

Active Disk Meets Flash: A Case for Intelligent SSDs

Sangyeun Cho (University of Pittsburgh), Chanik Park (Samsung Electronics Co., Ltd.),
Hyunok Oh (Hanyang University), Sungchan Kim (Chonbuk National University),
Youngmin Yi (University of Seoul) and Gregory R. Ganger (Carnegie Mellon University)

CMU-PDL-11-115

November 2011

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Intelligent solid-state drives (iSSDs) allow execution of limited application functions (e.g., data filtering or aggregation) on their internal hardware resources, exploiting SSD characteristics and trends to provide large and growing performance and energy efficiency benefits. Most notably, internal flash media bandwidth can be significantly (2–4× or more) higher than the external bandwidth with which the SSD is connected to a host system, and the higher internal bandwidth can be exploited within an iSSD. Also, SSD bandwidth is quite high and projected to increase rapidly over time, creating a substantial energy cost for streaming of data to an external CPU for processing, which can be avoided via iSSD processing. This paper makes a case for iSSDs by detailing these trends, quantifying the potential benefits across a range of application activities, describing how SSD architectures could be extended cost-effectively, and demonstrating the concept with measurements of a prototype iSSD running simple data scan functions. Our analyses indicate that, with less than a 4% increase in hardware cost over a traditional SSD, an iSSD can provide 2–4× performance increases and 5–27× energy efficiency gains for a range of data-intensive computations.

Acknowledgements: This collaborative work was done while Sangyeun Cho and Chanik Park were visiting PDL in 2011. We also thank the members and companies of the PDL Consortium (including APC, EMC, Facebook, Fusion-IO, Google, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, NEC, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, and VMware) for their interest, insights, feedback, and support.

Keywords: Active Storage, Data-Intensive Computing, Energy-Efficient Data Processing, Reconfigurable Processor.

1 Introduction

A large and growing class of applications process large quantities of data to extract items of interest, identify trends, and produce models of and insights about the data’s sources [7, 14]. Increasingly, such applications filter large quantities (e.g., TBs) of semi-structured data and then analyze the remainder in more detail. One popular programming model is Google’s MapReduce [12] (as embodied in the open source Hadoop system [34]), in which a data-parallel map function is generally used to filter data in a grep-like fashion. Another builds on more traditional database systems and makes extensive use of select and project functions in filtering structured records. A common characteristic is streaming through data, discarding or summarizing most of it after a small amount of processing.

Over a decade ago, research efforts proposed and explored embedding of data-processing functionality within storage or memory components as a way of improving performance for such applications. For example, there were several proposals for so-called “active disks” [1, 18, 29, 27, 28], in which limited application functions could be executed on a disk drive’s embedded CPU to increase parallelism and reduce reliance on host bandwidth at marginal cost. Also, there were proposals for executing functions on processing elements coupled with memory banks [26, 24], with similar goals. In developing these ideas, researchers developed prototypes, programming models, and example application demonstrations. Although interesting, few real systems adopted these proposals, for various technical reasons including manufacturing complexity (especially for active RAM proposals) and independent advances that marginalized the benefits (e.g., distributed storage over commodity multi-function servers [4, 23, 10], which end up having conceptual similarities to the active disk concept).

The active disk concepts are poised for a comeback, in the context of flash-based SSDs, which are emerging as a viable technology for large-scale use in systems supporting data-intensive computing. Modern and projected future SSDs have characteristics that make them compelling points for embedded data processing that filters/aggregates. In particular, their internal bandwidths often exceed their external bandwidths by factors of 2–4 \times , and the bandwidth growth over time is expected to grow rapidly due to increased internal parallelism. Even without the internal-to-external bandwidth multiplier, the rates themselves are sufficiently high that delivering them all the way to a main CPU requires substantial energy; filtering most of it near the flash channels would avoid that energy usage.

This paper develops the case for and an architecture for the result, which we refer to as “intelligent SSDs” (iSSDs). We detail the relevant SSD technology characteristics and trends that create the opportunity and discuss their architectural consequences. Combined with analyses of ten data-intensive application kernels, the internal parallelism (via multiple flash memory channels) and bandwidths of modern and future SSDs push for optimized compute elements associated with each flash memory channel. We focus on use of low-power reconfigurable stream processors for this role.

The performance and energy efficiency benefits of iSSDs are substantial. We quantify these benefits and explore the design space via analytical performance and energy models, as well as some demonstration experiments with a prototype iSSD. Compared to the traditional approach of executing the entire application on a primary server CPU, the iSSD approach offers 2–4 \times higher throughput and 5–27 \times more energy efficiency. We show that the iSSD approach provides much of this benefit even when compared to alternate approaches to improving efficiency of data-intensive computing, including heterogeneous elements (e.g., GPUs or the same reconfigurable stream processors we envision for iSSDs) in the server CPU [33], reliance on so-called “wimpy” nodes [2], or embedding the processing in the RAM subsystem as in proposals discussed above. All of the bandwidth benefits and half of the energy efficiency can only be achieved by exploiting the SSD-internal bandwidth and avoiding the need to move the data to other components.

2 Background and Related Work

2.1 Active devices for data-intensive computing

The concept of “active devices”, in which limited application functionality is executed inside a memory or storage component, has been well-developed over the years. Active disk designs, especially, were motivated and studied extensively. The early proposals laid out the case for exploiting the excess computing cycles of the hard disk drive (HDD) controller’s embedded processor for useful data processing, especially filter and aggregation functions. For example, Riedel et al. [28] showed that, by aggregating the bandwidth and computing capabilities of ten (emulated) HDDs, measured performance improves more than $2.2\times$, $2.2\times$ and $2.8\times$ for workloads like nearest-neighbor search, frequent set mining and image registration, respectively. Substantial benefits have also been demonstrated for a range of other data-intensive workloads, including database scan, select, and aggregation operations, satellite data processing, image processing, and search of complex non-indexed data [1, 27, 17, 18]. Active memory system designs (sometimes called “intelligent RAM”) have also been proposed and studied [26, 24], allowing simple functions to transform and filter ranges of memory within the memory system without having to move it to the main CPU.

Programming models for active disks were developed and shown to fit such data-parallel processing tasks, addressing both the data delivery and software safety issues; these same models should be suitable for iSSDs. As a representative example, Acharya et al. [1] proposed a stream-based programming framework and the notion of sandboxed “disklets” (disk-resident codes) for processing data ranges as they are made available by the underlying device firmware.

Active devices have not become the norm, for various reasons. In the case of active memories, complex manufacturing issues and memory technology changes arose. In the case of active disks, the demonstrated benefits were primarily from parallelized data-local execution; for many data-intensive computing activities, those benefits have instead been realized by spreading stored data across collections of commodity servers (each with a few disks) and partitioning data processing across those same servers [4, 23, 10]. The same approach has been more efficiently implemented with so-called wimpy nodes and low-power CPUs [2], as well. But, we believe that flash-based SSDs have characteristics (e.g., parallel channels and very high internal bandwidths) that make “active-ness” compelling, where it wasn’t for these other devices.

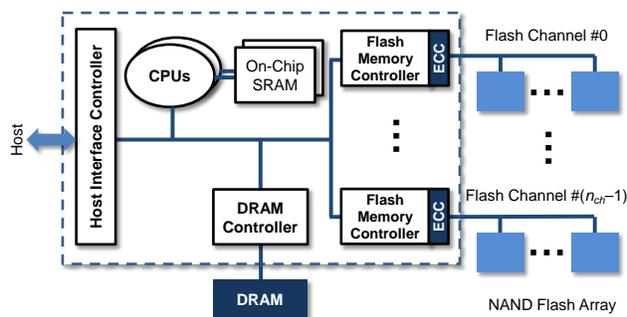
2.2 Architecture and evolution of SSDs

Figure 1 illustrates the general architecture of an SSD and how SSDs have evolved with the introduction of new higher bandwidth host interfaces [31]. Key hardware components in an SSD are the host interface controller, embedded CPU(s), on-chip SRAM, DRAM and flash memory controllers connected to the flash chips. On top of the hardware substrate runs the SSD firmware commonly referred to as *flash translation layer* (or FTL).

The host interface controller supports a specific bus interface protocol such as SATA, SAS and PCI Express (PCI-e). The host interface bandwidth has steadily increased, from PATA (1.5 Gbps max) to SATA (3 to 6 Gbps) in desktop systems and from SCSI (maxed out at about 5 Gbps) to SAS (about 5 Gbps max) in enterprise applications. PCI-e has a high bandwidth of 2 to 8 Gbps per lane and shorter latency than other interfaces.

