers have tried to use SSD in various capacities, such as transparently migrating virtual memory pages [6], using it as swap [12], or redeveloping applications specifically to use it [15].

We argue that a very desirable scenario is one where a little programming effort can deliver significant performance possible in a hybrid architecture. While using SSD as a swap layer will transparently provide some performance benefits, the cost-benefit of this approach has been disputed [7], and especially on very low-memory netbooks, swapping whole pages in and out of memory will needlessly tax the already moderate CPU, and bring RAM performance

down to that of the SSD. We believe that the application logic can better understand what data is frequently accessed and what is not, and can yield better performance by separating data on the basis of access frequencies of actual data entities and not whole pages. Our work aims at providing programming tools for application logic to express such characteristics to maximize the impact of using SSD.

SSDAlloc targets precisely the memory-intensive set of networking applications that we want to deploy in the developing world, such as Web servers, Web caches, WAN accelerators, and search engines. Our previous experience designing a web service for a low-footprint environment was the Web proxy cache HashCache [1], which decoupled the RAM consumption of the cache from the amount of disk storage being used. While that effort yielded a significant performance and capacity gain in our target environments, it required a complete redesign of the application, and is not likely to occur for a large number of applications without some concerted effort. Instead, we hope that SSDAlloc can allow applications to shift most of their memory usage to SSD, reducing RAM pressure, without having to invasively modify much of the application. While SSDAlloc requires more developer effort than completely transparent approaches [12, 5, 6, 16], we believe that for developing world scenarios using netbooks, the tradeoff is appropriate if the applications are going to be broadly deployed. Although, one can design automatic techniques to migrate to this hybrid setting using static analysis of code and dynamic analysis of the runtime behavior of the application. We discuss more about this in Section 6.

To illustrate why SSDAlloc is needed for the developing world, consider the case of a WAN accelerator where the average piece of content stored is 1KB. If the cache storage is 1 TB, then 1 billion index entries are needed, which would require 8GB of RAM if each entry is 8 bytes (a number derived from the HashCache-Log configuration [1]). This much RAM is far beyond the capacities of all netbooks, but many netbooks ship with this much SSD storage. We propose the usage of SSDs as RAM substitute. We explore the design space and provide with an efficient way of utilizing SSDs as a RAM substitute. In general, our techniques can be used for any flash based storage and not just SSDs. We used SSDs since that was the most prevalent form of flash storage available in systems at the time of writing this paper.

To motivate the problem we propose a very simplistic scenario where the SSD can be used as part of the virtual memory where it contains pages that can be accessed by the application similar to the manner in which applications access pages in RAM today. To speed up the process and to amortize the cost one could use RAM as a page cache for all the pages on the SSD using any of the existing page replacement policies prevalent in virtual memory system. Such a simple agnostic page replacement policy might potentially fault on every page access and refer to the flash for each request bringing down the performance of RAM to that of the SSD. The actual set of useful data in RAM may be much smaller, since only a few byte ranges from each page might actually be in use, meaning that most of RAM, while technically being used, is really providing little value in terms of the data truly needed by the application.

Utilizing SSD and RAM effectively can take different forms depending on the application. For very small data structures, one may opt to use RAM to cache recently-accessed data structures that are spread across the SSD. For data structures where fields have different usage patterns, one might pack heavily-used fields in RAM and leave the rest in SSD. For these reasons, we design SSDAlloc to manage RAM as an object cache rather than a page cache, which can improve small-memory server performance.

The rest of this paper is organized as follows: The design is dis-

cussed in Section 2, and we discuss our prototype in Section 3. Section 4 provides preliminary evaluation results using a system that we built using SSDAlloc. Section 5 presents a brief study of existing work in the area and we conclude in Section 6.

2 Design

In this section we describe the design of SSDAlloc, which allows applications to easily manage the hybrid SSD/RAM memory allocation. First we discuss why directly caching application objects in DRAM is better than caching pages containing these objects. Then we describe how such objects can be stored on SSD (and cached in RAM) and exposed to the application in a transparent manner. Finally, we discuss the optimizations that would enable the usage of SSD as a random access device by enabling efficient random reads and writes to these application objects.

