Bringing Modern Hierarchical Memory Systems Into Focus

A study of architecture and workload factors on system performance

Paul Tschirhart
University of Maryland, College Park
pkt3c@umd.edu

Jim Stevens
University of Maryland, College Park
jims@cs.umd.edu

Zeshan Chishti
Intel Labs
Hillsboro, Oregon, USA
zeshan.a.chishti@intel.com

Shih-Lien Lu
Intel Labs
Hillsboro, Oregon, USA
shih-lien.l.lu@intel.com

Bruce Jacob
University of Maryland, College Park
blj@umd.edu

ABSTRACT

The increasing size of workloads has led to the development of new technologies and architectures that are intended to help address the capacity limitations of DRAM main memories. The proposed solutions fall into two categories: those that re-engineer Flash-based SSDs to further improve storage system performance and those that incorporate non-volatile technology into a Hybrid main memory system. These developments have blurred the line between the storage and memory systems. In this paper, we examine the differences between these two approaches to gain insight into the types of applications and memory technologies that benefit the most from these different architectural approaches.

In particular this work utilizes full system simulation to examine the impact of workload randomness on system performance, the impact of backing store latency on system performance, and how the different implementations utilize system resources differently. We find that the software overhead incurred by storage based implementations can account for almost 50% of the overall access latency. As a result, backing store technologies that have an access latency up to 25 microseconds tend to perform better when implemented as part of the main memory system. We also see that high degrees of random access can exacerbate the software overhead problem and lead to large performance advantages for the Hybrid main memory approach. Meanwhile, the page replacement algorithm utilized by the OS in the storage approach results in considerably better performance on highly sequential workloads at the cost of greater pressure on the cache.

MEMSYS 2015, October 05-08, 2015, Washington DC, DC, USA

DOI: http://dx.doi.org/10.1145/2818950.2818975

CCS Concepts

•Computer systems organization → *Architectures;* **Heterogeneous (hybrid) systems; •Hardware** → **Non-volatile memory;**

Keywords

Hybrid Memory, SSD, DRAM Capacity, Full System Simulation, Non-Volatile Memory

1. INTRODUCTION

In the last few years, the distinction between the storage system and the memory system has become increasingly vague. Data sets have grown at a rate of 50% annually and have quickly outpaced the growth of the main memory system [14]. Ideally, the entire workload would be resident in the main memory for fast access. However, the limited capacity growth and cost per bit of DRAM constrain the amount of data that can be placed in main memory. This limitation of the main memory system has resulted in an effort among researchers to find a new way to increase the effective size of the main memory system. SSDs have already seen widespread adoption to improve the read performance of data centers and one direction of research seeks to further improve the performance of SSDs [6] [31] [13] [7]. Many of these suggestions center around improving the way in which the OS interacts with the SSD such as introducing object stores or reducing inter-core communication [6] [7]. The other direction of research seeks to expand the main memory system by using novel non-volatile memory technologies (such as PCM or flash) to build a Hybrid memory system that includes a memory level between the DRAM and the disk [26] [19] [12] [17]. Our work investigates the advantages and disadvantages of the SSD and Hybrid approaches.

Fundamentally these two architectures are expanding the main memory capacity in the same way. They are both adding a new, slower level to the main memory hierarchy and using the DRAM to cache it. Interestingly, the primary differences between these two approaches that are blurring the line between storage and memory have to do with the way that they utilize their DRAM cache. The SSD approach uses the virtual memory manager (or some other software) to manage the DRAM cache while the Hybrid system manages it in hardware. This difference shapes the constraints on

the cache management heuristics used by both approaches and ultimately results in the two systems utilizing system resources in different ways. So, analyzing these systems requires an understanding of all levels of the memory hierarchy: caches, main memory and storage.

The effectiveness of these two approaches is primarily determined by the miss rate of the DRAM cache and the average hit / miss latency ratio of the system. These two aspects of the system are, in turn, the cumulative product of multiple design attributes. For instance, the miss rate is affected by the size of the cache, its degree of associativity, and its replacement policy (including any prefetching that might be done). Similarly, the average hit/miss latency ratio is affected by the base access latencies of the cache and the backing store, the available concurrency at both levels, and the prefetching heuristic utilized. Each of these design attributes plays an important role in determining the overall performance of the system and in extreme cases can totally negate the positive effects of the others. Therefore, to understand the advantages and disadvantages of the storage and memory based approaches, it is necessary to understand how these approaches react to different aspects of the overall architecture.

To understand these interactions this work performs a series of full system simulation experiments that are intended to isolate and expose the effects and overheads of the SSD and Hybrid approaches. These experiments focus on simple, easy to understand single and multi-threaded workloads in order to provide the most straightforward picture of the architecture choices that favor one approach over the other. However, some additional representative storage workloads are also included to provide context for the simple workload results.

The goal of this work is to untangle some of the complicated relationships that shape the performance of these systems in order to create a general sense of which approaches are best suited to which architectures and workloads. To this end we find the following results to be of interest:

- Storage implementations enjoy full associativity, aggressive prefetching heuristics, and more complex replacement heuristics for their DRAM cache. As a result, storage based systems achieve better performance in architectures with large caches, very slow backing stores, and/or highly sequential workloads. However, these features come at the cost of a considerable software overhead that can greatly hamper performance when the features fail to provide a performance improvement.

- Context switching can also provide a source of speed up in storage based systems provided that the backing store is slow enough and there is other useful work to be done. In most other situations though it hinders performance and is best avoided.

- Memory implementations benefit from minimal overheads that allow them to perform better when the backing store is fast, when the cache is small and when workloads exhibit more random accesses.

- Relatively slow memory technologies still achieve better performance with a memory based approach for many workloads suggesting that technologies such as SLC flash can be used more aggressively than they currently are for certain applications. However, around typical MLC flash latencies, it makes more sense to context switch rather than wait.

2. ARCHITECTURE COMPARISON

The Hybrid architecture evaluated by this work resembles the Flash Hybrid architecture proposed in [17] and the PCM Hybrid architecture proposed in [26]. This architecture extends the functionality of the memory controller to manage the DRAM main memory as a cache for a large backing store. The backing store hardware has been kept largely identical to the hardware of an SSD to minimize the changes required by this architecture. The key difference is that the memory controller, rather than the operating system, is responsible for generating requests and managing whether a page is stored in the DRAM cache or the backing store. Figure 1 illustrates the diffences between the SSD and Hybrid approaches.

The following section will describe the key design differences of the Hybrid and SSD systems by comparing the Hybrid design to the design of a current state-of-the-art enterprise PCIe SSD. These differences are summarized in Table 1.

2.1 Stalling vs. Interrupting

Since the storage system was designed for long latency spinning disks, one of the key concerns was hiding disk latency. This was done on time-shared systems by having the OS scheduler switch to another task and using an I/O interrupt to indicate completion of the access. The process waiting for I/O would later be awoken by the OS scheduler to resume execution. Modern systems have inherited this feature, which still makes sense for most types of I/O operations. However, as with high speed networks, when the hardware is capable of high enough throughput or low enough latency, interrupting can create unnecessary overhead in the system. This can come from several possible sources including the time it takes to service an interrupt, the time it takes to perform a context switch to another task, the time it takes the OS scheduler to resume execution of the waiting task, and indirect costs such as cache pollution. At the current time, the typical design for a modern SSD still relies on the interrupt and task switch approach and we are assuming that for the SSD version of our experimental setup.