The CPU(s) and SRAM provide the processing engine for running FTL. SRAM stores time-critical data and codes. Typically, the CPU is a 32-bit RISC processor clocked at 200 to 400 MHz. Depending on the application requirements, multiple CPUs are provided to handle host requests and flash management tasks concurrently. DRAM stores user data and FTL metadata, and is operated at a clock rate of 667 MHz or higher.

Flash memory controllers (FMCs) are responsible for data transfer between flash memory and DRAM



Time frame	Characteristics
2007–2008	4-way, 4 channels, 30–80 MB/s R/W performance; mostly SLC flash based;
2008–2009	8–10 channels, 150–200+ MB/s performance (SATA, consumer); 16+ channels, 600+ MB/s performance (PCI-e, enterprise); use of MLC flash in consumer products;
2009–2010	16+ channels, 200–300+ MB/s performance (SATA 6 Gbps); 20+ channels, 1+ GB/s performance (PCI-e); adoption of MLC in enterprise products;
2010–	16+ channels; wider acceptance of PCI-e;

Figure 1: General architecture of an SSD (left): The dashed box is the boundary of the controller chip. SSD evolution with new host interface standards (right).

(or host interface). They also guarantee data integrity based on an error correction code (ECC). As NAND flash memory is continuously scaled down and adopt two-bits- or three-bits-per-cell schemes, the error rate grows and a stronger ECC capability is required. As a result, the ECC logic has become a dominant part of an SSD controller chip [35]. For high performance, FMCs utilize multi-way interleaving over multiple flash chips on a shared I/O channel as well as multi-channel interleaving. FMCs are implemented with dedicated hardware for efficiency or an application-specific instruction-set processor (ASIP) to support diverse flash commands.

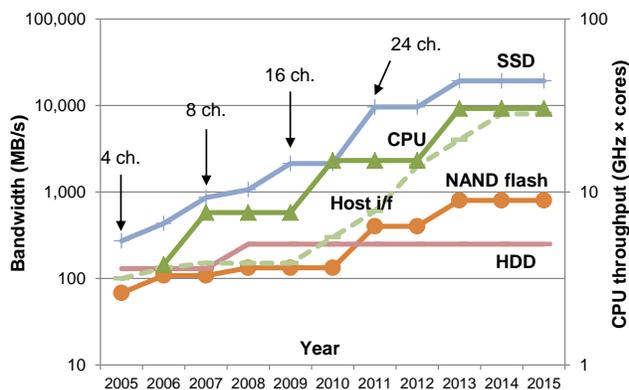


Figure 2: Bandwidth trends of key hardware components.

Bandwidths mean that substantial bandwidth capability and energy is required to deliver all data to an external processing unit, both of which could be reduced by filtering and/or aggregating within the SSD. Indeed, modern external SSD bandwidths lag raw internal bandwidths significantly—differences of 2–4× are common. So, an iSSD could process data 2–4× faster just by being able to exploit the internal data rate. Moreover, the natural parallelism inherent in SSD’s use of multiple channels couples well with parallelized data stream processing.

2.3 iSSD-relevant workload characteristics

iSSDs would be expected to provide benefits for data-intensive applications that explore, query, analyze, visualize, and, in general, process very large data sets [7, 14, 8]. In addition to simple data filtering and aggregation, some emerging data mining applications are characterized by substantial computation. This subsection examines a number of common data-intensive application kernels to gain insight into their rele-

The NAND flash memory interface has evolved from 40 Mbps single data rate to 400 Mbps double data rate; as higher performance is desired, the bandwidth will be raised even further [31]. Figure 2 plots the bandwidth trends of the HDD, NAND flash memory, SSD, host interface and CPU. The bandwidth improvement in NAND flash, especially resulting from increased use of multiple independent memory channels (with 8–16 being common today) for parallel data retrieval, results in much higher raw internal and external bandwidth than HDDs.

SSD bandwidth characteristics are what lead us to explore the iSSD concept. The very high band-

NAME [SOURCE]	DESCRIPTION	INPUT SIZE	CPB*	IPB [†]	CPI [‡]
word_count [36]	Counts the number of unique word occurrences	105MB	90.0	87.1	1.0
linear_regression [36]	Applies linear regression best-fit over data points	542MB	31.5	40.2	0.8
histogram [36]	Computes the RGB histogram of an image	1,406MB	62.4	37.4	1.7
string_match [36]	Pattern matches a set of strings against data streams	542MB	46.4	54.0	0.9
ScalParC [22]	Decision tree classification	1,161MB	83.1	133.7	0.6
k-means [22]	Mean-based data partitioning method	240MB	117.0	117.1	1.0
HOP [22]	Density-based grouping method	60MB	48.6	41.2	1.2
Naïve Bayesian [22]	Statistical classifier based on class conditional independence	126MB	49.3	83.6	0.6
grep (v2.6.3)	Searches for a pattern in a file	1,500MB	5.7	4.6	1.2
scan (PostgreSQL)	Finds records meeting given conditions from a database table	1,280MB	3.1	3.9	0.8

Table 1: Example data-intensive kernels. Measurements were made on a Linux box with a Nehalem-class processor using Intel’s VTune. Programs were compiled using `icc` (Intel’s C/C++ compiler) with the `-O3` flag. Measurement results in the table do not include the time to handle file I/O. *Cycles Per Byte. [†]Instructions Per Byte. [‡]Cycles Per Instruction.

vant characteristics.

Table 1 lists the kernels.¹ We show three metrics in the table to characterize and highlight the data processing complexity and the architectural efficiency. Among the three metrics, IPB captures the data processing complexity of a kernel. If IPB is high (e.g., `k-means`), implying that much work is needed to process a given input data, the workload is compute-intensive. In another example, `grep` and `scan` execute well less than ten instructions on average to process a byte. IPB is determined primarily by the specific data processing algorithm, the expressiveness of the instruction set, and the compiler’s ability to produce efficient codes.

By comparison, CPI reveals, and is determined by, the hardware architecture efficiency. On the testbed we used, the ideal CPI is 0.20 to 0.25 because the processor is capable of issuing four to five instructions per cycle. The measured CPI values range from 0.60 (`ScalParC` and `Naïve Bayesian`) to 1.7 (`histogram`). Compared with the ideal CPI, the values represent at least $2\times$ and up to $8.5\times$ efficiency degradation. There are three main reasons for the inefficiency: high clock frequency (i.e., relatively long memory latency), poor cache performance, and frequent branch mispredictions. According to Ozisikyilmaz et al. [25] the L2 cache miss rates of the classification kernels in Table 1 are as high as 10% to 68%. Moreover, HOP has a branch misprediction rate of 10%, considerably higher than other workloads in the same benchmark suite.

Another interesting observation is that kernels with a low IPB tend to have a high CPI. Kernels with an IPB smaller than 50 have the average CPI of 1.14 while the the average CPI of kernels with an IPB above 50 is only 0.82. We reason that high-IPB kernels are compute-intensive and utilize the available functional units and reuse the data in the cache memory relatively well. On the other hand, low-IPB kernels spend few instructions per unit data and the utilization of the hardware resources may slide (e.g., cache performs poorly on streams).

Lastly, CPB combines IPB and CPI ($CPB = IPB \times CPI$), and is the data processing rate of a platform. Hence, it is our goal to minimize CPB with the proposed ISSD. For the kernels we examined, CPB ranges from less than ten (`scan` and `grep`) to over a hundred (`k-means`).

¹A few kernels have been adapted to run on both an x86 platform and our ARM-based simulator platform described in Section 4.3.

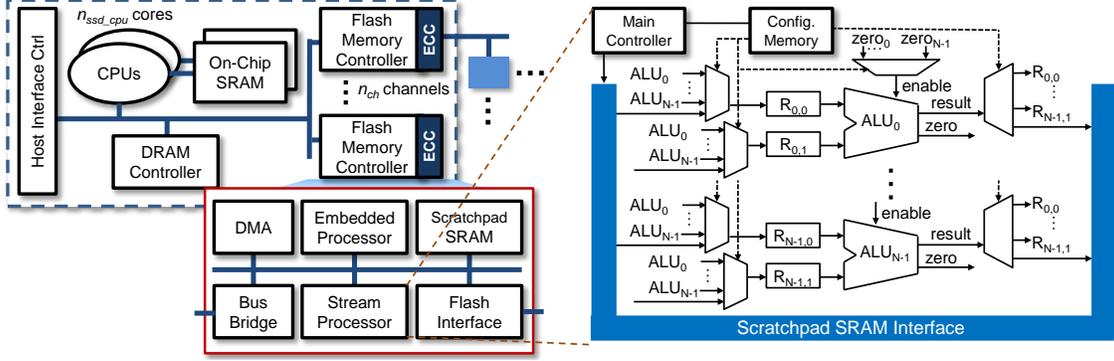


Figure 3: The organization of an iSSD’s FMC (in red box) and the reconfigurable stream processor (right).

