Active SSD Design for Energy-efficiency Improvement of Web-scale Data Analysis

Jian Ouyang[1], Shiding Lin[1], Zhenyu Hou[1], Peng Wang[2], Yong Wang[1], Guangyu Sun[2],
[1]Baidu, Inc.
[2]Center for Energy-efficient Computing and Applications, Peking University
{ouyangjian, linshiding, houzhenyu, wangyong03}@baidu.com, {wang_peng, gsun}@pku.edu.cn

***Abstract*— NAND flash based solid state drives (SSDs) have been widely adopted as storage devices in modern data centers to provide high performance I/O services. Recently, researchers proposed several schemes to improve energy efficiency of the system by off-loading specific computation tasks from generic processors to local processing elements in SSD controllers. However, it is inefficient to directly apply these approaches to the web-scale data analysis system equipped with modern SSDs using FPGA based controllers. More important, the design schemess proposed in prior work cannot work with our target system. In order to overcome the limitation, we present our Active SSD design, considering unique features of computation tasks in web-scale data analysis. In addition, we address an important issue about interference between normal data processing and local computation in Active SSDs. The detailed architecture of our Active SSD is described, and a prototype is implemented. Moreover, the modification to the whole system is also introduced to enable the Active SSD. Experimental results based on real applications show that the energy efficiency can be significantly improved with our design.**

***Keywords*— Active SSD, web-scale data analysis, FPGA controller**

I. Introduction

Compared to traditional hard disk drives (HDDs), NAND flash based SSDs have many advantages, such as high throughput, low power consumption, light weight, etc. SSDs have been widely adopted as storage devices in various systems ranging from portable devices to high performance clusters [9], [10], [5], [13]. Recently, as the per-bit price of NAND flash keeps decreasing, SSDs have also been employed in enterprise-scale data-centers to provide high throughput, low latency data access for web-scale data analysis. For example, some famous companies, such as Amazon, Baidu, and Dropbox, etc., have started using SSDs in their data centers.

Besides the advantage of high performance, prior research has shown that SSDs can provide energy-efficient computing capability[5], [13]. The basic idea is to offload some specific computing tasks from generic processors of the system to the processing elements inside SSDs. Boboila *et al* first proposed Active Flash to accelerate high performance computing (HPC) by leveraging a single active SSD device [5]. Later in Tiwari's work, an analytical model is presented to find out the solution of using multiple Active Flash devices for high-end computing machine under both performance and power constraints [13]. Cho *et al* proposed a design called intelligent SSDs (iSSDs) to enable execution of limited applications functions of Hadoop System [14] on SSDs using an extra streaming processor [6]. Abbani *et al* presented a reconfiguration model of using active SSD platform for data intensive workloads [3]. In the rest of this work, we call all these types of SSD designs, which provide computing capability with their internal hardware, as **Active SSDs** to simplify discussion.

The benefits of using Active SSDs come from three folds. First, since data are processed locally on SSDs, the energy consumption of moving data between SSDs and generic processors can be saved. Second, the processing elements in SSDs can achieve better computation energy-efficiency because they are normally based on low power designs [2]. Third, the I/O contention can be alleviated by leveraging internal high bandwidth inside SSD. However, these approaches are inefficient for systems running web-scale data analysis. They have limitations from both hardware design and task selection, which are discussed as follows.

First, most processing elements in these Active SSDs are based on embedded processors (e.g. ARM cores). The computation tasks are executed either on the embedded processors [5], [13] directly or on the extra hardware controlled by them [6]. For the former case, besides the ISA limitation, the tasks executed on these low-performance processors need to be carefully selected to avoid significant system performance degradation. For the later case, extra hardware (e.g. stream processors) is required. Both design complexity and cost of SSDs can be increased.

Second, the processing elements on Active SSDs are not optimized for tasks. The overhead of running tasks on Active SSDs is not fully considered. For example, some HPC algorithms running on Active SSD contain tens of even hundreds of instructions [13]. These codes have to be copied to local memory of Active SSD before being executed. In addition, the problem of interference among normal data processing and task execution on Active SSDs is not addressed. For example, the local computation on Active SSDs can interfere with some routine functions, like garbage collection and wear leveling. The interference can result in unpredictable delay of response time.