Since backing store accesses are hidden from the OS in the Hybrid memory design, the interrupt and task switch approach is replaced with a simpler stalling approach. This is similar to how the CPU cores simply wait on the memory controller to perform DRAM accesses. While this may seem non-intuitive due to the much longer latency of some backing store technologies such as flash, if the overhead of the interrupt and task switch approach is sufficiently large, then the stalling approach can offer better performance. The work in [20] indicated that the delay incurred by a context switch can be as much as 1.5 ms. Therefore, in some situations a system could benefit greatly from stalling on a read access to the backing store rather than task switching and waiting for an interrupt.

2.2 Page Placement

Currently, the OS virtual memory system determines which

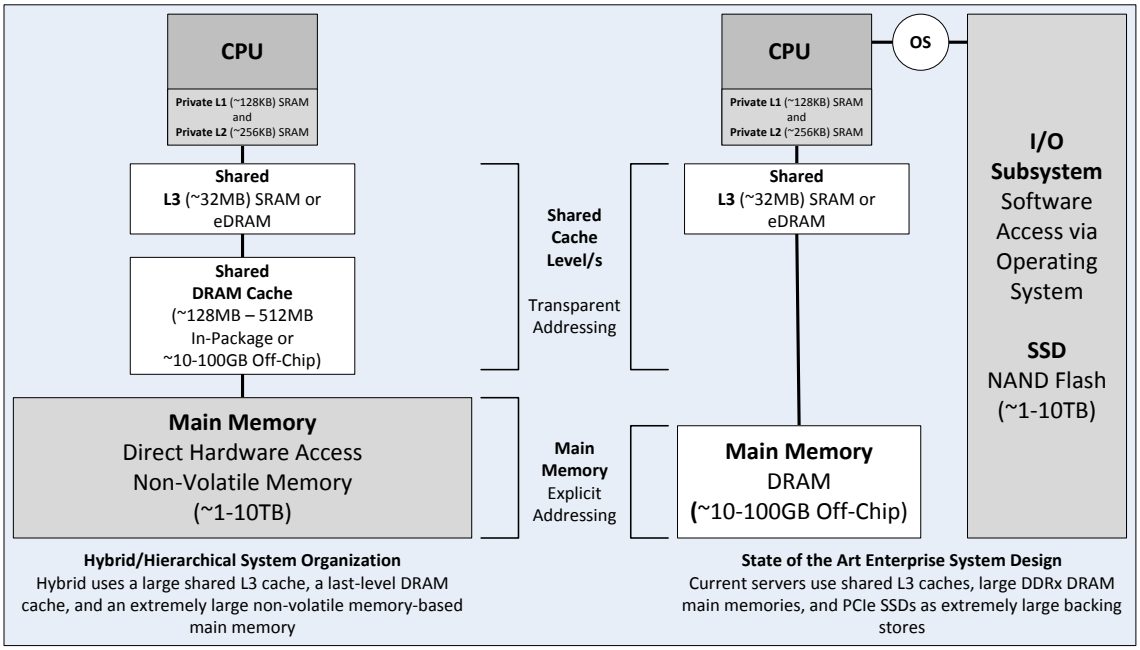

Figure 1: Hybrid organization versus a typical enterprise-class SSD organizataion

Table 1: Hybrid Memory vs. SSD Comparison

	Hybrid	**SSD**
Page Placement	CPU memory controller	OS - virtual memory/buffer cache
Garbage Collection (where necessary)	flash controller chip	flash controller chip
Virtual to Physical Address Translation	OS - virtual memory	OS - virtual memory
Physical to Backing Store Address Translation	CPU memory controller	OS - block layer
Backing Store access scheduling	backing store controller chip	OS - I/O scheduler
Host Interface	direct connection to CPU memory controller	PCIe root complex
File System	tmpfs ramdisk	ext3

virtual pages are kept in the main memory page frames and which virtual pages are stored on the SSD backing store (either the original file for file-backed page or the swap for anonymous virtual pages). This process is described in Figure 2-a. Step 1 of this diagram starts with the application generating a request to the virtual memory system. Step 2 occurs on a page miss; here the virtual memory system selects and evicts a virtual page from the main memory. The virtual memory system also passes the requested virtual page to the I/O system. During step 3 the I/O system generates a request for the SSD. This request is then sent to the PCIe root complex which directs it to the SSD in step 4. To specify which virtual page to bring in from the SSD, the OS sends the SSD controller a logical block address. The SSD then uses that logical block address to determine the actual physical location of the virtual page associated with that address and issues a request to the device which contains that virtual page. For the virtual page that is evicted from the main memory, the SSD allocates a new physical page for that virtual page and issues a write to the appropriate device. This occurs between steps 4 and 5. After the SSD handles the request, it sends the data back to the CPU via the PCIe root complex, step 5. The PCIe root complex the passes the data to the main memory system where it is written, step 6. Once the write is complete the PCIe root complex raises an interrupt alerting the OS scheduler that an applications request is complete. This is step 7. Finally, during step 8, the application resumes, reissues its request to the virtual memory system and generates a page hit for the data. The division of the virtual memory address space between the different physical address spaces in SSD-based systems is described in Figure 3-a.

In the Hybrid system, the backing store is presented to the OS virtual memory manager as the entire physical memory address space. This address space organization is presented in Figure 3-b. It appears to the OS that the computer's main memory is the size of the backing store. The actual DRAM main memory address space is hidden from the OS and is managed by the memory controller as a cache. Together the backing store and DRAM cache form the Hybrid memory. Accesses to this Hybrid memory have a granularity of 64 bytes, just like DRAM. The cache lines in the DRAM cache have a granularity of 4KB because that is the typical size of an OS virtual memory page. Keeping the fill granularity the same in both the SSD and Hybrid systems removes a possible source of confusion from the results. Figure 2-b. shows the access process for the Hybrid architecture which is considerably simpler than the SSD process. Step 1 begins with the application generating a request. Step 2, varies somewhat depending on the degree of associativity

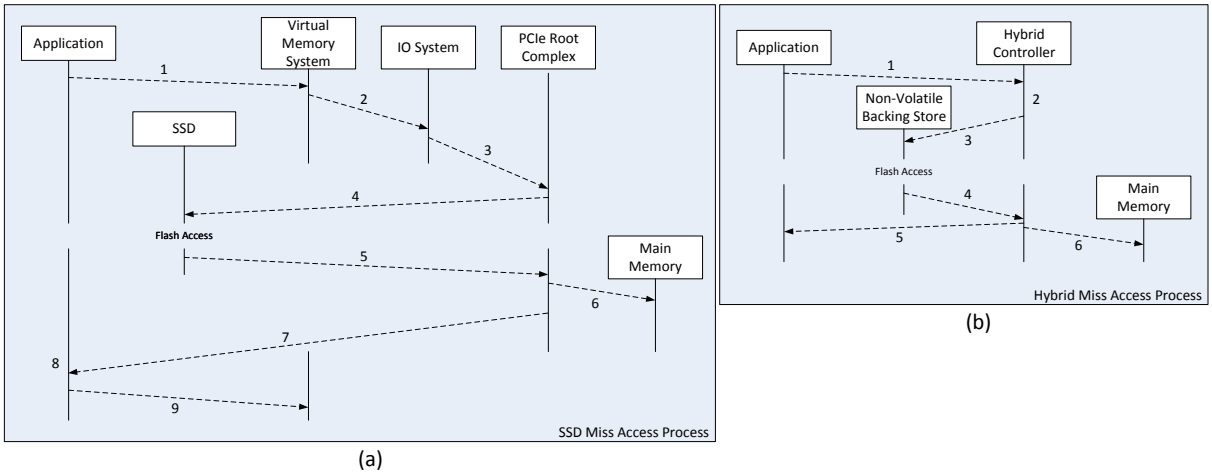

Figure 2: Comparison of the steps involved in servicing a miss of the DRAM for both the SSD (a) and Hybrid (b) organizations.