3 Intelligent SSDs

iSSDs offer the earliest opportunity to process data after they are retrieved from the physical storage medium. This section describes the iSSD—its hardware architecture and software architecture. Feasible data processing strategies as well as challenges to exploit iSSDs are also discussed.

3.1 Hardware architecture

The main difference in hardware between a regular SSD and an iSSD is their compute capabilities; iSSDs must provide substantially higher compute power to translate the raw flash memory bandwidth into high data processing rates. In conventional HDDs and SSDs, in-storage compute power is provisioned to meet the firmware performance requirements (e.g., host command processing, block mapping, error handling). The active disk work [1, 18, 27] optimistically predicted that future storage controller chips (in HDDs) will have increasingly more excess compute capabilities. However, that didn’t happen because there is no immediate merit for a storage vendor to add higher horsepower to a device than needed at cost. We argue that the iSSD internal architecture must be designed specially and offer both high raw performance and flexible programmability.

There are at least two ways to add compute resources in the SSD datapath: by integrating more (powerful) embedded CPUs and by augmenting each FMC with a processing element. While we consider both in this work, we note that SSD designers have kept adding more flash memory channels to continue the SSD bandwidth growth. Hence, it is necessary to scale the amount of compute power in the iSSD according to the number of flash memory channels, justifying the second approach. This approach has an added benefit of not increasing the bandwidth requirements on the shared DRAM. This work proposes to employ a *customized ASIP* and a *reconfigurable stream processor* as the FMC processing element, as shown in Figure 3.

The diagram in the bottom shows the components inside an FMC—flash interface, scratchpad SRAM, DMA, embedded processor, reconfigurable stream processor and bus bridge. Raw data from the flash memory are first stored in the scratchpad SRAM before data processing begins. Data are then processed by the per-FMC embedded processor (and the stream processor). The embedded processor has a custom instruction set for efficient data processing and small area cost. However, as Table 1 suggests, certain algorithms may still require many cycles for each input byte and may render the FMC stage a serious bottleneck. For example, word count has an IPB of 87. If the flash memory transfers data at 400 MB/s and the embedded processor is clocked at 400 MHz, the actual achievable data processing rate would be slowed down by the factor of 87, compared to the full raw data bandwidth (assuming a CPI of 1)!

In order to effectively increase the FMC’s data processing rate, we propose to incorporate a reconfigurable stream processor, highlighted in the right diagram. Once configured, the stream processor performs

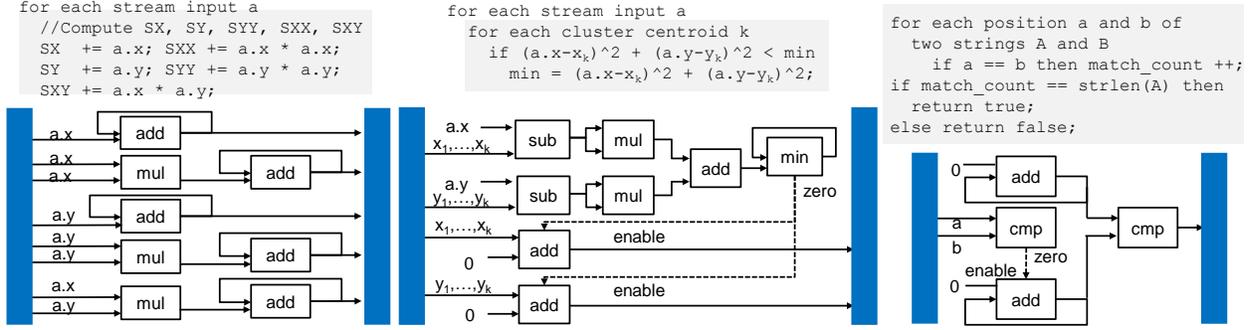


Figure 4: Reconfigurable stream processor instances for linear_regression, k-means and string_match from left.

like dedicated hardware, achieving very high data processing rates and low power [11]. At the heart of the stream processor are an array of ALUs, configurable connections and the main controller. An ALU’s output is either stored to the SRAM or forwarded to another ALU. How processed data flow within the processor is configurable (by programming the “configuration memory”). The main controller is responsible for the configuration and sequencing of operations. Figure 4 presents three example instances of the stream processor. For these examples, the CPB improvement rate is $3.4\times$ (linear_regression), $4.9\times$ (k-means) and $1.4\times$ (string_match).

Data processing with the proposed FMC is expected to be efficient. First, the stream processor exploits high fine-grained parallelism with multiple ALUs. Second, the embedded processor and the stream processor feed their data from a scratchpad memory, not cache. Third, the embedded processor has a shallow pipeline and does not suffer from a high branch misprediction penalty like the host CPU. Hence, the effective average CPB achieved with the FMCs is expected to be very competitive. This observation is revisited in Section 4.

Beyond FMCs, further data processing can be done by the embedded CPUs. The performance of the CPUs can be scaled with the core type (e.g., ARM7 vs. Cortex-A8), core count, and clock frequency. Because the CPUs see data from all flash memory channels, they can perform “wide-scope” tasks like the Reduce stage of a MapReduce job. The performance of the embedded CPUs is limited by the shared DRAM bandwidth, however.

3.2 Software architecture

This section considers programming frameworks to utilize the proposed iSSD architecture efficiently. Since an iSSD platform includes multiple heterogeneous processor cores, its programming framework should be able to express parallelism for heterogeneous processors. Among many frameworks, for example, OpenCL [19] meets the requirements. It not only supports data parallel executions on each parallel device but also provides a unified programming framework to coordinate heterogeneous devices and host CPUs to collaborate. Another framework that is particularly interesting to us is MapReduce [12]. It was developed primarily for distributed systems and assumes that individual processing units (computers) read data from a storage device.

The MapReduce framework provides two basic primitives: *Map* and *Reduce*. They are user specified and easy to be parallelize and distribute on the computing elements (FMC processors and embedded CPUs in the context of iSSD) with the assistance of the run-time system. In addition, the MapReduce model matches well with the hardware organization of iSSD as shown in Figure 5(a). Input data are saved in flash memory prior to being transferred to FMCs. They execute the Map phase and the intermediate results are stored in DRAM or flash memory temporarily. The output data are produced and transferred to the host system or

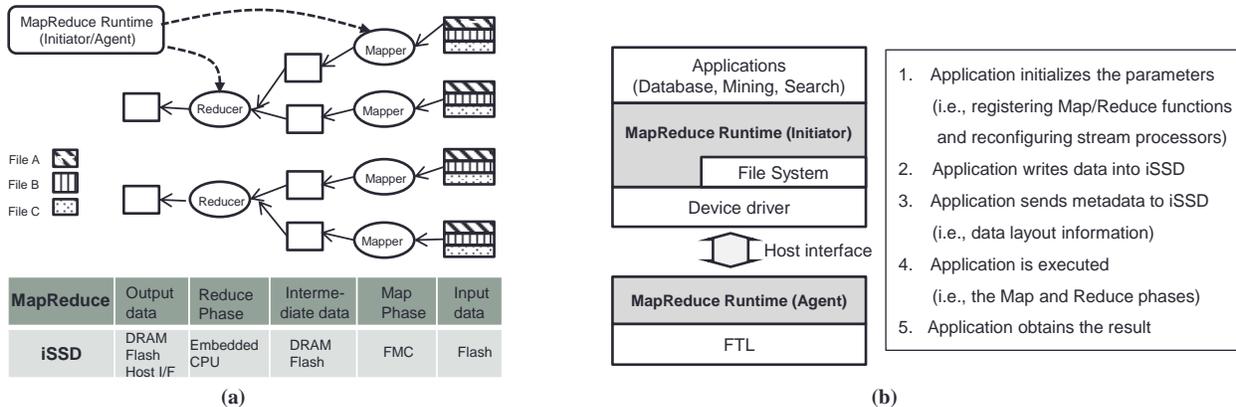


Figure 5: Mapping the MapReduce framework to iSSD. (a) Associating MapReduce phases to iSSD hardware organization. (b) Software structure and execution flow of iSSD.

flash memory after the Reduce phase that is run on the embedded CPUs. The Map and Reduce functions are assigned to the iSSD or the host system depending on the cost and benefit projection. The job partitioning and resource assignment are managed by the MapReduce run-time system that is split to run on the host system and the iSSD. We call the run-time system “Initiator” (on the host) and “Agent” (on the iSSD) (Figure 5(b)). Initiator, essentially a run-time library, provides programmers with a programming interface to iSSDs. Initiator hides some iSSD details such as the number of embedded CPUs and FMCs, the capability of stream processors, and flash memory parameters. Agent is responsible for allocating and scheduling Map and Reduce tasks inside the iSSD. It communicates with Initiator through a tunneling mechanism available in storage protocols like SATA, SAS and PCI-e or via an object storage interface [20].² An iSSD based job execution scenario using the suggested framework is given in Figure 5(b).

iSSDs face an issue that was thoroughly explored in previous active disk work [27, 1, 18]: enabling applications to target data even though legacy file systems are not aware of underlying active-ness and underlying devices are not aware of file system organization details. One approach is to use a separate partition and direct I/O for the data, which worked for modified databases and work for our MapReduce framework. In our iSSD prototype on a real SSD (Section 4.3), this approach is taken. Another is to modify legacy file systems to allow applications (e.g., Initiator) to obtain block lists associated with files, and request that they do not change for a period of time, which can then be provided to the iSSD (e.g., Agent).

