A Journaled, NAND-Flash Main-Memory System

Technical Report UMD-SCA-2010-12-01 — December 2010, updated 2014

Bruce Jacob, Ishwar Bhati, Mu-Tien Chang, Paul Rosenfeld, Jim Stevens, Paul Tschirhart

Electrical & Computer Engineering Dept
University of Maryland, College Park
blj@umd.edu ◆ www.ece.umd.edu/~blj

Zeshan Chishti, Shih-Lien Lu

Intel Corporation
Hillsboro, Oregon
www.intel.com

James Ang, Dave Resnick, Arun Rodrigues

Sandia National Labs
Albuquerque, New Mexico
www.sandia.gov

Abstract

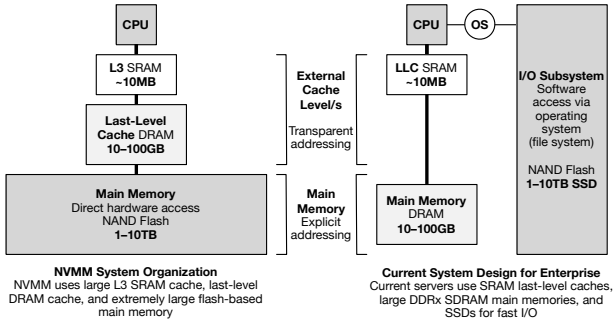
We present a memory-system architecture in which NAND flash is used as a byte-addressable main memory, and DRAM as a cache front-end for the flash. NAND flash has long been considered far too slow to be used in this way, yet we show that, with a large cache in front of it, NAND can come within a factor of two of DRAM's performance. The memory-system architecture provides several features desirable in today's large-scale systems, including built-in checkpointing via journaled virtual memory, extremely large solid-state capacity (at least a terabyte of main memory per CPU socket), cost-per-bit approaching that of NAND flash, and performance approaching that of pure DRAM. It is also non-volatile.

Introduction

Today's main memory systems for datacenters, enterprise computing systems, and supercomputers fail to provide high per-socket capacity [Ganesh *et al.* 2007; Cooper-Balis *et al.* 2012], except at extremely high price points (for example, factors of 10–100x the cost/bit of consumer main-memory systems) [Stokes 2008]. The reason is that our choice of technology for today's main memory systems—i.e., DRAM, which we have used as a main-memory technology since the 1970s [Jacob *et al.* 2007]—can no longer keep up with our needs for density and price per bit. Main memory systems have always been built from the cheapest, densest, lowest-power memory technology available, and DRAM is no longer the cheapest, the densest, nor the lowest-power storage technology out there. It is now time for DRAM to go the way that SRAM went, many years ago: move out of the way and allow a cheaper, slower, denser storage technology to be used as main memory … and instead become a cache.

This inflection point has happened before, in the context of SRAM yielding to DRAM. There was once a time that SRAM was the storage technology of choice for all main memories [Tomasulo 1967; Thornton 1970; Kidder 1981]. However, once DRAM hit volume production in the 1970s and 80s, it supplanted SRAM as a main memory technology because it was cheaper, and it was denser. It also happened to be lower power, but that was not the primary consideration of the day. At the time, it was recognized that DRAM was much slower than SRAM, but it was only at the supercomputer level (for instance the Cray X-MP in the 1980s and its follow-on, the Cray Y-MP, in the 1990s) that could one afford to build ever-larger main memories out of SRAM—the reasoning for moving to DRAM was that an appropriately designed memory hierarchy, built of DRAM as main memory and SRAM as a cache, would approach the performance of SRAM, at the price-per-bit of DRAM [Mashey 1999]. Today it is quite clear that, were one to build an entire multi-gigabyte main memory out of SRAM instead of DRAM, one could improve the performance of almost any computer system by up to an order of magnitude—but this option is not even considered, because to build that system would be prohibitively expensive.

It is now time to revisit the same design choice in the context of modern technologies and modern systems. For reasons both technical and economic, we can no longer afford to build ever-larger main memory systems out of DRAM. Flash memory, on the other hand, is significantly cheaper and denser than DRAM and therefore should take its place. While it is true that flash is significantly *slower* than DRAM, one can afford to build much *larger* main memories out of flash than out of DRAM, and we will show that an appropriately designed memory hierarchy, built of flash as main memory and DRAM as a cache, will approach the performance of DRAM, at the price-per-bit of flash.



NVMM Organization versus a Typical Enterprise-Class Organization. *NVMM uses the same storage technologies as in present-day enterprise systems, in a slightly different organization. DRAM is used as a large last-level cache, and the main memory is NAND flash.*

This paper introduces Non-Volatile Main Memory (NVMM), pictured above. NVMM is a new main-memory architecture for large-scale computing systems, one that is specifically designed to address the weaknesses described previously. In particular, it provides the following features:

non-volatility: The bulk of the storage is comprised of NAND flash, and in this organization DRAM is used only as a cache, not as main memory. Furthermore, the flash is journaled, which means that operations such as checkpoint/restore are already built into the system.

1+ terabytes of storage per socket: SSDs and DRAM DIMMs have roughly the same form factor (several square

inches of PCB surface area), and terabyte SSDs are now commonplace.

performance approaching that of DRAM: DRAM is used as a cache to the flash system.

price-per-bit approaching that of NAND: Flash is currently well under $0.50 per gigabyte; DDR3 SDRAM is currently just over $10 per gigabyte [Newegg 2014].

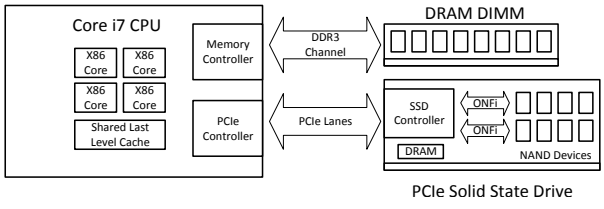
Even today, one can build an easily affordable main memory system with a terabyte or more of NAND storage per CPU socket (which would be extremely expensive were one to use DRAM), and our cycle-accurate, full-system experiments show that this can be done at a *performance* point that lies within a factor of two of DRAM.

Background and Related Work

The most relevant comparisons are to existing computer systems such as enterprise computing systems that use SSD architectures as their back-end I/O subsystem, and other studies involving non-volatile main memories.

Solid-State Disk Architectures and Operation

A block diagram of a system using typical flash-based solid state drive is shown in the figure below.

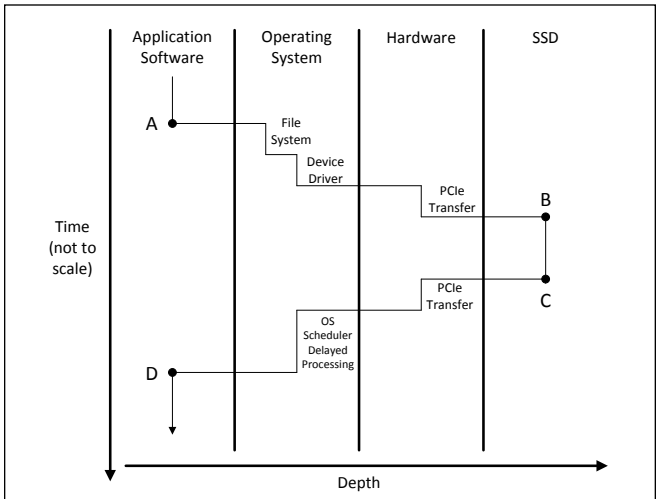


System Design for SSD. *Typical systems today (e.g. based on Intel's i7) use DRAM as main memory and an SSD with a SATA or PCIe interface, both of which have controllers integrated onto the CPU.*

The system consists of three main components: host interface, an SSD controller, and a set of NAND flash devices. The host interface is typically SATA or PCIe—for instance, the high performance SSDs produced by Fusion IO [Fusion IO 2012], OCZ [OCZ Technology 2012], and Intel [Intel 2012] all utilize between 4 and 16 PCIe lanes. Due to the design of currently available flash controllers, some of these drives still utilize sets of SATA SSD controllers internally in a parallel RAID 0-style configuration to achieve higher bandwidth; the NVM Express standard will enable pure PCIe SSD controllers in future products. The SSD controller performs tasks such as memory mapping, garbage collection, wear leveling, error correction, and access scheduling. It also typically has a small amount of SRAM or DRAM to cache metadata and to buffer writes [Marvell 2012].