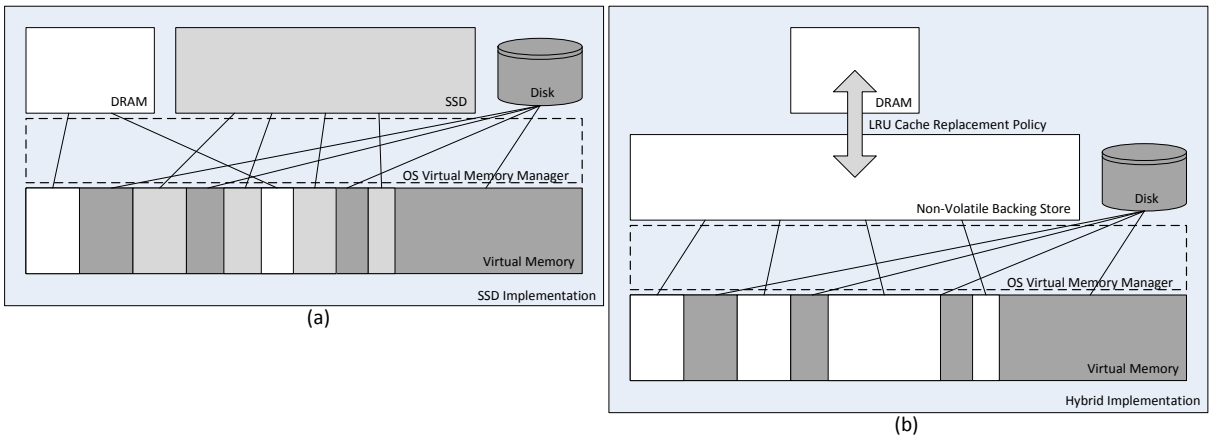

Figure 3: Comparison of the address spaces involved in the SSD (a) and Hybrid (b) organizations.

implemented in the Hybrid system. If the system is using a direct mapped cache like the one proposed in [25] then the appropriate location in the DRAM is determined from the address and accessed. The tag is retrieved with the data in a single accesses. If the system is implementing associativity we assume that it is using an SRAM tag store. In this system a DRAM cache access begins with the Hybrid memory controller checking its tag database to determine if a particular cache line is present in the DRAM cache. If the cache line is present in the DRAM cache, then the access is serviced by the DRAM as a normal main memory access (not shown in Figure 2-b.). When an access misses the DRAM cache, the Hybrid controller selects a page in the DRAM to evict and schedules a write-back if the page is dirty. In the current implementation of our Hybrid memory controller, a least recently used (LRU) algorithm is used to determine which page to evict. The missed page is then read in from the backing store and placed in the DRAM. This is step 3. The Hybrid memory controller can also prefetch additional pages into the DRAM or write back cold dirty pages preemptively, similar to how the virtual memory memory works, to further improve read performance. Currently, our Hybrid system implements sequential prefetching. More complex prefetching schemes such as stream buffers, stride prefetching, and application directed prefetching are also compatible with this design. Step 4 is the backing store handling the request. Once the data has been received from the backing store the Hybrid controller passes the data to the application, step 5. Finally, during step 6 the data is written into DRAM from the Hybrid memory controller.

2.3 Associativity

The page table utilized by the OS is functionally fully associative and so the cache in our Hybrid system is 16 way set-associative in order to provide a more fair comparison. We also analyze the effects of different levels of associativity on Hybrid performance later in the study to determine the roll that associativity plays in the performance of both systems.

In order to implement associativity efficiently a Hybrid main memory design must store its cache tags in an efficient manner. The tag size is computed as b - c + a + s,

where b is the number of bits in the backing store address space, c is the number of bits in cache address space, a is the number of bits in the cache associativity, and t is the number of tag state bits including the valid bit, the dirty bit, replacement policy state bits, and other related data. For sufficiently large Hybrid memories, the tag store can become too large in terms of transistor budget to store directly on the CPU. There have been several solutions that have been proposed to solve the problem of tag overhead in DRAM caches. These range from adding a tag cache to temporarily store tags [16], only storing tags alongside data in DRAM [21] and only implementing DRAM caches as direct mapped [25]. To keep matters as straightforward as possible in this work we assume that the tags of the associative Hybrid implementations are stored in SRAM and can be accessed efficiently. Such a system could be realized with either a dedicated tag store or a tag cache in a real world implementation. Additionally, we find in our experimentation that associativity has a relatively limited effect on many of these workloads. As a result, the design suggested in [25] would also produce many of the same results without the need for tag storage. We note that an interesting subject for future work would be to expand upon this study by comparing different DRAM cache designs.

2.4 Prefetching

The operating system prefetches data off out of the backing store on a page fill in a effort to reduce future misses and to amortize the cost of the backing store access. To keep the comparison as fair as possible we have implemented a sequential prefetcher as part of the Hybrid design. On a cache fill this prefetcher grabs the next 16 pages after the missed address in addition to the missed page itself. We arrived at the 16 page window as a result of experimentation that is not included in this study. More advanced prefetching schemes are another area of potential future work that the researchers would like to explore.

3. EXPERIMENTAL SETUP

3.1 Base System Parameters

The simulation environment used in this work is based on the MARSSx86 [24] cycle-accurate full system simulator, which consists of the PTLSim and QEMU subcomponents. PTLSim models an x86-64 multicore processor with full details of the pipeline, micro-op front end, trace cache, reorder buffers, and branch predictor. This processor model includes a full cache hierarchy model and implements several cache coherency protocols. In addition, MARSSx86 utilizes QEMU as an emulation environment to support any hardware not explicitly modeled by the full system environment, such as network cards and disks. This simulation environment is able to boot a full, unmodified operating system, such as any modern Linux distribution, and run unmodified benchmarks. The simulator captures both the user-level and kernel-level instructions, unlike other simulations that are user-level only, enabling the study of the operating system.