An issue for iSSDs that was not present for active disks relates to how data is partitioned across the flash memory channels. Striping across channels is used to provide high streaming bandwidth, which could create difficulties for processing of data on individual per-channel processing elements; with striped data, no single processing element would have it all. Fortunately, the stripe unit size is fairly large (e.g., 8KB or 16KB) and consistent with common file system block sizes and database page sizes [27, 1, 18]. For applications that can process individual blocks/pages, one by one, channel striping should present no issue for our proposed iSSD architecture. Fortunately, almost all active disk work found this to hold for applications of interest, which was important because of files being partitioned into independently allocated blocks. Many more recent data mining activities can also be arranged to fit this requirement, given exposure of the internal block (stripe unit) size. When it is not possible, the embedded processor can still process data (more slowly), as it sees the unstriped data.

²There are interesting trade-offs between using a commodity legacy interface like SATA (large installed base) and the object storage interface (rich information within storage). Exploring the trade-offs in detail is beyond the scope of this paper.

3.3 Data processing strategies with iSSDs

Allocation of Map and Reduce functions to different processing elements affects the system performance significantly. Therefore, one must carefully map tasks onto heterogeneous hardware resources in an iSSD platform. This work proposes two data processing strategies: *pipelining* and *partitioning*.

Pipelining. The pipelining strategy assigns tasks to a set of computing resources in a hierarchical manner. For instance, Map functions could be mapped onto processors inside FMCs and Reduce functions onto the iSSD’s embedded CPUs. As such, a set of homogeneous processors execute their mapped tasks in parallel and send the computed results to the next layer of processors in a pipeline fashion. For example, FMCs can perform pattern search in parallel. They can then pass the matched values to the shared DRAM buffer. Next, the iSSD’s embedded CPUs can accomplish complex operations that require global information, such as Reduce tasks, join in database, and sort over filtered data sets (preprocessed by FMCs). Intermediate data are stored in temporary memory (SRAM and DRAM) and computation results are passed to the higher level computing stages.

Because the available physical processing resources are organized in a hierarchical manner in an iSSD platform—from FMCs to embedded processors (inside the iSSD) to host CPUs—, this data processing strategy fits nicely with the available hardware infrastructure. However, certain applications may not naturally and efficiently exploit this strategy (e.g., lack of low-level parallelism needed to utilize FMCs).

Partitioning. In general, host CPUs are higher performance than any single computing resource in the iSSD. If we map a given data-intensive computing job entirely to an iSSD we may end up under-utilizing the host CPUs. To fully utilize the computing resources in both the host platform and the iSSD, we could assign Map functions to both entities by “partitioning” the job into properly sized sub-jobs. For example, if the input data set is composed of 1,000 files, 400 files could be delegated to the iSSD and the remaining 600 files could be processed by the host CPUs.

For best results, we need to balance the amount of data (computation) the host CPUs and the iSSD take based on their performance. This decision can be made at run time (“dynamic load-balancing”) or before the job is launched (“static load-balancing”). For stream oriented applications, the simple, static load-balancing would work well (the actual balancing decisions are reached based on accurate performance models like the ones in Section 4). This is because the execution time of each function varies little in stream applications. Still, if the execution time of a function varies or it is hard to predict the execution time *a priori*, a dynamic load-balancing strategy will be effective.

Finally, the pipelining and the partitioning strategies can be used in combination. In fact, combining both strategies will likely lead to better system performance with higher overall resource utilization. We will come back to this point when we discuss evaluation results in Section 5.2.

4 Modeling iSSD Benefits

This section develops models to evaluate the benefits of the proposed iSSD approach. We take a data-centric perspective and express performance and energy as a function of *input data size*. In essence, the overall performance (or inversely, total time) and energy are determined by the average time and energy to process a single input byte. We will separately discuss models for pipelining and partitioning.

4.1 Models for pipelining

Performance. A data-intensive workload may involve multiple execution phases. For example, a typical MapReduce workload would go through three phases, namely, *map*, *sort* and *reduce*. Hence, once we

compute the times of individual phases, we can combine them to obtain the overall execution time.³ The execution time of a given phase is, simply put, D/B , where D is the input data size and B the overall processing bandwidth for the phase. Our performance modeling effort accordingly focuses on computing B .

We assume that D is sufficiently large for each phase. Moreover, B is determined primarily by the underlying hardware architecture and the workload characteristics. Then, each phase's execution time is determined by D and the steady data processing bandwidth ($=B$) of the system during the phase. Note that Riedel [27] makes the same assumptions in his active disk performance models.

With the pipelining workload mapping strategy, data being processed go through multiple steps in a pipelined manner. Each step can be thought of as a pipeline stage and the step that takes the longest amount of time determines the overall data processing bandwidth at the system level. The steps are: (a) Data transfer from NAND flash chips to an FMC on each channel; (b) Data processing at the FMC; (c) Data transfer from the FMC to the DRAM; (d) Data processing with the SSD embedded CPU(s) on the data stored in the DRAM; (e) Data transfer from the SSD (DRAM) to the host via the host interface; and (f) Data processing with the host CPU(s). If the time needed for each of the above steps is $t_{nand2fmc}$, t_{fmc} , $t_{fmc2dram}$, t_{ssd_cpu} , $t_{ssd2host}$ and t_{host_cpu} , respectively, the bandwidth is the inverse of the total time (T_{total}). Hence, we have:

$$T_{total} = (1 - p) \cdot t_{conv} + \max(t_*), \quad B = \frac{D}{T_{total}} \quad (1)$$

where t_* is the list of the time components defined above (for steps (a)–(f)) and $(1 - p)$ is the portion of the execution that is not pipelinable [27] using the iSSD scheme, and t_{conv} is the time that would be consumed with the conventional scheme. Let us now tackle each term in t_* .

(a) Data transfer from NAND flash chips to an FMC on each channel. Given n_{ch} flash channels and the per-channel effective data rate of r_{nand} , $t_{nand2fmc} = D/n_{ch} \cdot r_{nand}$. We assume that data have been split onto the NAND flash chips evenly and all data channels are utilized equally.

(b) Data processing at the FMC. Once data are retrieved from NAND flash chips, they are processed in parallel using n_{ch} FMCs. Each ASIP processor at an FMC is assumed to run at the clock frequency of f_{fmc} . Furthermore, to process a single byte, the processor would require CPB_{fmc} cycles on average. Hence,

$$t_{fmc} = \frac{D \cdot CPB_{fmc}}{n_{ch} \cdot f_{fmc}} = \frac{D \cdot IPB_{fmc} \cdot CPI_{fmc}}{n_{ch} \cdot f_{fmc}}. \quad (2)$$

In the above formulation, IPB_{fmc} exposes the efficiency of the instruction set chosen (and the compiler), CPI_{fmc} the microarchitecture, and f_{fmc} the microarchitecture and the circuit-level processor implementation.

(c) Data transfer from the FMC to the DRAM. We introduce a key parameter α to express the amount of residual data for further processing after FMCs finish processing a batch of data. α can be referred to as *reduction factor* or *selectivity* depending on the workload semantics and has a value between 0 and 1. The time needed to push the residual data from the FMCs to the DRAM is, then: $t_{fmc2dram} = \alpha \cdot D/r_{dram}$. We assumed that the aggregate data rate of the n_{ch} NAND flash channels is at least r_{dram} , the DRAM bandwidth.

(d) Data processing with the embedded CPU(s). Our formulation here is similar to that for Equation (2).

$$t_{ssd_cpu} = \frac{\alpha \cdot D \cdot CPB_{ssd_cpu}}{n_{ssd_cpu} \cdot f_{ssd_cpu}} = \frac{\alpha \cdot D \cdot IPB_{ssd_cpu} \cdot CPI_{ssd_cpu}}{n_{ssd_cpu} \cdot f_{ssd_cpu}}.$$

CPB_{ssd_cpu} , f_{ssd_cpu} , IPB_{ssd_cpu} and CPI_{ssd_cpu} are defined similarly. n_{ssd_cpu} is the number of embedded CPUs.

