

Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications

Adrian M. Caulfield Laura M. Grupp Steven Swanson

Department of Computer Science and Engineering
University of California, San Diego
{acaulfie,lgrupp,swanson}@cs.ucsd.edu

Abstract

As our society becomes more information-driven, we have begun to amass data at an astounding and accelerating rate. At the same time, power concerns have made it difficult to bring the necessary processing power to bear on querying, processing, and understanding this data. We describe Gordon, a system architecture for data-centric applications that combines low-power processors, flash memory, and data-centric programming systems to improve performance for data-centric applications while reducing power consumption. The paper presents an exhaustive analysis of the design space of Gordon systems, focusing on the trade-offs between power, energy, and performance that Gordon must make. It analyzes the impact of flash-storage and the Gordon architecture on the performance and power efficiency of data-centric applications. It also describes a novel flash translation layer tailored to data-intensive workloads and large flash storage arrays. Our data show that, using technologies available in the near future, Gordon systems can out-perform disk-based clusters by 1.5× and deliver up to 2.5× more performance per watt.

Categories and Subject Descriptors C.5.5 [Computer Systems Organization]: Computer System Implementation—Servers; C.3 [Computer Systems Organization]: Special-Purpose and Application-Based Systems; C.4 [Computer Systems Organization]: Performance of Systems

General Terms Performance

Keywords Cluster architecture, data centric, Flash memory, solid-state storage

1. Introduction

We live in a world overflowing with data. From the handheld to the data center we are collecting and analyzing ever-greater amounts of information. Companies like Google and Microsoft routinely process many terabytes of data, and users of desktop search engines routinely pose queries across the 100s of gigabytes of data stored on their hard drives. There is no reason to expect our appetite for collecting and processing data to stop growing at its current breakneck speed.

To satiate our appetite for large-scale data processing current technology must overcome three challenges. First, the recent slowdown in uni-processor performance and the difficulties in programming their CMP replacements makes it increasingly difficult to bring large computing power to bear on a single problem. Second, while hard drive capacity continues to grow, the latency and bandwidth that hard drives can deliver do not. Third, power constraints due to cooling, economic, and ecological concerns severely limit the range of possible solutions for both of these problems.

On each of these fronts some progress has been made: For data-centric applications, programming models such as Map-Reduce [17] and Dryad [31] largely automate the task of parallelizing data-processing programs. Solid state storage devices offer increased bandwidth and reduced latency for mass storage. Finally, processor manufacturers have developed very capable, yet very power-efficient processors.

For data-centric computing, the most fundamental of these three advances is the rise of solid-state storage. Flash memory's performance characteristics enable systems far outside the design space covered by existing technologies such as conventional servers, processor-in-disk, and processor-in-memory systems. The highest density flash memories available today (or in the near future) offer 16× the density per package of DRAM at 1/16 the power [9, 36]. In the near future, an array of four flash packages will be able to deliver 4× the read bandwidth of a high-end disk at 1/30 the power and a fraction of the latency. These advantages, combined with the fact that solid state storage arrays comprise many discrete chips, instead of a few large drives, provide vastly more flexibility in the *architecture* of a combined computing and storage platform.

This paper describes *Gordon*, a flash-based system architecture for massively parallel, data-centric computing. Gordon leverages solid-state disks, low-power processors, and data-centric programming paradigms to deliver enormous gains in performance and power efficiency. In designing and evaluating Gordon, we make the following contributions:

1. A description of the Gordon architecture.
2. A flash management software layer that allows highly-parallel operation of large arrays of flash devices.
3. An evaluation of flash management techniques for data-centric applications.
4. A thorough analysis of the Gordon system design space and the trade-offs between power, energy, and performance that Gordon must make.
5. A discussion of cost, virtualization, and system-level issues in Gordon machines.

Our results show that Gordon systems can deliver up to 2.5× the computation per energy of a conventional cluster-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'09, March 7–11, 2009, Washington, DC, USA.
Copyright © 2009 ACM 978-1-60558-215-3/09/03...\$5.00

based system while increasing performance by a factor of up to 1.5. We also demonstrate that our flash management system can deliver up to 900MB/s of read and write bandwidth.

The rest of this paper is organized as follows. Section 2 provides background on the technologies that Gordon utilizes. Section 3 describes the architecture of a Gordon system. Sections 4 and 5 describe Gordon's flash storage system and our design space exploration. Section 6 discusses usage models for Gordon systems and directions for future work. Section 7 concludes.

2. Background

Gordon relies on three technologies: large-scale data-parallel programming systems, solid-state storage, and low-power processors. Data-centric computing has been a topic of research for some time, and we provide a brief overview below. Solid-state storage is a more recent development, and the architectural trade-offs it offers are less well-known, so we discuss them briefly here and in more depth in Section 4. We discuss the recent advances in commercially available low-power processors that Gordon leverages in the next section.

2.1 Peta-scale data-centric parallel programming

Industrial researchers have recently developed several systems for processing extremely large data sets in parallel. Two of these systems, Dryad [31] and MapReduce [17], offer an alternative to parallel databases [18, 25, 19] for managing and computing with massive data sets. They focus specifically on handling large-scale data in enterprise-scale clusters composed of commodity hardware. Both systems provide simple abstractions for specifying data-parallel computations and then automate the task of actually making them run in parallel.

MapReduce, in particular, has received a great deal of recent attention. Several implementations are publicly available [1, 38], and researchers have used MapReduce in domains ranging from natural language processing [17], to machine learning [16], to scientific computing [35]. Extensions have also been proposed to enhance and extend its functionality [15].

MapReduce programs operate on a set of input key-value pairs. A user-supplied map function takes an input key-value pair and generates a set of intermediate pairs. The system groups the intermediate pairs by key and then applies the reduce function to each group. The reduce function generates the final output key-value pairs. MapReduce runs on top of a distributed, reliable file system [24].

Dryad takes a different approach and supports a wider range of application structures, but increases complexity. Dryad programs are a generalization of UNIX pipes to arbitrary directed acyclic graphs of processing steps connected by uni-directional channels. The system maps the processing steps and channels onto available hardware, virtualizing resources as needed.

2.2 Solid-state storage

Flash memory has risen to prominence over the last decade due to the growing popularity of mobile devices with large storage requirements (iPods, digital cameras, etc.). These devices make up the vast majority of the NAND flash memory market. They do not require particularly high performance but they do require cheap, high-density, low-power, persistent storage. In response to these requirements, flash manufacturers have primarily pursued density and cost improvements and paid less attention to performance. This pressure has allowed flash density to increase by 85% per year for the past decade (Figure 1), with 64Gb devices due out next year [34, 10].

Despite these gains in density, flash's full potential for high-performance storage is far from realization. While density has improved, other aspects of flash performance have not kept pace. For instance, the bandwidth on flash devices' 8-bit bus (typically 40MB/s) has not increased since 1995. Likewise, the pin-level interface is primitive by modern standards. Industrial efforts [7, 8] are underway to remedy these problems and promise to raise peak bus bandwidth to at least 133MB/s.

In addition to device advances, system architecture, software, and management technology for flash memory are advancing as well. Several efforts have sought to improve or refine flash storage performance in general-purpose systems. These include new chip-level interfaces [39], solid-state disk organizations [12], improvements in the flash translation layer that abstracts away flash's idiosyncrasies [13], and system-level interfaces [2].

Taken together, these advances signal the beginning of flash memories' "coming of age" as a high-performance storage technology. In the next section we describe a system architecture designed to exploit flash's unique capabilities.

3. Gordon's system architecture

The Gordon system architecture uses the technology described in Section 2 to put terabytes of solid-state storage in close proximity with 100s or 1000s of power-efficient processors. The processors communicate over a simple interconnect and execute independent instances of a stripped-down operating system along with the software stack needed to implement a data-parallel execution environment like MapReduce.