To achieve high throughput, SSDs leverage multiple NAND flash devices organized into parallel channels with multiple devices per channel. Internally, the NAND devices are organized into planes, blocks, and pages. *Planes* are functionally independent units that allow for concurrent operations on the device. Each plane has a set of registers that allow for interleaved accesses and provide access to a number of *blocks*, the physical granularity at which erase operations occur. Each block consists of multiple *pages*, which are the physical granularity at which read and write operations occur.

Early NAND flash chips used an asynchronous interface that ran at speeds in the tens of MB/s. These early interfaces were acceptable for many years, as the access latency of flash was still faster than other external storage media of the time, and the bandwidth was not the bottleneck in the applications that utilized flash [Dirik & Jacob 2009]. However, as flash has been used increasingly in high-performance systems, its transfer times matter more. Noting that the array of flash cells within the chip are actually capable of producing data at a rate of 330 MB/s without any modifications [Cooke 2009], manufacturers have developed synchronous DDR standards for NAND flash's external interface—for instance, the latest ONFI standard is capable of bandwidths up to 400 MB/s [Intel *et al.* 2013].

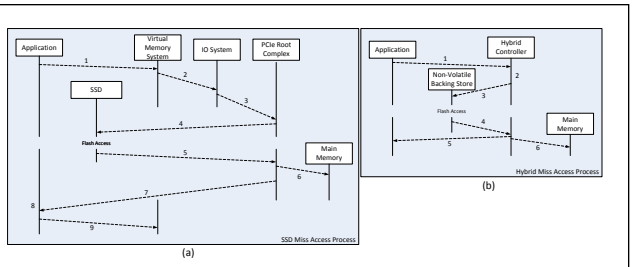


Hardware and Software Involved in Servicing SSD Request. *A single I/O request moves through multiple layers of both software and hardware.*

As shown in the figure above, there are many intermediate software and hardware layers involved in an SSD access. The software side on a Linux-based system includes the virtual memory system, the virtual file system, the specific file system for the partition that holds the data (e.g. NTFS or ext3), the block device driver for the disk, and the device driver for the host interface such as the Advanced Host Interface Controller (AHCI) for Serial ATA (SATA) drives [Bovet & Cesati 2005]. At the hardware level, the interfaces involved include the host interface to the drive, the direct memory access (DMA) engine, and the SSD internals. When the host interface is SATA, it resides on the southbridge, which means that the request must first cross the Intel Direct Media Interface (DMI) or equivalent before crossing the SATA interface. However, higher performance systems (and our model for this paper) assumes the pure PCIe 3.0 NVM Express interface, using 16 lanes, which brings the performance to an enterprise-class solid state drive. The DMA engine accesses memory on behalf of the disk controller without requiring the CPU to perform any actions. A DMA read operation must happen before an SSD write, and a DMA write operation must happen after an SSD read.

In terms of memory-system performance, the metric that NVMM targets, an access delay to a solid state drive begins

when the user application issues a request for data that triggers a page fault; it ends when the operating system returns control to the user application after the request has completed. At the hardware level, the SSD controller receives an access for a particular address and then later the controller raises an interrupt request (IRQ) on the CPU to tell the operating system the data is ready. A typical access to an SSD, behavior that our experiments capture in its entirety, is shown in the figure below (figure (a)).



Access to SSDs and NVMM. *Steps to access an SSD are shown on the left; steps involved in an NVMM access are shown on the right.*

In Step 1, the application generates a request to the virtual memory system. Step 2 represents 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 the request 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 uses that logical block address to determine the physical location of the virtual page associated with that address and issues a request to the device or devices that contain that virtual page (in enterprise SSDs, page data is typically striped in a RAID manner across multiple flash devices to increase both performance and reliability). For the virtual page that is evicted from the main memory, the SSD allocates a new physical page slot 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, in Step 6. Once the write is complete, the PCIe root complex raises an interrupt alerting the OS scheduler that an application's 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.

In NVMM, the flash-based backing store is presented to the OS virtual memory manager as the entire physical memory address space—i.e., it appears to the OS that the computer's main memory is the size of the flash backing store (terabytes instead of gigabytes). The actual DRAM physical address space is hidden from the OS and is managed by the memory controller as a cache. Together, the flash-based backing store and DRAM cache form a hybrid memory that is NVMM. Accesses to NVMM have the same granularity as a typical main memory system today: i.e., 64 bytes per access. The cache lines in the DRAM cache have a much larger granularity to match the read/write access granularity NAND flash, typically 4KB, 8KB, or 16KB.

The previous figure shows the access process for NVMM (figure (b)). In Step 1 the application generates a request to the virtual memory system. In Step 2, NVMM's "hybrid" memory controller performs a lookup 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 the figure). When an access misses the DRAM cache, the controller selects a page to evict from the DRAM cache and performs a write-back to the flash subsystem if the page is dirty, Step 3. The missed page is then read in from the flash backing store and placed in the DRAM, Step 4. This involves translating the address for the request into the physical address of the data in the backing store (e.g., flash channel, device, plane, row, and page). A read command is issued to the appropriate flash device, and the resulting data is returned. The controller can also prefetch additional pages into the DRAM or write back cold dirty pages preemptively, similar to how current virtual memory systems work, to further improve read performance. Once the data has been received, the controller passes the requested data at a 64B granularity to the application, in Step 5. Finally, during Step 6 the page read from the flash subsystem is written into the previously emptied DRAM cache block.

SSD Optimizations and Non-Volatile Main Memories

A number of similar 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 [Yang *et al.* 2012; Foong *et al.* 2010; Caulfield *et al.* 2010]. This is similar to our design in that it eliminates interrupts, but it still requires polling on the CPU side.

Another way to redesign the OS to work with SSDs is to build persistent object stores. These designs require careful management at the user and/or system level to prevent problems such as dangling pointers and to deal with allocation, garbage collection, and other issues. SSDAlloc [Badam & Pai 2011] builds persistent objects for boosting the performance of flash-based SSDs, particularly the high end PCIe Fusion-IO drives [Fusion IO 2012]. NV-Heaps [Coburn *et al.* 2011] 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 [Mogul *et al.* 2009]. 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 [Condit *et al.* 2009].

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 the amount of DRAM needed by systems [Qureshi *et al.* 2009; Lee *et al.* 2009; Ferreira *et al.* 2010]. Some of the architectures that have been suggested for use with PCM are similar to our storage system design in that they also utilize the DRAM as a cache that is managed by the memory controller [Qureshi *et al.* 2009]. However, our work differs from these approaches in that our design only utilizes existing technologies and does not assume a low-latency DRAM replacement (PCM, unlike flash, has access times comparable to DRAM).

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 [Wu & Zwaenepoel 1994]. This design is actually very similar to both our hybrid architecture and the hybrid PCM architectures, except that it utilizes NOR flash as its non-volatile backing store technology, which at the time had access time extremely close to that of DRAM. In addition, a very similar architecture was also proposed by FlashCache which utilized a small DRAM caching a larger NAND flash system [Kgil & Mudge 2006]. 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 in our system. In 2009, a follow-up paper to FlashCache proposed essentially the same design with the same goals using PCM [Roberts *et al.* 2009].