To model the memory system, DRAMSim2 [28], a cycle accurate, hardware verified DRAM memory system simulator has been integrated into MARSSx86 to service last level cache misses. The Hybrid memory system simulator extends the base MARSSx86 + DRAMSim2 system further by adding two additional modules: OpenMemorySimulator (OMS) and HybridSim. OMS provides detailed simulation of the backing store. HybridSim simulates the memory controller for a Hybrid architecture, providing cache management mechanisms to utilize the DRAM and backing store efficiently. OMS has been verified against DRAMSim2 when simulating DRAM and against the publicly available flash access protocols when simulating non-volatile memory. The full-system infrastructure models the full behaviors of the studied capacity (i.e., it uses 1MB of DRAM to model 1MB of flash), so we simulate our 1TB backing store and cache with a scaled-down capacity model that represents the timing of the full-size hierarchy.

Table 2: System Under Study

Processor	
Number of cores	4-core
Issue Width	4
Frequency	2GHz
On Chip Caches	
L1I (private)	128 KB, 8-way, 64 B block size
L1D (private)	128 KB, 8-way, 64 B block size
L2 (private)	256 KB, 8-way, 64 B block size
L3 (shared)	32 MB, 20-way, 64 B block size
DRAM Cache	
Organization	64GB, 16-way, 4 KB page size
DRAM Bus Frequency	DDR3-1333
DRAM Bus Width	64 bits per channel
DRAM Channels	1-16
DRAM Ranks	1 Ranks per channel
DRAM Banks	8 Banks per rank
Row Size	1024 Bytes
tCAS-tRCD-tRP-tRAS	10-10-10-24
Backing Store	
Organization	1TB, 4 KB page size
Backing Store Bandwidth	PCIe 3.0 x16 equivalent

To understand the performance of the Hybrid architecture relative to current systems that utilize solid state drives, we created a full-system SSD simulation and integrated it into MARSSx86. In this configuration, the main memory consists only of DRAMSim2. To the best of our knowledge, this is the first full-system SSD simulation. Prior SSD simulation work [5] [11] utilizes trace-based simulation only to study SSD performance. The SSD simulator is implemented with a module wrapped around HybridSim called PCI_SSD to simulate the host interface and DMA engine simulation. The internals of the SSD are simulated with OMS, making the non-volatile memory simulation identical for both the Hybrid and SSD versions. The SSD model explicitly simulates direct memory access via a callback to the DRAMSim2 main memory. The addresses for the DMA requests are extracted from QEMU's scatter-gather lists. These lists consist of pairs of pointers and sizes to enable the DMA request to access non-contiguous locations in the DRAM address space. In addition, we have also modified QEMU to utilize AHCI drivers instead of the default IDE drivers. This enables Native Command Queuing and allows the SSD-based system to take advantage of hardware parallelism in multi-threaded runs.

The full scale system that is modeled in all of our simulations is shown in Table 2.

3.2 Benchmarks

To simulate random read access, we utilize GUPS and

MMAP. GUPS is based on an implementation of the Giga-Updates Per Second benchmark by Sandia National Labs [2]. In our experiments, this benchmark allocates a table that is one eighth the size of the simulated backing store in the virtual memory space and then randomly updates locations within it 5000 times. Since the table is larger than main memory, GUPS forces the virtual memory system to swap the tableâĂŹs anonymous pages to the backing store.

Our MMAP experiment maps a file that is half the size of the simulated backing store into the virtual address space with the mmap() system call and then performs 10000 random reads with the file. Once again, since the file is larger than main memory, the file system cannot read in the entire file into the buffer cache and is forced to read the random addresses from the backing store. For the Hybrid memory version, we overlay a filesystem on top of the memory address space using a tmpfs ramdisk. The purpose of MMAP is to provide a filesystem workload in contrast to the swapping workload of GUPS so we can understand different aspects of OS overhead when comparing the SSD and Hybrid architectures.

Both MMAP and GUPS also incorporate OpenMP to allow for multiple threads performing random accesses to utilize more parallelism at the hardware level for our study. To understand the effect of varying the degree of random access within our workloads, the gups benchmark also has a âĂIJrandom probabilityâĂİ, which determines if the next access will be sequential following the current access or if it will be an independent random access. Finally, because both of these benchmarks are run to completion in our tests, execution cycles are used to measure performance.

To test the Hybrid and SSD systems with a large sequential workload, we built microbenchmarks DD_READ and DD_WRITE based on the dd Linux utility. Both benchmarks measure the time required to move a 64 MB file between the backing store and DRAM. For DD_READ, the 64 MB file is created and forced to the backing store with sync and cache flush operations. The dd program is then used to copy that into memory. In the case of the Hybrid memory, a file system is created using a tmpfs ramdisk and we force the sections of main memory that make up the ramdisk to be flushed to the backing store before the DD_READ starts. DD_WRITE performs a dd run to copy data from /dev/zero and write the data to the disk. For the SSD case, we ensure that the DD_WRITE actually writes to the disk rather than the RAM buffer cache by using the conv\=fdatasync option to dd. For the Hybrid case, we use a special MMIO operation to tell the memory controller to flush dirty pages in the cache to the backing store.

Together we refer to these four workloads as the targeted benchmarks as their intent is to isolate certain behaviors of the systems in question.

To provide context for the results from our targeted workloads, we also utilize several representative benchmarks from the Filebench workload generator [1]. We use the varmail, webserver, webproxy, and fileserver workloads from this suite. Unlike the targeted benchmarks, performance for these workloads is measured in IOP/S because they run for a fixed time of 250 ms in the ROI of each workload.

4. EXPERIMENTAL RESULTS

The overall goal of this work is to better understand how certain features of the underlying hardware architecture or workloads affect the performance of systems that utilize a slow backing store as either storage or memory. To achieve that understanding we began by identifying the two principle factors that affected performance in both types of system: the miss rate of the DRAM cache and the average hit/miss latency ratio. The experiments that make up this study therefore focus on the features that affect those two principle factors such as backing store latency, cache size, associativity, etc. Together these experiments provide a clear picture of which architectural and workload features benefit or harm the performance of both SSD and Hybrid systems and why. For the purposes of this study we do not focus on any particular backing store technology but instead examine points of interest on the range of possible backing store latencies.

4.1 The Effect of Backing Store Latency

We begin our investigation by looking at the effects of the backing store latency on the performance of the SSD and Hybrid approaches. This is a logical starting point since the difference between storage and memory technologies has traditionally largely been determined by their access latency. The potential to task switch and perform useful work while a page fill is taking place makes the SSD approach better suited to longer latency technologies. However, it is unclear exactly how long that latency should be in order to see any benefit from the SSD approach. So, we are looking for the crossover point where the backing store is too slow for the Hybrid approach and better suited to the SSD approach. To find this crossover point, we steadily increase the read latency of the backing store by a factor of 10 from 125 to 12500 ns. In addition, we also include some additional latencies at particular points of interest. For instance, 25000 ns is the read latency of SLC NAND Flash and 75000 ns is the read latency of MLC NAND Flash. Because most non-volatile technologies feature asymmetric read to write latencies, we use a write latency that is 10x the read latency for these experiments. However, with the exception of DD_Write, the benchmarks in this study do not feature write traffic to the backing store as a system bottleneck.