2.1 Chunking the RAM

One of the main design decisions in SSDAlloc is managing memory allocations at the object level, and then aggregating application objects into units we call Chunks. The reason for this approach is twofold – to gather useful data into less memory, and to mitigate the performance difference between RAM and SSD. When objects are allocated, by default their permanent homes are on the SSD, which allows for much more storage space than the RAM. In the kind of server-style applications we are targeting, allocation order and location may have little to do with usage order or popularity. As a result, objects of varying popularity may share the same page of SSD memory.

When these objects get used, they are either fully or partially cached in RAM at the object level, rather than the page level. This decision provides two benefits. The first is that objects cached in RAM can be accessed much faster than the SSD, mostly because of systems architectures. The I/O bus used to connect drives to the system is much slower than RAM, simply because peripherals are generally not designed to have the kind of bandwidth that RAM can provide. The second benefit is that when objects can be cached based on usage, rather than caching entire pages, the relatively small main memory can cache more useful objects. Given the density trends of SSD and RAM, object caching is likely to continue being a useful optimization going forward.

To access objects regardless of location, SSDAlloc provides what we call Chunk Tables, which are similar to the concept of page tables in the virtual memory design. The Chunk Table contains the location of each set of objects, regardless of location (SSD or RAM). Each allocation of objects generates its own Chunk Table.

The Chunk Table allows applications to perform random accesses (both reads and writes), regardless of where an object is currently stored. While RAM can easily handle both random reads and writes, SSD can only handle random reads well. Due to the non-overwrite behavior of SSDs, a truly RAM-like random write behavior would mean that part of the SSD would have to be erased and re-written with the new data. Not only would this cause a performance problem due to the long delay of the erase operation, but reliability would also be impacted, since SSDs also have limited numbers of erase cycles.

To address this problem we adopt the approach used by flash-based filesystems [3], and manage the Chunks on the SSD as a log-structured system [11]. Whenever something is written to a Chunk, it is remapped to a new location which is essentially the current offset in the log. This new location is reflected appropriately in the Chunk Table for the application to use.

This remapping process also means that the older locations for Chunks are now garbage, and have to be garbage collected. The simplest garbage collector is essentially the log manager. It reads in a certain number of erase blocks from the current head and then first aggregates the live Chunks in these blocks. Later it flushes its dirty Chunk cache and the present live Chunks on to these erase blocks so that wastage is minimized. It also updates the Chunk Tables appropriately. While this kind of approach would have been too simplistic for disk-based log-structured file systems [11], the difference is that SSDs have no mechanical disk head to move, so eager cleaning and other optimizations have less benefit.

This approach solves several problems related to SSDs – it minimizes writes, minimizes fragmentation and also does wear leveling. Another advantage of having Chunks as the logical entity rather than pages is that writes are aggregated as Chunks and not pages leading to lesser number of erases for the same random write workload.

2.2 Chunk Sizes

Applications can need Chunks that are different in sizes and to minimize SSD storage wastage the size of the Chunk must be configurable. While allowing arbitrary Chunk sizes minimizes space wastage it can lead to a higher utilization of RAM since the Chunk Table would have to store the size of the Chunk in addition to the location of the Chunk on the SSD. Arbitrary Chunk sizes can also lead to a higher overhead in the garbage collection mechanism which has to perform Chunk cleaning and aggregation for arbitrarily sized Chunks. To minimize such a complexity while also minimizing space wastage we allow the Chunk to be chosen from among a few predetermined sizes. For example, each Chunk can only be either 32 bytes, or 64 bytes, or 128 bytes or 256 bytes etc. Each Chunk Table can then represent the locations of Chunks of the same size to avoid storing the size of the Chunk on a per Chunk basis in the Chunk Table. Such a choice for the size of a Chunk provides the right tradeoff between space wastage, performance of garbage collection and restricts the size of the Chunk Table. Such a design, leads to a garbage collector that has to monitor only a few locations per an actual SSD page to determine all the Chunks that that particular page has.