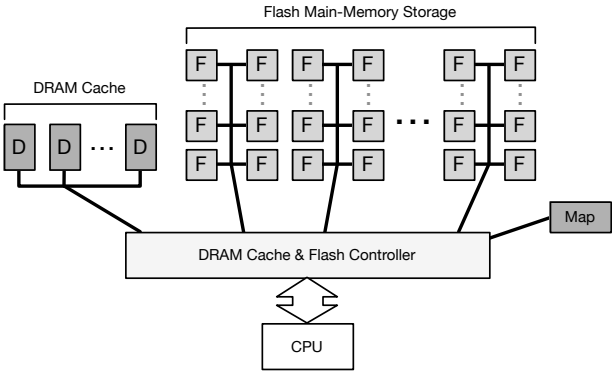
There have also been several industry solutions that address the problem of the backing store bottleneck [Oracle 2010; OCZ 2012; Fusion IO 2012; Spansion 2008; Tom's Hardware 2012]. These solutions tend to fall in one of three categories: software acceleration for SSDs, PCIe SSDs, and Non-Volatile DIMMs. 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 [Oracle 2010]. This approach is similar to ours in that it recognizes that flash should be used as an additional storage system tier between the DRAM and hard disks. However, our approach consists of hardware and organizational optimizations rather than software optimizations. 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. Our work differs in that it is trying to provide a better interface to access the flash for main memory, rather than improving just the file system.

For several years, companies such as Fusion IO [Fusion IO 2012], OCZ [OCZ 2012], and Intel [Intel 2012] have been producing SSDs that utilize the PCIe bus for communication rather than the traditional SATA bus. This additional channel bandwidth allows for much better overall system performance by alleviating one of the traditional storage system bottlenecks. Our solution draws upon these designs in that it also provides considerable bandwidth to the flash in an effort to eliminate the bandwidth bottleneck between the CPU and the backing store.

Finally, in 2008 Spansion proposed EcoRAM which was a flash based DRAM replacement [Spansion 2008; InsideHPC 2009]. Like our solution, EcoRAM allowed the flash to interface directly with a special memory controller over the fast channel. However, EcoRAM utilized non-standard proprietary flash parts to construct its DIMMs and it was meant to be pin-compatible with existing DRAM-based memory channels.

Nonvolatile Main Memory System Architecture

As shown in the figure below, NVMM uses a DRAM cache, comprised entirely of DRAM (tags are held in DRAM, not in SRAM), and the main memory, comprised of a large number of flash channels—each of which contains numerous independent, concurrently operative banks. The controller acts as the flash translation layer [Dirik & Jacob 2009] for the collection of flash devices, and it uses a dedicated mapping block to hold the translation information for the flash storage while running—this mapping information is in effect the system's virtual page table. Just as in SSDs, the mapping information is kept permanently in flash and is cached in a dedicated DRAM while the system is running.



NVMM Organization. *The CPU connects through a high-bandwidth interface to the NVMM hybrid DRAM/flash controller—this device controls both a large, last-level cache made from DRAM, and the flash subsystem. The NVMM controller maintains the flash mapping information in a dedicated DRAM while operating. When the system is powered down, the mapping information is stored into a dedicated flash location.*

Also just as is done in an SSD, NVMM extends its effective write lifetime by spreading writes out across numerous flash chips. As individual pages wear out, they are removed from the system (marked by the flash controller as bad), and the usable storage per flash chip decreases. Pages within a flash device obey a distribution curve with respect to their write lifetimes—some pages wear out quickly, while others can withstand many times the number of writes before they wear out [Micron 2014]. With a DRAM cache of 32GB and a moderate to light application load, a flash system comprised of but a single 8Gb device would lose half its storage capacity to the removal of bad pages in just under two days and would wear out completely in three. Thus, a 1TB flash system comprised of 1,000 8Gb devices (or an equivalent amount of storage in a denser technology point) would lose half its capacity in two to three years and would wear out completely in four to five.

The DRAM cache uses blocks that are very large, to accommodate the large pages used in NAND flash. It is also highly banked, using multiple DRAM channels, each with multiple ranks, so as to provide high sustained bandwidth for requests—both requests from the client processor and requests to fill cache blocks with data arriving from the (also highly banked and multi-channel) flash subsystem.

Every logical flash page in the address space of the non-volatile memory is mapped into a cache set in the DRAM system using an LRU replacement policy. The tag store for the cache is located in the DRAM subsystem connected to the controller. The controller also contains a small TLB-like memory to cache mappings currently in use, and in our experiments we simulated the servicing that is required when this cache experiences a miss.

As indicated in the figure, the non-volatile subsystem is comprised of numerous 8-bit ONFI channels (plus command signals), each with multiple volumes (logically equivalent to DRAM ranks). Flash devices are organized into packages, dies and planes. Packages are the organization level that is

connected to the 8 bit interface of the device. That interface is then shared by one or more dies that are internal to the package. Those dies are in turn made up of one or planes, and the planes of the flash device actually perform the access operations. To enable better performance, the planes on most flash devices feature two registers which allow for the interleaving of reads and writes. One register can contain incoming read or write data while the other holds the data currently being used by the plane. In this way the transfer time of the 8 bit flash interface can be somewhat hidden. To take advantage of these interleaving registers, the controller needs to schedule operations appropriately. The flash controller in NVMM accomplishes this by giving commands priority over return data on the package interface. This ensures that a plane can begin working on its next access while simultaneously sending back the data from its last access. Alternatively, the return data using the interface would prevent the command from being sent, and the plane would sit idle during the data transmission.

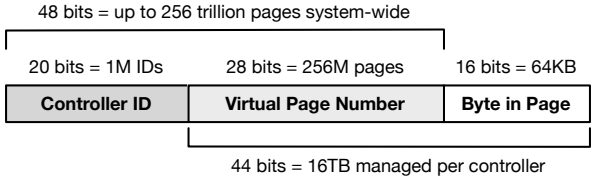
The I/O scheduler of the OS uses several scheduling algorithms to prioritize certain accesses over others, to maximize performance while maintaining fairness between threads. In Linux, these algorithms include *completely fair queuing* (the default), *deadline*, *first come first serve*, and *anticipatory*. The SSD controller then handles the scheduling for the addresses via the Native Command Queuing protocol, which enables the OS to send multiple outstanding requests to the SSD. The scheduling algorithms used by the SSD controller attempt to balance the concerns for high throughput, efficient request merging, load balancing among individual flash devices, wearout, and low latency reads.

In addition, to fully utilize the die parallelism of the backing store, the backing store flash controller has two layers of queues: the flash translation layer (FTL) queue and the die queues. The flash translation layer queue holds incoming accesses until the FTL is able to convert the flash logical address into the flash physical address. The die queues are then used to manage flow control at the die level. The bulk of on chip memory in the controller is devoted to the die queues with only a small amount devoted to the FTL master queue. This is because, relative to normal flash operations, the translation step incurs a very low latency. Also, because the FTL queue is used to feed commands to many flash devices, it is a potential source of delay for the entire system. If the command at the head of the FTL queue cannot be added to its appropriate die queue, then no other commands in the FTL queue can proceed until a space has opened up for that particular die. Allowing for longer die queues reduces the probability that this event will occur. Queue reordering could also be used to address the queue delay problem by allowing commands to jump past the command which cannot be currently accommodated in the appropriate die queue. However, this is only useful if enough commands are being issued to just one die. In most situations queues of only a few entries deep are enough to prevent most queuing delays. In this work, most of the workloads did not generate enough traffic to fill the die queues.

Software Interface

The main memory system is non-volatile and journaled. Flash memories do not allow write-in-place, and so to over-write a page one must actually write the new values to a new page. Thus, the previously written values are held in a flash device until explicitly deleted—this is the way that all flash devices work. NVMM exploits this behavior by retaining the most recently written values in a journal, preferring to discard the oldest values first, instead of immediately marking the old page as invalid and deleting its block as soon as possible.

The system exports its address space as both a physical space (using flash page numbers) and as a virtual space (using byte-addressable addresses). Thus, a system can choose to use either organization, as best suits the application software. This means that software can be written to use a 64-bit virtual address space that matches exactly the addresses used by NVMM to keep track of its pages. The following figure illustrates the address format, indicating its role in multiprocessor systems. Note that the bottom page-offset bits are only used in the access of the DRAM cache and are thus ignored when the controller is accessing the flash devices. Two controller ID values are special: all 0s and all 1s, which are interpreted to mean local addresses—i.e., these addresses are not forwarded on to other controllers.



NVMM Virtual Address. *The NVMM architecture uses a 64-bit address, which allows the address to be used by a CPU's virtual memory system directly, if so desired. The top 20 bits specify a home controller for each page, supporting up to 1M separate controllers. Each controller can manage up to 16TB of virtual storage, in addition to several times that of versioned storage. 64KB pages are used, which is independent of the underlying flash page size.*

This organization allows compilers and operating systems either to use this 64-bit address space directly as a virtual space, i.e. write applications to use these addresses in their load/store instructions, or to use this 64-bit space as a physical space, onto which the virtual addresses are mapped. Moreover, if this space is used directly for virtual addresses, it can either be used as a Single Address Space Operating System organization [Chase *et al.* 1993; 1994], in which software on any CPU can in theory reference directly any data anywhere in the system, or as a set of individual main-memory spaces in which each CPU socket is tied only to its own controller.

NVMM exports a modified load/store interface to application software, including a handful of additional mechanisms to handle non-volatility and journaling. In particular, it implements the following functions:

alloc. Equivalent to malloc() in a Unix system—allows a client to request a page from the system. The client is given an address in return, a pointer to the first byte of the allocated page, or an indication that the allocation failed. The function takes an optional **Controller ID** as an argument, which causes the allocated page to be located on the specified controller. This latter argument is the mechanism used to create address sets that should exhibit sequential consistency, by locating them onto the same controller.

read. Equivalent to a load instruction. Takes an address as an argument and returns a value into the register file. Reading an as-yet-un-**alloc**'ed page is not an error, if the

page is determined by the operating system to be within the thread's address space and readable. If it is, then the page is created, and non-defined values are returned to the requesting thread.

write. Equivalent to a store instruction. Takes an address and a datum as arguments. Writing an as-yet-un-**alloc**'ed page is not an error, if the page is determined by the operating system to be within the thread's address space and writable. If it is, then the page is created, and the specified data is written to it.

delete. Immediately deletes the given flash page from the system, provided the calling application has the correct permissions.

setperms. Sets permissions for the identified page. Among other things, this can be used to indicate that a given temporary flash page should become permanent, or a given permanent flash page should become temporary. Note that, by default, non-permanent pages are garbage-collected upon termination of the creating application. If a page is changed from permanent to temporary, it will be garbage-collected upon termination of the calling application.

sync. Flushes dirty cached data from all pages out to flash. Returns a time token representing the system state [Lamport 1978].

rollback. Takes an argument of a time token received from the **sync** function and restores system state to the indicated point.

The sync/rollback mechanism allows for long-running applications to perform checkpointing without having to explicitly move application data to permanent store, and without having to overwrite data that is already there, as the sync only flushes dirty data from the DRAM cache.

Page Table Organization for NAND Main Memory

When handling the virtual mapping issues for a flash-based main memory system, there are several things that differ dramatically from a traditional DRAM-based main memory. Among them are the following:

- The Virtual Page Number that the flash system exports is smaller than the physical space that backs it up. In other words, traditional virtual memory systems use main memory as a cache for a larger virtual space, so the physical space is smaller than the virtual space. In NVMM, because flash pages cannot be overwritten, and we use this fact to keep previous versions of all main memory data, the physical size is actually *larger* than the virtual space.
- Because the internal organization of the latest flash devices changes over time—in particular, block sizes and page sizes are increasing with newer generations—one must choose a virtual page size that is independent of the underlying physical flash page size. So, in this section, unless otherwise indicated, "page" means a virtual-memory page managed by NVMM.

The NVMM flash controller requires a page table that maps pages from the virtual address space to the physical device space and also keeps track of previously written page data. We use a direct table that is kept in flash but is cached in a dedicated DRAM table while the system is operating. Each entry of the page table contains the following data:

34 bits	Flash Page Mapping (channel, device, block, & starting page)
30 bits	Previous Mapping Index—pointer to entry within page table
32 bits	Bit Vector—Sub-Page Valid Bits (Remapping Indicators)
24 bits	Time Written
8 bits	Page-Level Status & Permissions
16 Bytes	**Total Size**

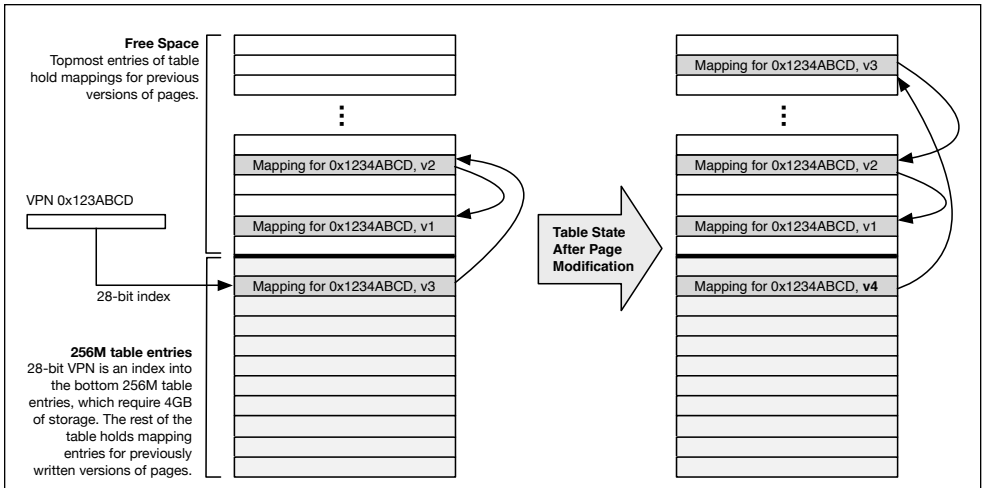
The **Flash Page Mapping** locates the virtual page within the set of physical flash-memory channels. A page must reside in a single flash block, but it need not reside in contiguous pages within that block.

The **Previous Mapping Index** points to the table entry containing the mapping for the previously written page data. The **Time Written** value keeps track of the data's age, for use in garbage-collection schemes.

The **Sub-Page Valid Bits** bit vector allows the data for a 64KB page to be mapped across multiple page versions written at different times. It also allows for pages within the flash block to wear out. This is described in detail later.

The Virtual Page Number is used directly as an index into the table, and the located entry contains the mapping for the most recently written data. As pages are overwritten, the old mapping info is moved to other free locations in the table, maintaining a linked list, and the indexed entry is always the head of the list. The figure below illustrates.

When new data is written to an existing virtual page, in most cases flash memory requires the data written to a new



NVMM Page Table. *NVMM uses a direct-mapped table and stores mappings for previously written pages as well as the most recent. Each VPN is a unique index and references the page's primary entry; if older versions of a page exist, the primary entry points to them. When the primary mapping is overwritten, its old data is copied to an empty entry in the table, and this new entry is linked into the version chain.*

physical page. This will be found on the free list maintained by the flash controller (identical to the operation currently performed by a flash controller in an SSD), and this operation will create new mapping information for the page data. This mapping information must be placed into the table entry for the virtual page. Instead of deleting or overwriting the old mapping information, the NVMM page table keeps the old information in the topmost portion of the table, which cannot be indexed by the virtual page number (which would otherwise expose the old pages directly to application software via normal virtual addresses). When new mapping data is inserted into the table, it goes to the indexed entry, and the previous entry is merely copied to an unused slot in the table. Note that the pointer value in the old entry is still valid even after it is copied. The indexed entry is then updated to point to the previous entry. The Previous Mapping Index is 30 bits, for a maximum table size of 1B entries, meaning that it can hold three previous versions for every single virtual page in the system. The following pseudo-code indicates the steps performed when updating the table on a write-update to an already-mapped block:

```
existing mapping entry is at index VPN

find a new, available entry E in top section of table

copy existing mapping from entry #VPN into entry #E
  i.e., table[E] <- table[VPN]

write "E" into table[VPN].previousMappingIndex

find free page N in flash system
  (where N={chan|volume|LUN|block|page})

write dirty data from 64K page to flash pages N..N+7
  at 8K granularity

table[VPN].flashPageMapping = N

table[VPN].bitVector = set to indicate
  which 8K chunks were written (as data from all
  other chunks are in previously written pages)

table[VPN].timeWritten = now()
```

The Virtual Page Size is 64KB, and pages are written to flash at the granularity dictated by the flash device (the examples use 8KB, but it could also be 4KB or 16KB accordingly). The 8KB sections are called page segments, with eight such segments per 64KB page.

Use of the Bit Vector's Sub-Page Valid Bits

The fact that the page segments are 8KB, within a 64KB page, would suggest a Bit Vector of 8 bits, but the Bit Vector data structure shown above in the page table entry is 32 bits, not 8. This is chosen to support multiple features: it keeps track of data even if there are worn-out pages in the flash block, and it allows for page data to be spread out across multiple flash blocks, so as to avoid re-writing non-dirty data.

If all the data in a Virtual Page is in the cache and is dirty—for example, say this is the first time that the Virtual Page is written—then all 64KB would be written to eight consecutive flash pages in the same flash block, and the first 8 bits of the Bit Vector area would be set to "1," the remaining 24 set to a value of "0" as follows (spaces inserted every 8 bits to show 64KB-sized page groupings):

```
11111111 00000000 00000000 00000000
```

If, however, one or more of the flash pages in the first eight has exceeded its write endurance and is no longer usable, or if it is discovered to be "bad" when it is written, then the flash page cannot be used. In this scenario, the controller will make use of the pages at a distance of eight away instead, or at a distance of 16, or 24. The 32-bit vector allows each 8KB page-segment of the 64KB Virtual Page to lie in one of four different locations in the flash block, starting at the given flash page-number offset within the block (note that the flash page number within the flash block need not be a power of 32). In this scenario, say that there are two bad pages in the initial set of eight—at the positions for page segments 3 and 6—but the other pages are free, valid, and can be written. Assume also that the Starting Flash Page Number is 53—thus, flash pages 56 and 59 within the given flash block are worn out and cannot be written, but pages 53, 54, 55, 57, 58, and 60 can be written. The controller cannot write the data corresponding to page segment 3 to flash page 56, and so it will attempt to place the data at flash-page numbers 64, 72, or 80; assume that page 64 is available, writable, and can accept data. The controller cannot write the data corresponding to page segment 6 to flash page 59, and so it will attempt to place the data at flash-page numbers 67, 75, or 83; assume that page 67 already has data in it and that page 75 is available, writable, and can accept data. Then, once the data is written to the flash pages and the status confirmed by the controller, the Bit Vector is set to the following:

```
11101101 00010000 00000010 00000000
```

Suppose that the next time that data is written to this page and must be written back from the cache, that not all 64KB is "dirty" data—not all of it has been written. Assume, for example, that only page segments 2 and 5 have been modified since the previous write-out to main memory. Only these page-segments should actually be written to flash pages, as writing non-dirty data is logically superfluous (the previous data is still held in the table) and also would cause pages to wear out faster than necessary. In this scenario, a new location in the flash subsystem is chosen, representing a different device and a different block number. Suppose that the Starting Flash Page Number is 17 and that both pages 19 and 22 in this block are valid. The data corresponding to page segment 2 is written to flash page 19; the data corresponding to page segment 5 is written to flash page 22; and the Bit Vector for the new Page Table Entry is set to the following:

```
00100100 00000000 00000000 00000000
```

As flash blocks become fragmented (the pages in the blocks will not be written consecutively when the 64KB virtual pages start to age), the controller can exploit the Bit Vector. In the previous example, the controller would only need to find free writable pages at one of several possible distances from each other within the same flash block:

```
distance 3   00100100 00000000 00000000 00000000
distance 5   00000100 00100000 00000000 00000000
distance 11  00100000 00000100 00000000 00000000
distance 13  00000100 00000000 00100000 00000000
distance 19  00100000 00000000 00000100 00000000
distance 21  00000100 00000000 00000000 00100000
distance 27  00100000 00000000 00000000 00000100
```

When a flash block needs to be reclaimed, in most cases it means that multiple page-segments need to be consolidated. Often, this would entail reading the entire chain of page-table entries, loading the corresponding flash pages, and coalescing

all of the data into a new page. This suggests a natural page-replacement policy in which blocks are freed from the longest chains first. This frees up the most space in one replacement and also improves performance in the future by reducing the average length of linked lists that the controller needs to traverse to find cache-fill data.

Experimental Setup

Our simulation environment is based on the MARSSx86 cycle accurate full system simulator [Patel *et al.* 2011], which consists of 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 benchmarks such as PARSEC or SPEC. The simulator captures both the user-level and kernel-level instructions, enabling the study of all operating system activity.

To model the memory system, we integrated into MARSSx86 the DRAMSim2 simulator [Rosenfeld *et al.* 2011], a cycle accurate, hardware verified DRAM memory-system simulator. We also wrote a detailed cycle-accurate simulator to model the hybrid controller and all non-volatile technologies, including PCM and flash at several different speed grades, as well as a detailed SSD model. To the best of our knowledge, this represents the first full-system SSD simulation—as opposed to trace-based simulation [Agrawal *et al.* 2008; Dirik & Jacob 2009]. 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, which 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 Queueing and allows the SSD-based system to take advantage of hardware parallelism.

The non-volatile memory parameters used in our experiments are shown in the table below. Note that for several of the parameters, minimum values are given, even though in practice, the value exhibits a wide distribution of values (for example, DRAM read and write latencies, which are affected by the scheduling of the controller).

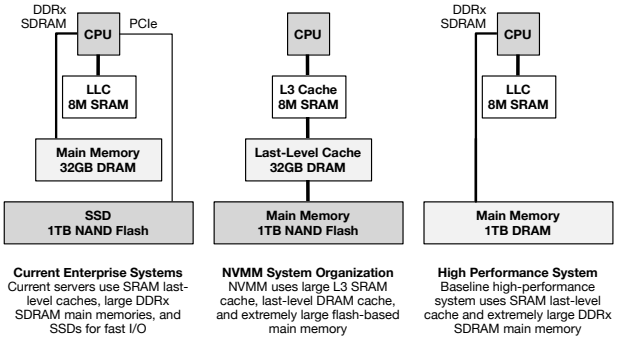
We average across 3 runs for all data values. All IPC results given are user instructions committed divided by total cycles, since the file system running in kernel mode creates overhead for the SSD that does not apply to the hybrid memory.

Technology	Read ns	Write ns	Erase	References
DDR3 SDRAM	50 (min)	50 (min)	n/a	[Micron 2012]
PCM Optimistic	50	150	n/a	[Lee et al. 2009]
PCM Expected	125	1000	n/a	[Qureshi et al. 2012]
SLC / 8 ***	3125	200000	1.5 ms	[Micron 2012; Micheloni et al. 2010]
SLC / 4	6250	200000	1.5 ms	[Micheloni et al. 2010]
SLC / 2	12500	200000	1.5 ms	[Micron 2012]
SLC NAND	25000	200000	1.5 ms	[Micron 2012]
MLC NAND	50000	1200000	3 ms	[Micron 2012]

*** The SLC/x values represent potential read latencies that could theoretically be achieved via design tradeoffs discussed in the cited sources. These were included to provide a range of latencies between SLC, MLC, and PCM.

Performance Studies

In the following studies, we compare NVMM to two different baseline systems, shown below.

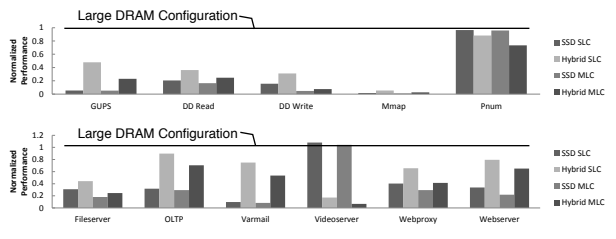


Performance Comparisons. *In these studies, NVMM is compared to two different baselines. The first is the typical design practice today: using DRAM as main memory and SSDs as I/O. The second is the upper bound on achievable performance, simply extending current practice to use a DRAM main memory with as much capacity as in the SSD approach.*

The sizes of the various levels of the hierarchy are given just for comparison, to indicate that this is a meaningful comparison. Each system uses the same sized caches and the same-sized lowest level of the hierarchy. The system on the left represents current SSD-based architectures; the system on the right represents the upper bound on performance—a main memory of DRAM with the same capacity as an SSD (for example, 1TB). The system on the left is achievable today. The system on the right would be prohibitively expensive for most applications.

Performance, Power, and Energy Comparisons

Performance numbers are normalized to that of the high-performance all-DRAM system, as shown in the figure below. We also present the results of NVMM and SSDs using both MLC and SLC NAND flash.



Performance Comparison. *IPC is normalized to the performance of the large-DRAM system, shown as the horizontal lines across the graphs.*

Perhaps the most notable data point is that all of the performance numbers are within the same ballpark. Many of the applications running with flash-based main memory are reasonably close to the performance of the all-DRAM main memory, which represents an organization that is "ideal" in the sense that it is what we would like to build but cannot afford to build. Using SLC flash approaches 95% of ideal system performance for realistic single threaded workloads and 80% of ideal system performance for some realistic multi-threaded workloads. Both NVMM and SSD architectures fail to approach the performance of all-DRAM systems for the file system workloads, which is due to the much higher bandwidth of DDR channels versus the ONFi 3.0 channels (note that this should become a non-issue when devices following the recently released ONFI 4.0 specification appear, as it increases individual device bandwidth up to 800 MB/s). The file-system benchmarks (DD and mmap) both use files that exceed the size of the DRAM system, forcing accesses to go to the non-volatile memory for both NVMM and the SSD configurations. Averaging across all of the benchmarks, NVMM tends to outperform SSD systems by a factor of 1.5 to 2 and comes within a factor of 2 of an all-DRAM system.

Another interesting data point is that the SSD results for the Videoserver benchmark show that an SSD can slightly outperform an all-DRAM system. This is not a bug. It is due to the interplay between MARSSx86 and Linux: Linux detects when it is connected to an SSD as opposed to a spinning disk, and when it does, it activates an extremely aggressive software prefetching mode that, for highly sequential workloads, actually performs better than the baseline hardware prefetcher built into MARSSx86. This should settle the debate once and for all as to whether modern operating systems are mistakenly trying to control SSDs by relying on legacy I/O timing heuristics designed for spinning disks: Linux, at least, clearly is not.

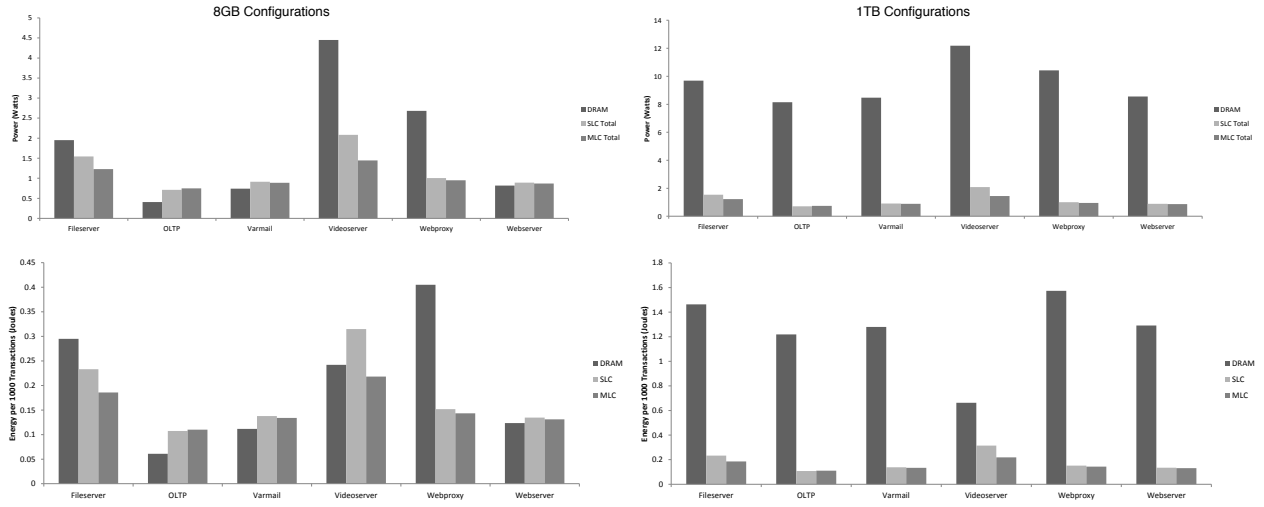
The power and energy for two different configurations are presented below: the graphs represent all-DRAM and hybrid systems of 8GB and 1TB. As the all-DRAM system increases from 8GB to 1TB, the contribution of background power (e.g., refresh power) changes dramatically. At 8GB the background power is relatively small, and the active power of precharging, activating, and reading/writing across the bus dominates. At 1TB, the refresh costs dominate. Note that in these configurations, the unused DRAM is allowed to be put into low-power modes, otherwise the power for a 1TB system would approach 100W.

At 8GB, both the power (top) and energy (bottom) are roughly equivalent for all systems; at 1TB, the flash-based systems have significantly lower power and energy than the equivalent-capacity DRAM system. This is to be expected. Power and energy costs of a main memory system scale with both the bandwidth and the capacity of that memory. Both DRAM and flash scale with the desired read/write bandwidth, but in the graphs below the bandwidth is held constant. As main memory capacity increases, the power and energy costs of DRAM increase with it, as the cost of refresh scales with the number of bits that need refreshing. The power and energy costs of flash, however, do not scale with the capacity, as flash has no background power costs. Clearly, for extremely large main memories, non-volatile technologies are preferable to DRAM.

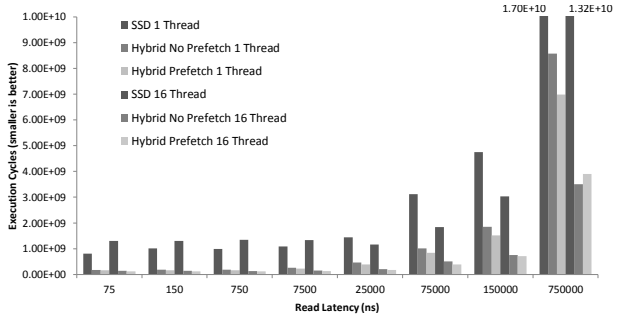
Latency Sensitivity

In this study, we sweep the latency parameter to see how well both NVMM and SSD architectures can tolerate long-latency non-volatile memories. The same latencies are used for both NVMM and the SSD internals. The figure below shows the results for the GUPS benchmark (*Giga-Updates Per Second*: the benchmark creates a table larger than the DRAM cache and then randomly updates locations within the table 5000 times).