Third, the design scheme based on timing modeling in prior approaches cannot work with Active SSDs for web-scale data analysis. In our design, data processing bandwidth rather than computation latency is more critical because of the unique features of tasks.

Fourth, there is no clear discussion on how to adapt the whole system to support the Active SSD, especially in OS and application level. Apparently, an API must be provided to programmer, and the compiler and OS need to understand the commands so that these request can be sent to Active SSD.

In order to overcome these limitations, we propose an Active SSD design enhanced with processing elements in FPGA-based SSD controller. Both the processing hardware and tasks selection are optimized for applications of web-scale data analysis. The contribution of this work is concluded as follows.

- We leverage the computing capability of FPGA in modern SSD to achieve significant high energy-efficiency for computation in Active SSD.
- We select proper computation tasks, which are dominating in web-scale data analysis, for our Active SSD.
- We proposed a simple but effective design strategy based on feature of tasks, considering different constraints. Active SSD can be easily reconfigured according to computation pattern.
- We demonstrate that, in our Active SSD, the interference

This work was supported by the National Natural Science Foundation of China (No. 61202072) and National High-tech R&D Program of China (No. 2013AA013201). Corresponding author: Guangyu Sun.

between normal data access and computation on Active SSD is avoided without extra design overhead.
- A prototype of our Active SSD is implemented on processing nodes in a real data center.
- A comprehensive evaluation is presented in difference levels of scenarios. Experimental results show significant improvement of energy-efficiency.

The rest of this paper is organized as follows. Section II provides a brief introduction to the background of web-scale analysis and SSDs equipped FPGA based controllers. We present detailed design of our Active SSD in Section III, which includes both hardware architecture and corresponding system support. The experimental setup and evaluation results are presented in Section IV, followed by a conclusion.

II. Preliminaries

In this section, we first present a brief introduction to web-scale data analysis and FPGA based SSD controller. Then, we provide the definition of some terminologies that are used in this work.

A. Web-scale Data Analysis

Web-scale data analysis is important for modern internet services. The analysis is usually employed to extract some information or features from huge volume of data, such as queries and logs. A typical example is to find out the top 10 search queries in a period. These applications have the following common features:

- The data scale is huge (e.g. hundreds of PetaByte), and bandwidth requirement is high.
- Many computation tasks are simple and can be efficiently executed on FPGA design.
- The type of tasks are diverse among applications, but one or two types can dominate in a single application.
- Computation intensity is very high so that systems seldom enter the idle mode.

B. FPGA based SSD controller

In consumer-level storage products, SSDs usually adopt the traditional host-to-disk interface, such as SATA, for compatibility reasons. The embedded ARM processor in the SSD controller is powerful enough to deal with the maximum throughput of the interface [12]. However, in the environments that require high bandwidth, such as in systems of web scale data analysis, the overall bandwidth of the traditional host-to-disk interface is a bottleneck. The high-end SSDs for enterprise servers [1] in the market leverage the PCI-E interface with gigabytes of bandwidth. PCI-E SSDs provide lower latency path and offer higher throughput compared to SATA SSDs. In addition, the adapter form factor of PCI-E SSDs allots more space for NAND flash and associated controllers, maximizing both SSD performance and available capacity [1], [12]. Current PCI-E SSD solutions are primarily based on FPGA [11]. In addition, the reconfiguration capability of FPGA is important to adopt Active SSDs for applications with diverse computation tasks.

C. Terminology Definition

In order to simplify discussion, we define following terminologies in this work. More examples of kernel and task can be found in Table I.

- **Host**: The computer node equipped with Active SSDs.
- **Computation Engine**: The whole design entity used for data processing on Active SSD.
- **Kernel**: Refer to basic functions can be run on FPGA implementation, such as *sum(), max()*, etc.