3 Preliminary Implementation

We have implemented a preliminary version of SSDAlloc, which requires roughly 2500 lines of C code. It currently supports SSD as the only form of flash memory, though we believe that it could later be expanded, if necessary, to support flash-based USB sticks, etc. The main functions it includes are as follows:

ssdalloc: *ssdunit ssdalloc(int numChunks, int ChunkSize, int cacheSize)*: This function creates a new object called *ssdunit* which is essentially a Chunk Table for the *numChunks* Chunks of size *ChunkSize* it creates on the SSD. It also creates a Chunk cache of size *cacheSize* in RAM for these Chunks. This cache is managed as a simple LRU queue. The allocation function simply gives out a Chunk Table for future references. This Chunk Table contains the location of each of the Chunks on the SSD which is garbage collected in a log structured manner. For reading and writing one needs to just reference the Chunk Table and use the metadata available there to know about the actual location of the Chunk on the SSD.

ssdalloc: *ssdunit ssdalloc(int numChunks, int ChunkSize, int cacheSize)*: This function is like *ssdalloc*, but zero-fills the allocated chunks. Given the cost of erasing SSD, the performance difference between this and *ssdalloc* warrants having two functions.

ssdrealloc: *ssdunit ssdrealloc(ssdunit unit, int numChunks):* This function grows the size of the allocated Chunk Table to the requested number of Chunks.

readChunk: *int readChunk(ssdunit unit, int ChunkNum, void* buffer):* reads the Chunk *ChunkNum* of *unit* and copies it to *buffer*. It first checks the RAM cache for chunks before checking the SSD.

writeChunk: *int writeChunk(ssdunit unit, int ChunkNum, void* buffer):* writes to the Chunk *ChunkNum* of *unit* from *buffer*. If the Chunk is in the RAM cache then it writes to the cache and marks it as dirty. If the Chunk is not in the RAM cache, then an existing entry in the cache may have to be evicted in order to make space. If the eviction process chooses a dirty entry, all dirty entries, including the chunk from this call, are written to SSD. However, only the oldest entry on the LRU list is actually evicted. This process also triggers the garbage collector, described later.

flushUnit: *int flushUnit(ssdunit unit):* Any dirty chunks in RAM are flushed to the SSD. This function may be called periodically while the server runs in order to reduce any problems due to power failure, and it can be called before the server gracefully terminates.

3.1 Garbage Collection and Dirty Cache

The Garbage Collector (GC) activates whenever the RAM cache has enough dirty entries to amortize the cost of writing to the SSD or during idle CPU times when there are enough objects to be flushed to flash.. When this happens, the GC writes them to the SSD and updates the Chunk Tables appropriately. Each Chunk on the SSD has a back pointer (metadata) to the Chunk Table that contains it. This back pointer helps the GC update the Chunk Table. The GC is a simple copy-and-compact garbage collector, and ensures that it reads enough partially-filled blocks to make enough space for the new writes. To minimize the number of reads per each iteration of GC on the SSD we maintain in RAM the amount of free space per each erase block. These numbers can be updated whenever a Chunk in an erase block is moved elsewhere or when new Chunks are written to the whole block when it is garbage collected. The entire per-chunk metadata in our implementation is only 4 bytes.

3.2 Migration to SSDAlloc

We believe that SSDAlloc is suited to the memory-intensive portions of server applications, and that migration should not be difficult in most cases. The reason for this belief is that our experience suggests that a small number of data types are responsible for most of the data usage in these applications, and that their use is tied to transactions with explicit start and end behaviors. For example, when a Web server receives a request, it has to operate on a file and use it until the request is satisfied, and then it no longer needs the file. As a result, the program has explicit points at which it can request and release the objects for that transaction.

The migration process is relatively straightforward, and while we cannot automate it at the moment, we can provide a relatively simple process for migrating applications. The first step is to identify which data structures are responsible for most of the memory usage, and focus on those. The other allocations can be handled by the standard malloc system, which can coexist with SSDAlloc. If objects are individually allocated, the next step is to write a simple pool allocator per data type. The pool allocator simply allocates an array of objects each time from SSDAlloc, and then provides them individually to the caller. The final step is to add the *readChunk* calls to get the object's current location before each use, and to call