The NVMM system ("hybrid" controller) performs extremely well, especially when the controller uses prefetching. Compared to the SSD built with MLC NAND Flash (75000 ns), the NVMM architecture provides a 3x improvement for a single thread of GUPS with no prefetching. However, Linux is performing prefetching in the SSD case as a result of its adaptive readahead mechanism. As a comparison to prefetching,



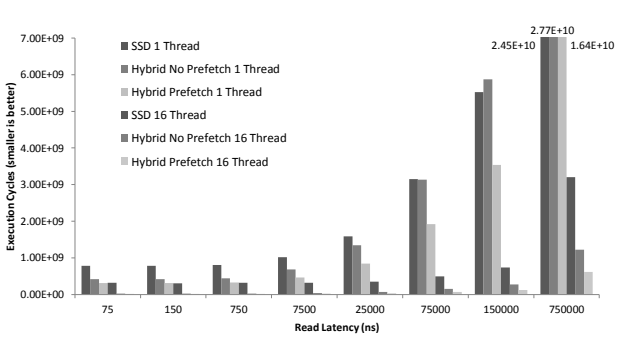
Power and Energy Comparisons. *The graphs on the left-hand side represent 8GB configurations; the graphs on the right represent 1TB configurations. The graphs on the top present the average power consumed; the graphs on the bottom represent the energy per 1000 transactions.*



Latency Sensitivity for GUPS. *The y-axis is execution time in cycles.*

NVMM fetches the next 16 pages after the one that caused a miss. This scheme is less intelligent than the readahead mechanism but still manages to provide a boost in performance resulting in a 3.7x improvement in the single threaded case versus the SSD for the MLC-like latency. Although one would expect prefetching to not help a random access workload, there is some benefit due to a "birthday attack"-like effect, in which prefetching the pages from the backing store will help some future accesses with a certain probability. The boost provided by prefetching also indicates that there is additional bandwidth available to the backing store. This is because the backing store is not being fully utilized by the stream of traffic generated by GUPS, even though it is generating far more requests to the backing store than a typical workload. However, at the 750000 ns latency point, the additional traffic generated by the prefetching decreases performance rather than helping. This is because the additional read latency reduces the available bandwidth. The introduction of multiple threads increases the performance margin between Hybrid and the SSD even more to a 4.7x speedup.

For the MMAP benchmark (a file larger than the DRAM system is mapped into virtual memory via *mmap()* and then read 10000 times at random locations), the performance benefits of multithreading and prefetching are even greater than they were with GUPS. These effects can be seen in the results in the figure below.

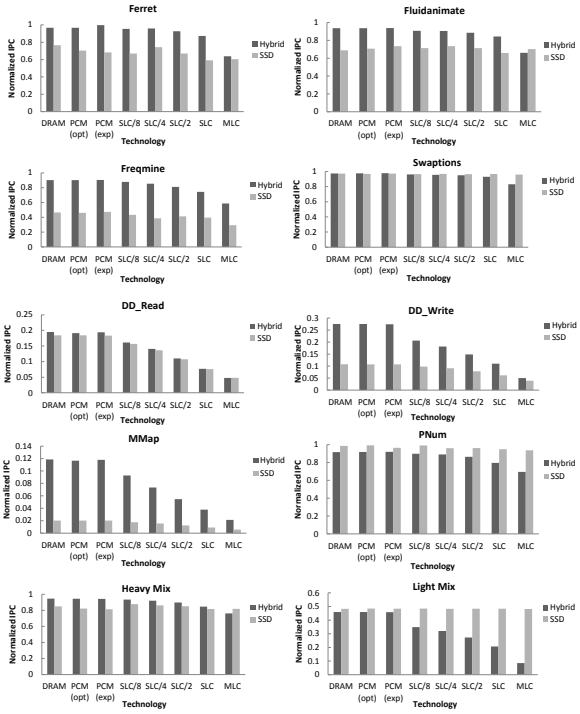


Latency Sensitivity for MMAP. *The y-axis is execution time in cycles.*

For the single threaded version of MMAP, the Hybrid architecture has roughly the same performance as the SSD for an MLC backing store. Turning on prefetching for the single threaded case does provide a boost of 1.6x. However, switching to the multithreaded case results in a large increase in performance for both the SSD and the Hybrid architecture. NVMM has a 3x speedup over the SSD for the MLC latency backing store; with prefetching this becomes a 7x speedup. Prefetching has a greater effect for MMAP than it did for GUPS because MMAP has roughly twice as many accesses as GUPS so the "birthday attack" effect is amplified.

Another interesting feature of this experiment is that we can see a crossover point where the performance of the SSD surpasses the performance of the Hybrid architecture. At the 150000 ns backing store latency, the performance of the SSD surpasses the Hybrid architecture for the single threaded case, when no prefetching enabled. This trend is extended at the 750000 ns point. Multithreading and prefetching push this crossover point beyond the 750000 ns point. This demonstrates that NVMM is viable for MLC NAND Flash with a few simple optimizations such as sequential prefetching.

We also look at a number of other workloads in non-prefetching organizations, and with several more latency values, to try to highlight the behavior of systems as a function of technology latency.



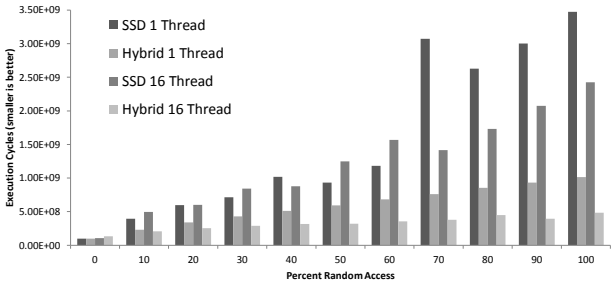
Latency Sensitivity. *IPC is normalized to the performance of the large-DRAM system.*

In general, PCM-based systems perform extremely well (usually close to the upper limit of an all-DRAM system); NAND-based systems approach the performance of PCM-based ones; and SSD architectures do well on largely sequential workloads. In addition, SSDs appear to be the least sensitive of the three architectures to variations in non-volatile latency. For workloads with many random accesses, NVMM outperforms SSDs because Linux is highly optimized for sequential file-system access, prefetching a substantial amount in software. By comparison, NVMM is configured in these results for single-block lookahead. Additional OS optimizations are re-

sponsible for the performance gap between the SSD system and NVMM in the light mix workload. This workload fits entirely into main memory, so the OS uses the swap token technique [Jiang & Zhang 2005] to ensure that the working set of each scheduler quantum does not cause swapping, which enables the over-all system performance to be much higher. This shows the importance of tailoring the operating system to the memory system's technology and is an important area of future work.

A Closer Look at Random Accesses

This experiment demonstrates the effect of varying degrees of randomness in the memory access pattern on both NVMM and a DRAM/SSD system when 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. This experiment establishes the substantial benefit enjoyed by NVMM for even a small degree of random read traffic. This because the operating system's heuristics for SSD control are targeted towards sequential access and are not particularly tolerant to random accesses.



Varying Randomness. *GUPS Benchmark.*

Even programs that are largely sequential can benefit from NVMM's efficient handling of random reads, as the SSD architecture is affected by even small amounts of randomness. In the multi-threaded version of the workload, the relatively sequential threads interfere with one another and produce traffic that is largely random at the memory and storage levels. 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. NVMM is more deterministic due to its reduced dependence on software.

The Software Overhead of SSDs

To measure the software component of a storage access in the standard SSD-based system we instrumented our MMAP benchmark to log when a request began and ended at the application level. To accomplish this we used x86 *rdtsc* instructions that ran immediately before and after each accesses. 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 subtract the hardware time from those raw values to compute the actual time 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 the table below.

	Total Time (ns)		HW Time (ns)		SW Time (ns)	
Latency	Mean	Std Dev	Mean	Std Dev	Mean	SW Delay
SLC	85,400	33,800	38,900	6,700	46,500	54%
MLC	162,200	61,100	88,800	13,000	73,400	45%

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.

Note 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 non-determinism than the Hybrid architecture.

Conclusions

The entire point of a memory hierarchy is that, properly configured and organized, it can offer average latency approaching that of the fastest technology and at the same time the cost per bit approaching that of the densest technology. Thus it should be no surprise that one could build a main memory out of an extremely dense but extremely slow memory technology such as modern flash, representing a roughly thousand-fold slowdown relative to DRAM.

We have demonstrated such a system to be viable and have shown that, at large capacities, it approaches the performance of an all-DRAM system, in particular an all-DRAM system that would be extremely expensive to build. Moreover, the DRAM system would also be expensive to operate, as, at terabyte-sized DRAM capacities, the power is significantly higher than a flash-based system.