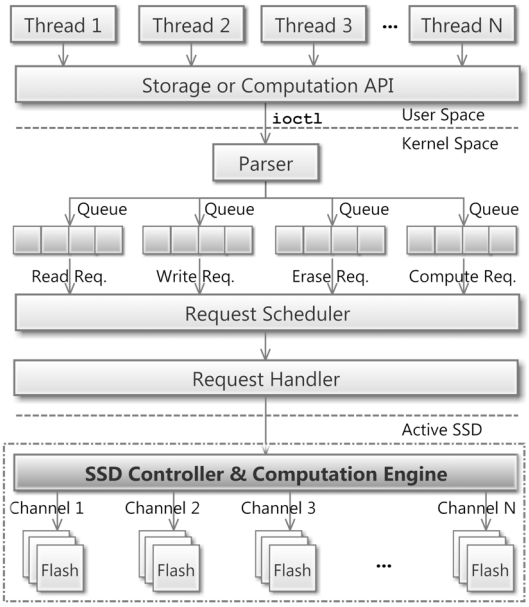

Fig. 1. Illustration of the system structure.

- **Task**: Refer to the unit of load that can be executed on either host or Active SSD. A task is composed of several kernels.
- **PE**: Refer to processing element on Active SSD, which is a functional component that can execute a complete task.

III. Active SSD Design

In this section, we first introduce system level workflow with Active SSD. Then, we present the detailed design and discuss some related design issues.

A. System Level Overview

Figure 1 illustrates the overall structure of a system that supports Active SSD. In this figure, the modification to both user space and kernel space, which includes user API, I/O command parser, system request queues, etc., are illustrated. The details of these modification and the flow of issuing tasks on Active SSDs are introduced as follows.

Threads $1-N$ represent applications of web-scale data analysis running simultaneously on the host. Each of these threads can issue requests of either normal data read/write or computation tasks on host and Active SSDs. In order to explicitly issue tasks to Active SSDs, we provide a user-friendly computation API similar as those for normal data accesses. The format of API is shown below. Note that these APIs are tailored for SSDs in data center of Baidu.

```
Compute_Op(uint64 src_addr0, uint64 len0,
           uint64 src_addr1, uint64 len1,
           uint64 dst_addr0, uint64 dst_len0,
           uint64 dst_addr1, uint64 dst_len1,
           uint32 cmd, datatype para0,
           [para1], ...)
```

The different parts in the function is explained as follows,

1) Source Address (src_addr): the starting address of source data on Active SSD.
2) Data Length (len): the length of data for computation on Active SSD.
3) Task (cmd): the type of task for processing.
4) Destination Address (dst_addr): the output address on Active SSD.

5) Parameter (para): this region is used to pass parameters to computation engine, such as datatype, which can be one of following, *int*, *float*, *double*, etc.

With the help of this API, functions called for computation on Active SSDs are translated into I/O commands (e.g. *ioctl*) and sent to a *Parser*, as shown in the figure. The *Parser* is responsible for decoding these I/O commands into low level requests of Active SSDs. Then, these requests are inserted into corresponding queues for further kernel level scheduling by a request scheduler in operating system. Note that there are various scheduling methods employed in modern systems and specific optimization can be applied to coordinate different requests [4], [8]. The discussion is out of scope of this work, since we focus on the design of Active SSD. After these requests are scheduled for processing, they will be sent to *Request Handler* in the device driver of Active SSD, as illustrated in Figure 1. Then, the driver will process these requests to Active SSD by writing corresponding registers in the Active SSD.

The lowest region of Figure 1 is the abstract of our Active SSD design, which is composed of two main parts. The first one is the storage part based on many NAND flash chips. The second part includes SSD controller and computation engine implemented with FPGA. The SSD controller is responsible for basic functions, such as normal data processing, FTL, error correction, etc. The computation engine handles execution of tasks on Active SSD. Note that these computation engines can be reconfigured on demand to work with different applications. The details about reconfiguration will be discussed in Subsection III-C.

B. Active SSD Architecture

The architecture of our Active SSD is demonstrated in Figure 2. The working flows of normal data access and local computation on Active SSD are described separately as follows.

Normal Data Access Flow. The normal data access process is the same as that of traditional SSD. We take the data storage process as an example. When data in main memory is ready to be written back, the request is initialized by DMA controller (③ in Figure 2). The data fetched by DMA controller are sent on data bus connected to DRAM buffer (④ and ⑤ in Figure 2). DRAM buffer is employed to improve access throughput. Data from DRAM buffer are checked with error correction component (e.g. BCH ⑥ in Figure 2) then input into FTL (⑦ in Figure 2). The physical block address to be written is generated by FTL. Finally, data are sent to the corresponding channels and updated in NAND flash chips. For a read request, the process is a reversed one of a write request.