The goal of the Gordon design is to reduce the mismatch between network bandwidth, disk bandwidth, and CPU performance. In a typical high-density cluster (e.g., a rack of blade servers) a few power-hungry, high-performance processors provide several GOP/s of computing power, but share access to a small number of (also power-hungry) hard drives that provide limited bandwidth. In a Gordon system, the ratio of GOP/s to MB/s is quite different: Each processor has access to a flash array that provides enormous bandwidth and very low latency.

We call the combination of a processor and its flash storage system a Gordon node. Figures 2(a) and (b) show a scale drawing of a single Gordon node and an enclosure containing sixteen nodes. Within an enclosure, nodes plug into a backplane that provides 1Gb ethernet-style network connectivity and power. A standard rack would hold about sixteen enclosures for 256 nodes, for 64TB of storage and 230GB/s of aggregate I/O bandwidth.

As Figure 2 demonstrates, Gordon nodes can be very compact. This is largely due to the high density of flash devices (16GB/package). Each node contains 256GB of flash storage, a flash storage controller (with 512MB of dedicated DRAM), 2GB of ECC DDR2 SDRAM, a 1.9Ghz Intel Atom processor, and other supporting circuitry. The configuration in the figure delivers the most performance per watt of all the configurations we examine in Section 5.

A Gordon node offers two principle advantages over a conventional server. First, it consumes much less power: Our power model (see Section 5) estimates the configuration shown would consume no more than 19W, compared to 81W for a full-blown server. Second, it provides the processor 900MB/s of read and write bandwidth to 256GB of "disk."

Each Gordon node is a complete, independent computer. They each run a full-blown operating system (e.g., a minimal Linux installation) and communicate using conventional network interfaces (TCP/IP). The benchmarks we use for this study use Hadoop [1], an industrial-strength implementation of Google's MapReduce parallel programming envi-

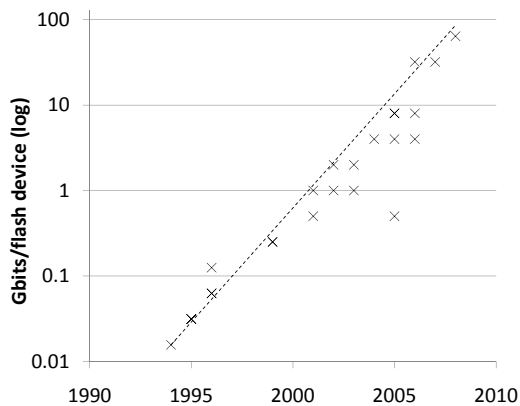


Figure 1. Flash scaling: Flash memory density has been increasing by 85% per year.

ronment. Hadoop provides automatic parallelization and a reliable, distributed file system similar to the Google File System [24]. From the software and users’ perspectives, a Gordon system appears to be a conventional computing cluster.

Our goal in designing Gordon is to significantly improve performance and power efficiency for large-scale data-intensive applications that process many terabytes of data. To achieve this goal, we have developed a high-performance flash storage system tuned for data-intensive applications. We describe this storage system in the next section. Then, in Section 5, we carry out a systematic exploration of the Gordon system design space to find the optimal balance between storage configuration, CPU performance, and power.

4. Gordon’s storage system

Gordon’s flash-based storage system is the key to its power efficiency and performance advantage. This section describes the physical organization of the flash storage system as well as the firmware layer that manages it. Although Gordon’s storage system targets data-intensive applications, several of the approaches we describe are applicable to more general-purpose storage systems as well.

Before we describe Gordon’s flash storage system, we briefly outline current flash technology, likely future developments, and the opportunities and limitations that flash presents. Then, we explore the architecture of the flash memory system including the flash controller, the buses that connect flash chips, and the firmware layer that manages the array.

4.1 Flash memory overview

Flash memory technology presents unique challenges and opportunities that system architects must address and exploit in order to realize flash’s full potential. Below we briefly describe NAND flash technology, its organization, its interface, its performance, its reliability behavior, and the system software support it requires. The performance parameters we list below are typical and are the ones we assume for this work. Actual devices may vary slightly from these values.

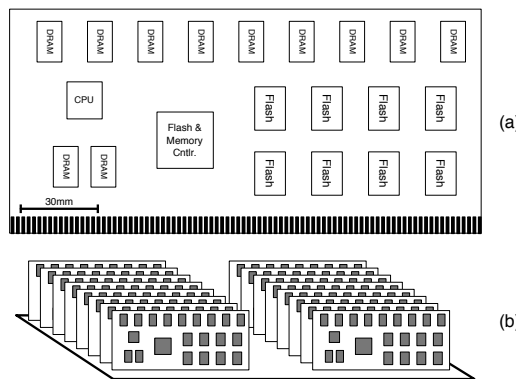


Figure 2. A Gordon system: A scale drawing of a Gordon node PCB (a) and a sketch of 16 nodes in a single enclosure. Gordon nodes can be very compact. This configuration (with flash on both sides) holds 256GB of flash memory and 2.5GB of DDR2 SDRAM. The enclosure holds 4TB of storage and provides 14.4GB/s of aggregate I/O bandwidth.

Flash technology Flash memories store data as charge trapped on a floating gate between the control gate and the channel of a CMOS transistor. Each gate can store one (single-level cell or SLC) or more (multilevel cell or MLC) bits of information. Commercially available devices store just one or two bits per cell. Modern SLC flash devices achieve densities of 1 bit per $4F^2$ where F is the process feature size (currently 30nm), allowing for very high-density flash arrays. Emerging 3D flash technologies [22, 33] promise to push densities even higher.

High-density flash applications use NAND flash devices, which provide less flexible access to data than lower-density NOR flash. We only consider NAND flash in this work.

Organization Flash memories are organized differently than SRAM or DRAM both logically and physically. A flash device contains a set of *blocks* each made up of 64 (SLC) or 128 (MLC) *pages*. Each page contains 2112 bytes. This includes a 2048-byte primary data area as well as an “out of band” data area used to store bad block information, ECC, and other meta-data.

Devices typically divide blocks among two or four “planes.” Each plane has its own buffer to hold data for programs and reads. Planes can perform some operations in parallel, although they contend for the package pins.

Operations NAND flash devices support erase, program, and read operations. Erase operates on entire blocks and sets all the bits in the block to 1. It takes 2ms to erase one block. Program operations program entire pages at once. The time to program a page includes moving the data over the pins and onto the device. Once the data is in an internal buffer, programming typically takes between $200\mu s$ (SLC) and $800\mu s$ (MLC). Program operations can only change 1s to 0s, so an erase operation (of the entire block) is required to arbitrarily modify the page’s contents. Read operations read an entire page in parallel. Reading data from the plane into the internal buffer takes $25\mu s$.

Capacity Current SLC and MLC technology allows for 32Gb/die and 64Gb/die, respectively [34, 10]. Manufacturers stack between one and four chips in a single package, allowing for between 4GB (one die, SLC) and 32GB (four die,

MLC) per package. While increased density is desirable, it comes at a cost: All chips in a single package contend for a single bus (8 or 16 bits wide, depending on the device) of I/O pins.

Performance Flash offers significant performance gains over conventional hard drives. Currently, 8-bit flash devices can transfer data on and off chip at 40MB/s, although 133MB/s chips are beginning to appear and 400MB/s chips are projected [27]. This peak bandwidth is higher than the maximum achievable sustained bandwidth because reading and programming latencies are large (see above). A single 40MB/s SLC flash chip with a single plane can achieve a maximum of 30.7MB/s for reads and 12.9MB/s for writes. Erase bandwidth is not affected by bus speed since it does not require any data transfer. As a result, erase bandwidth (166MB/s for a single plane) far exceeds program bandwidth. Increasing the number of planes per die and/or dies per package can increase performance significantly by increasing bus utilization (see below).

Reliability Flash memories can fail in several ways. Most notoriously, devices wear out with use. After many repetitions, the erase and program process can cause cells to become stuck due to physical damage to the device. The expected lifetime or *erase budget* of one block in a flash device is 10,000 for MLC and 100,000 for SLC. Flash devices report erase and program failures due to wear out, and manufacturers recommend that the entire block be removed from service if an error occurs that cannot be corrected with ECC. To maximize the lifetime of a flash chip, flash systems use wear-leveling [14, 32, 42] to ensure that blocks are erased with equal frequency.

Whether flash wear-out is a problem depends on the application. With good wear-leveling, it is possible to write 25.6PB to the storage array in a Gordon node. The peak program bandwidth for a node is 900MB/s, so wear out could, in theory, occur in one year. However, in practice none of our applications sustain program rates near that level and the average across the applications is much lower, leading to much longer lifetimes for the array. Nonetheless, techniques for carefully managing the write budgets of large flash arrays warrant further study.

Software Flash storage systems typically include a “flash translation layer” (FTL) that manages flash storage. The FTL serves two primary purposes: The first is to provide wear leveling. FTLs maintain a layer of indirection between the logical block addresses (LBA) that the system uses to address data, and the physical location of the data in flash. This allows FTLs to write data wherever they wish, and in particular, to spread writes out across the available flash storage to improve wear leveling. The second purpose is to provide high-performance access to the array by scheduling accesses across the chips to exploit as much parallelism as possible between and within the flash devices. We discuss flash translation layers in more detail below.

4.2 Gordon’s storage system

The flash storage system comprises two components – a flash controller and the flash memory itself. The flash controller provides the link between the CPU and the flash array and implements Gordon’s FTL. The physical organization of the flash memories and the logical organization that the FTL imposes on them has a large impact on the storage system’s performance.

4.2.1 The flash array hardware

The flash controller implements Gordon’s FTL and provides the hardware interface to the flash storage array. We would like the controller to be able to manage as much stor-

Parameter		SLC	MLC
Chip Configuration			
Density Per Die (GB)		4	8
Page Size (Bytes)		2048+32	2048+64
Block Size (Pages)		64	128
Bus Width (Bits)		16	16
Operational Latencies (μs)			
	Read	25	25
	Write	200	800
	Erase	2000	2000
Peak Bandwidth (MB/s)			
40MHz Bus	Read	75.8	75.8
	Program	20.1	5.0
133MHz Bus	Read	126.4	126.4
	Program	20.1	5.0
400MHz Bus	Read	161.1	161.1
	Program	20.1	5.0

Table 1. Flash characteristics: We assume near-future flash memory technology for this study. MLC values are from [34, 37]. SLC numbers are from [9]. Per-die capacity is based on [34]. Bus speed is a projection from [8].

age as possible, but hardware constraints limit its capacity. Flash chips connect to the controller over shared buses. Each bus supports up to four flash packages, each of which contains four dies. We expect that attaching more packages to a 133Mhz (66Mhz DDR) bus would be challenging. Likewise, additional buses would be expensive. Each bus comprises 24 shared pins (eight control and 16 data) and a unique chip enable and ready line for each die, for 56 pins, or a total of 224 pins across four buses. In addition to the flash buses, the controller must also interface with the host processor (150 pins), support the system DRAM interface (105 pins), and the narrower, private DRAM interface used to store FTL meta-data (41 pins), for a total of 304 signal pins. Other signals (JTAG, etc.) are needed as well. For comparison, the Atom processor’s System Controller Hub (which uses aggressive packaging technology) has 474 I/O pins and a total of 1249 pins [30].

For flash devices, we use the parameters in Table 1. The values in the table represent flash technologies that should be commercially available in the next 1-2 years.

4.2.2 The Gordon FTL

Gordon’s FTL defines the logical organization of the flash array and the interface the processor uses to access the flash storage. Gordon’s FTL is an extension of the FTL described in [13]. This FTL allows the application to write to and read from any logical block address (LBA) at random, hiding flash’s idiosyncrasies from the rest of the system.

The FTL performs program operations at a *write point* within the array. The write point is a pointer to a page of flash memory. When the FTL receives a write command, it programs the data to the location indicated by the write point, updates the LBA table and advances the write point to the next page in the block. When the write point’s block is full, the FTL allocates a new, erased block for the write point. If there is an insufficient supply of empty blocks, the FTL may have to clear a block by copying data within the array (see [13] for details).

The LBA table is held in volatile memory, but the FTL must keep a persistent version as well. The FTL stores this

data as a summary page in each block. The summary records the LBA-to-physical mapping for each page in the block. Since multiple, stale copies of an LBA may exist, the FTL also gives each block a sequence number. The freshest copy of an LBA's data is the last copy written to the block with the highest sequence number.

A key limitation of the FTL in [13] is that it allows for only a single write point. As a result, it will never allow two operations to proceed in parallel, except in the case of cleaning blocks in the background. For small flash storage systems (like a "USB key"), this is acceptable, but for Gordon it is not.

We use three techniques to solve this problem. The first is to aggressively pursue dynamic parallelism between accesses to the flash array. We have extended our FTL to support multiple write points and spread accesses between them. To maintain the sequence number invariant, each write point has its own sequence number, and once an LBA has been written to a particular write point, future writes must go to the same write point or another write point with a larger sequence number.

The policy for spreading programs across write points selects the write point with the smallest sequence number that can accept data for the LBA while also balancing load across the busses. Using multiple write points does not affect read bandwidth significantly, but it can improve write bandwidth dramatically. Our data show that increasing the number of write points per 133Mhz bus from 1 to 4 increases write bandwidth by 2.8 \times .

The second approach is to combine physical pages from several dies into "super-pages" (and, therefore, "super-blocks" for erase operations) and manage the flash array at this larger granularity. We explore three ways to create super-pages: horizontal striping, vertical striping, and 2-dimensional (2D) striping (Figure 3).

In horizontal striping each physical page in a super-page is on a separate bus. In this case, access to the physical pages proceeds in parallel. The number of buses in the system limits the size of horizontal super-pages. Horizontal striping is similar to the "ganging" scheme described in [12].

Vertical striping places the physical pages in a super-page on a single bus. The bus transfers each physical page's data in turn, but the program and read operations can occur in parallel across the dies. The number of dies per bus limits the size of vertical super-pages.

The final scheme, 2D striping, combines horizontal and vertical striping to generate even larger super-pages. With 2D striping, it is possible to divide the array into rectangular sets of chips that are part of the same horizontal and vertical stripes. Our FTL provides one write point for each of these sets.

2D striping trades parallelism between operations for parallelism within a single operation. It also reduces management overhead by reducing the total number of super-pages in the array. This is very important, since the LBA table for a large flash array can be very large. For instance, for 256GB of flash and 2KB pages, the LBA table is 512MB. The same array with 64KB super-pages requires only 16MB. Since our flash controller has 512MB of storage and our storage array is 256GB, super-pages must be at least 4KB in size. In this case the LBA table consumes 256MB, leaving space for other FTL data.

Large super-pages cause two problems. First, they increase the latency of sub-page accesses, since the FTL will need to read or program more data than requested.

The second danger is that wear-out failures will effect a much larger portion of the array. In a striped system, if a single physical block wears out, the entire super-block must be removed from service. We have stress-tested SLC flash chips

to measure their failure rate directly, and our data show that for 64KB super-pages, only 98% of super-blocks will last to 100,000 erase cycles, compared to 99.9% for 2KB pages. For 128KB, only 96% will survive this long. In a striped disk-based system, RAID techniques would provide the means to restore this lost reliability. However, the reliability of RAID systems stems from the possibility of *replacing* failed components. Since flash components are soldered to the Gordon nodes, this is not practical.

Vertical and horizontal striping have complimentary effects on how the FTL utilizes the buses that connect the flash devices. Horizontal striping effectively creates a wider, high-bandwidth bus, increasing the raw performance of the array for large reads. Vertical striping further enhances throughput by increasing bus utilization. When writing to a vertical stripe, the FTL can queue up all the data for a program operation and stream out the data without interruption, transferring data to each chip in turn. For reads, the FTL can initiate the read operation on all the chips, and then read data from each one in turn. Interleaving [12] accesses between multiple vertical pages can further increase performance.

To evaluate these alternative organizations, we have developed a detailed trace-driven flash storage system simulator. It supports parallel operations between flash devices and across planes within a single device. It also models the flash buses and implements our FTL. We generated traces of 200,000 random accesses (reads or writes, depending on the trace) of between 2KB and 256KB. To evaluate performance for real applications, we use traces from our suite of data-intensive workloads (Table 2).

Figure 4 demonstrates the impact of increasing super-page size on performance. For super-pages with four or fewer pages, we use horizontal striping, but going beyond four requires vertical striping as well. The data show write bandwidth for varying transfer sizes. The data also show the negative impact on bandwidth for large super-page sizes and small transfers. For instance, bandwidth for 32KB and 128KB transfers is the same until the page size exceeds 32KB. Then, the extra work required to read the rest of the super-page reduces performance.

Our workloads present a mix of large and small transfers. While 88-93% of bytes read and written are part of transfers of 120KB or larger, roughly half of individual transfers are only 8KB. As a result, setting our page size to 128KB would roughly double the amount of data the FTL had to read and write.

Careful examination of traces from our workloads shows a clear pattern in accesses: an 8KB access aligned on a 128KB boundary followed immediately by a 120KB access to the rest of the same 128KB region. To exploit the pattern, we added a simple bypassing mechanism to our FTL that merges incoming read requests with pending requests to the same page and also caches the result of the last read. Our FTL also performs limited write combining by merging write requests to the same page when possible. Figure 4 shows that this mechanism nearly eliminates the negative impact of large pages for small sequential reads. The small dip in performance is due to the overhead of scanning the queue of pending reads. The effect is similar for reads. As expected, bypassing and combining do not improve random access performance.

The data also show the necessity of interleaving requests to allow some chips to program or read data while data are flowing over the bus. For 256KB pages, the super-page spans the entire array, so only one request can be active at once. As a result, performance degrades even with write combining. Using 64KB pages gives the best performance and allows four operations to proceed in parallel. This is approximately what we would expect: Transferring 64KB of data over four

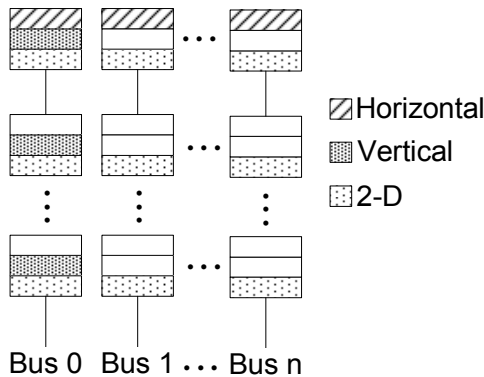


Figure 3. Three approaches to striping data across flash arrays: Horizontal, vertical, and two-dimensional striping reduce overhead by increasing effective page size.

133Mhz buses takes $58\mu s$, or about 0.29 times the program time for our flash chips. If the transfer time were $1/3$ the program time, bus utilization would be nearly 100%.

Figure 5 shows how page size and bypassing affect overall storage performance of our data-intensive workloads, normalized to 8KB pages. In the left-hand figure, bypassing is disabled, and large pages benefit Index, Identity, RandomWriter, and Sort. The right-hand figure shows performance with bypassing. There, 64KB pages provide between $1.1\times$ and $6.3\times$ speedups for all applications.

For sequential accesses (and random accesses to super-pages of similar size) our storage array delivers 900MB/s of bandwidth for reads and writes. Our design uses 16-bit SLC flash devices and assumes a 133Mhz bus. Currently, 40Mhz devices are commonly available. Moving to these devices would reduce peak bandwidth by 60% and reduce average I/O performance for our applications by the same amount. Conversely, moving to a 400Mhz (200Mhz DDR) bus would increase performance by between 20% for write-intensive applications and $2.5\times$ for read-intensive applications. It would also push peak read bandwidth to 2.2GB/s. Peak write bandwidth would rise to 1.1GB/s. Write performance is limited by the aggregate program bandwidth of the chips on each bus. Moving to MLC devices also reduces performance because of their long program times, reducing performance by between 2% (read-intensive) and 70% (write-intensive).

5. Configuring Gordon

Having tuned Gordon’s flash storage system for data-intensive applications, we turn our attention to the remainder of the Gordon node design. First, we describe our workloads, power model, and simulation methodology. Then we describe our design space survey of Gordon configurations. Section 6 presents further results and discussion.

5.1 Workloads

To motivate Gordon’s design, we use a set of benchmarks that use MapReduce for parallel computation. Table 2 summarizes the workloads. Two of the benchmarks focus specifically on I/O (Identity and RandomWriter). The other bench-

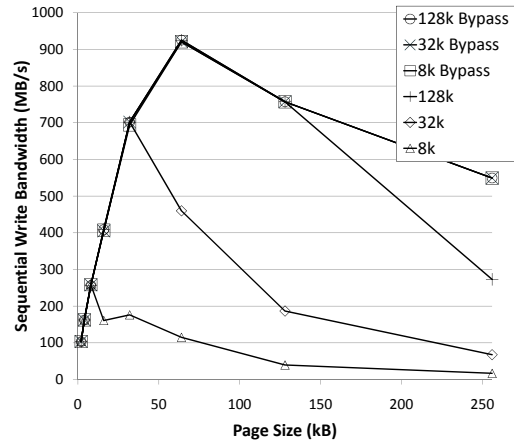


Figure 4. Flash storage array performance: Without bypassing or write-combining, large super-page sizes lead to decreased performance for small transfers. Adding these features nearly eliminates the effect for sequential accesses.

marks represent more realistic applications from a range of domains. WebIndex is our most sophisticated application. It generates an index for a web search engine. ComplexGrep is the most computationally demanding of the workloads. The I/O measurements in the table do not include accesses hidden by the distributed file system’s caching mechanism. For instance, Sort writes more than it reads, because some accesses to temporary files are satisfied by the cache.

To run the benchmarks we use Hadoop [1], an industrial-strength open-source MapReduce implementation written in Java. It provides many of the features of Google’s MapReduce [17] including a distributed file system (Hadoop DFS), similar to Google’s GFS [24]. All our experiments run under Linux 2.6.24 using Sun’s 64-bit Java implementation (JDK 1.6).

To characterize the workloads we ran each of them on a cluster of eight 2.4GHz Core 2 Quad machines, with 8GB of RAM, and a single, large SATA hard disk. A private 1Gb ethernet network connects the machines.

5.2 Power model

To measure power consumption, we developed a power model for Gordon and more conventional systems. Table 3 summarizes the model. For the server components, our data come from direct measurements of a running system. Our model uses activity factors similar to the Mantis full system power model [20] and our results broadly match their results as well as those in [28, 21, 20]. For flash memory and the Atom processor, we use maximum power numbers from product data sheets. As [21] points out, these numbers are often up to 50% greater than actual consumption. As a consequence, the results we report for Gordon’s efficiency are conservative. We model flash controller power on Intel’s System Controller Hub, but with a second DRAM channel [30].

The power model describes machine configurations as a set of identical nodes, each of which contains one or more processors, disk (or flash), and DRAM. Within each node, we model each component’s power as $P = IdlePower \times (1 - ActivityFactor) + ActivePower \times (ActivityFactor)$. The activity factors come from our traces, or, in the case of flash, from our flash simulator.

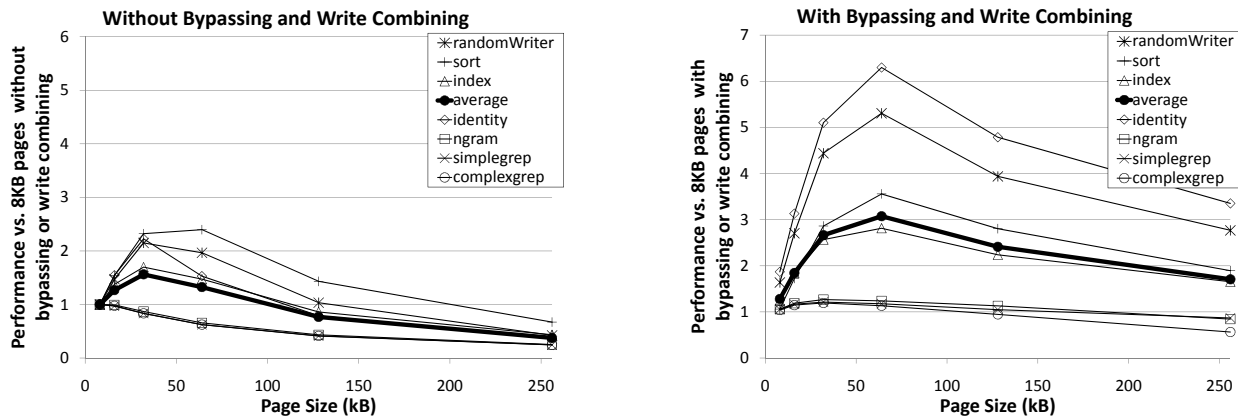


Figure 5. Application-level IO performance with and without bypassing: Read bypassing and write combining (right) allow all applications to benefit from larger page sizes. Without these two optimizations, large pages degrade performance (left).

Name	Description	Workload size	Disk Use (MB)	
			Read	Write
RandomWriter	Output random data	10GB	351	26,860
Identity	Copy all inputs to the output	15GB	45,106	103,650
Sort	Sort random numbers	1GB	1,373	5,672
Grep (simple)	Search for “the” in multi-lingual text	8GB	8,448	483
Grep (complex)	Complex regular expression search in multi-lingual text	8GB	9,199	958
N-Gram	Find frequently occurring N-word phrases in multi-lingual text	4GB	40,133	90,688
WebIndex	Indexing of web pages	13GB	18,925	62,808

Table 2. Workloads: The workloads we use to evaluate Gordon vary in the amount of read and write bandwidth they require.

Component	Power (W)	
	Idle	Active
Core 2 CPU (1 core @ 2.4Ghz)	12	20.8
Atom CPU	0.16	2.4
2GB DDR2 SDRAM	3	5.3
Hard disk (read)	9	13
Hard disk (write)	9	17
4GB SLC Flash (Rd/Wr/Eraser)	0.003	0.04
Flash Ctlr./DRAM Ctlr.	0.16	2.3
Flash Ctlr. 512MB SDRAM	0.7	1.2
System overhead power	See text.	

Table 3. Power model: The values for the Core 2, DRAM, and disk components are from measurements of a running system. For other components, we use values taken from the manufacturer [11, 9]. Core 2 idle power is from [29]. We model flash controller power on Intel’s System Controller Hub, but with a second DRAM channel [30].

Component	Performance
Core 2 CPU (one core)	1.5 insts/cycle
Atom CPU	0.5 insts/cycle
Hard disk latency	simulated
Hard disk read BW	simulated
Hard disk write BW	simulated
Flash read BW	simulated
Flash write BW	simulated
Network BW	1Gb/s

Table 4. Baseline component performance metrics: For the servers, these numbers are from measurements of a running system. Atom performance numbers are based on [3, 4] and our measurements.

To model system overhead power, we measured the idle power of one of our servers and subtracted out the idle power of memory, disk, and CPU components. We then measured average total system power under load, and took the ratio of the two. Using this ratio, we set the ratio of idle power to average power to be the same for each of our configurations.

We have validated this model against the servers in our cluster by enabling and disabling system components and measuring power consumption on varying load. It gives results within 10% of our measurements. For experiments in which we vary CPU speed, we scale voltage with frequency across the range supported by each processor (0.75-1.2V for Atom [26]; 0.8-1.6V for the Core 2 [29]).

5.3 Measuring cluster performance

To evaluate the performance of an architecture for a cluster of machines we use two simulators. The first is a high-level, trace-driven cluster simulator to measure total system performance. The second simulator provides detailed storage system simulations that allow us to explore architectural options for flash storage arrays.

High-level simulator We use a trace-driven high-level simulator to measure overall performance. We collect traces from running systems that contain second-by-second utilization information for each component using performance counters and the `/proc` file system on Linux. The traces describe the number of instructions executed, the number of L2 cache misses, the number of bytes sent and received over the network, the number of bytes written to and read from the disk drive, and the number of hard drive accesses performed. We collect the traces on otherwise idle machines, but the traces include *all* system activity (OS, system daemons, etc).

We model a 32-node cluster by running four VMWare [5] virtual machines on each of our eight servers (giving each VM its own CPU and 2GB of memory) and gather independent traces for each one. Since VMWare does virtualize performance counters, we gather instruction and L2 miss counts for the VM itself.

The simulator processes a set of traces (one per node) in parallel. For one sample of data in the trace, it computes the time needed for instruction execution, disk accesses, and network transfers during that second. It then takes the maximum of these values as the simulated execution time for that sample. Taking the maximum effectively models perfect parallelism between disk, network, and the CPU.

We calculate two sets of results from the simulator using two different methods that model different amounts of inter-node synchronization. The first, called *sync*, models a barrier at the end of each trace sample, forcing the nodes to synchronize frequently and preventing nodes that do little work from finishing early. The time to execute each sample in the trace is set by the slowest node for that sample. The *sync* model provides an upper bound on execution time. The second method, *nosync*, models zero synchronization between nodes. The execution time for the trace is set by the running time of the slowest node. The *nosync* model provides a lower bound on execution time.

To measure performance of p -way multi-processor configurations, we model the execution of p parallel instances of the workload and divide run-time by p . We assume that performance scales linearly with clock speed.

To measure power, the simulator uses the power model described above to compute the power consumption each second for each node in the cluster. The *sync* model includes power for the idle periods when one node is waiting for the other nodes to finish. Nodes are never idle in the *nosync* model until they are finished executing. Once execution on a

Parameter	Value	Parameter	Value
Processors	1,2,4	Processor type	Atom,Core 2
Flash dies	0,64	Atom freq (Ghz)	0.8,1.5,1.9
Hard drives	0,1,2,4	Core 2 freq (Ghz)	0.6,1.2,1.8,2.4

Table 5. Node design space: The parameters for our design space exploration. For all node configurations, we model a cluster of 32 nodes.

node is complete, we assume it goes into a deep power-done state.

Table 4 contains the simulator parameters for all system components. For Core 2 Duo performance, disk, and network, these are peak values. Atom performance (based on [3, 4]) are given for comparison. We use this simulator to generate all of our system-level performance results.

Storage simulator Since we study machines with two types of storage systems (flash and hard drives), we use two different simulators. For disk simulations we use `disksim` [23] configured to model a 1TB, 7200rpm, SATA II hard drive with a 32MB disk cache. To model flash behavior we use the simulator described in Section 4. Both simulators process block-level traces taken from running systems and generate latency and bandwidth measurements. We use these values as inputs to the high-level cluster simulator.

5.4 Design space survey

We have carried out a systematic survey of the design space for Gordon nodes. To understand how Gordon compares to the alternatives, our survey includes both flash-based and hard drive-based designs.

Table 5 shows the parameters and values we varied in our survey. We restrict ourselves to a single storage type (disk or flash) and processor type (Core 2 or Atom) in each configuration. We set the power budget for a single node to 300W. The result is a set of 88 node configurations. All flash-based designs use the maximum number of flash chips possible per node, because adding additional chips cannot hurt performance and their contribution to power consumption is small. We assume the flash storage system described in the previous section with 64KB super-pages.

We use our simulators to measure the performance of a cluster of 32 of each node configuration using the *sync* model. Figure 6 contains the results for three individual benchmarks and the average across our benchmark suite. Each point represents a single node configuration and measures the energy consumption of the whole system versus run-time. All data are normalized to a four-processor Core 2 configuration with one disk.

The lower-left points (circled) are the Pareto optimal node designs: Shorter run times are not possible without increasing energy and vice versa. For all the workloads, the same designs are Pareto optimal. All of them are flash-based. Table 6 summarizes the Pareto optimal designs and the lowest-power design. The designs labeled MinT, MaxE, and MinP, are the fastest (minimum time), most efficient (performance/watt), and minimum average power configurations, respectively.

The table also summarizes the improvements in performance, efficiency, and power consumption that flash provides for each configuration. For instance, MaxE is between 2.2 and 2.5 \times more efficient than the most efficient disk configuration, while MinP saves over 68% in power. The gains in performance are substantial as well: MinT is between 1.5 \times and 1.8 \times as fast as a similar disk-based system while spending nearly equal energy.

Name	CPU configuration	Average Power (W)		Power vs MinP Disk		Perf/watt vs MaxE Disk		Speedup vs MinT Disk	
		NoSync	Sync	NoSync	Sync	NoSync	Sync	NoSync	Sync
MinP	1 Atom; 0.8GHz	1.43	3.91	0.33	0.32	1.74	1.12	0.07	0.07
MaxE	1 Atom; 1.9GHz	2.32	4.81	0.54	0.39	2.54	2.15	0.16	0.16
	1 Core 2; 2.4GHz	9.14	19.89	2.11	1.63	2.31	1.88	0.59	0.56
	2 Core 2; 2.4GHz	23.82	45.66	5.51	3.74	1.64	1.45	1.09	1.00
	4 Core 2; 1.8GHz	47.56	92.74	11.00	7.59	1.11	0.92	1.48	1.29
MinT	4 Core 2; 2.4GHz	58.50	106.18	13.54	8.69	1.08	0.93	1.77	1.49

Table 6. Optimal Gordon configurations: For all three design goals (performance, performance/watt, and power consumption) Gordon achieves substantially better results than disk-based designs. Results are presented for both sync and nosync methods. MinP is not Pareto optimal.

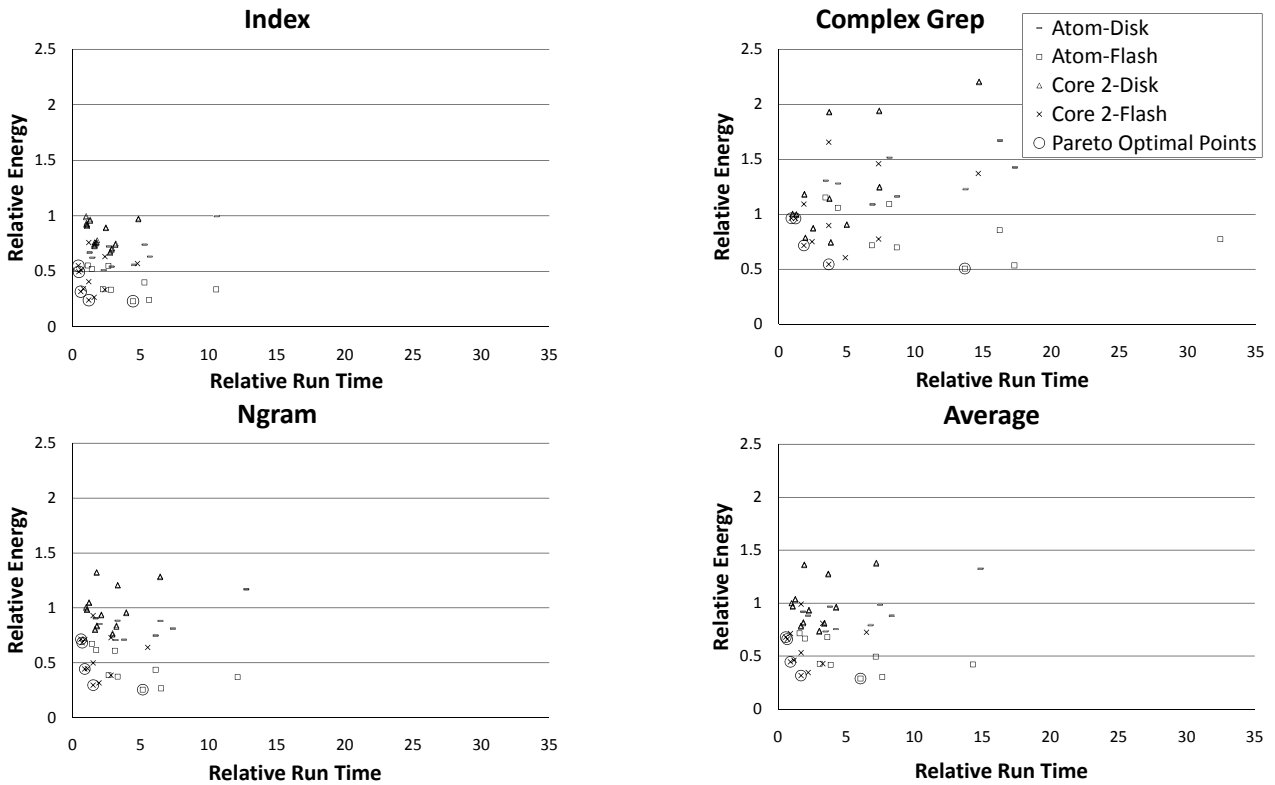


Figure 6. Pareto-optimal Gordon system designs: Results of our design space survey for Index, NGram, Complex Grep, and the average across our benchmark suite. The Pareto-optimal designs are circled.

Figure 7 shows how the network, processor, and storage systems contribute to execution time for MaxE and MaxE-disk, the most efficient disk-based configuration (4 Atom processors @ 1.9Ghz, 1 disk). The stacked bars show the portion of execution when different combinations of components are fully utilized (i.e., when those components limit performance). The disk-based configuration spends about equal time disk bound (second section of the bar from the top) as CPU bound (third from top). For the MaxE flash configuration, the CPU is clearly the bottleneck, but since the MaxE configuration is Pareto-optimal, adding additional processors is not a good power/performance trade-off. The MinT flash configuration, however, includes four processors and its performance varies from strongly CPU bound (ComplexGrp) to substantially disk/flash bound (Identity).

5.5 Gordon power consumption

Figure 8 shows the per-component energy consumption for the MaxE configuration relative to MaxE-disk. Per-component energy consumption is mostly uniform across the applications. On average, MaxE-flash consumes 40% of the energy of the disk-based configuration, but execution times are longer, leading to a factor of two increase in performance per watt. The data show that the increased idle power of disk has a twofold impact on efficiency. It causes the disk to burn excess power, but also encourages the design to use higher-performance, less efficient processors, since these processors reduce the amount of time the disk sits idle. Flash eliminates the vast majority of the storage system's idle power, allowing the design to take advantage of more efficient processors.

Reducing Gordon's power consumption further is challenging. DRAM and overhead power account for most of the remaining power. Reducing the amount of DRAM may be possible, especially if the in-memory working sets of our applications are small. Overhead power is a widely reported problem at the system level [21, 20]. Reducing this source of waste would benefit a wide range of systems, and make the power savings that flash can offer even more significant.

6. Discussion

Incorporating flash memory into Gordon's system architecture required us to reevaluate the trade-offs between power, performance, efficiency, and cost in the design of a Gordon node. At a larger scale, fully exploiting the efficiency gains that Gordon offers requires careful considerations of larger-scale trade-offs. We explore several of the trade-offs below and examine usage models for large Gordon systems and their potential roles in a large data center.

6.1 Exploit disks for cheap redundancy

Using a distributed, replicated filesystem increases the cost of storage for both disk and flash-based systems. We can mediate this problem for some applications by combining Gordon nodes with other servers that have conventional hard drives. Gordon's file system can keep one replica in flash and redundant copies on disk. In the case of a failure, recovery will be automatic. When the disk-based replicas are not needed (the vast majority of the time) the conventional servers can be put into deep sleep with their hard drives spun down.

This system works well for reads, since the replicas do not need to be updated. For write-intensive workloads, the replica must be stored in flash at least temporarily, otherwise the disk bandwidth for updating the hard drive-based replicas would limit performance and significantly increase power consumption. One approach to mitigating this effect is to treat the flash storage array as a *replica cache*. All replicas of frequently updated data would be kept in flash, but replicas of less frequently updated data could be kept on disk. If

	Capacity	Cost	Active power (W)
SAS disk	300GB	\$340	15
SATA disk	1.5TB	\$129	17
4GB DIMM	4GB	\$193	6
FusionIO 160GB SLC	160GB	\$2000	9
2.6GHz Intel Quad core	n/a	\$500	60
Max DRAM enclosure	n/a	\$1795	110
Max flash enclosure	n/a	\$1025	90
Max disk enclosure	n/a	\$1600	90

Table 7. Cost and power model for commodity systems:

To model the cost and power consumption of commodity systems, we retail pricing information for the disks and DRAM. Enclosure power data are estimates based on measurements of our cluster.

	# of servers	Media cost	Non-media cost	%Non-media cost	%Non-media power
4GB DIMMs	10,923	\$50M	\$41M	45%	68%
NAND	2185	\$13M	\$3M	22%	91%
SAS Disks	218	\$1.2M	\$627K	35%	53%
SATA Disks	44	\$100K	\$125K	54%	68%

Table 8. The fiscal and power costs of storing a petabyte:

Differences in cost, performance, density, and power requirements affect the distribution of costs across the system. Flash spends more dollars and less power on the storage media itself, so specializing the rest of the system leads to smaller increases in costs and greater savings in power.

writes become less frequent, the replicas could be migrated to disk. Indeed, if a piece of data has not been accessed at all in a very long time, all replicas could be moved to disk.

6.2 Cost

Currently, cluster-based systems rely heavily on commodity components to reduce cost. For current systems that need to store very large amounts of data, this is a wise trade-off: Disks are slow and cheap, so there is little point in spending large sums to provide fast processors and exotic, high-speed networks.

Tables 7 and 8 contain a cost model and the results it generates to show how flash storage alters these trade-offs. The model computes the cost of storing 1PB of data in DRAM DIMMs, SLC flash-based PCIe storage cards [2], SAS disks, and commodity SATA disks. The model includes non-storage costs, such as the enclosures needed to house the storage media and the CPUs required to access it. Using prices and specifications for currently-available commodity servers [41], it chooses the densest option for each storage technology (for DRAM, a machine with 24 DIMM slots; for disks, the machine with sixteen drive bays; for flash, a machine with four PCIe slots to accept flash storage cards).

Currently, the retail price for 160GB FusionIO devices is \$7200, but we do not believe this reflects a reasonable cost for the device they provide. In bulk, SLC flash currently

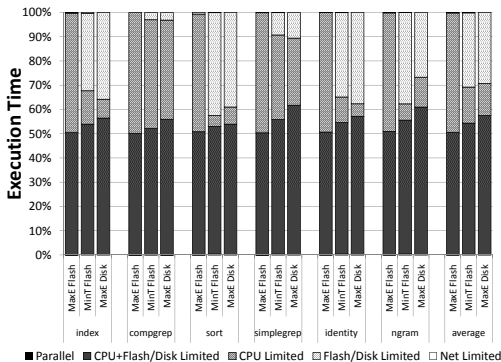


Figure 7. Per-component contributions to execution time: The run-time of each benchmark broken down by the active hardware components.

costs about \$5/GB (\$800 for 160GB) [6], so \$7200 reflects a very large profit margin that will undoubtedly diminish with competition. We estimate the price of the 160GB PCIe device at \$2000.

The model makes it clear why commodity components are a good choice, especially for disk-based systems. In those systems between 35 and 54% of the cost is the non-storage components of the system and those components account for 53-68% of power consumption. If specialized hardware doubled the cost of the non-disk components and reduced their power consumption by half, the resulting machine would cost 35-54% more and use 16-32% less power. Since disks are inherently slow, the gains from specialization would likely be modest.

For NAND, specialization could cost less (22%) and lead to greater power savings (45%). Furthermore, since flash delivers much greater raw performance than disks, the performance gains from specialization will be greater (as our results for Gordon demonstrate).

We expect flash prices to continue to drop at between 50 and 60% per year [40], but hard drive prices are falling as well. However, in terms of cost per bandwidth, flash is a clear winner. Gordon’s flash array delivers over 900MB/s using ~\$1280 worth of flash at current prices. The disks for a RAID array that delivers the same bandwidth would cost \$4500 and would consume 100 times more power than Gordon’s flash storage system. As flash bus speeds increase and prices fall further, flash’s bandwidth advantage will continue to grow.

Ultimately, whether flash is an economically wise design decision depends on the benefits it can deliver and the new applications it can enable. Flash offers huge gains relative to disk in terms of performance, efficiency, density and relative to DRAM in terms of density and power consumption. If these gains are, for instance, sufficient to move a data-intensive application from off-line to on-line, flash could easily justify its extra cost. More generally, we expect flash’s capabilities to enable applications that are simply not feasible with conventional storage technology. This work takes the first step in that direction, by understanding flash’s strengths and weakness in existing applications.

6.3 Virtualizing Gordon

Gordon’s strength is providing high-bandwidth, highly-parallel access to large volumes of data, not in storing that

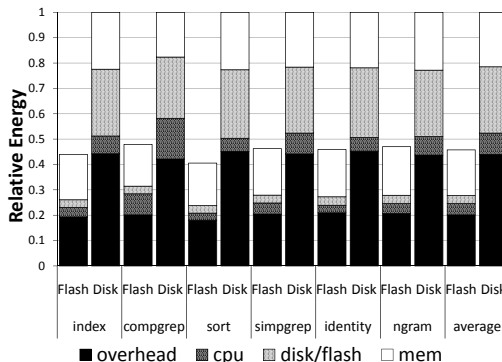


Figure 8. Relative energy consumption: Node energy consumption for MaxE relative to the disk-based configuration with highest performance per watt.

data. For storage, disks are more cost effective. This means that a Gordon system is being effectively utilized if the data it stores are being accessed frequently. Consequently, it makes sense to manage Gordon systems to maximize bandwidth utilization.

For some latency-critical workloads and workloads that need to process all the data stored by a group or organization, it will make sense to store data in a Gordon system and process it in place. In other scenarios, we imagine that the total data stored will be much larger than Gordon’s capacity. In these cases, we will virtualize the Gordon system so its data-processing abilities can be brought to bear on much larger data sets than it can store.

This usage model treats all or part of a Gordon system as a specialized co-processor for performing computations on a small part of a much larger quantity of data stored in a disk-based “data warehouse.” Before a job runs, Gordon will load the necessary data into flash storage, and once the job completes, data for another job will take its place.

For instance, a 1024-node system could be partitioned into 16, 64-node slices, each with NVM storage for 16TB of data. Each slice provides a total of 112GB/s of I/O bandwidth. Assuming dual 1Gb network connections between the Gordon slice and the data warehouse, loading 10TB of data (leaving room for scratch data in the flash storage array) would take about 11 hours. Gordon would perform the computations on the data, transfer the results to the warehouse and load the data for the next job. Network limitations mean that transferring data between the Gordon array and the warehouse utilizes only 0.4% of Gordon’s bandwidth resources. We can improve the situation by overlapping data transfer with execution and reducing the storage space allocated to each job by 50% (to accommodate storing both data sets simultaneously). The impact on execution time would be minimal, and if jobs spent at least 4 seconds processing each GB of data, the time cost of loading the next job would be fully hidden by useful processing.

7. Conclusion

This paper has presented Gordon, a flash memory-based cluster architecture for large-scale data-intensive applications. We describe a flash-based system that is carefully tuned for data-intensive applications. Gordon systems combine this storage array with a conventional low-power processor and programming abstractions for large-scale dis-

tributed programming. The result is a highly-efficient, high-performance, highly-parallel computing system that is easy to use.

Compared to disk-based systems, Gordon systems are $1.5\times$ faster and deliver $2.5\times$ more performance per watt. As flash performance improves, these performance gains will only increase. Gordon demonstrates that flash affords the opportunity to re-engineer many aspects of system design, and, therefore, enables a new class of computing systems.

Acknowledgements

The authors would like to thank the ASPLOS program committee reviewers as well Nathan Goulding and Joel Coburn for their helpful comments. This work is supported in part by NSF awards NSF0811794 and NSF0643880.

References

- [1] <http://hadoop.apache.org/core/>.
- [2] <http://www.fusionio.com/>.
- [3] <http://xtreview.com/addcomment-id-4801-view-Intel-atom-1.6-GHz-benchmark.html>.
- [4] <http://laptoping.com/intel-atom-benchmark.html>.
- [5] <http://www.vmware.com/>.
- [6] DRAMeXchange. <http://www.dramexchange.com/>.
- [7] Open nand flash interface specification 1.0. <http://www.onfi.org/documentation.html>.
- [8] Open nand flash interface specification 2.0. <http://www.onfi.org/documentation.html>.
- [9] Samsung k9f8g08uxm flash memory datasheet.
- [10] Samsung debuts 64gbit mlc nand flash memory. *EE Times Asia*, October 2007. http://www.eetasia.com/ART_8800485916_499486_NP_-11c4687c.HTM.
- [11] New intel centrino atom processor technology ushers in 'best internet experience in your pocket', 2008. Intel Press release.
- [12] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. June 2008.
- [13] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. Technical Report MSR-TR-2005-176. Microsoft Research, December 2005.
- [14] L.-P. Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1126–1130, New York, NY, USA, 2007. ACM.
- [15] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040, New York, NY, USA, 2007. ACM.
- [16] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In B. Schölkopf, J. C. Platt, and T. Hoffman, editors, *NIPS*, pages 281–288. MIT Press, 2006.
- [17] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.
- [18] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.
- [19] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):44–62, 1990.
- [20] D. Economou, S. Rivoire, C. Kozyrakis, and P. Ranganathan. Full-system power analysis and modeling for server environments. June 2006.
- [21] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *ISCA*, pages 13–23, New York, NY, USA, 2007. ACM.
- [22] Y. Fukuzumi, Y. Matsuoka, M. Kito, M. Kido, M. Sato, H. Tanaka, Y. Nagata, Y. Iwata, H. Aochi, and A. Nitayama. Optimal integration and characteristics of vertical array devices for ultra-high density, bit-cost scalable flash memory. *Electron Devices Meeting, 2007. IEDM 2007. IEEE International*, pages 449–452, 10–12 Dec. 2007.
- [23] G. Ganger, B. Worthington, and Y. Patt. Disksim. <http://www.pdl.cmu.edu/DiskSim/>.
- [24] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.
- [25] G. Graefe. Encapsulation of parallelism in the volcano query processing system. *SIGMOD Rec.*, 19(2):102–111, 1990.
- [26] T. R. Halfhill. Intel's tiny atom. *Microprocessor Report*, April 2008.
- [27] A. Huffman. Onfi: Leading the way to higher performance. <http://www.onfi.org/docs/ComputexDRAMeXchange.pdf>.
- [28] Intel. Increasing data center density while driving down power and cooling costs, June 2006. White paper.
- [29] Intel. Quad-core intel xeon processor 3200 series datasheet, 2007.
- [30] Intel. Intel system controller hub datasheet, 2008. <http://download.intel.com/design/chipsets/embedded-datashts/319537.pdf>.
- [31] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, 2007.
- [32] D. Jung, Y.-H. Chae, H. Jo, J.-S. Kim, and J. Lee. A group-based wear-leveling algorithm for large-capacity flash memory storage systems. In *CASES*, pages 160–164, New York, NY, USA, 2007. ACM.
- [33] S.-M. Jung, J. Jang, W. Cho, H. Cho, J. Jeong, Y. Chang, J. Kim, Y. Rah, Y. Son, J. Park, M.-S. Song, K.-H. Kim, J.-S. Lim, and K. Kim. Three dimensionally stacked nand flash memory technology using stacking single crystal si layers on ild and tanos structure for beyond 30nm node. *Electron Devices Meeting, 2006. IEDM '06. International*, pages 1–4, 11–13 Dec. 2006.
- [34] D. Kwak, J. Park, K. Kim, Y. Yim, S. Ahn, Y. Park, J. Kim, W. Jeong, J. Kim, M. Park, B. Yoo, S. Song, H. Kim, J. Sim, S. Kwon, B. Hwang, H. kyu Park, S. Kim, Y. Lee, H. Shin, N. Yim, K. Lee, M. Kim, Y. Lee, J. Park, S. Park, J. Jung, and K. Kim. Integration technology of 30nm generation multi-level nand flash for 64gb nand flash memory. *VLSI Technology, 2007 IEEE Symposium on*, pages 12–13, 12–14 June 2007.
- [35] A. W. McNabb, C. K. Monson, and K. D. Seppi. Mrps0: Mapreduce particle swarm optimization. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 177–177, New York, NY, USA, 2007. ACM.
- [36] Micron. Micron ddr3 sdram mt41j256m8 datasheet rev d, 2008. http://download.micron.com/pdf/datasheets/dram/ddr3/2Gb_DDR3_SDRAM.pdf.
- [37] Y. Park, J. Choi, C. Kang, C. Lee, Y. Shin, B. Choi, J. Kim, S. Jeon, J. Sel, J. Park, K. Choi, T. Yoo, J. Sim, and K. Kim. Highly manufacturable 32gb multi-level nand flash memory with 0.0098 μm^2 cell size using tanos(si - oxide - al2o3 - tan) cell technology. *Electron Devices Meeting, 2006. IEDM '06. International*, pages 1–4, 11–13 Dec. 2006.
- [38] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.
- [39] R. Schuetz, H. Oh, J.-K. Kim, H.-B. Pyeon, S. Przybylski, and P. Gillingham. Hyperlink nand flash architecture for mass storage applications. *Non-Volatile Semiconductor Memory Workshop, 2007 22nd IEEE*, pages 3–4, 26–30 Aug. 2007.
- [40] A. Shah. *PC World*, February 2008. <http://www.pcworld.com/article/id,142684-page,1/article.html>.
- [41] <http://www.supermicro.com/products/system/>.
- [42] D. Woodhouse. Jffs2: The journalling flash file system, version 2. <http://sources.redhat.com/jffs2/>.