Upcoming technologies such as HMC and ONFI 4.0 will be extremely useful in addressing any bandwidth shortcomings; in particular, the channel between DRAM cache and Flash devices requires significant bandwidth to be effective.

As we have shown, terabyte-scale main memories can be a reality today, and the wear-out issues of flash are addressed by spreading writes across enormous numbers of chips. In addition, future solutions such as active flash management and on-chip annealing processes that "heal" the cells promise to make wear-out a thing of the past.

References

N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, and A.R. Panigrahy (2008). "Design tradeoffs for SSD performance." In *Proceedings of the 2008 USENIX Technical Conference (USENIX'08)*.

A. Badam and V.S. Pai (2011). "SSDAlloc: Hybrid SSD/RAM memory management made easy." In *In Proc. 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*.

D.P. Bovet and M. Cesati (2005). *Understanding the Linux Kernel.* O'Reilly Media.

A.M. Caulfield, A. De, J. Coburn, T.I. Mollow, R.K. Gupta, and S. Swanson (2010). "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*, pp. 385–395.

J. Chase, M. Feeley, and H. Levy (1993). "Some issues for single address space systems." In *Proc. Fourth Workshop on Workstation Operating Systems.*

J.S. Chase, H.M. Levy, M.J. Feeley, and E.D. Lazowska (1994). "Sharing and protection in a single-address-space operating system." *ACM Transactions on Computer Systems (TOCS)*, *12*(4), 271–307.

J. Coburn, A.M. Caulfield, A. Akel, L.M. Grupp, R.K. Gupta, R. Jhala, and S. Swanson (2011). "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories." *SIGARCH Comput. Archit. News*, *39*(1), 105–118.

J. Condit, E.B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee (2009). "Better I/O through byte-addressable, persistent memory." In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 133–146.

J. Cooke (2009). "Choosing the right NAND for your application." In *Denali MemCon.*

E. Cooper-Balis, P. Rosenfeld, and B. Jacob (2012). "Buffer-on-board memory systems." In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, pp. 392–403.

C. Dirik and B. Jacob (2009). "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*, pp. 279–289.

A.P. Ferreira, B. Childers, R. Melhem, D. Mosse, and M. Yousif (2010). "Using PCM in next-generation embedded space applications." In *Real-Time and Embedded Technology and Applications Symposium*, pp. 153–162.

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

Fusion IO (2012). *Fusion IO*. Retrieved 2012, from http://www.fusionio.com

B. Ganesh, A. Jaleel, D. Wang, and B. Jacob (2007). "Fully-Buffered DIMM memory architectures: Understanding mechanisms, overheads and scaling." In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 109–120.

InsideHPC (2009). *Spansion packs a whole lotta RAM into your server*. Retrieved 2009, from http://insidehpc.com/2009/04/24/spansion-packs-a-whole-lotta-ram-into-your-server/

Intel, Micron, Phison, SanDisk, SK Hynix, Sony, and Spansion (2013). *Open NAND Flash Interface Specification Revision 3.2.* ONFI Working Group.

Intel (2012). *Intel Solid-State Drive 910 Series: Product Specification*. Retrieved 2012, from http://www.intel.com/content/www/us/en/solid-state-drives/ssd-910-series-specification.html

B. Jacob, S.W. Ng, and D.T. Wang (2007). *Memory Systems: Cache, DRAM, Disk.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

S. Jiang and X. Zhang (2005). "Token-ordered LRU: An effective page replacement policy and its implementation in linux systems." *Perform. Eval.*, *60*(1-4), 5-29.

T. Kgil and T. Mudge (2006). "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*, pp. 103–112.

T. Kidder (1981). *The Soul of a New Machine.* Little, Brown & Co. Inc.

L. Lamport (1978). "Time, clocks, and the ordering of events in a distributed system." *Communications of the ACM*, *21*(7), 558–565.

B.C. Lee, E. Ipek, O. Mutlu, and D. Burger (2009). "Architecting phase change memory as a scalable DRAM alternative." *ACM SIGARCH Computer Architecture News*, *37*(3), 2-13.

Marvell (2012, March). *Marvell Unveils Third-Generation SSD 6Gb/s SATA Controller*. Retrieved 2014, from http://www.marvell.com/company/news/pressDetail.do?releaseID=2176

J. Mashey (1999). *Why No More SRAM Main Memories?* Retrieved March 3, 2014, from the yarchive.net database, http://yarchive.net/comp/sram_main_mem.html

R. Micheloni, L. Crippa, and A. Marelli (2010). *Inside NAND Flash Memories.* Springer.

Micron (2014, January). Personal Communication.

Micron (2012). *Small-Block vs. Large-Block NAND Flash Devices*.

J.C. Mogul, E. Argollo, M. Shah, and P. Faraboschi (2009). "Operating system support for NVM+DRAM hybrid main memory." In *Proceedings of the 12th conference on Hot topics in operating systems*, pp. 14–14.

Newegg (2014, February 28). *Newegg.com - Computer Parts, Laptops, Electronics, and More!*. Retrieved February 28, 2014, from the Newegg database, http://www.newegg.com/

OCZ Technology (2012). *PCI Express OCZ Technology*. Retrieved 2012, from the OCZ Technology database, http://www.ocztechnology.com/products/solid_state_drives/pci-e_solid_state_drives

Oracle (2010). *Achieving New Levels of Datacenter Performance and Efficiency with Software-optimized Flash Storage*. Retrieved 2012, from http://www.oracle.com/us/products/servers-storage/storage/tape-storage/software-optimized-flash-192597.pdf

A. Patel, F. Afram, S. Chen, and K. Ghose (2011). "MARSSx86: A full system simulator for x86 cpus." In *Design Automation Conference 2011 (DAC'11)*.

M.K. Qureshi, M. Franceschini, A. Jagmohan, and L. Lastras (2012). "PreSET: Improving performance of phase change memories by exploiting asymmetry in write times." In *International Symposium on Computer Architecture*, pp. 380–391.

M.K. Qureshi, V. Srinivasan, and J.A. Rivers (2009). "Scalable high performance main memory system using phase-change memory technology." In *Proc. 36th Annual International Symposium on Computer Architecture*, pp. 24–33.

D. Roberts, T. Kgil, and T. Mudge (2009). "Using non-volatile memory to save energy in servers." In *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 743–748.

P. Rosenfeld, E. Cooper-Balis, and B. Jacob (2011). "DRAMSim2: A cycle accurate memory system simulator." *IEEE Computer Architecture Letters*, *10*(1), 16–19.

Spansion (2008). *Using Spansion EcoRAM To Improve TCO and Power Consumption in Internet Data Centers*. Retrieved 2012, from http://www.spansion.com/jp/About/Documents/spansion_ecoram_whitepaper_0608.pdf

J. Stokes (2008, February 25). "MetaRAM quadruples DDR2 DIMM capacities, launches 8GB DIMMs." *Ars Technica.* Retrieved February 28, 2014, from the arstechnica.com database, http://arstechnica.com/gadgets/2008/02/metaram-quadruples-ddr2-dimm-capacities-launches-8gb-dimms/

J.E. Thornton (1970). *Design of a Computer—The Control Data 6600.* Scott Foresman & Co.

Tom's Hardware (2012). *Samsung Intros NAND Flash-Friendly File System*. Retrieved 2012, from http://www.tomshardware.com/news/NAND-Flash-Flash-Friendly-File-System-F2FS-Jaegeuk-Kim,18229.html

R.M. Tomasulo (1967). "An efficient algorithm for exploiting multiple arithmetic units." *IBM Journal of Research and Development*, *11*(1), 25–33.

M. Wu and W. Zwaenepoel (1994). "ENVy: A non-volatile, main memory storage system." In *Proceedings of the 1994 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

J. Yang, D.B. Minturn, and F. Hady (2012). "When poll is better than interrupt." In *FAST'12: Proceedings of the 10th USENIX conference on File and Storage Technologies*, pp. 3–3.