Local Computation Flow. In order to simplify design of datapath, the data fed to FPGA for local computation on Active SSD are all from NAND flash chips. Consequently, if some data for computation are in main memory, the host processor need to issue a write request to write them back to SSD before being processed. This scenario is really rare for web-scale data analysis. Data in these applications is usually organized as a batch (e.g. large data block) and is written back to SSD after being processed by host processors. As shown in Figure 2, the *computation engine* is also connected to FTL to fetch data from NAND flash chips. The data fetching is similar to a read operation. Data fetched go through FTL and BCH before being allocated in DRAM buffer. Then, data are sent from DRAM to *computation engine* instead of DMA controller.

One important design issue is the interference between local computation and other functions of Active SSD. For example, Active SSD is also responsible for the normal data access requests from host processors. Note that we focus on interference on processing logic instead of flash chip/channel competition. More important,

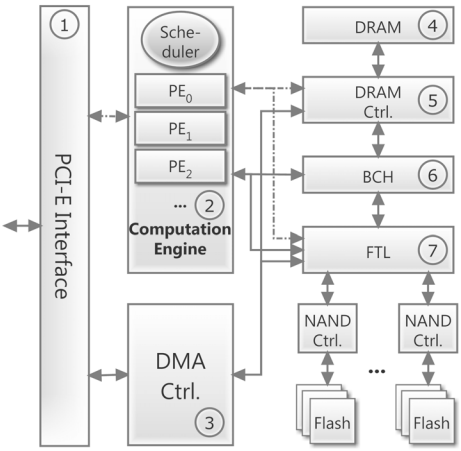

Fig. 2. Active SSD Architecture.

the local computation should not interrupt some routine functions, such as garbage collection and wear-leveling, to avoid severe performance degradation. Fortunately, our FPGA-based Active SSD solve these problems without introducing extra design effort.

First, when computation task is offloaded to Active SSD, the computation engine leverages the high data bandwidth inside SSD to save the load of moving data between storage and host processors. To this end, the local computation can help reduce pressure on host I/O. On the other side, computation engine needs all data to be allocated on NAND flash before being processed. As we mentioned, the extra I/O requests caused by such operation are trivial because the operations are really rare. Second, different from prior Active SSD approaches based on embedded ARM processor, computation engine in our design is isolated from routine logic functions of normal SSDs. As shown in Figure 2, computation engine can only access data on NAND flash chips through FTL component. From the perspective of FTL, there are no difference for requests from computation engine or DMA (e.g. host processor). Thus, when routine functions are running, it can block requests from both sources so that these functions are not interrupted. Note that, if one computation request has already been issued before them, these routine functions need to wait for completion of the task before being processed.

Computation Engine. The structure of a computation engine is illustrated in Figure 3. It is composed of a task scheduler and several computation processing elements (PE). The task scheduler is responsible for scheduling tasks to corresponding PEs. The scheduling policy is a first-come-first-serve policy. Such a simple scheduling works efficiently with Active SSD design because only one type of task is scheduled for most of cases. In order to achieve high throughput of computation engine, local computation tasks can be scheduled in parallel to fully utilize the computation kernels. Note that the data hazard is handled by host processor instead of scheduler in computation engine. It means that computation requests running at the same time on Active SSD have no data dependency.

Apparently, the computation throughput is related to both application and design of PEs in computation engine. As we mentioned, the types of PE may be diverse for different applications. Thus, the goal is to design the optimized number of PE for each type of task with the consideration of different constraints, such as FPGA resource constraint, data bandwidth constraint, etc. We discuss our strategy of PE design and how to reconfigure them for different applications in the next subsection.

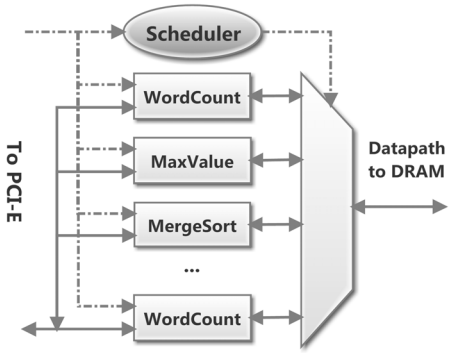

Fig. 3. Structure of a Computation Engine.