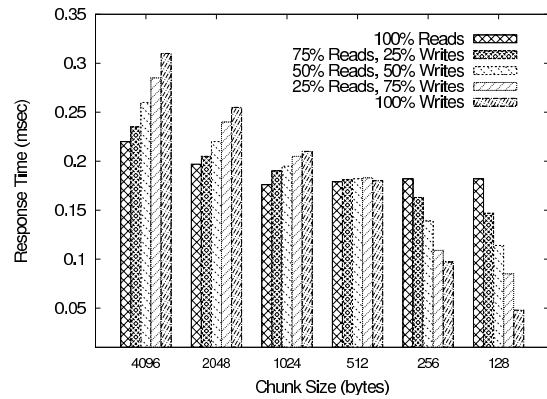


Figure 1: Average response time for 2 million request microbenchmark with varying amounts of reads and writes. As the Chunk size decreases, the write performance increases due to fewer erases on the SSD for the same workload.

writeChunk to write back any modified object when it is no longer needed.

Cases like doubly-linked lists (by themselves or in chained hash tables) can make coding more complicated, since every operation that rearranges elements also modifies the pointers in the previous and next elements in the list. In this case, the simplest step is to keep the pointers in RAM, and just use SSDAlloc to allocate the other elements within the structure. In this case, linked list manipulations do not require extra calls to *writeChunk*. In the case where each element in the list is relatively small compared to the pointers, we recommend keeping the contents of multiple elements in one Chunk, rather than operating on a per-element basis. We admittedly are still in the process of gaining more experience with applications that have these constraints.

4 Preliminary Evaluation

Our preliminary evaluation of SSDAlloc includes some microbenchmarks as well as one real server application. In particular, we run a HashCache system [1] in its memory-intensive high-performance configuration, known as HashCache-Log. We use this configuration to index two billion objects on a system with 4GB of RAM and 32GB of flash (an SSD), which can do 6000 random reads/sec. The actual amount of RAM used in our tests varies in size from netbook-like to server-like.

The results of a random read and write microbenchmark is shown in Figure 1, where SSDAlloc is used to allocate 16GB of data with chunk sizes ranging from 128 bytes to 4096 bytes. The test performs 2 million operations, at random offsets, with a mixture of reads and writes. Each operation acts on 128 bytes of data, so the chunk sizes show the range of tightly-packed useful data down to useful data scattered among useless data.

The throughput on this microbenchmark is nearly six times as high with small chunks versus large chunks for writes. The RAM cache used is minimal in size to test the performance in netbook settings, so the tests demonstrate the effect of data transfers from the SSD. There is no significant difference in read performance for Chunk sizes below 512 bytes because the minimum reads that the device allows is 512 bytes. The 100% write workload improves for small chunks because these can be packed together into one 512-byte sector when being written.

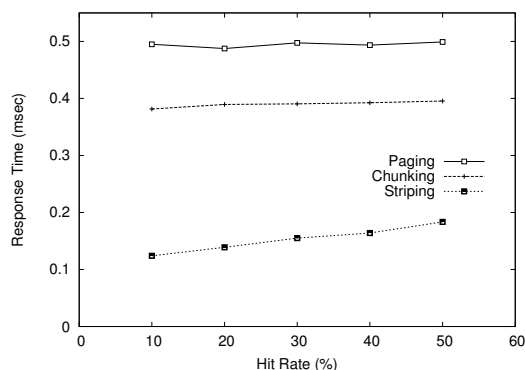


Figure 2: Average response time for 2 million index operations for each technique. Performance of Striping is dependent on hit rate and is in general much higher than Chunking and Paging which have a uniformly high overhead at all hit rates.

The application test of the HashCache server provides more insight into SSDAlloc’s benefits. With two billion index entries in this configuration, HashCache requires 16GB of index space, far more than the RAM on the machine. We use such a high number of index entries to perform stress tests, in reality the number of index entries needed for a deployment would be much smaller. We test three configurations:

1. **HashCache with Paging:** We configure SSDAlloc to use a Chunk size of 4096 bytes, the same as the OS page size, to measure the performance we can expect from swapping to SSD. We use a RAM cache of 2GB, enough, in theory to cache 12.5% of the workload.
2. **HashCache with Chunking:** We use a 128 byte Chunk size to maximize the utility of the RAM cache. The modification to HashCache are minimal and maintain the layout of all data structures. With 2GB of RAM, we can expect to cache 12.5% of the workload with a better expected hit rate in RAM cache than Paging. Chunking is interesting because it can be used for migrating applications in a trivial manner, just by replacing malloc calls with ssdalloc calls.
3. **HashCache with Striping:** We break HashCache’s main data structure into two parts – the frequently-used portion, the hash bits, are kept in an ssdunit with a large RAM cache size (expected 50% hit rate), and the less-used portion, the metadata, is kept in a different ssdunit with a smaller RAM cache size (expect 15% hit rate). The total size of the two RAM caches is still 2GB.

Figure 2 shows the response time for 2 million index operations on HashCache for varying hit rates. We assume a high constant insert rate of 70% representative of web workloads. This figure shows how the performance of Striping scales with the hit rate. Striping uses fewer reads than Paging and Chunking since the hash bits are more likely to be found in RAM, and the application accesses the rest of the metadata only if the hash bits match. Paging and Chunking on the other hand use the same number of reads at all hit rates since the data structures are kept intact. Paging and Chunking also have the disadvantage of having to edit LRU information on flash for hits leading to a higher number of writes. For this test, Paging and Chunking will need reads and writes for almost every index operation, leading to a high performance impact. The average response

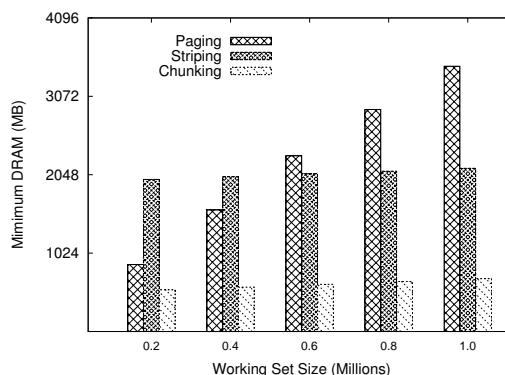


Figure 3: Minimum DRAM needed to hold working set size. Striping uses more memory than Chunking but provides performance benefits in doing so. Paging consumes more memory and performs worse as the working-set size increases.

time also includes the total time needed to flush the units after the 2 Million operations complete.

Figure 3 shows the amount of RAM needed to keep a random working set in RAM. With working set sizes varying from 200,000 to 1 million (less than 0.1% of the entire workload) index entries we show how Chunking uses substantially less memory than Paging. Chunking and Striping are created with a 128 byte Chunk, so the Chunk Tables for Chunking and Striping are 32 times larger than that of Paging, but still the Chunk Tables are only 512 MB in size for the entire 2 Billion entry index. Also, we use a 50% cache for hashbits and LRU bits in Striping (that is hashbits for 50% of the index entries are cached in main memory and the rest are on the SSD), around 1.5GB. Striping uses more memory, by aggressively caching hashbits and LRU bits, than Chunking but saves SSD bandwidth (read and write) for other applications to use. Paging performs poorly compared to both Chunking and Striping requiring higher RAM due to the fact that each object in the working set could be on a different page. Striping with a higher constant overhead than Chunking presents a trade-off of SSD bandwidth vs RAM consumption. Although, for many cases 2 Billion entries is very large. Smaller settings will have smaller Striping overhead.

5 Related Work

While alternative memory technologies have championed for more than a decade [16], the improvements in DRAM had largely kept them sidelined until recently. With advancements in capacity and reliability, flash memory has recently received widespread adoption for nonvolatile storage, even though its appropriateness as good intermediate layer between RAM and disk have been long recognized [2].

Subsequent research has focused on using flash as a storage layer to improve disk seek performance [5, 15]. More recent research has examined how to transparently use flash in the memory hierarchy [12, 6], focusing on providing power efficiency without performance loss. Some of these approaches focus on making flash a memory layer inside the VM and implement paging across DRAM and flash or the disk [5, 6]. The disadvantage of these techniques for our environment is that in making the same pages span across DRAM and flash reduce the goodput of DRAM.

SSDAlloc, in comparison, tries to exploit the memory density of SSD to augment RAM. Given the cost of low-end SSD storage and

its widespread use in netbooks, using it as a RAM replacement can provide server-like behavior that is well-suited for developing-world environments. While we find that the maximum benefit is achieved with application modification, we believe this tradeoff is acceptable given the reduction in deployment and maintenance complexity.