Comparing the results in Figure 4 it is clear that for the random workload GUPS the Hybrid architecture performs considerably better at all backing store latencies. This suggests that the cost of a cache miss is considerably greater for the the storage implementation than it is for the memory one. In addition, the Hybrid approach is clearly advantageous at all backing store latencies for the single threaded case of MMAP. However, increasing the thread count from 1 to 16 improved the performance of the SSD based system considerably more than the Hybrid system and resulted in the SSD system out performing the Hybrid system. This was not the case with GUPS and that suggests that tmpfs may be responsible for some of the lost performance in the multi-threaded Hybrid MMAP runs.

DD_Read shows the performance of the SSD to be slightly faster than the Hybrid system. At this point we hypothesized that the superior sequential performance of the SSD system was probably due to its prefetching and associativity which helped to keep its miss rate lower. This also explains why DD_Write does not show a similar advantage for the SSD. The prefetching and associativity of the storage implementation cannot help with a write heavy workload even if it is streaming. Also, because the writes are triggering

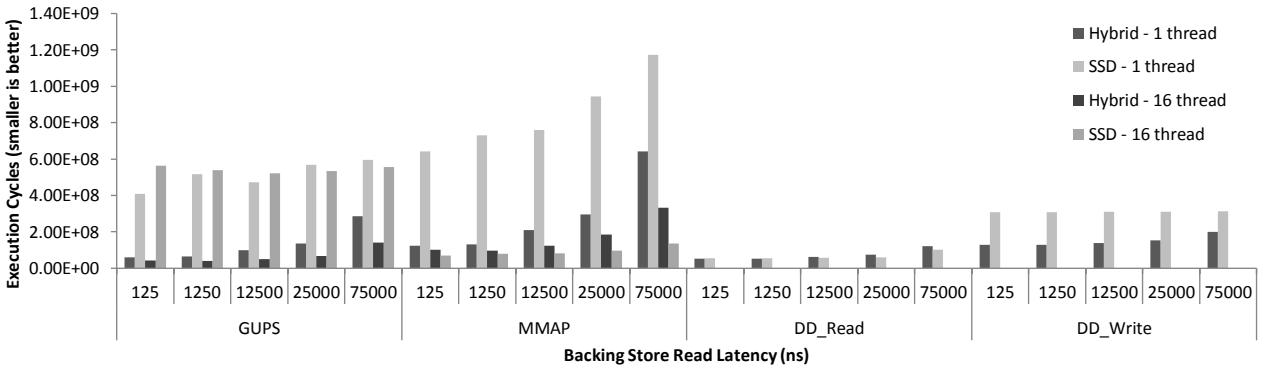

Figure 4: The effect of different backing store read latencies on System Performance for the targeted benchmarks. The Y-Axis is the execution time in cycles, so smaller is better. DD_Read and DD_Write are single threaded only.

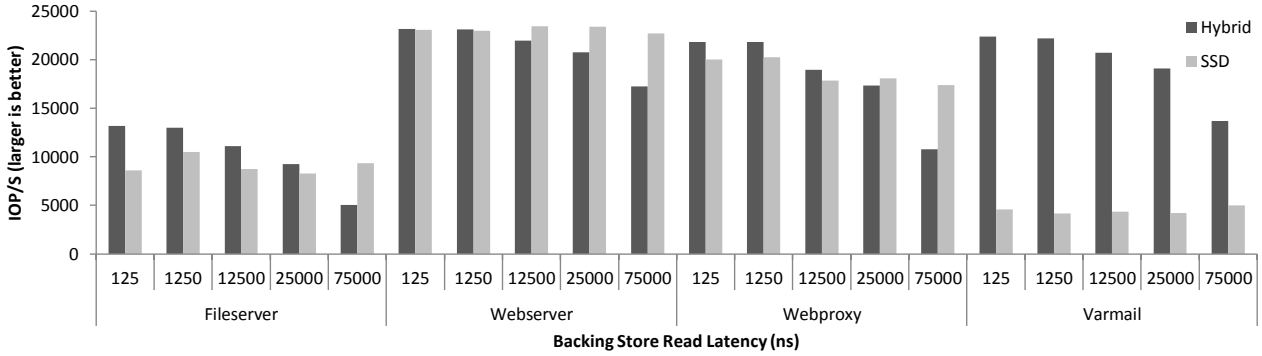

Figure 5: The effect of different backing store read latencies on System Performance for the representative file system benchmarks. The Y-Axis is IO operations per second, so larger is better.

page faults and cache misses there is a slight edge for the Hybrid system in the DD_Write benchmark due to the additional page fill overhead suffered by the storage system.

The results in Figure 5, however, tell a slightly different story. In these representative workloads we see the SSD system pulling ahead in quite a few benchmarks. At MLC latencies the Hybrid system is only better in the Varmail workload. This indicates that many realistic workloads benefit from the prefetching and associativity advantages that enabled the SSD system to outperform the Hybrid system in the DD_Read workload. Also, even though DD_Read is only one of three target workloads, this result demonstrates how important this workload is for capturing the more easily prefetched and cached address streams of many common storage workloads.

An important feature of this experiment is that we can see the crossover points where the performance of the storage system surpasses the performance of the memory architecture. The crossover points are really only visible in the representative workloads as the targeted workloads tend to always favor either the SSD or the Hybrid system. Among those workloads that have a crossover point we see that at SLC latencies the Hybrid and SSD architectures are very close in terms of performance while the storage system performs much better at MLC latencies in almost all cases. These results show that technologies such as SLC NAND Flash could be used more aggressively as the backing store technology of some Hybrid systems. SLC NAND backed Hybrid systems break even with SSD systems in most cases and achieve significant performance gains in others.

4.2 The Software Overhead of the Storage System

From the latency sweep results we can clearly see that at lower backing store latencies there is some overhead in the storage system that is negatively affecting its performance. In order to characterize the nature of the storage overhead we need to quantify the contribution of the software portion of a storage access. We accomplished this by instrumenting our MMAP benchmark to log when a request began and ended at the application level. We created the instrumentation by using x86 rdtsc instructions that ran immediately before and after each access. We also implemented code in the SSD host interface to record when accesses began and ended at the hardware level. The hardware time includes the host interface time, the time it took to process the access in the SSD controller, and the time it took to perform the DMA. Since the raw time from the software level logs include the hardware time, we must subtract the hardware time from those raw values to compute the actual time

Table 3: Software and Hardware Access Time

	Total Time (ns)		Hardware Time (ns)		Software Time (ns)	
	Mean	Stdev	Mean	Stdev	Mean	Percent Software Delay
SLC Latency	85360.93	33837.39	38900.67	6689.59	46460.26	54.43
MLC Latency	162148.72	61060.33	88750.12	13016.83	73398.60	45.27

spent processing the access in software. To provide an accurate picture of the associated delays, 10000 accesses were measured and their delays were averaged. Also, the same analysis was performed with a backing store that had a read latency roughly equivalent to SLC NAND Flash (25000 ns) and MLC NAND Flash (75000 ns). The resulting values are presented in Table 3.

From these results it is clear that the software level of a storage system access represents a significant portion of the total delay. As the latency of the backing store increases, the relative percentage of the delay that is due to software decreases because it remains relatively constant. However, even at MLC NAND Flash latencies the software delays represent almost half of the time it takes to access the backing store.

It is also worth pointing out that the standard deviation of the total delay is roughly five times the standard deviation of the hardware delay. This indicates that the software layer introduces a significant amount of nondeterminism to the system. This same effect can clearly be seen in our other results where the traditional SSD approach exhibits considerably more nondeterminism than the Hybrid architecture.