C. Computation Engine Design

In this subsection, we first list several common computation kernels that can be implemented efficiently on Active SSDs. Then, we propose a strategy of designing PEs using these kernels. At last, we discuss how to reconfigure them for different applications.

Since we focus on web-scale data analysis, the design of computation kernels inside the computation engine should be adapted based on two basic rules. First, the implementation of these kernels on FPGA must be feasible. Second, the FPGA implementation of these kernels can achieve significant higher energy-efficiency compared to execution on host processors. The purpose of these rules is to make sure that offloading these tasks on Active SSD can have an significant impact on energy efficiency of computation in the system.

With a comprehensive analysis of various applications, we propose several commonly used computation kernels and can be implemented on FPGA directly with direct logic synthesis, which are listed as follows.

- ***sum()*** This is a basic kernel for addition and counting.
- ***memcmp()*** This is a kernel used for text comparison, which is widely used in some workloads, such as *WordCount*. When implemented on FPGA, it can compare multiple bytes in parallel to speed up the performance.
- ***max(), min()*** These two kernels are used to find maximum and minimum values in a set of data.
- ***merge()*** This kernel is used to merge two sets of data into one set and is commonly used in workloads, such as *MergeSort*.

In the real design, multiple computation kernels compose a PE for execution of different tasks. For example, the PE for task *WordCount* is composed of several *memcmp* kernels and one *sum* kernel. In order to simplify the design, all processing elements are designed as combinational logic. It means that data go through from input to output as a streaming without any iteration. This rule limits the tasks that can be supported by Active SSDs. Fortunately, this is not a problem for our targeting applications because most tasks in web-scale data analysis can be executed on PE of combinational logic design. For example, the abstract logic design of *WordCount* is illustrated in Figure 4. In this example, the PE can compare N queries at the same time with N *memcmp* kernels. Each kernel contains eighteen M-bit comparators which support different length of queries. The counter shown in Figure 4 is based on kernel *sum*.

Having these PEs, we propose a simple but effective design strategy for computation engine design. The design goal is to achieve maximum computation throughput under constraints of both FPGA resources and SSD internal I/O bandwidth. Our design strategy is based on three simplifications, considering the features of targeting applications.

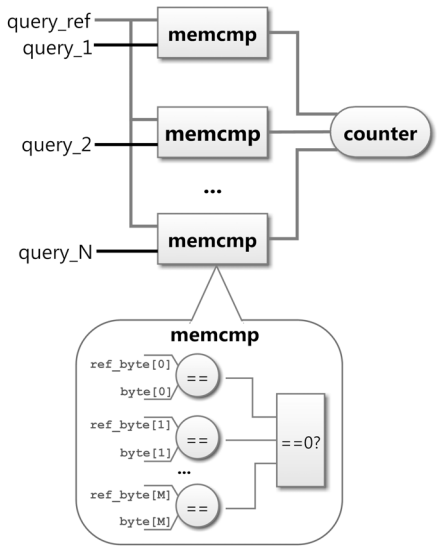

Fig. 4. Illustration of PE design for WordCount.

1) There are always enough computation tasks for both Active SSDs and host processors. This simplification is reasonable for application of web-scale data analysis, especially for those that generate huge data volume.
2) There is only one type of processing element active for computation at the same time. This simplification is also based on the character of applications. For example, in programming model like "map-reduce", the computation task allocated to one node normally contains a single type of task, such as WordCount, MergeSort, etc.
3) The average bandwidth requirement of host processors is kept in a stable level. This one is related to the first simplification. Since there are always enough commutation tasks for host processors, the data requirement bandwidth (B_{host}) can be assumed as the case that the processors running at the peak performance. Although B_{host} can vary during execution, in Section IV, our experimental results show that this simplification works well with our Active SSD design.

With these simplifications, the design goal is just to maximize the total computation throughput, as shown in Equation (1) to Equation (3). The details are explained as follows.