Other flash-based techniques examine redesigning programs to get the maximum performance possible [8, 15]. With a tool like SSDAlloc, the necessary changes can focus on the core data structures and can leave all of the SSD-related decisions to the library. The most closely related previous work was performed by Wu et al [15], in which they design flash-aware data structures. SSDAlloc provides a software layer so that developers can very easily make all their data structures flash-aware by providing the right amount of write caching and leaving wear leveling to the library. Previous work in data structures for flash includes Birrell et al [3], which proposes augmenting flash with main memory storage. Our Chunk Table is essentially a mapping similar to the one proposed by them, but in reverse – our goal is to maximize overall memory and performance, rather than focusing in overcoming the limitations of flash.

6 Conclusion and Future Work

In this paper we present the design of SSDAlloc, a hybrid main memory architecture suitable for the development of web services for Developing World. Such hybrid architectures are easily available in the form of netbooks today. These netbooks are more sturdy and resistant to dust, power failures, and low voltages and provide the right kind of infrastructure for web services in developing regions. SSDAlloc enables the conversion of netbooks to servers, eliminating the management and supply headaches of maintaining special servers. Our prototype implementation and modified application demonstrate the potential of SSDAlloc and this server-less approach to service deployment, and we expect to gain experience with more applications and live deployments going forward.

While programming tools like SSDAlloc can definitely provide better and faster server applications, involvement of the programmer must be minimized. We are working towards minimizing programmers' involvement in these hybrid memory management techniques to obtain the best possible benefits. Towards that goal we are making efforts in two different directions that when combined can provide the transparency needed. First, we aim to exploit source code, when available, to make an automatic migration from malloc to ssdalloc. Second, we aim to analyze the runtime behavior of the application's virtual memory usage and determine the right parameters for application's random access data in this hybrid setting.

References

- [1] A. Badam, K. Park, V. Pai, and L. Peterson. Hashcache: Cache storage for the next billion. In *Proceedings of NSDI'09*, 2009.
- [2] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. In *Proceedings of ASPLOS'92*, 1992.
- [3] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. *Operating Systems Review*, 42(2):88–93.
- [4] Intel. Classmate PC, <http://www.classmatepc.com/>.
- [5] T. Kgil and T. N. Mudge. Flashcache: A NAND flash memory file cache for low power web servers. In *Proceedings of CASES'06*, 2006.
- [6] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating system support for NVM+DRAM hybrid main memory. In *Proceedings of HotOS'09*, 2009.
- [7] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating server storage to ssds, analysis of tradeoffs. In *Proceedings of EuroSys'09*, 2009.
- [8] S. Nath and A. Kansal. FlashDB: Dynamic self tuning database for nand flash. In *Proceedings of IPSN'07*, 2007.
- [9] One Laptop Per Child. <http://www.laptop.org/>.
- [10] R. Patra, S. Nedeveschi, S. Surana, A. Sheth, L. Subramanian, and E. Brewer. Wildnet: Design and implementation of high performance wifi based long distance networks. In *Proceedings of NSDI'07*, 2007.
- [11] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of SOSP'91*, 2009.
- [12] Spansion Inc. Using spansion EcoRAM to improve TCO and power consumption in internet data centers. http://www.spansion.com/about/news/events/spansion_ecoram_whitepaper_0608.pdf.
- [13] S. R. Sterling, J. W. O'Brien, and J. K. Bennett. Advancement through interactive radio. In *Proceedings of ICTD 2007*, 2007.
- [14] S. Surana, R. Patra, S. Nedeveschi, M. Ramos, L. Subramanian, Y. Ben-David, and E. Brewer. Beyond Pilots: Keeping Rural Wireless Networks Alive. In *Proceedings of NSDI'08*, 2008.
- [15] C.-H. Wu, L.-P. Chang, and T.-W. Kuo. An efficient b-tree layer for flash-memory storage systems. In *Proceedings of RTCSA'04*, 2004.
- [16] M. Wu and W. Zwaenepoel. eNVY: A non-volatile main memory storage system. In *Proceedings of ASPLOS'94*, 1994.