4.3 The Effect of Random Accesses

The impact of high miss rate applications on the performance of both Hybrid and SSD systems led to questions about the importance of miss rate on the overall performance of both systems. As a result, the following experiment was developed to gauge the effect of varying degrees of randomness in the memory access pattern on both the Hybrid and SSD systems when they are swapping. For this experiment, each access has a probability of being either sequential or random and by changing the probability, we can adjust the degree of randomness in the workload. For this experiment we held the latency constant at 25000ns for all data points.

In Figure 6, we can see that only 10 percent random reads introduces significant performance degradation compared to the purely sequential case. This suggests that even programs which are largely sequential can benefit from the Hybrid architecture's efficient handling of random reads. In addition, the multithreaded version of the workload appears to introduce additional randomness that further degrades performance for the SSD versions of the system when the workload is relatively sequential. This is because the relatively sequential threads interfere with one another and produce memory traffic that is largely random. However, as the randomness of the workload increases the performance benefits of the multithreaded version begin to outweigh the randomness introduced by the multithreading interference. This transition appears to occur at around 70 percent random access. However, the considerable involvement of the OS in the SSD version of the system introduces some nondeterminism to the results. The Hybrid architecture, on the other hand, is much more deterministic due to the reduced dependence on the OS. The Hybrid architecture also benefits from the additional threads for all levels of random access. As the degree of randomness increases, the performance boost provided by the Hybrid architecture increases from around 2X at the 10 percent randomness point to 5x when the workload is totally random. Furthermore, the Hybrid architecture handles the purely sequential workload almost as well as the SSD despite lacking many of the optimizations that have been developed for sequential disk accesses.

It is important to note though that the performance differences seen in this experiment do not reflect the results from the representative workloads. In those workloads the SSD performed better or equal in most cases at 25000ns but in this experiment it is surpassed by the Hybrid organization at just 10 percent random reads. To further investigate these results we next analyze the aspects of the cache that tend to affect the miss rate in an effort to determine if some other factor besides randomness is contributing to the representative workload performance of the SSD.

4.4 The Effect of Associativity

Associativity is often an effective way to reduce the miss rate of a workload. However, while the page table of the OS is functionally fully associative the Hybrid system was limited to 16 ways of associativity in the prior experiments. To investigate the effects of associativity we swept the degree of associativity available to the Hybrid system from direct mapped to 64 way set associative. For this experiment and the remaining two experiments we set the backing store latency to 17000 ns. This value was selected because it represents a middle value that does not overly favor any particular aspect of the system that has been studied thus far.

The results in Figure 7 show that associativity has a relatively minimal effect on performance. Both MMAP and DD_Read experience a roughly 30% speedup as a result of increasing the associativity from direct mapped to 64 way. It is interesting to note that the multi-threaded workloads appear to benefit more from the added associativity. This indicates that the different threads are creating some set contention in the lower associativity cases. However, the multi-threaded MMAP runs exhibit some additional nondeterminism as a result of the OS scheduler and so it is difficult to accurately gauge the precise speedup. Finally, the direct mapped and 2 way associative runs of DD_Write failed to complete in all attempts. The additional write pressure on the backing store created by the lack of associativity seems to have overwhelmed the backing store resulting in extremely long access latencies.

Given the 30% improvement seen in DD_Read it seemed that the superior associativity of the storage implementation might have been a contributing factor to its performance. So, we repeated this same experiment with the representative workloads. However, we found that only fileserver exhibited any sensitivity to associativity. Moreover, fileserver was slowed by reducing the associativity to direct mapped but did not receive any boost from increasing associativity beyond 4 way. Therefore, associativity on its own could not

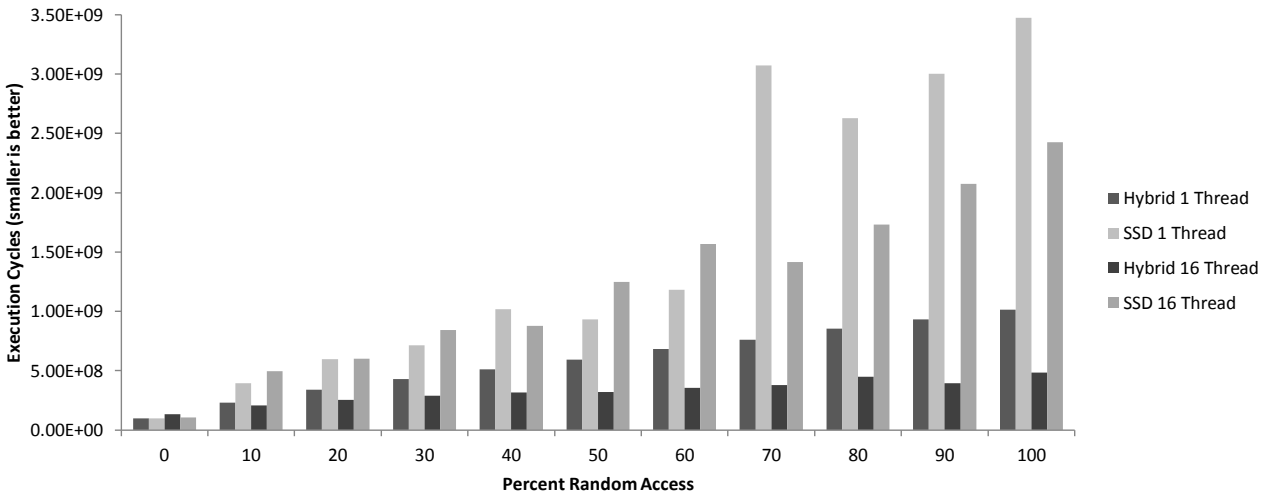

Figure 6: The effects on system performance from different percentages of random access for the GUPS benchmark. The Y-Axis is the execution time in cycles, so smaller is better.

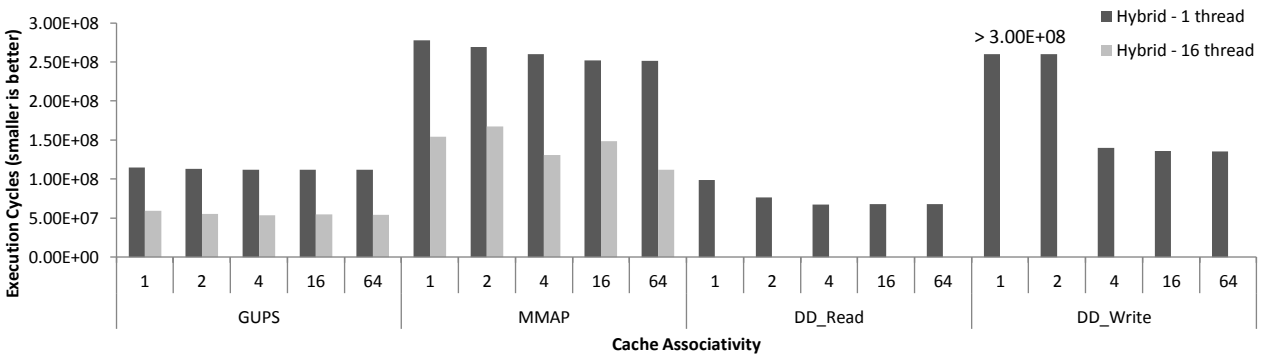

Figure 7: The effect of different levels of DRAM cache associativity on the performance of the Hybrid system for the targeted benchmarks. The Y-Axis is the execution time in cycles, so smaller is better.

be the source of the SSD's performance advantage.

4.5 The Effect of Cache Size

Another possible source of misses could come from the cache being too small for the working set. However, this can be mediated by clever cache management schemes that retain only the most useful data thereby maximizing the available cache space. By varying the size of the cache we can expose the cache manage scheme and determine how effective it is at utilizing the space in the cache.

We can see right away in Figure 8 that the cache management schemes in the SSD and Hybrid approaches are doing something very different. The size of the cache has almost no effect on the Hybrid system while the SSD benefits greatly from increasing the cache size. This suggests that the SSD is prefetching very aggressively as the randomness of GUPS should limit the effectiveness of most prefetches. We confirmed this by checking the size of the requests being issued by the OS and noted that they can grow to be quite large (on the order of megabytes). By prefetching so much the SSD is benefiting from a âĂIJbirthday attackâĂİ-like effect, in which prefetching the pages from the backing store is helping some future accesses with a certain probability. However, this prefetching results in greatly reduced performance when the cache is small because the replacement policy is unable to prevent the cache from being polluted by prefetched data.

4.6 The Effect of Cache Concurrency

In addition to potentially polluting the cache, aggressive prefetching to can also place a greater strain on the concurrency of the cache. In order to determine if the available cache concurrency was a bottleneck we performed an experiment where we swept the number of channels in the DRAM cache from 1 to 16.

From the results in Figure 9 we can see that increasing the available concurrency in the cache does not significantly speed up most of the SSD runs. So, we can be reasonably sure that though the SSD is exerting a lot of prefetch pressure on the cache, that pressure is not interfering with normal requests in most cases. However, the single threaded GUPS SSD benchmark shows a nearly 2x speedup due to increasing the concurrency. This suggests that cache pressure

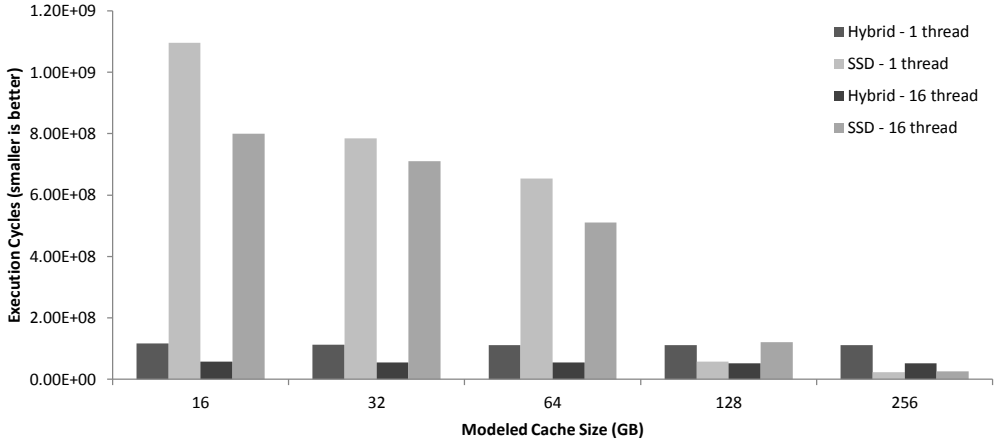

Figure 8: The effect of cache size on System Performance for the GUPS targeted benchmark. The Y-Axis is the execution time in cycles, so smaller is better.

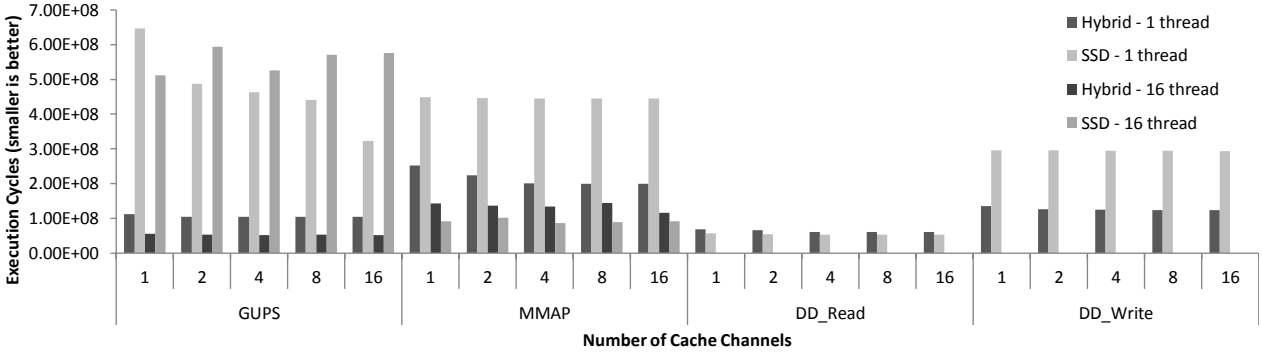

Figure 9: The effect of cache concurrency on system performance for the targeted benchmarks. The Y-Axis is the execution time in cycles, so smaller is better.

is a major bottleneck for the SSD in the single threaded GUPS runs. The ability to switch to other threads while waiting on a DRAM cache accesses appears to negate this effect in the multithreaded GUPS runs. Interestingly, some of the Hybrid runs also show some improvement with increased cache concurrency suggesting that the Hybrid system is using the cache less efficiently in certain workloads. In particular, the MMAP workload shows a considerable speed up. This could account for some of the performance difference between the SSD and Hybrid systems that was noted earlier for this workload.

5. RELATED WORK

A number of projects exist that have modified the software interface to solid state drives by polling the disk controller rather than utilizing an IO interrupt to indicate when a request completes [31] [13] [8]. This is similar to the Hybrid architecture in our study in that the memory controller and the application poll when a request is outstanding to the backing store. The key differences are that these designs still utilize the same PCIe interface and basic operating system structures as current PCIe SSD designs. The performance advantages/limitations that IO polling would have compared to the Hybrid architecture approach are an open area of study and hopefully a subject for future work.

Another way to redesign the OS to work with SSDs is to build persistent object stores. These designs require the programmer to determine which objects should be persistent and modify the code to utilize special allocation functions. A section of the DRAM is then allocated as a cache for the persistent objects. These designs require careful management at the user and/or system level to prevent problems such as dangling pointers and deal with allocation, garbage collection, and other management issues. A performance improvement for certain types of workloads is possible with these designs by customizing the caching algorithm for persistent objects to be more efficient than the generic operating system paging mechanism. SSDAlloc [6] builds persistent objects for boosting the performance of flash-based SSDs, particularly the high end PCIe Fusion-IO drives. NV-Heaps [9] is a similar system designed to work with upcoming byte-addressable non-volatile memories such as phase change memory.

Other work describes file system approaches for managing non-volatile memory. One example is a file system for managing Hybrid main memories [22]. This work assumes

that the OS rather than the memory controller handles tasks such as page placement, garbage collection, and wear leveling. Another proposed file system is optimized for byte-addressable and low latency non-volatile memories (e.g. phase change memory) using a technique called short-circuit shadow paging [10].

Over the past few years, a significant amount of work has also been put into designing architectures that can effectively use PCM to replace or reduce that amount of DRAM needed by systems [26] [19] [12]. This body of work anticipates a slow down in the scaling of DRAM and proposes PCM based systems as a way to continue increasing the capacity of main memory to meet demand. Some of the architectures that have been suggested for use with PCM inspired the Hybrid architecture studied in this paper in that they also utilize the DRAM as a cache that is managed by the memory controller [26].

However, these PCM designs were not the first to utilize a Hybrid architecture. In 1994, eNVy was proposed as a way to increase the size of the main memory by pairing a NOR flash backing store with a DRAM cache [30]. This design is actually very similar to both the Hybrid architecture studied in this paper and the Hybrid PCM architectures except that it utilizes NOR flash as its non-volatile backing store technology. In addition, a very similar architecture was also proposed by FlashCache which utilized a small DRAM caching a larger NAND flash system [18]. However, it is engineered to focus on low power consumption and to act as a file system buffer cache for web servers, which means the performance requirements are significantly different than the more general purpose merged storage and memory system. In 2009, a follow-up paper to FlashCache proposed essentially the same design with the same goals using PCM [27].

There have also been several industry solutions that attempt to improve the performance of the storage system [23] [4] [3] [29] [15]. These solutions tend to fall in one of three categories: software acceleration for SSDs, PCIe SSDs, and Non-Volatile DIMMs. Recently, several companies including Oracle have released software to improve the access times to SSDs by treating the SSD differently than a traditional hard disk [23]. Similarly, Samsung recently released a file system for use with its SSDs that takes into account factors such as garbage collection which can affect access latency and performance.

Finally, for several years, companies such as Fusion IO, OCZ and Intel have been producing SSDs that utilize the PCIe bus for communication rather than the traditional SATA bus. The additional channel bandwidth provided by PCIe allows for much better overall system performance by alleviating one of the traditional storage system bottlenecks. The SSD design utilized in this study was based on these products.

6. CONCLUSION

In this paper, we have presented a series of experiments which clearly illustrate the advantages and disadvantages of Hybrid and SSD-based hierarchical memory architectures. These experiments have demonstrated that SSD-based systems perform best when workloads are highly sequential, DRAM cache sizes are large, and backing store technologies are slow. Conversely, the Hybrid-based system performs best when the workload is more random and the backing store is fast. In addition, we have quantified the direct delay introduced by the software during storage system access and have shown that the total cost of a software overhead (file system, task switch, etc.) is a significant component of the storage approachâĂŹs access latency. Finally, we have also shown that slower technologies such as SLC NAND Flash can be successfully utilized in more aggressive Hybrid systems. Overall, this work has shown that there is a clear place for both the Hybrid and the SSD approaches to building hierarchical memory systems. Furthermore, by taking certain specific system and workload attributes into account it is possible to safely decide which approach is best suited to a particular use case.

7. ACKNOWLEDGMENTS

The authors would like to thank Dr. Ishwar Bhati, Dr. Mu-Tien Chang and the reviewers for their valuable input. This research was funded in part by Intel Labs University Research Office and Sandia National Labs.

8. REFERENCES

[1] Filebench. http://www.fsl.cs.sunysb.edu/~vass/filebench.

[2] Gups. http://www.dgate.org/~brg/files/dis/gups.

[3] Fusion IO. http://www.fusionio.com, 2012.

[4] PCI Express OCZ Technology. http://www.ocztechnology.com/products/solid_state_drives/pci-e_solid_state_drives, 2012.

[5] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, , and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the 2008 USENIX Technical Conference (USENIX'08)*, USENIX '08, 2008.

[6] A. Badam and V. S. Pai. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. In *In Proc. 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*, 2011.

[7] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet. Linux block io: Introducing multi-queue ssd access on multi-core systems, 2013.

[8] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 385–395, Washington, DC, USA, 2010. IEEE Computer Society.

[9] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *SIGARCH Comput. Archit. News*, 39(1):105–118, Mar. 2011.

[10] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[11] C. Dirik and B. Jacob. The performance of PC Solid-State Disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system

organization. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 279–289, 2009.

[12] A. P. Ferreira, B. Childers, R. Melhem, D. Mosse, and M. Yousif. Using PCM in Next-generation Embedded Space Applications. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:153–162, 2010.

[13] A. Foong, B. Veal, and F. Hady. Towards SSD-ready Enterprise Platforms. In *1st International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, 2010.

[14] J. Gantz and D. Reinsel. The digital universe decade-are you ready. *External Publication of IDC (Analyse the Future) Information and Data*, pages 1–16, 2010.

[15] T. Hardware. Samsung intros nand flash-friendly file system. http://www.tomshardware.com/news/NAND-Flash-Flash-Friendly-File-System-F2FS-Jaegeuk-Kim, 18229.html, 2012.

[16] C.-C. Huang and V. Nagarajan. Atcache: Reducing dram cache latency via a small sram tag cache. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 51–60, New York, NY, USA, 2014. ACM.

[17] B. Jacob. The memory system: you can't avoid it, you can't ignore it, you can't fake it. *Synthesis Lectures on Computer Architecture*, 4(1):1–77, 2009.

[18] T. Kgil and T. Mudge. FlashCache: a NAND Flash Memory File Cache for Low Power Web Servers. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded systems*, CASES '06, pages 103–112, New York, NY, USA, 2006. ACM.

[19] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable Dram Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM.

[20] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[21] G. H. Loh and M. D. Hill. Efficiently enabling conventional block sizes for very large die-stacked dram caches. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 454–464, New York, NY, USA, 2011. ACM.

[22] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating System Support for NVM+DRAM Hybrid Main Memory. In *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS'09, pages 14–14, Berkeley, CA, USA, 2009. USENIX Association.

[23] Oracle. Achieving new levels of datacenter performance and efficiency with software-optimized flash storage. http://www.oracle.com/us/products/servers-storage/storage/tape-storage/software-optimized-flash-192597.pdf, 2010.

[24] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSSx86: A Full System Simulator for x86 CPUs. In *Design Automation Conference 2011 (DAC'11)*, 2011.

[25] M. K. Qureshi and G. H. Loh. Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 235–246, Washington, DC, USA, 2012. IEEE Computer Society.

[26] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable High Performance Main Memory System Using Phase-Change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[27] D. Roberts, T. Kgil, and T. Mudge. Using non-volatile memory to save energy in servers. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 743–748, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[28] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *Computer Architecture Letters*, 10(1):16 –19, Jan.-June 2011.

[29] Spansion. Using spansion ecoram to improve tco and power consumption in internet data centers. http://www.spansion.com/jp/About/Documents/spansion_ecoram_whitepaper_0608.pdf, 2008.

[30] M. Wu and W. Zwaenepoel. envy: A non-volatile, main memory storage system. In *ASPLOS*, pages 86–97, 1994.

[31] J. Yang, D. B. Minturn, and F. Hady. When poll is better than interrupt. In *FAST'12: Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.