Assume that there are N types of PE designed for N types of tasks. Each element M_i in the set $< M_0, M_1, M_2, \ldots, M_N >$ represents the total number of PE instances implemented for i_{th} type of PE. The total throughput of executing task i can be calculated as $M_i \times T_i$, where T_i denotes the raw throughput of i_{th} type of PE. We further assume that the percentage that the i_{th} task occupies in the total computation on Active SSD is introduced as P_i. Thus, based on our simplifications, the total throughput T_{total} can be calculated as in Equation (1).

$$T_{total} = \sum_{i=1}^{N} P_i \times M_i \times T_i \tag{1}$$

There are several potential constraints that can limit computation engine design. The first one is resource constraint on FPGA. Note that in the real design, we normally use the spare resource of a traditional SSD controller. As shown in Equation (2), the first part on the left side of the equation represents the total resource used by all PE logic, where R_i denotes the resource for i_{th} type PE. R_{per} counts the necessary resource for peripheral circuitry such as connection, scheduler, and input/output circuit to DRAM and

TABLE I
LIST OF TASKS

Task Name	Kernel	Task Description
WordCount	Memcmp()	Given a word (e.g. IP or URL), count the total word number appears in data.
MaxValue	Memcmp() Max()	From a list of $< k, v >$ pairs, find out the maximum v for the same k. Key is already sorted.
SumValue	Memcmp() Sum()	From a list of $< k, v >$ pairs, calculate the sum of v for the same k. Key is already sorted.
MergeSort	Merge()	Merge sort algorithm

PCI-E. R_{FPGA} is total available resource that constrains the total processing power of an Active SSD. Later in Section IV, we will show that the resource constraint is not a critical limitation.

$$\sum_{i=1}^{N} M_i \times R_i + R_{per} < R_{FPGA} \quad (2)$$

The second limitation comes from the data bandwidth that can be provided by DRAM on Active SSD. Assume that the peak bandwidth of DRAM is B_{DRAM}, which is the constraint data bandwidth consumed by both host and computation engine on Active SSD. Let B_i denote bandwidth required by i_{th} type of PE, the second constraint is shown in the following equation. Note that the host bandwidth is assumed as a stable value based on our third simplification. According to our second simplification, there is only one type of PE active at the same time. Thus, peak bandwidth required by PE is calculated as $max(M_i \times B_i)$. Later in Section IV, we will demonstrate that the second constraint has an impact on the design of Active SSD.

$$max(M_i \times B_i) + B_{host} < B_{DRAM} \quad (3)$$

When different applications are running on the same system, the pattern of computation requests can vary a lot from each other. The most attractive feature of FPGA is that it can be reconfigured to optimize computation engine for different applications. Prior research has mentioned the reconfiguration of Active SSDs [3]. Our design also support reconfiguration, which also follows the design strategy based on equation (1)(2)(3). It should be mentioned that the process of reconfiguration only consumes several minutes and can be operated during period of light load. Thus, compared to the total data processing time, its impact on total system performance can be neglected.

IV. EVALUATION

In this section, we will first introduce our experimental setup and our design prototype. Then, the experimental results are listed and compared.

A. Experimental Setup

Our experiments are evaluated on nodes (a.k.a. host machines), in the real data center system from Baidu. Note Baidu is the largest search engine company in China that provides various internet services. The targeting applications are running on a platform similar to Hadoop [14]. The workload allocated to the evaluation node is based on request traces captured in the real system. In the workload, we extract the following tasks that can be executed on our Active SSD, as listed in Table I. For the workload we use for experiments, the first three tasks occupy about 35% of the total computation tasks, and the last task (MergeSort) consumes about another 10% of the total tasks. Thus, these tasks that can be executed on Active SSD contribute more about half to the total execution. It means that the energy-efficiency improvement of these tasks has an important impact on that of the whole system.

The host processors are two Intel E5620 [7] 2.4GHz including 4 cores on two slots of the motherboard. The processor supports SSE instruction set, which can be used to speed up computation kernel such as *memcmp()*. The size of main memory is 32GB. The host is equipped with eleven 2TB HDDs with the SATA interface. The operating system running on host is Linux with a customized kernel, which is modified to support computation on Active SSD. In addition, the PCI-E driver is also customized for Active SSD. The host machine can be connected to a fully customized SSD design, working as a prototype of our Active SSD. The details of Active SSD prototype is introduced in the next subsection.