(e) Data transfer from the SSD to the host. If there remains further processing after step (d), data transfer occurs from the iSSD to the host. With a data reduction factor β , the time to transfer data is expressed as: $t_{ssd2host} = \alpha \cdot \beta \cdot D/r_{host}$, where r_{host} is the host interface data transfer rate.

³Tasks of different phases may overlap in time depending on how the “master” coordinates task allocations [12].

(f) Data processing with the host CPU(s). In the last step, data processing with the host CPU takes:

$$t_{host_cpu} = \frac{\alpha \cdot \beta \cdot D \cdot CPB_{host_cpu}}{n_{host_cpu} \cdot f_{host_cpu}} = \frac{\alpha \cdot \beta \cdot D \cdot IPB_{host_cpu} \cdot CPI_{host_cpu}}{n_{host_cpu} \cdot f_{host_cpu}},$$

where n_{host_cpu} is the number of host CPUs. CPB_{host_cpu} , f_{host_cpu} , IPB_{host_cpu} and CPI_{host_cpu} are CPB, clock frequency, IPB and CPI for the host CPUs.

The above time components ((a) through (f)) are plugged into Equation (1) to obtain the data processing bandwidth of a particular phase in a workload. Note that the effect of parallel processing with multiple computing resources (n_{ch} , n_{ssd_cpu} and n_{host_cpu}) is exposed.

Finally, let us derive the bandwidth of the conventional data processing scheme by adapting the above formulation. Because there is no data processing within the SSD, $\alpha = \beta = 1$ (all data will be transferred to the host). Furthermore, step (b) and (d) are skipped, i.e., $t_{fmc} = t_{ssd_cpu} = 0$. With these changes, Equation (1) remains valid for the conventional data processing scheme.

Energy. The main idea of our energy models is: The overall energy is a function of input data size and energy to process a single byte. Energy consumption of a system is the sum of two components, dynamic energy and static energy ($E_{total} = E_{dyn} + E_{static}$). We assume that E_{static} is simply $P_{static} \cdot T_{total}$ where P_{static} is a constant (static power); accordingly, we focus in this section on deriving E_{dyn} .

Dynamic energy consumption is split into energy due to computation and energy for data transfer. That is, $E_{dyn} = E_{comp} + E_{xfer} = D \cdot (EPB_{comp} + EPB_{xfer})$, where EPB_{comp} is the average energy spent on processing a single byte of input and EPB_{xfer} is the average energy spent on data transfer per single byte. EPB_{comp} can be further decomposed into terms that represent the contributions of different hardware components: $EPB_{comp} = EPB_{fmc} + \alpha \cdot EPB_{ssd_cpu} + \alpha \cdot \beta \cdot EPB_{host_cpu}$. Similarly, $EPB_{xfer} = EPB_{nand2fmc} + \alpha \cdot EPB_{fmc2dram} + \alpha \cdot \beta \cdot EPB_{ssd2host}$, where EPB_{A2B} is the energy needed to move a byte from A to B .

To expose the processor design choices and their impact, we can further decompose the terms of EPB_{comp} : $EPB_{fmc} = EPI_{fmc} \cdot IPB_{fmc}$, $EPB_{ssd_cpu} = EPI_{ssd_cpu} \cdot IPB_{ssd_cpu}$, and $EPB_{host_cpu} = EPI_{host_cpu} \cdot IPB_{host_cpu}$, where EPI_* is the average energy per instruction, an architectural and circuit design parameter.

4.2 Models for partitioning

Performance. With the partitioning strategy, data are split between the iSSD and the host for processing. Hence, $D = D_{issd} + D_{host}$. Then, based on the partition, the execution time $T_{total} = \max(D_{issd}/B_{issd}, D_{host}/B_{host})$, where B_{issd} and B_{host} stand for the data processing bandwidth of the iSSD and the host. Clearly, the above formulation captures the importance of partitioning, because T_{total} is minimized when the execution times of the iSSD and the host are equal. B_{issd} and B_{host} can be easily obtained with our formulation of Section 4.1. For example, B_{host} can be expressed as $n_{host_cpu} \cdot f_{host_cpu}/CPB_{host_cpu}$.

Energy. Like before, we assume that E_{static} is $P_{static} \cdot T_{total}$ and focus on deriving the dynamic energy, $E_{dyn} = E_{comp} + E_{xfer} = D_{issd} \cdot EPB_{issd_comp} + D_{host} \cdot EPB_{host_cpu} + D_{host} \cdot EPB_{ssd2host}$. EPB_{issd_comp} is the average energy spent on processing a single byte of input within the iSSD and can be computed using our formulation for the pipelining case with $\beta = 0$. EPB_{host_cpu} and $EPB_{ssd2host}$ were previously defined.

4.3 Validation

The basic performance modeling framework of this work was previously validated by Riedel [27]. They compare the performance of a single server with fast, directly attached SCSI disks against the same machine with network-attached (emulated) active disks. Because the iSSD architecture stresses high data access parallelism inside a single device, the emulation approach is too limiting in our case. Instead, we develop

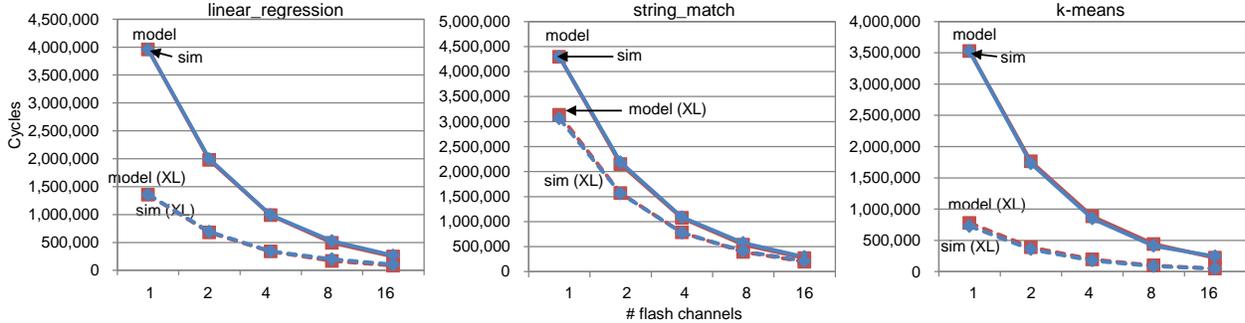


Figure 6: Comparison of cycle counts of three benchmarks obtained with our models and SoC Designer simulation. Dotted lines and solid lines are for iSSD processing with and without stream processor acceleration (XL), respectively.

and use two iSSD prototypes to validate our models and evaluate the iSSD approach. Our first prototype is a detailed execution-driven iSSD simulator that runs on SoC Designer [9]. Our second prototype is built on a real SSD product platform provided by Samsung [30]. Let us discuss each in the following.

Model validation through simulation of iSSD hardware. Our simulation is both functional and timing. We describe the hardware components in Figure 3 using SystemC and realistic timings. We use an existing cycle-accurate ARM9 model to simulate embedded CPUs and FMC processors. Additionally, we build a timing-only wrapper simulation component for reconfigurable stream processors to avoid designing a full-fledged stream processor at the RTL level. This component reads program execution traces and generates necessary events (e.g., bus transactions) at specified time points.

We have ported a MapReduce framework similar to Stanford Phoenix [36] to natively run on our simulator. An embedded CPU in the iSSD becomes a “master” and triggers FMCs to execute Map tasks (more experimental settings in Section 5.1). It also manages buffers in DRAM to collect intermediate data. If the output buffer in DRAM gets full, the embedded CPU flushes the data and triggers the FMCs for further data processing.

Figure 6 plots how well the results obtained from our model and the simulator agree. Overall, the model is shown to predict the performance trends very well as we change the number of flash memory channels. The maximum error between the two results was 17.9% and the average error was 5.1%.

Scan experiments with prototype iSSD. We also separately studied how database scan is accelerated and system-level energy consumption is saved with the iSSD approach by porting PostgreSQL’s scan algorithm to the Samsung SSD platform, having sixteen 40 MB/s FMCs (i.e., internal bandwidth is 640 MB/s) and a SATA 3 Gbps interface (~ 250 MB/s on the quad-core system used). The SSD has two ARM processors and 256 MB DRAM. As for software implementation, simple Initiator and Agent were implemented in the host system and the SSD. They communicate with each other through a system call, `ioctl()` with the `ATA_PASS_THROUGH` parameter. A SATA reserved command was used for parameter and metadata passing. The scan algorithm is implemented into a Map function and loaded into the code memory of an embedded CPU.

For input data, a 1 GB TPC-H Lineitem table [32] was written into flash memory with striping. Selectivity was 1% and projectivity was 4 bytes (out of 150 bytes). After input data was stored, the data layout information such as table schema and projected column was delivered to the SSD. As soon as the host system invoked the scan function through Initiator, the input data were read into the internal buffer memory (DRAM) of the SSD and compared with the key value for scan operation. Matched data items are transferred to the host periodically by Initiator until there is no more data to be scanned.

The measured performance improvement with the iSSD over the host system was $2.3\times$. Because database scan is not compute-intensive, this performance improvement came closely by the ratio between the

SSD-internal NAND flash bandwidth and the host interface bandwidth (640 MB/250 MB=2.56). Through careful mapping of the scan algorithm, we were able to achieve a near-maximum performance gain with only a single embedded CPU. In particular, scan operation and FTL execution time (tens of microseconds) are effectively hidden by data transfer time between flash memory and the internal buffer memory. Our analytical models are capable of capturing this effect, and the performance predictions in Figure 7 agree with our measurement result. We also measured dynamic energy improvement (difference of current flowing from the wall outlet to the system) of $5.2\times$ using a multimeter.

5 Quantitative Evaluation

To complement the prototype measurements above, this section uses the analytical models to study the performance and energy benefits of the iSSD approach across a broader range of configurations. Our main focus is to prove the concept of the iSSD and identify conditions when the iSSD obtains the most and least benefit.

5.1 Evaluation setup

The right table lists the values of the parameters to plug into our models. Hardware parameters (for the iSSD and the host platform) are based on the technology trends summarized in Section 2.2. For intuitive discussions, we focus on four selected kernels that have different characteristics: `linear_regression`, `string_match`, `k-means`, and `scan`. In terms of computation complexity (expressed in IPB), `scan` is the simplest and `k-means` is the most complex. Except `scan`, which we implement its kernel to directly run on FMCs, we ported the Map stage of the remaining kernels to FMCs. We focus on the Map stage of the kernels only because there are many options for running the Reduce stage (e.g., in iSSD or host? Overlapped or not?) and the Map stage is often the time-dominant phase (76–99% in our examples). We use hardware acceleration to speed up the kernels using the stream processor instances in Figure 4. For `scan`, we estimate CPBs for iSSD to be 4 and 1 (with hardware support for fast matching) based on our experiments on a real SSD platform (Section 4.3).

Throughout this section, we fix r_{nand} to 400 MB/s to keep the design space to explore reasonably bounded. This way, we focus on n_{ch} , the only parameter to control the internal raw data bandwidth of the storage. In the case of iSSD, n_{ch} also determines the raw data processing throughput. Lastly, we set $p = 1$ [27].

5.2 Results

Performance improvement potential. Figure 7 compares the data processing rate of different iSSD and conventional machine configurations, as we change the number of FMCs. We assume that the host machine has eight CPUs. iSSD configurations are shown to effectively increase the data processing bandwidth as we add more FMCs while conventional machine configurations (having equivalently fast SSDs) gain little.

Interestingly, the performance of the conventional configurations were compute-bound except for `scan`. For example, with `k-means` and `string_match` (our “compute-intensive” kernels), the host interface speed

iSSD parameters	
r_{nand}	400 MB/s
r_{dram}	3.2GB/s (@800 MHz, 32-bit bus)
n_{ch}	8–64 with the step of 8
f_{fmc}	400 MHz, 800 MHz
n_{ssd_cpu}, f_{ssd_cpu}	4, 400 MHz
System parameters	
r_{host}	600 MB/s (SATA), 4 or 8 GB/s
n_{host_cpu}	4, 8, 16
f_{host_cpu}	3.2 GHz
Workload parameters— <code>linear_regression</code> , <code>string_match</code> , <code>k-means</code> , <code>scan</code>	
p	1
α	0.05 for <code>scan</code> ; <0.05 for others;
CPB_{fmc}	33.6, 38.8, 157.6, 4.0
CPB_{fmc} (w/ accel.)	10.1, 28.3, 32.4, 1.0
CPB_{host_cpu}	31.5, 46.4, 117.0, 3.1

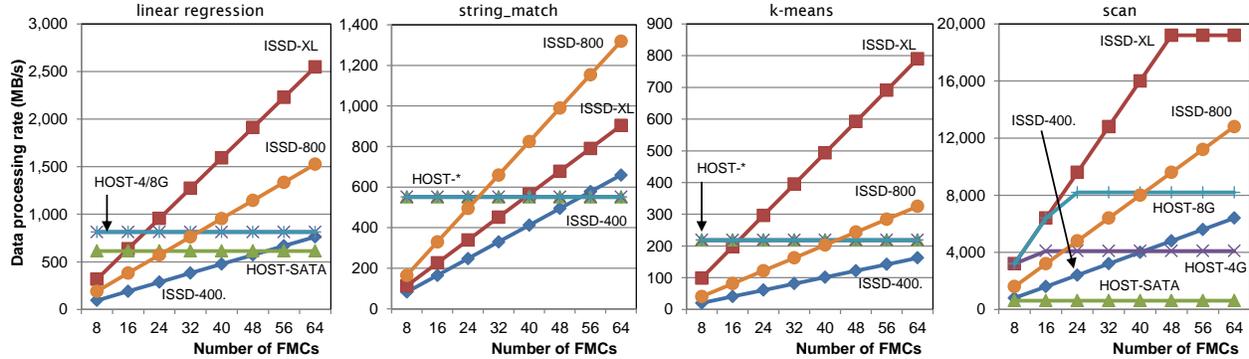


Figure 7: Performance of selected kernels on five system configurations: HOST-SATA, HOST-4G, HOST-8G, iSSD-400, iSSD-800, and iSSD-XL. HOST-“speed” represents a conventional, host CPU based processing scheme with the storage interface speed of “speed” (“SATA”=600 MB/s, “4G”=4 GB/s, “8G”=8 GB/s). iSSD-400 and iSSD-800 employ FMCs running at 400 MHz and 800 MHz for data processing. iSSD-XL uses reconfigurable stream processors for processing. iSSD-400 uses the SATA interface, iSSD-800 the PCI-e 4 GB/s interface, and iSSD-XL the PCI-e 8 GB/s interface.

mattered little. With `linear_regression`, the 4 GB/s and 8 GB/s host interface speeds made no difference. iSSD configurations show scalable performance in the whole FMC count range, except with `scan`. In this case, the performance of iSSD-XL was saturated due to the internal DRAM bandwidth wall (not the host interface bandwidth) when the number of FMCs exceeded 48. By comparison, all conventional machine configurations suffered the host interface bandwidth limitation, making `scan` the only truly storage bandwidth bound workload among the kernels examined. Still, iSSD configurations scale the `scan` throughput for the most part because the iSSD does not transfer filtered data to the host, reducing the required data traffic substantially. Our results clearly highlight the importance of efficient data handling with optimized hardware, parallel data processing and data filtering.

Among the examined kernels, `string_match` and `k-means` have an IPB larger than 50 and are compute-intensive (see Table 1). In both cases, the data processing rate on the host CPUs is not bounded by the host interface bandwidth at all. Interestingly, their performance behavior on the iSSD is quite different. `k-means` was successfully accelerated and performs substantially better on the iSSD than host CPUs with 24 or more FMCs. In contrast, `string_match` required as many as 40 FMCs to outperform host CPUs. The main reason is that the stream processor is not as effective for this kernel as other kernels and produced a small gain of only 37% compared with the 400 MHz FMC processor (also indirectly evidenced by the large improvement with the 800 MHz FMCs). If the iSSD does not have sufficient resources available, one would execute workloads like `string_match` on host CPUs [27]. Still, the particular example of `string_match` motivates us to investigate (in the future) a broader array of workload acceleration opportunities, especially to handle unstructured data streams efficiently.

In a conventional system, parallelism is exploited by involving more CPUs in computation. To gain further insight about the effectiveness of parallel processing inside the iSSD (with FMCs) as opposed to on the host platform (with multiple CPUs), Figure 8 identifies *iso-performance* configurations of the iSSD and conventional systems in two plots, each assuming a different host interface speed (600 MB/s vs. 8 GB/s).

Both plots show that hardware acceleration makes iSSD data processing much more efficient by increasing the effective per-channel throughput. Consider `linear_regression` for example: When r_{host} is 600 MB/s, the iSSD obtains the performance of the 16-CPU host configuration with 52 FMCs (without acceleration) and 15 FMCs (with acceleration). Note that there is a large difference between the maximum raw instruction throughput between the iSSD and the host configurations; the 4-CPU host configuration ($4 \times 3.2 \text{ GHz} \times 4 \text{ instructions/cycle}$) corresponds to twice the computing capacity of the 64-FMC iSSD con-

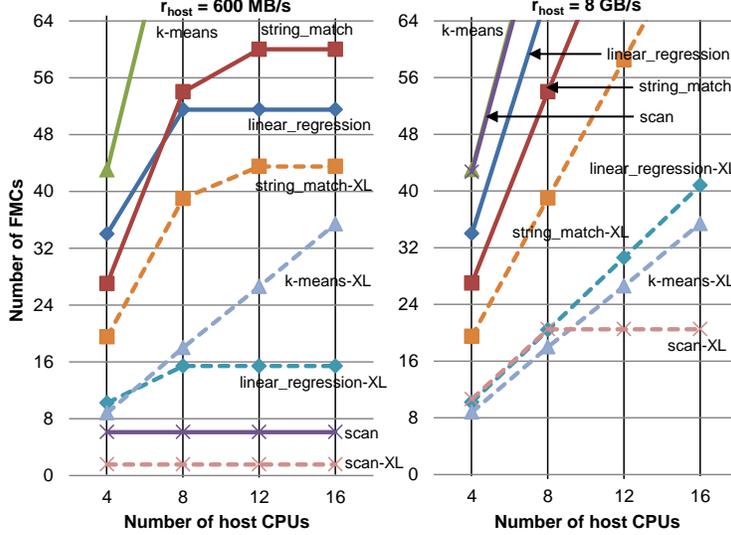


Figure 8: Iso-performance iSSD (Y axis) and host (X) configurations.

figuration ($64 \times 400 \text{ MHz} \times 1 \text{ instruction/cycle}$). The fact that we find iso-performance points in each column reveals the strength of the iSSD approach. Simple kernels like `scan` and `linear_regression` perform much more efficiently on the iSSD. Moreover, relatively complex kernels like `k-means` will also be quite suitable for running on the iSSD, given adequate hardware acceleration support.

With a high-bandwidth host interface, the host CPU configurations gain on performance, and hence, the curves are pushed toward the upper-left corner. Still, the iSSD provides robust performance with acceleration for `scan`, `k-means` and `linear_regression`. With fewer than 41 FMCs, the iSSD performed as well as 16 CPUs for these kernels.

We also gain from above results an insight that having more powerful embedded CPUs in an iSSD (e.g., Cortex-A8s at 1.2 GHz) will not produce equivalent cost-effective performance benefits because they are subject to the same architecture inefficiency issues that plague host CPUs. Moreover, with significantly more compute power in the embedded CPUs, the shared DRAM bandwidth will become a new bottleneck. Attacking data at FMCs—the front line of computing resources—appears essential for scalable data processing in iSSDs.

Lastly, we explore the potential of applying both partitioning and pipelining strategies in the iSSD architecture. We employ all computing resources, including FMCs and host CPUs, to maximize data processing throughput. Figure 9 presents the result, assuming optimal work partitioning between the iSSD and host CPUs. It is shown that higher data processing throughput is achievable with partitioning, compared with either the host only or the iSSD only configuration. Except for `scan`, the maximum achievable throughput is the sum of the throughputs of the host CPUs and the iSSD. This is because the ample internal flash memory bandwidth ($n_{ch} = 32$) can feed the host CPUs and the FMCs simultaneously. This is why data

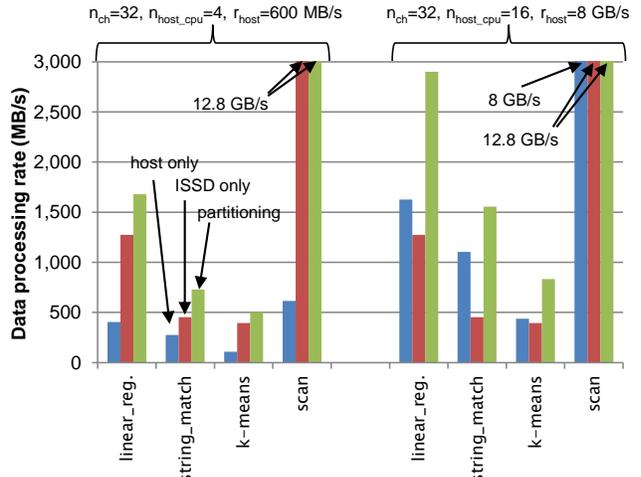


Figure 9: Performance of host only, iSSD only, and partitioning based combined configuration.

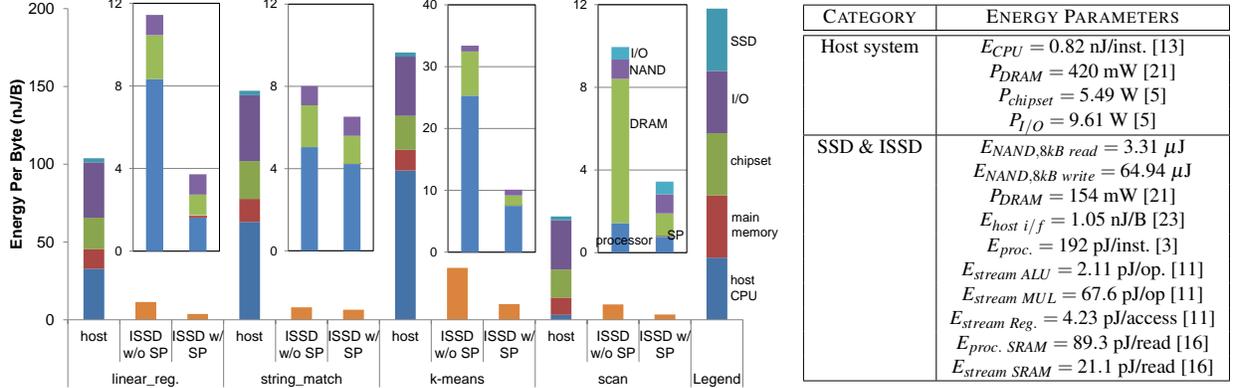


Figure 10: Breakdown of energy into key system components (left) and input parameters to the energy model (right).

processing throughput of nearly 1,700 MB/s is achieved with `linear_regression` when the host interface bandwidth is merely 600 MB/s. In case of `scan`, pipelining with the iSSD already achieves the highest possible data throughput (12.8 GB/s) that matches the raw flash memory bandwidth and partitioning brings no additional benefit. In this case, the host CPUs may perform other useful computation or enter into a low power mode.

The partitioning strategy brings practical benefits because partitioning is relatively straightforward when the input data is large and the partitioning overhead becomes relatively small. We are encouraged by our result, and specific research issues of static and dynamic partitioning (and load balancing) are left for future work.

Energy reduction potential. To investigate potential energy savings of the iSSD approach, we compute EPB for three configurations: conventional host processing, iSSD without acceleration, and iSSD with acceleration. The examined configurations follow the setup of Section 5.1 and $n_{ch} = 8$. We break down energy into key system components. In case of host processing, they are: CPUs, main memory, chipset, I/O bus and SSD. For iSSD, we consider: FMC processor, stream processor, DRAM, NAND flash and I/O.

We use the widely practiced event-based energy estimation method (e.g., [6, 5]). We determine our energy parameters based on publicly available information. For example, energy of the host CPU and the iSSD processors are derived from [13, 3, 11]. SRAM scratchpad memory and DRAM energy are estimated using [16] and [21]. Flash memory energy is obtained from data books and in-house measurement. For fair comparison, we assume that the host CPU is built with a 22-nm technology while the SSD/iSSD controllers are built with a 45-nm technology (i.e., two technology generations apart). Figure 10 is the result and the parameters used.

It is shown that overall, the iSSD configurations see large energy reduction in all examined kernels by at least $5\times$ (k-means) and the average reduction was over $9\times$. Furthermore, energy is significantly smaller when the stream processor was used—the maximum reduction was over $27\times$ for `linear_regression` ($9\times$ without the stream processor). The stream processor was very energy-efficient, consuming only 0.123 nJ/byte in the worst case (`string_match`), compared with 63.0 nJ/byte of the host CPU for the same kernel. Clearly, hardware acceleration proves to be effective for both performance improvement and energy saving in iSSDs.

In host processing, typically more than half of all energy is consumed for data transfer (I/O, chipset and main memory). iSSD addresses this inefficiency by migrating computation and eliminating unnecessary data transfer. In this sense, the iSSD approach is in stark contrast with other approaches toward efficient data processing, such as intelligent memory [26], on-CPU specialization [33], GPGPUs [15] and low-power CPUs [2]; they offer more efficient computing than powerful CPUs but do not eliminate the overheads of

data transfer. For example, even if these approaches employ the same energy-efficient reconfigurable stream processors, their energy improvement would be limited to $1.04\times$ to $1.60\times$ (when iSSD achieves $17\times$ to $27\times$ with acceleration).

Cost analysis. Finally, we estimate and compare the hardware cost of an SSD and an iSSD. Standard components like flash and DRAM chips are identical in both designs. With that the main cost difference comes from the controller chip. We break down the chip cost in the right table and estimate the die cost difference to be 40%.

The SSD bill of material (BOM) is dominated by the NAND flash chips. For example, commodity 256 GB SSDs are priced at as low as \$370 in on-line stores as of July 2011, whereas 64 Gbit MLC NAND flash chips are sold at \$9.92 (obtained at www.dramexchange.com on July 13, 2011). According to this price, the cost for the flash memory chips is \$317, accounting for over 86% of the SSD end user’s price. With a larger capacity SSD, it won’t be surprising that the flash cost exceeds 90% of the drive’s BOM. Hence, even if the dominant portion of the remaining cost—around 10% of the drive’s BOM—is attributed to the controller chip, *the additional cost of iSSD will not exceed 4%*. We believe that this cost is substantially smaller than upgrading the host system (e.g., CPUs with more cores) to get the same performance improvement the iSSD can bring and is well exceeded by the expected energy savings.

CHIP COMPONENT	SSD	iSSD
CPUs	700K (int core \times 2)	2,200K ((int+fp) \times 4)
SRAM (256 KB)	1,500K	1,500K
FMC (incl. ECC, 16 channels)	7,200K (450K \times 16)	11,200K (700K \times 16)
PCI-e (mac only)	500K	500K
DRAM controller	50K	50K
Peripherals	60K	60K
Misc. (glue, pads, ...)	2,000K	2,000K
TOTAL	12,010K	17,510K

6 Conclusions

Intelligent solid-state drives (iSSDs) offer compelling performance and energy efficiency benefits, arising from modern and projected SSD characteristics. Based on analytic models and limited experimentation with a prototype iSSD, we show that iSSDs could provide 2–4 \times higher data scanning throughput and 5–27 \times better energy efficiency relative to today’s use of commodity servers. Based on analysis of ten data-intensive application kernels, we describe an architecture based on reconfigurable stream processors (one per internal flash memory channel) that could provide these benefits at marginal hardware cost increases (<4%) to traditional SSDs. Most of these benefits would also be realized over non-iSSD approaches based on efficient processing outside of SSDs, because they do not exploit SSD-internal bandwidth or avoid costly high-bandwidth transfers of all processed data.

References

- [1] ACHARYA, A., UYSAL, M., AND SALTZ, J. Active disks: programming model, algorithms and evaluation. ASPLOS-VIII, pp. 81–91.
- [2] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. Fawn: a fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), SOSP ’09, ACM, pp. 1–14.
- [3] ARM LTD. Cortex a9 processor. <http://www.arm.com/products/processors/cortex-a/cortex-a9.php>.
- [4] BARROSO, L., DEAN, J., AND HOLZLE, U. Web search for a planet: The google cluster architecture. *IEEE Micro* 23, 2 (March-April 2003), 22–28.

- [5] BIRCHER, W., AND JOHN, L. Complete system power estimation: A trickle-down approach based on performance events. *ISPASS*, pp. 158–168.
- [6] BROOKS, D., TIWARI, V., AND MARTONOSI, M. Wattch: a framework for architectural-level power analysis and optimizations. *ISCA '00*, pp. 83–94.
- [7] BRYANT, R. E. Data-intensive supercomputing: The case for disc. Tech. Rep. CMU-CS-07-128, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 2007.
- [8] CANNATARO, M., TALIA, D., AND SRIMANI, P. K. Parallel data intensive computing in scientific and commercial applications. *Parallel Comput.* 28 (May 2002), 673–704.
- [9] CARBON SOC DESIGNER PLUS. <http://carbondesignsystems.com/SocDesignerPlus.aspx>.
- [10] COOPER, B. F., BALDESCHWIELER, E., FONSECA, R., KISTLER, J. J., NARAYAN, P. P. S., NEERDAELS, C., NEGRIN, T., RAMAKRISHNAN, R., SILBERSTEIN, A., SRIVASTAVA, U., AND STATA, R. Building a cloud for yahoo! *IEEE Data Eng. Bull.* 32, 1 (2009), 36–43.
- [11] DALLY, W. J., KAPASI, U. J., KHAILANY, B., AHN, J. H., AND DAS, A. Stream processors: Programmability and efficiency. *Queue* 2 (March 2004), 52–62.
- [12] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In *OSDI* (2004), pp. 137–150.
- [13] GROCHOWSKI, E., AND ANNAVARAM, M. Energy per instruction trends in intel microprocessors. *Tech.@Intel Magazine*, March 2006.
- [14] HALEVY, A., NORVIG, P., AND PEREIRA, F. The unreasonable effectiveness of data. *IEEE Intelligent Systems* 24 (March 2009), 8–12.
- [15] HE, B., FANG, W., LUO, Q., GOVINDARAJU, N. K., AND WANG, T. Mars: a mapreduce framework on graphics processors. *PACT '08*, pp. 260–269.
- [16] HP LABS. Cacti 5.3. <http://quid.hpl.hp.com:9081/cacti/>.
- [17] HUSTON, L., SUKTHANKAR, R., WICKREMESINGHE, R., SATYANARAYANAN, M., GANGER, G. R., RIEDEL, E., AND AILAMAKI, A. Diamond: A storage architecture for early discard in interactive search. *FAST*, pp. 73–86.
- [18] KEETON, K., PATTERSON, D. A., AND HELLERSTEIN, J. M. A case for intelligent disks (idisks). *SIGMOD Record* 27, 3 (1998), 42–52.
- [19] KHRONOS GROUP. Opencl. <http://www.khronos.org/opencl/>.
- [20] MESNIER, M., GANGER, G. R., AND RIEDEL, E. Object-based storage. *IEEE Communication Magazine* 41 (August 2003), 84–90.
- [21] MICRON TECHNOLOGY, INC. Sdram power calculator. http://download.micron.com/downloads/misc/SDRAM_Power_Calc_10.xls.
- [22] NARAYANAN, R., OZISIKYILMAZ, B., ZAMBRENO, J., MEMIK, G., AND CHOUDHARY, A. Minebench: A benchmark suite for data mining workloads. *IISWC*, pp. 182–188.
- [23] OPEN COMPUTE PROJECT. <http://opencompute.org>.

- [24] OSKIN, M., CHONG, F., AND SHERWOOD, T. Active pages: a computation model for intelligent memory. In *Computer Architecture, 1998. Proceedings. The 25th Annual International Symposium on* (June 1998), pp. 192–203.
- [25] OZISIKYILMAZ, B., NARAYANAN, R., ZAMBRENO, J., MEMIK, G., AND CHOUDHARY, A. An architectural characterization study of data mining and bioinformatics workloads. IISWC, pp. 61–70.
- [26] PATTERSON, D., ANDERSON, T., CARDWELL, N., FROMM, R., KEETON, K., KOZYRAKIS, C., THOMAS, R., AND YELICK, K. A case for intelligent ram. *Micro, IEEE* 17, 2 (March/April 1997), 34–44.
- [27] RIEDEL, E. *Active Disks—Remote Execution for Network-Attached Storage*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1999.
- [28] RIEDEL, E., FALOUTSOS, C., GIBSON, G., AND NAGLE, D. Active disks for large-scale data processing. *Computer* 34, 6 (June 2001), 68–74.
- [29] RIEDEL, E., GIBSON, G. A., AND FALOUTSOS, C. Active storage for large-scale data mining and multimedia. VLDB '98, pp. 62–73.
- [30] SAMSUNG ELECTRONICS CO., L. http://www.samsung.com/global/business/semiconductor/Greenmemory/Products/SSD/SSD_Lineup.html.
- [31] SCHUETZ, R., AND JEONG, S. Looking ahead to higher performance ssds with hlnand. In *Flash Memory Summit* (2010).
- [32] TPC. Tpc-h. <http://www.tpc.org/tpch/default.asp>.
- [33] VENKATESH, G., SAMPSON, J., GOULDING, N., GARCIA, S., BRYKSIN, V., LUGO-MARTINEZ, J., SWANSON, S., AND TAYLOR, M. B. Conservation cores: reducing the energy of mature computations. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems* (New York, NY, USA, 2010), ASPLOS '10, ACM, pp. 205–218.
- [34] WHITE, T. *Hadoop: The Definitive Guide (Chapter 6: How MapReduce Works)*. O'Reilly, 2009.
- [35] WONG, W. A chat about micron's clearnand technology. *electronic design* (December 2010).
- [36] YOO, R. M., ROMANO, A., AND KOZYRAKIS, C. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. IISWC, pp. 198–207.