B. Prototype Design

Fig. 5. Prototype of Active SSD.

The prototype of our Active SSD is shown in Figure 5. There are four Spartan-6 150T FPGAs [15] on the board. Note that the four low-end FPGAs are selected for the consideration of design cost. Each FPGA has a 4-channel DDR3 with the size of 2GB, which can provide 4×1.6 GB/s bandwidth. Since there are four FPGAs, the peak total DRAM buffer bandwidth (B_{DRAM} of the Active SSD is $4 \times 4 \times 1.6$ GB/s = 25.6GB/s. The internal NAND flash chips can provide 2.6GB/s and 1.25GB/s bandwidth for read and write operations, respectively. The total capacity of NAND flash is 700GB. The bandwidth of PCI-E to host is 2GB/s.

The SSD controller is customized to provide proper FTL for Baidu systems. For the design without computation engine, about 60% of total LUT is used. It means the rest 40% LUT is free resource that can be used for computation engine design.

C. Experimental Results

We evaluate our Active SSD in three different cases: (1) single node case, (2) distributed system case, and (3) data-center level case. In the first case, the Active SSD is connected to a single processing node and the performance and power results are collected through measurement. In the second case, the experimental results are based on injected traces captured from distributed system. For the last one, the benefits of using our Active SSD are estimated by scaling to a real data center environment.

The results for single node evaluation are listed in Table II. For Active SSD design, we first list theoretical peak bandwidth of each PE without any constraints. In the third column, we present the number of PEs implemented in the Active SSD. We can find that the total resource consumed is less than 6% of FPGA resource. It means that the design overhead is trivial. The PE design follows our design strategy in Subsection III-C. The number of each type of PE is decided by the constraint of DRAM bandwidth. From the fifth column, we can find that each type of PE can achieve almost the maximum bandwidth of DRAM buffer. The performance results for each task are also listed in the table. The results show that the power consumption for each type of PE is less than 1 Watt.

TABLE II
EXPERIMENTAL RESULTS ON A SINGLE HOST

	Active SSD				Host Only		
Task Name	Peak Throughput per PE (no constraints)	Number of PE FPGA resource	Total Throughput (DRAM constraint)	Power (Watt)	Core Throughput	Node Throughput	Power
WordCount	$32GB/s$	1, $< 1\%$	$\approx 24GB/s$	< 1 W	$\approx 4GB/s$	$\approx 24GB/s$	160W / 8 core
MaxValue	$32GB/s$	1, $< 1\%$	$\approx 24GB/s$	< 1 W	$\approx 1.9GB/s$	$\approx 15GB/s$	160W / 8 core
SumValue	$32GB/s$	1, $< 1\%$	$\approx 24GB/s$	< 1 W	$\approx 1.7GB/s$	$\approx 13GB/s$	160W / 8 core
MergeSort	9.6GB/s	4, $< 3\%$	$\approx 24GB/s$	< 1 W	$\approx 1.5GB/s$	$\approx 12GB/s$	160W / 8 core

For comparison, the results of running tasks on host only node are also included in Table II. In the sixth column, the peak computation throughput on a single core is listed for each task. It is easy to find that our PE can achieve much higher performance than host processor. In addition, we also show the total processing throughput that can be provided by all cores on the host. From the last column, it is easy to tell that our PE can improve the energy efficiency by hundreds of times for the single case.

For the evaluation of distributed system case, we need to clarify the total percentage of tasks that can be executed on Active SSD, as shown in Table III. The experiments are based on the trace of $100TB$, and we assume there is 100 Active SSDs and enough host machines so that all data are processed within same execution time for host only configuration and host plus Active SSD configuration. The energy consumption is compared in Figure 6. Note that the energy consumption consumed on network is also estimated for a reasonable evaluation. The results show that, after using Active SSDs, the energy consumption for each workload is significantly reduced. It is easy to find that the improvement is related to the total percentage of task in the workload. For example, the improvement for task *MergeSort* is the lowest because its percentage is also the lowest, as shown in Table III. On average, the total power consumption is reduced by about 63%.

TABLE III
PERCENTAGE OF TASKS ON ACTIVE SSD FOR EXPERIMENTS ON DISTRIBUTED SYSTEM

Task Name	WordCount	MaxValue	SumValue	MergeSort
Percent	$\approx 85\%$	$\approx 70\%$	$\approx 67\%$	$\approx 35\%$

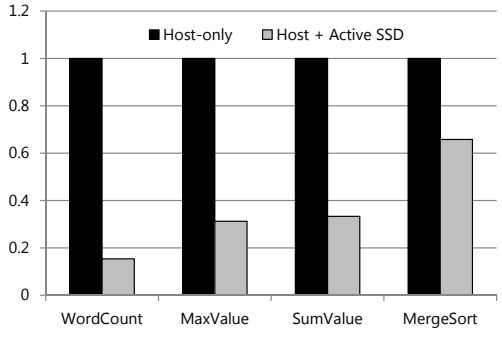

Fig. 6. Energy comparison for distributed mode.

In the last case, we estimate the energy-efficiency improvement based on real workload in Baidu's data center. In the data center, there are about $15,000$ tasks processed everyday. The length of data ranges from a few $KByte$ to several $TByte$. Assume that $10PByte$ total data are processed with *map-reduce* model in a day, and there are $2K$ nodes working. The first three tasks in Table I contribute about 35% to total tasks, and the last one occupies about 10% of the total. As a consequence, we can save about 30% total energy consumption after changing the traditional SSD to Active SSD on each node.

V. CONCLUSION

Active SSD design can help improve computation energy-efficiency of data centers. For web-scale data analysis, the unique features of applications make FPGA based computation engine have more advantages than embedded processor based ones. In addition, the design strategy for FPGA based design is more limited by data bandwidth. More important, the FPGA based design can isolate local computation on SSD from other functions in SSD. Our prototype demonstrates significant improvement of energy-efficiency for the whole system.

REFERENCES

[1] Fusion-io: A New Standard for Enterprise-class Reliability, 2011. http://www.fusionio.com/white-papers/fusion-io-a-new-standard-for-enterprise-class-reliability/

[2] Samsung PM830 datasheet, 2011. http://tinyurl.com/co9zyq7

[3] N. Abbani, A. Ali, D. Al Otoom, M. Jomaa, M. Sharafeddine, H. Artail, H. Akkary, M. Saghir, M. Awad, and H. Hajj. A distributed reconfigurable active ssd platform for data intensive applications. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 25–34, 2011.

[4] J. Axboe. Linux Block IO C Present and Future. In *Proceedings of the Ottawa Linux Symposium*, pages 51–61, 2004.

[5] S. Boboila, Y. Kim, S. Vazhkudai, P. Desnoyers, and G. Shipman. Active Flash: Out-of-core Data Analytics on Flash Storage. In *Mass Storage Systems and Technologies (MSST)*, pages 1–12, 2012.

[6] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger. Active disk meets flash: A case for intelligent ssds. In *Technical Report CMU-PDL-11-115*, 2011.

[7] Intel. Intel Xeon Processor E5620. http://ark.intel.com/products/47925/Intel-Xeon-Processor-E5620-12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI

[8] S. Iyer and P. Druschel. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 117–130, 2001.

[9] Y. Joo, J. Ryu, S. Park, and K. G. Shin. FAST: Quick Application Launch on Solid-State Drives. In *Proceedings of FAST'11*, pages 259–272, 2011.

[10] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting Storage for Smartphones. *Trans. Storage*, 8(4):14:1–14:25, Dec. 2012.

[11] S. Kung. Native PCIe SSD Controllers, 2012. http://www.marvell.com/storage/system-solutions/native-pcie-ssd-controller/assets/Marvell-Native-PCIe-SSD-Controllers-WP.pdf

[12] C. Mellor. OCZ samples twin-core ARM SSD controller, 2011. http://www.theregister.co.uk/2011/07/25/ocz_indilinx_everest/

[13] D. Tiwari, S. Boboila, S. S. Vazhkudai, Y. Kim, X. Ma, P. J. Desnoyers, and Y. Solihin. Active Flash: Towards Energy-Efficient, In-Situ Data Analytics on Extreme-Scale Machines. In *Proceedings of FAST'13*, pages 119–132, 2013.

[14] Hadoop Homepage. http://hadoop.apache.org

[15] Xilinx. Spartan-6 Family Overview, 2011. http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf