Summarizer: Trading Communication with Computing Near Storage

Gunjae Koo*
University of Southern California
gunjae.koo@usc.edu

Kiran Kumar Matam*
University of Southern California
kmatam@usc.edu

Te I
North Carolina State University
ti@ncsu.edu

H. V. Krishna Giri Narra
University of Southern California
narra@usc.edu

Jing Li
University of California, San Diego
jil261@eng.ucsd.edu

Hung-Wei Tseng
North Carolina State University
htseng3@ncsu.edu

Steven Swanson
University of California, San Diego
swanson@cs.ucsd.edu

Murali Annavaram
University of Southern California
annavara@usc.edu

ABSTRACT

Modern data center solid state drives (SSDs) integrate multiple general-purpose embedded cores to manage flash translation layer, garbage collection, wear-leveling, and etc., to improve the performance and the reliability of SSDs. As the performance of these cores steadily improves there are opportunities to repurpose these cores to perform application driven computations on stored data, with the aim of reducing the communication between the host processor and the SSD. Reducing host-SSD bandwidth demand cuts down the I/O time which is a bottleneck for many applications operating on large data sets. However, the embedded core performance is still significantly lower than the host processor, as generally wimpy embedded cores are used within SSD for cost effective reasons. So there is a trade-off between the computation overhead associated with near SSD processing and the reduction in communication overhead to the host system.

In this work, we design a set of application programming interfaces (APIs) that can be used by the host application to offload a data intensive task to the SSD processor. We describe how these APIs can be implemented by simple modifications to the existing Non-Volatile Memory Express (NVMe) command interface between the host and the SSD processor. We then quantify the computation versus communication tradeoffs for near storage computing using applications from two important domains, namely data analytics and data integration. Using a fully functional SSD evaluation platform we perform design space exploration of our proposed approach by varying the bandwidth and computation capabilities of the SSD processor. We evaluate static and dynamic approaches for dividing the work between the host and SSD processor, and show that our design may improve the performance by up to 20% when compared to processing at the host processor only, and 6× when compared to processing at the SSD processor only.

*Gunjae and Kiran contributed equally to the paper.

MICRO-50, October 14–18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3124553

CCS CONCEPTS

• **Computer systems organization → Secondary storage organization**; *Distributed architectures*; *Firmware*;

KEYWORDS

Near Data Processing, SSD, Storage Systems, Dynamic workload offloading

ACM Reference format:
Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. 2017. Summarizer: Trading Communication with Computing Near Storage. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 13 pages.
https://doi.org/10.1145/3123939.3124553

1 INTRODUCTION

Processing large volumes of data is the backbone of many application domains such as data analytics and data integration. In this context the cost of transferring data from storage to compute nodes starts to dominate the overall application performance. Applications can spend more than half of the execution time in bringing data from storage to CPU [38]. In magnetic disk storage systems the medium access time, such as seek and rotational latencies, dominates the data access time. However, with the rapid adoption of solid state non-volatile storage technologies the performance bottleneck shifts from medium access time to the operating system overheads and interconnection bandwidth. As such the prevalent computational model which assumes that storage medium access latency is an unavoidable cost must be rethought in the context of solid state storage. In particular, the computing model must adapt to the realities of bandwidth and OS overheads that dominate storage access.

As modern solid state drives (SSDs) for data centers integrate large DRAM buffers as well as multiple general-purpose embedded cores that are often under-utilized, moving computation closer to data storage becomes feasible. Previous work demonstrated the

potential of using near-data processing models to offload data analytics, SQL queries, operating system functions, graph traversal, image processing, and MapReduce operations [4, 5, 8, 15, 16, 30, 32, 34, 36, 41] to data storage devices. Many of these prior works assume the presence of a commodity general purpose processor near storage. However, even high-end data center SSDs are equipped with wimpy embedded cores. Apart from the cost and power consumption constraints, one reason for the inclusion of only wimpy embedded cores is that they already provide sufficient computing capability to handle most of the current SSD operations, such as protocol management, I/O scheduling, flash translation and wear leveling. For instance, recent SSD controllers that support NVMe protocol and high bandwidth PCIe interconnects may partially utilize only three low-end ARM Cortex R5 cores to support firmware operations [23, 31]. These cores are utilized up to 30% even in the worst case. Hence while there is slack in utilizing the embedded cores, it is not practical to offload large computation kernels to these wimpy cores.

With wimpy embedded cores the choice of computing near storage must be carefully managed. Sometimes it can be advantageous to move computation to the embedded cores to reduce host-storage communication latency by alleviating heavy traffic and bandwidth demands. On the other hand, with wimpy embedded cores in-storage computation requires much longer processing time compared to computing on host processors. Thus there is a trade-off between the computation overhead suffered by wimpy cores and the reduction in communication latency to transfer data to the host system. In this paper we propose Summarizer, a near storage computing paradigm that uses the wimpy near-storage computing power opportunistically whenever it is beneficial to offload computation. Summarizer automatically balances the communication delay with the computing limitations of near-storage processors.

This paper makes the following contributions:

(1) This work proposes Summarizer — an architecture and computing model that allows applications to make use of wimpy SSD processors for filtering and summarizing data stored in SSD before transferring the data to the host. Summarizer reduces the amount of data moved to the host processor and also allows the host processor to compute on filtered/summarized result thereby improving the overall system performance.
(2) A prototype Summarizer system is implemented on a custom-built flash storage system that resembles existing SSD architectures but also enables fine grain computational offloading between the storage processor and host. We enhanced the standard NVMe interface commands to implement the functionality of Summarizer, without changing the NVMe compatibility. Using this prototype, we demonstrate the benefits of collaborative computing between the host and embedded storage processor on the board.
(3) We evaluated the trade-offs involved in communication versus computation near storage. Considering several ratios of internal SSD bandwidth and the host to SSD bandwidth, ratio of host computation power and SSD computation power, we perform design space exploration to illustrate the trade-offs.

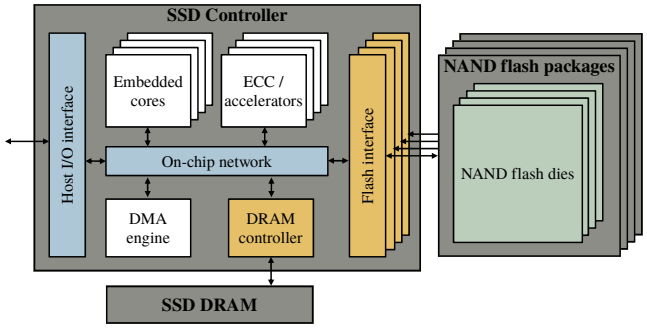


Figure 1: The architecture of a modern SSD

(4) Summarizer dynamically monitors the amount of workload at the SSD processor and selects the appropriate work division strategy among the host processor and the SSD processor. Summarizer's work division approach quantifies the potential of using both the host processor and SSD processor in tandem to get better performance.

The rest of this paper is organized as follows: Section 2 describes the architecture of modern data center SSDs, performance of NVMe for PCIe SSDs and motivates near SSD computing. Section 3 introduces the architecture and the implementation of Summarizer. Section 4 describes the applications used in this paper. Section 5 describes our methodology and implementation details. Section 6 presents the experimental setup and our results. Section 7 provides a summary of related work to put this project in context, and Section 8 concludes the paper.

2 BACKGROUND AND MOTIVATION

Summarizer relies on the processing capabilities embedded in modern SSDs and the NVMe (Non Volatile Memory Express) commands used by the host to interface with SSDs. This section provides a brief overview of SSD architectures, the processing potential near modern SSDs, and the NVMe protocol that the host uses to communicate with the SSD.

2.1 Architecture of modern data center SSDs

Figure 1 depicts the hardware architecture of a modern SSD supporting NVMe protocol via PCI Express (PCIe) bus. Most SSDs use NAND flash memory packages as the primary non-volatile storage elements. Each SSD may contain dozens of flash chips which are organized into multiple channels. Each flash chip can be accessed in parallel and all chips that share a single channel may be multiplexed on that channel. Furthermore, multiple channels can be accessed in parallel. Both the chip and channel level parallelism provides significant internal bandwidth in SSDs. Typical high-end data center SSDs can provide massive internal parallelism with multi-channel topology of flash memory packages and die-stacked fabrication per package [6]. For instance, a commercial NVMe SSD today can support 32 channels of MLC NAND flash memory packages [26] and achieve up to 4.5 GB/s total internal bandwidth [25]. Recent advances in 3D NAND flash technology achieve even higher data bandwidth per package, thus recent SSD systems are capable of providing much higher internal bandwidth [24]. Rather than increasing

the internal bandwidth SSD designers use fewer packages of 3D NAND to maintain the similar internal bandwidth level over many generations. Increasing the internal bandwidth beyond the current level is not very advantageous in SSD platforms since the external bandwidth of even high-end NVMe SSDs currently saturates at less than 4 GB/s due to bandwidth limitation on PCIe lanes.

To summarize we note that although increasing the internal bandwidth of an SSD can be achieved by equipping multiple flash memory channels and advanced fabrication process, extending the external bandwidth between host CPUs and SSDs would require additional PCIe lanes from the processor. In fact the recent move from SATA SSDs to NVMe SSDs in data centers was triggered by the fact that SSD's internal parallelism far exceeds the maximum bandwidth supported by SATA, even though NVMe interface is much more expensive than SATA [27].

To effectively manage the channel parallelism and internal bandwidth, modern SSDs integrate embedded multi-core processors as SSD controllers. These processors handle I/O request scheduling, data mapping through flash translation layer (FTL), wear-leveling, and garbage collection. The controllers connect to flash memory chips through channels and issue flash memory commands to perform I/O operation in each channel in parallel. SSDs also provision a DRAM controller to interface with DRAM which acts as a temporary storage for flash data and also to store the controller data structures. In addition to the embedded cores each SSD may also contain several hardware accelerators to provide efficient error correction code (ECC) processing or data encryption.

The NVMe SSD attaches to the host computer system's PCIe interconnect through a standard PCIe slot or an M.2 slot. The PCIe/NVMe interface fetches I/O commands such as read and write operations from the host CPU and performs Direct Memory Access (DMA) operations between the system interconnect and the SSD. For instance, the host CPU may specify the memory address into which the data must be placed after reading the data from SSD. The NVMe interface then enables a DMA transfer from the SSD controller to the host memory. Current generation NVMe SSDs support dozens of host requests to be queued within the SSD controller's command queues. The controller may schedule these requests so as to maximally utilize the available chip and channel level parallelism.

The write behavior of flash memory is significantly different from that of magnetic memory technologies. Every *page* in flash memory becomes immutable after being written once. The page cannot be updated again without the *block* containing it being fully erased. One block can contain between 64 to 512 pages. Each block can only be erased for limited number of times during its lifetime and block erase operations are significantly slower than page reads and writes. To address these limitations and provide longer lifetime, the SSD implements Flash Translation Layer (FTL) in the SSD controller. The SSD uses general-purpose embedded processors to run the firmware for FTL operations. Basically, the FTL firmware maps the logical block address (LBA) requested from host applications to the physical page address (PPA) in the flash memory chips. This LBA-to-PPA mapping table is cached in the DRAM on the SSD system for faster access. In order to guarantee longer lifetime for entire flash memory cells, the wear-leveling algorithm is also applied for the mapping table management process. In addition, the FTL firmware periodically executes garbage collection (GC) to reclaim space in blocks with invalid pages.

2.2 Potential of in-SSD computing

While SSDs provision multiple embedded processors to improve the performance of SSD controller functions, such as FTL management and garbage collection, much of the compute power in the controller remains under utilized. These underutilized embedded cores provide opportunities for offloading computation from the application. In particular, as we look into the future the processing power of even the embedded cores will continue to grow, even though there is likely to be a large gap in the computing capability of a host processor and the embedded core within each SSD. This section will analyze the potential of using these embedded processors to accelerate applications.

Note that even though the embedded cores are underutilized the primary rationale for provisioning multiple cores is still performance. The firmware can partition the code to enable concurrent execution of different operations such as parsing commands, looking up addresses, locking addresses that are being accessed, and interfacing with flash memory chips. The firmware code also uses several cores to perform garbage collection and wear-leveling in the background. In our lab evaluations concurrently performing 4096 operations in a data center class SSD, the average utilization of these SSD processors was always lower than 30% and there was always at least one processor in idle state. The under utilization remains the same even when performing garbage collection or wear-leveling. While there is plenty of concurrency in FTL operations, each of the concurrent operation itself is relatively simple. Hence, while multiple cores are useful to exploit concurrency, each core's utilization during any given operation remains low. Such under utilization reveals the potential for using existing hardware in these data center-scale SSDs for computation. For example, we can add one more stage to the flash data access pipeline to perform low compute intensity operation without impacting the SSD controller performance. As long as the additional stage is shorter than the current critical operation, the SSD will not suffer from any degradation in throughput.

Among the steps in the flash data access pipeline, we found that the most time consuming step is accessing the flash medium. Even for the fastest flash operation, read operation, a high-performance single-level cell flash memory chip still takes more than 20 microseconds to complete the operation [25]. This delay is even longer for multi-level flash cells. Assuming that the flash interface can access b flash channels simultaneously and the accessed data are evenly distributed among all channels, and if each read operation takes t seconds to complete, the "slack" that allows the computation stage to finish without hurting the throughput is $\frac{t}{b}$. This slack grows even more when using multi-core processors. If each flash page contains p bytes of data, each embedded processor can process n instructions each second, we can obtain the per-byte operation on the file data without affecting the throughput will be $\frac{t \times n}{b \times p}$ when we spare one core for in-storage computing. However, if the SSD is provisioned with more cores or if the number of parallel operations do not reach the peak performance, we may expect much larger slack than this first order estimate. For example, if we can spare

m cores for computation and perfectly parallelize the computation, the operation we can perform on each byte without affecting throughput can be close to $\frac{t \times n \times m}{b \times p}$.

For the MEX SSD controller that Samsung SSDs (such as the EVO range) typically use, each processor core can execute 4×10^8 instructions per second. If we use this processor in an SSD with 32 banks and 8KB flash pages, we find that the per-byte operation must be restricted to only 1 instruction execution per-byte, or 4 operations per 4-byte word.

2.3 NVMe

NVM Express (NVMe) is a protocol for the SSDs attaching to PCIe bus or M.2 interface [2]. NVMe avoids the disk-centric legacy of SATA and SCSI interfaces and leverages PCIe to provide scalable bandwidth. For example, 4-lane Generation 3 PCIe used in datacenter NVMe SSDs supports up to 3.9 GB/sec full-duplex data transfer, while SATA can typically only achieve 600 MB/sec. NVMe also supports more concurrent IO requests than SATA or SCSI by maintaining a software command queue that may hold up to 64K entries for each processor core, and its command set includes scatter-gather data transfer operations with out-of-order completion, further improving performance.

NVMe is highly scalable and capable of servicing multiple I/O requests in parallel. This makes NVMe a good candidate for modern data processing where the application needs to pull enormous amounts of data from secondary storage and feed it into highly parallel computing devices like GPUs.

NVMe supports a streamlined yet powerful set of commands that can initiate and complete I/O operations. Each command has a fixed length of 64 bytes containing information including a command identifier, the logical block address, and the length of the requesting data. An NVMe command can also contain a list of Physical Region Page (PRP) entries which enables scatter-gather data transfers between the SSD and other devices. The PRP entry can specify a list of pairs of base address and offset in host memory corresponding to multiple sub-transfers that the device can execute out-of-order.

3 SUMMARIZER

As detailed in the previous section, there is sufficient compute capacity within the SSD controller to enable modest computation near storage. As SSDs integrate more powerful computing resources then the complexity of near-storage computing can also scale. In this section we describe Summarizer which is our proposed near-storage computing paradigm that automatically scales the near-store computing capability without the need to rewrite the application software with each new generation SSD. We first describe the system architecture and then present our extended NVMe command support provided for the near-storage computation paradigm.

3.1 SSD Controller architecture

Figure 2 illustrates the overall architecture of NVMe SSD controller and also highlights the additional components that are introduced for enabling Summarizer (which are described in detail in the next subsection). The host applications interact with NVMe device through the host-side driver and the SSD controller firmware running on an embedded processor in SSD. The host driver and the

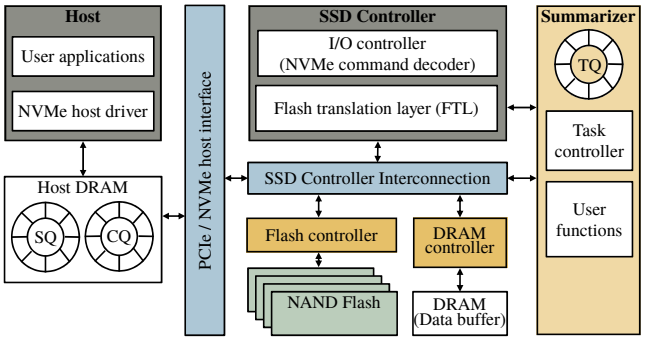


Figure 2: The overall architecture of NVMe controller and Summarizer

controller firmware on NVMe SSD device communicate via PCIe bus. NVMe commands issued by the NVMe host driver are registered in the submission queues (SQs) within the host DRAM space, and the doorbell signal corresponding to the requested command is sent to the SSD controller to notify a new command request from the host.

The major functions of the SSD controller are I/O control and flash translation layer (FTL) processing. The SSD controller receives request commands from the host by reading the registered request from the head of SQ. The SSD controller can fetch host request commands as long as the registered requests exist in SQs. After fetching the NVMe command it is decoded into single or multiple page-level block I/O commands. Each page-level request has a logical block address (LBA), which is translated to a physical page number (PPN) by the FTL processing. The flash memory controller accesses flash memory chips by the page-level commands.

For a NVMe read command the requested page data is fetched from flash memory chips through a series of physical page reads and the fetched data is buffered in the DRAM on the SSD device. Then the page data is transferred to the host memory via the direct memory access (DMA) mechanism. After NVMe command handling completes, the SSD controller notifies the completion of the previously submitted command by registering the NVMe command and its return code in the completion queue (CQ) on the host memory.

3.2 Summarizer architecture and operations

In this section we describe the necessary hardware and software modifications to the SSD controller architecture described above to enable Summarizer. Summarizer can be implemented with some minor modifications to the NVMe command interpreter and a software module added to the SSD controller. We envision majority of *Summarizer* functionality to be implemented in the interface between the SSD controller and the flash memory controller.

Summarizer has three core components: (1) a task queue structure, (2) a task controller module and (3) user function stacks as shown in Figure 2. The task queue (TQ) is a circular queue which stores a pointer to the appropriate user function that must be invoked when the host requests in-SSD processing on a given I/O request. The task controller decides whether in-SSD processing is performed for the fetched page data or not. If the controller decides

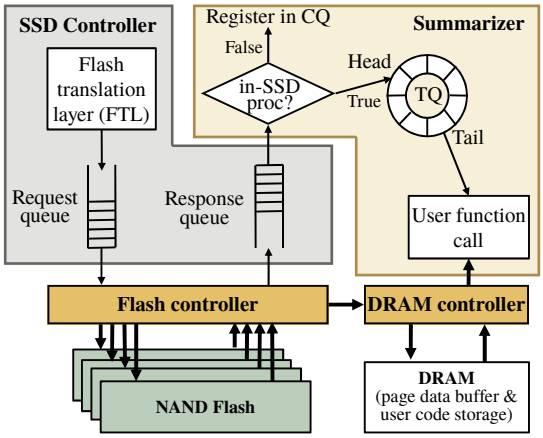


Figure 3: Detailed in-SSD task control used by Summarizer

to execute in-storage computation the target function is executed from the user function stacks. In some cases the task controller may not perform in-SSD processing even if the host requested for such a processing. We explored two different options for how the controller can make such a determination, and these two options are described later.

Summarizer-enabled SSDs allow the host to associate a specific user-defined function to be executed on the SSD controller with every data access request. To enable such an association we define the new NVMe commands to specify *initialization*, *computation* and *finalization* operations. The extended NVMe commands triggering these three steps are listed in Table 1. In the list *n* after TSK represents an identifier of task *TSK*.

Command	Description
INIT_TSK*n*	Initialize variables or set queries
READ_PROC_TSK*n*	Read page data and execute the computation kernel for task *n* with the data
READ_FILT_TSK*n*	Read page data and filter the data by predefined queries
FINAL_TSK*n*	Transfer outputs of FN*n* to the host

Table 1: New NVMe commands to support Summarizer

INIT_TSK*n*: When the host NVMe driver issues INIT_TSK*n* command, the SSD controller calls the initialization function for TSK*n*. This command essentially informs the Summarizer's task controller that the host intends to execute a user-defined task *n* near storage. During the *initialization* step the task's local variables or any temporary data that may be used by that task are initialized.

READ_PROC_TSK*n*: READ_PROC_TSK*n* command resembles the conventional NVMe READ command, except that this command carries information regarding the desired task that may be executed on the SSD controller once a page is read from the flash memory. Note that existing NVMe READ command has multiple reserved bytes that are not used for any processing. We use these unused bytes to specify the task id in the READ_PROC_TSK*n* command. Like the conventional NVMe READ command, the SSD controller issues the read request to flash memory controller to fetch page data. In addition the SSD controller also recognizes that the host is requesting in-SSD processing for this data request and the processing task is specified in the task identifier field (TSK*n*) of the NVMe command itself. This information is tagged with the request. For this purpose the request queue entry carries two additional fields, a 1-bit in-SSD compute flag, and the task id field. These fields are set by the SSD controller when executing the READ_PROC_TSK*n* command. In addition, the SSD controller adds the request to the Summarizer's task queue.

The flash controller processes the read request by accessing the appropriate channel and chip ids. The data fetched is first buffered in the SSD DRAM and the completion signal is sent to the response queue as in any regular SSD. In Summarizer the flash controller also transfers the two additional in-SSD computing fields to the response queue.

The response queue data is usually sent back to the host via DMA by the SSD controller. However, with Summarizer the SSD controller checks the in-SSD compute flag bit. If the bit is set then it is an indication that the host requested in-SSD computation for this page. In this case the task controller decides whether in-SSD processing is performed for the fetched page data or not. If the controller decides on in-SSD processing then the computation task pointed by the user function pointer registered in the TQ entry is invoked. The buffered page data is used as an input of the computation kernel. The intermediate output data produced by the computation kernel updates the variables or the temporary data set that was initiated by the *initialization* step.Then the special status code is returned to the host to indicate that in-SSD computation is performed for the corresponding page data instead of transferring entire page data to the host's main memory.

Task controller modes: The task controller in Summarizer can execute either in static or dynamic mode. In the static mode whenever in-SSD computing flag is set then that computation is always completed on the fetched data irrespective of the processing delay of the embedded processor. In the static mode, when in-SSD computation request is not possible since TQ is full, the return process is simply stalled.

We also explored a dynamic task controller approach. When Summarizer is running in the dynamic mode, if in-SSD computation is delayed in the SSD controller due to lack of computation resource, the buffered page data is transferred to the host even though READ_PROC_TASK*n* is issued by the host. This situation happens when the service rate (execution time) of embedded processor is slower compared to the incoming rate of in-SSD computation requests. Such congestion happens frequently in the presence of very wimpy SSD cores if near data processing is applied aggressively on fetched page data.

READ_FILT_TSK*n*: The operation of READ_FILT_TSK*n* is similar to that of READ_PROC_TSK*n* except filtering is performed and filtered data is transferred to the host. A filtering request is also a computation task but in this paper we consider a request as a filtering task if the host processor only offloads part of the computation task to the SSD processor and it retains some of the computing for execution on the host. For instance, a filtering task may use a simple compare operations on specific data fields within a page to remove some data that is not needed at the host. The filtering conditions are pre-defined during the *initialization* step

by INIT_TSK*n* command. The filtered data size is recorded in the reserved 8 byte region in the NVMe command and registered in CQ when filtering execution is complete.

FINAL_TSK*n*: The host machine can gather the output result of the computation kernel for task *n* using FINAL_TSK*n* command. When that command is issued the results stored in DRAM on SSD is transferred to the host memory. The size of transferred data is also logged in the reserved 8 byte field of the NVMe response command.

3.3 Composing Summarizer applications

As stated in Section 3.1 Summarizer piggybacks on page-level flash read operations to execute user-defined functions before returning processed data to the host. As such there are some basic restrictions on data layout and computing that must be followed. For instance, the input data for Summarizer should be aligned at the page granularity (4 KB – 16 KB). If data overlaps across page boundaries, a more complex Summarizer data management strategy is necessary. In this work we instead provide data layout and computing API to the programmer to satisfy the page granularity based computing restrictions. In particular, we provide the following Summarizer methods which serve as wrappers that allow conventional user programs to use the proposed Summarizer NVMe commands.

STORE: To exploit the Summarizer it is necessary to align data sets in a page size memory space. To support page-level alignment *STORE* primitive of Summarizer API first assigns user data sets in 4 KB or 16 KB data space and then directly issues store block I/O commands to the host NVMe driver. If the valid data is less than one page then that page meta data stores the valid data size.

READ: The application programmer can use READ API to specify the data set to compute and the desired computation (i.e. SSD functions) to apply on the data set. As data sets are aligned at the page-granularity, the READ API will be translated into READ _PROC_TSK*n* or READ_FILT_TSK*n* NVMe commands at page granularity. If the SSD does not support Summarizer functionality the READ command will map to the default NVMe read command thereby preserving compatability with all SSDs. Note that we assume that the READ command is mapped to READ_PROC_TSK*n* or READ_FILT_TSK*n* explicitly by the programmer.

COMPUTE: Recall that the SSD controller may optionally execute the user function or may return the entire page data back to the host. Thus the dynamic task controller approach requires bit more effort on the host side code to determine whether a page needs processing on the host or not, based on the response received from the SSD controller. As such, the application programmer uses the COMPUTE function as a wrapper to handle the different return values from invoking the READ function. The COMPUTE wrapper simply encapsulated all host function invocations under a conditional statement that checks for the return code from the SSD controller before initiating host side execution on a page.

GATHER: Since computation is distributed on both the host CPUs and SSD devices it is necessary to gather output of kernel computation performed on the SSD devices. So the GATHER wrapper function in the application program issues the *finalization* NVMe command to collect processing output from the SSD devices. And then the collected output is merged with the output from the CPU computations by the programmer.

The programmer can compose in-storage Summarizer programs using imperative programming languages like C and C++. Using the Summarizer API, it is easy to extend the programs that execute only on host system to also execute functions on the processor near SSD. For the applications that we describe later in section 4, it took us 3 – 10 person hours for each application on average (and note that the effort level reduced once the first application conversion was completed). As Summarizer inherits imperative programming model, Summarizer leverages existing ARM programming toolchains to generate the machine code running on SSD controller.

4 CASE STUDIES

Summarizer can provide benefits to a wide range of applications. To evaluate the proposed model, we present several case studies suited for Summarizer execution from database to data integration areas and demonstrate how Summarizer helps avoid redundant data transfer and improve application performance.

4.1 Data analytics

Decision Support Systems (DSS) are a class of data analytics where a user performs complex queries on a database to understand the state of their business. DSS queries are usually exploratory and prefer early feedback to help identify *interesting* regions. Many of the DSS queries perform significant amount of database filtering and only use a subset of database records to perform complex computations. The amount of computation per byte of data transferred is quite low in these applications. Using Summarizer to enable data filtering or even executing the entire query near SSD helps reduce the data bandwidth demands to the host.

Algorithm 1 TPC-H query 1

```
select l_returnflag, l_linestatus,
  sum(l_quantity) as sum_qty,
  sum(l_extendedprice) as sum_base_price,
  sum(l_extendedprice*(1–l_discount)) as sum_disc_price,
  sum(l_extendedprice*(1–l_discount) * (1+l_tax)) as sum_chrg,
  avg(l_quantity) as avg_qty,
  avg(l_extendedprice) as avg_price,
  avg(l_discount) as avg_disc,
  count(*) as count_order
from lineitem
where l_shipdate <= date '1998–12–01' – interval '[DELTA]'
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus;
```

Algorithm 2 TPC-H query 6

```
select sum(l_extendedprice * l_discount) as revenue
from lineitem
where l_shipdate >= date '[DATE]'
  and l_shipdate < date '[DATE]' + interval '1' year
  and l_discount between [DISCNT]–0.01 and [DISCNT]+0.01
  and l_quantity < [QUANTITY];
```

We run the TPC-H benchmark to test the performance of data analytics. TPC-H is a well-known data warehouse benchmark. It

consists of a suite of business oriented ad-hoc queries. We select TPC-H queries 1, 6, and 14 that require several operations such as where condition, join, group by and order by. These operations are also performed in many other TPC-H queries. TPC-H queries 1,6, and 14 are shown in Algorithm 1, 2 and 3 respectively. In our experiments, we evaluate these queries on the TPC-H databases with scale factors 0.1 (~100MB). Note that this scale factor is simply a limitation of our prototype board (described in the next section) due to the limited amount of capacity, not a limitation of Summarizer.

Algorithm 3 TPC-H query 14

```
select 100.00 * sum(case when p_type like 'PROMO%'
  then l_extendedprice*(1-l_discount) else 0 end)
  / sum(l_extendedprice * (1 - l_discount)) as promo_revenue
from lineitem, part
where l_partkey = p_partkey
  and l_shipdate >= date '[DATE]'
  and l_shipdate < date '[DATE]' + interval '1' month;
```

4.2 Data integration

Data integration is the problem of combining data from different sources and/or in different formats. This problem is crucial for large enterprises that maintain different kind of databases, for better cooperation among government agencies, each with their own data sources, and for search engines that manage all kinds of web pages on the Internet.

Similarity join is an important step in the data integration process. While SQL provides support (such as *join*) to combine complementary data from different sources, it fails if the attribute values of a potential match are not exactly equal due to misspelling or other structuring issues. Similarity join is an effective way to overcome this limitation by comparing the similarities of attribute values, as opposed to exactly matching corresponding values.

The similarity join problem can be defined as given a collection of records, a similarity function $sim()$, a similarity threshold t and a query record q finding all the pairs of records, $< q, x >$ such that their similarity values are at least above the given threshold t, i.e, $sim(q, x) \geq t$. We adopt the Overlap similarity which can be defined as: $O(q, x) = \frac{|q \cap x|}{min(|q|, |x|)}$. We use the DBLP dataset which is a snapshot of the bibliography records from the DBLP website. It consists of nearly 0.9M records. Each record consists of the list of authors and the title of the publication. The dataset is preprocessed to tokenize each record using white spaces and punctuation. The tokens in each record are sorted based on this frequency in the entire dataset. The records are then sorted based on their lengths (number of tokens). The prefix filtering based similarity join algorithm that we implemented is shown in Algorithms 4, 5. First we filter each record x which is similar to q from the dataset using the prefix filtering principle [42]. The prefix filtering principle is as follows: Let the p-prefix of a record x be the first p tokens of x. If $O(q, x) \geq t$, then the $(|q| - \lceil t.|q| \rceil + 1)$-prefix of q and the $(|x| - \lceil t.|x| \rceil + 1)$ of x share at least one token. Only the records that pass the prefix filtering stage are verified to check if they meet the overlap similarity threshold.

Algorithm 4 Prefix filtering similarity join

```
Input: query Record q, tokenized dataset D, threshold t
Output: set of Records similar to q in D
S ← φ
for each x∈R do
    a = |q| − ⌈t.|q|⌉ + 1
    b = |x| − ⌈t.|x|⌉ + 1
    for i = 1 to a do
        for j = 1 to b do
            if q[i] == x[j] then
                match ← true
    if match is true then
        similar ← Verify(q, x, t)
    if similar is true then
        S ← S ∪ {x}
    return S
```

Algorithm 5 Verify(q, x, t)

```
Input: Query Record q, matched record x, threshold t
minLength ← min(|q|, |x|)
overlap ← 0
for i = 1 to minLength do
    for j = 1 to minLength do
        if q[i] == x[j] then
            overlap ← overlap + 1
similarity ← overlap/minLength
if similarity > t then
    return true
```

5 METHODOLOGY AND IMPLEMENTATION DETAILS

Summarizer essentially trades computation near-storage with reduced bandwidth to the host. We evaluated four different strategies to study this trade-off.

(1) The first strategy is our baseline where the entire computation is done on the host processor as is the case today. In this baseline the host processor will receive all the data required for computation from the SSD. (2) At the other extreme one may consider doing all the computation at the wimpy processor near SSD, which involves only communicating the output values and input values related to the query with the host processor. All the data required for computation is fetched and processed within the SSD. (3) As the cores near SSD have relatively lower processing power, it may be better to use host and wimpy SSD processors collaboratively. To evaluate this strategy, we used two different approaches. One approach is custom hand-coding of the workload. For the hand-coding approach we analyze the applications and map computations to processors according to the strengths of the processors. Intuitively, computations which help in filtering lots of data communication to host processor shall be mapped at the processors near SSD. Part of the program with high computational intensity shall be mapped to the host processor. While computational intensity of any function may be automatically quantified based on simple metrics such as total instruction count, in this paper we hand-classified the DSS

queries and data integration code into functions that are either data intensive or computation intensive. (4) And the last approach we evaluated is an automatic approach that dynamically selects which pages to compute on the embedded core and which pages to be processed on the host. The automatic approach is agnostic to the entire workload when distributing the computation tasks. For this mode, the host CPU sets all pages as *in-SSD computation* when block requests are issued to the SSD device. Once the page data is fetched from NAND flash, the SSD controller checks the empty slots in TQ. Recall TQ is a queue in Summarizer architecture where a page is registered to be considered for processing near SSD. If there are empty slots, the page is registered in TQ and the page data is computed in SSD. Otherwise, all page data is transferred to the host CPU without processing.

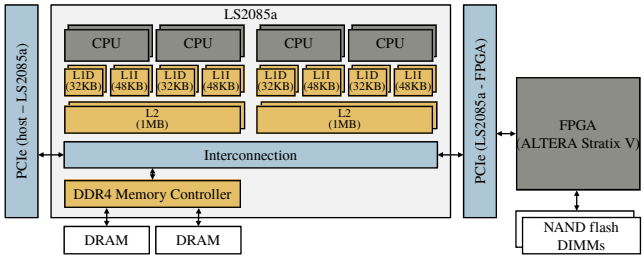
Clearly executing the entire workload on the host or SSD controller is trivial. The only challenge here is to compile the workload to run on two different ISAs: the host processor in our implementation is based on x86 while the SSD controller core is based on ARM core. However, collaborative execution on two processors requires workload distribution. For the hand-coded version we used the following division of work for each workload as shown in the list below. Note that there are several variants of this hand-coded version that can be implemented (and we in fact evaluated some of these variants), but as we show later in the results section hand-coded optimization, while better than baseline, is generally outperformed by the dynamic workload distribution approach. Dynamic approach is much easier to adopt in practice. The programmer does not have to worry about workload distribution and the system automatically determines where to execute the code based on dynamic system status.

- **TPC-H query 1,6:** For static workload distribution, we implement the where condition at processor near SSD and then transfer just the items of the record needed to do group by and aggregation operations.
- **TPC-H query 14:** We implement hash join algorithm to perform equi-join operation between the lineitem and part table. In this algorithm first is the build phase, in this phase using the part table a hash table is built with the table key being used for hashing and the item required in further processing as the values in the hash table. In the next iteration, i.e. probe phase, we traverse over the lineitem table and check if the key item is present in the hash table. For processing at both the processors strategy we check where condition at the processor near the SSD and transfer only item of the record that are needed for hashing and then aggregation.

6 EVALUATION

6.1 Evaluation platform

We evaluated the performance of Summarizer using an industrial-strength flash-based SSD reference platform. The architecture of the SSD development board is illustrated in Figure 4. The board is equipped with a multi-core ARM processor executing the SSD controller firmware programs (including FTL management, wear-leveling, garbage collection, NVMe command parsing and communication with the host), and an FPGA where the flash error correction logic and NAND flash memory controller logic are implemented.



(a) Architecture

(b) SSD development board

Figure 4: The SSD development platform

The ARM processor communicates with the host processor via PCIe Gen.3 ×4 bus. Also, the ARM processor and the NAND flash controller logic on the FPGA transfers NAND flash access commands and data through PCIe buses on the SSD development board.

The NAND flash controller on the FPGA accesses two NAND flash DIMMs that are equipped with 4 NAND flash chips per DIMM. The prototype board design faithfully performs all the functions of commercial SSDs. Unlike commercial SSD devices our prototype NAND flash interface on the development board has lower internal data bandwidth since the NAND flash DIMMs have fewer NAND flash stacks and fewer channels. In addition, the NAND flash controller on FPGA runs with lower clock frequency due to the error correction logic implementation limitations. These limitations are purely due to cost considerations in designing our boards. Hence, the internal SSD bandwidth observed in our board is significantly lower than commercial SSDs. To accommodate the lower internal bandwidth limitation, the host-SSD bandwidth is also set to be proportionally lower. In commercial Samsung SSDs the external bandwidth is typically 2–4× lower than the peak internal bandwidth. Hence, the host-SSD bandwidth is set as 2× lower in our board compared to the internal FPGA-ARM core bandwidth.

Another implementation difference between our board and commercial SSDs is that our board is equipped with more powerful embedded cores than what is seen typically on commercial SSDs. Compare to the reported clock frequency (400 or 500 MHz) of commercial NVMe SSD controllers, the ARM processor on our development platform runs at a faster peak clock frequency (1.6 GHz). Hence, the host-embedded performance ratios favor more embedded core computing. To mimic commercial SSDs we throttled the ARM core frequency. In the next subsection we describe how we select the throttling frequency.

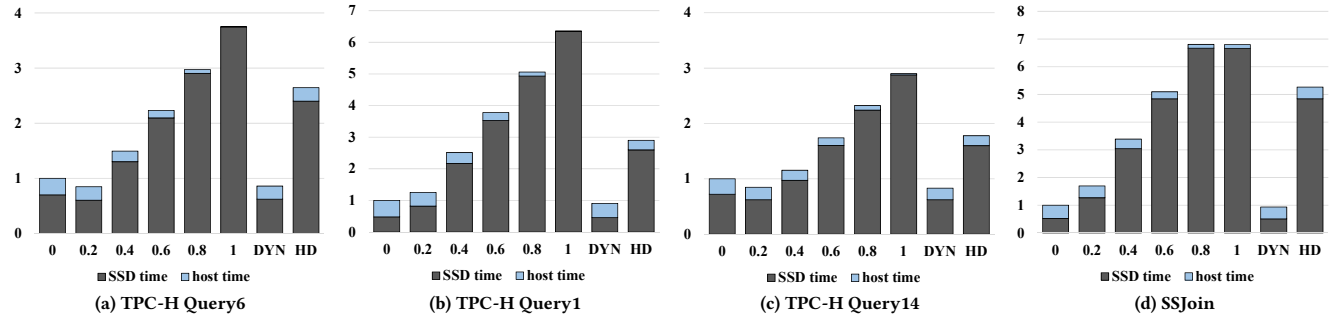


Figure 5: Execution time by the ratio of in-SSD computation

6.2 Calibration based on workload measurements

We tested several database applications introduced in Section 4 on an Intel NVMe SSD platform for this study. First we measured the ratio of I/O time to compute time of the applications in order to throttle down the host CPU as well as the embedded core performance of the SSD to meet the measured I/O and compute time ratio on the SSD development platform. Table 2 shows the compute time by I/O time ratio measured on the real NVMe SSD platform, which is equipped with an Intel i5-6500 (4 cores running at 3.2 GHz) with 8 GB DRAM and Intel NVMe SSD. The Intel NVMe SSD is a 750 Series SSD with PCI-Express 3.0 and 20 nm multi-level cell technology. We then ran the same workload on our platform to create an equivalent processing to I/O ratio. Based on this ratio we set the frequency of the ARM core to be 200 MHz. Note that some of the reduction in frequency is also due to the smaller internal bandwidth we have on our platform.

Data set	Processing by I/O ratio
TPC-H query 6	0.42
TPC-H query 1	1.08
TPC-H query 14	0.39
Similarity Join	0.93

Table 2: Processing by I/O ratio on data center SSDs

6.3 Summarizer Performance

As described in methodology section, we consider workload division at page-level granularity between the host processor and the embedded processor near SSD. Figure 5 shows the performance change by the degree of in-SSD processing. The X-axis indicates the ratio of number of pages processed in-SSD with Summarizer and number of pages processed at the host processor. Y-axis is the execution time normalized to the baseline, namely all data is processed in the host CPU. For this study we adjust the ratio of the pages marked as *in-SSD computation* in the host application and compare the performance change. Thus, zero on the X-axis means all data is processed by the host CPU, namely that is a baseline. If the ratio is one, it means all data is computed in SSD and the host CPU receives only the final result.

The results where the X-axis shows specific numbers between 0 and 1 correspond to the *static mode* operation of Summarizer as described in Section 3. The bar labeled DYN represents results obtained using the *dynamic mode* of Summarizer. In the dynamic mode, all page fetch requests from the host CPU are issued using READ_PROC_TSK*n* commands. Summarizer dynamically decides the in-SSD computation for the requested pages. HD means *hand-coded task offloading* for in-SSD computation. Simple tasks (e.g. a database field filtering function) are performed on the SSD to reduce data traffic. The READ_FILT_TSK*n* NVMe command is exploited to perform filtering operations in the HD mode. We tested the performance of the hand-coded version under Summarizer's static mode. Namely the filtering tasks are performed in the SSD regardless of available resources in the embedded processors.

Each execution time bar is split into two components: time spent on the host side (labeled as host time in the bar), and time spent on the SSD side (labeled as SSD time). When using the baseline (X-axis label 0) the time spent on the SSD side is purely used to read the NAND flash pages and transfer them to the host. But for other bars the time spent on the SSD side includes the time to read and process a fraction of the pages on SSD. Clearly processing only at the SSD (X-axis labeled 1) leads to significant performance degradation since the data computation takes longer on the wimpy SSD controller core. The hand-coded version (labeled HD) provides better performance than the static page-level SSD computation for all or large percentage of pages but in general hand-coding is a static approach that does not adopt to changing system state. As the wimpy cores on the SSD get overloaded, even though filtration tasks do not require lots of computation resources, our result shows static in-SSD offloading approach is ineffective as the I/O request rates exceed service rate of wimpy cores.

The result in Figure 5 also demonstrates that placing all computation on host processor or SSD processor does not deliver the best performance. As such each application has a sweet spot where collaborative computation between SSD and host gives the best performance. But this sweet spot varies from workload to workload and may even vary based on the input data. In the *dynamic* mode Summarizer dynamically decides where the user application functions are performed by observing the availability of the embedded processor in the SSD. This dynamic approach can reduces the burden of programmers in deciding the division of in-SSD computing to achieve better performance while exploiting the computation resources in the SSD.

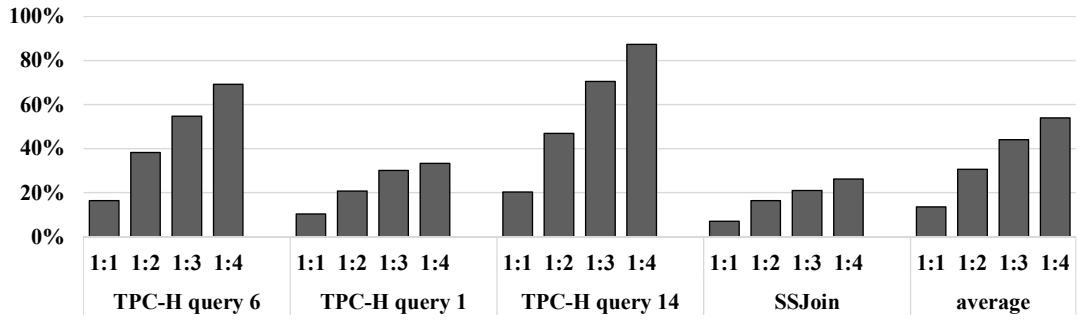


Figure 6: Performance improvement by internal/external bandwidth ratio

The performance of TPC-H query processing with DYN is improved by 16.4%, 10.3% and 20.3% for queries 6, 1 and 14 respectively. On average our current Summarizer prototype can improve the TPC-H performance by 15.7%. For similarity join the performance is improved by 6.9%. The percentage improvements that we observed are directly related to the amount of computation that we need to perform on each page of data. For TPC-H query 6 and 14 as most of the records are filtered by the where condition the amount of work that we do at each page is less and the improvements are higher. For TPC-H query 1 most of the records pass the where condition and the amount of work done at each page is higher when compared to TPC-H queries 6 and 14. Further, we observe that the amount of work involved at each page in similarity join is even higher and concomitantly the improvements are lower.

It is important to note that almost all the bars in the figure use collaborative computation between the host and SSD processor (except for 0 and 1). However, the performance improvements achieved by DYN are not simply due to the availability of additional wimpy CPU resource. As shown in the bars much of the gains come from reduced I/O processing time, rather than having an additional wimpy core on the SSD.

We must also emphasize that the performance improvements seen in this figure are somewhat constrained by the evaluation platform that has severely limited internal bandwidth than commercial SSDs. As such we believe that these results are only a demonstration of the Summarizer potential rather than an absolute performance gain.

6.4 Design space exploration: Internal/external bandwidth ratio

Figure 6 shows the performance change as a function of the ratio of internal data bandwidth between the SSD controller and NAND flash chips and external bandwidth of PCIe between the host processor and SSD. As stated earlier, even though the internal bandwidth of SSD is easier to increase, current SSD designers have no incentive to increase the internal bandwidth since the external bandwidth determines the system performance. Summarizer provides a compelling reason for decoupling the internal and external bandwidth growth. As shown in the results in-SSD computation is more beneficial if internal bandwidth is higher than external bandwidth.

6.5 Design space exploration: In-SSD computing power

As Summarizer exploits the underutilized computation power of SSD controller processors, it is expected that the performance benefits from in-SSD computation will improve with more powerful embedded processors in SSDs. In order to explore the performance impacts of powerful embedded processors for in-SSD computation, we measured the performance changes of the overall system by changing the computation power of the embedded processor. As mentioned in the previous section we throttled down the clock frequency of the embedded core to mimic the operation of commercial SSDs. We use the throttling capability to increase the frequency of the embedded core up to 1.6 GHz for 8× computation power, or increase the number of cores for in-SSD computation (2 cores running at 1.6 GHz for 16× computation power). While frequency is not a sole measure of performance we use it as a first order metric in this study.

Figure 7 shows the performance change as a function of improved computation power of the SSD controller when Summarizer runs in the dynamic mode. Our experimental results show that the overall performance can be improved up to 120% for TPC-H query 1 and 94.5% on average with in-SSD computation when the performance of the embedded controller core is increased by 16×. As Summarizer uses wimpy core to achieve the above result, Summarizer provides one compelling argument for including a more powerful embedded core in future SSD platforms.

6.6 Cost effectiveness

In addition to the model that Summarizer proposes, there are also system design options that can improve application performance. This section will compare the cost of Summarizer with other options.

Embedded processor vs Host processor: The result of Summarizer may encourage system designers to equip SSDs with more powerful processors. Even though using more powerful embedded processor may increase overall cost of the SSD platforms, it can still be more cost-effective considering the total cost of ownership (TCO) of the entire system. As shown in Figure 7 the overall performance is improved up to 94.5% on average with the more powerful embedded processor assigned to the in-SSD computation. This performance improvement is equivalent to doubling the host processor cores, assuming that the entire application performance

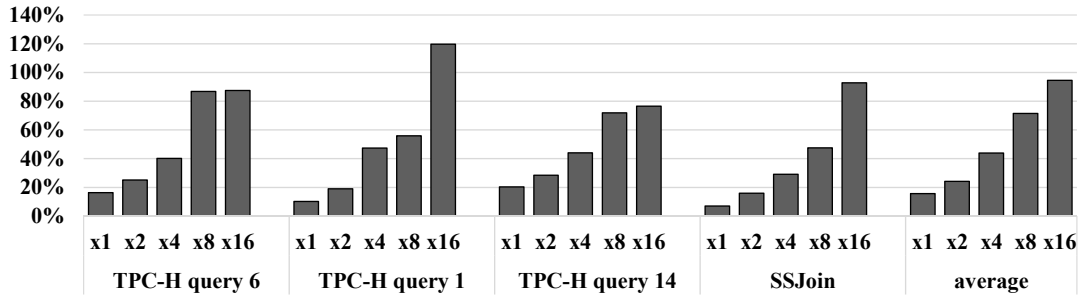


Figure 7: Performance improvement by SSD controller's computation power

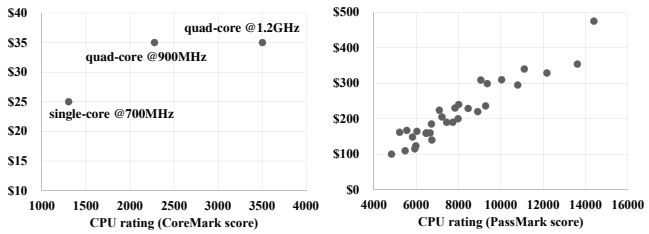


Figure 8: Price by processor performance

is dependent on compute power, and the I/O time is negligible. But in practice to achieve 95% performance improvement on data intensive applications with significant I/O component the host compute power and the rest of system components such as the amount of memory, number of PCIe lanes may need to be scaled up.

Figure 8 plots the performance improvement as a function of price for an x86 host CPU and ARM embedded cores. Unlike x86 CPUs most of ARM SoCs are directly delivered to product manufacturers, thus it is hard to get exact price information. Hence, we show the price changes of three versions of Raspberry Pi (RPi) that are available on the market today in Figure 8a. RPi is a popular single board computer that equips an ARM-based SoC including video processing engines and various peripheral control IPs [40]. X-axis of the figure is the CPU rating measured by CoreMark benchmarks [10] and Y-axis is the price of the RPi boards. While the three generation of RPis have different system capabilities the CoreMark is mostly a CPU benchmark. As such a 4× improvement in CoreMark rating is achieved with less than $20 increase in price. While we acknowledge that the price is determined by many factors in the market, this is a first order approximation to demonstrate how cost effective it is to improve wimpy core performance.

Figure 8b shows the price as a function x86 host CPU performance as measured by the PassMark CPU benchmark [28]. The prices and performance ratings are selected from Intel's 6th and 7th generation CPUs. The additional cost for doubling the performance of x86 desktop CPUs is around $150 as reported in Figure 8b. Again, we acknowledge that it should not come as a surprise to designers that doubling x86 desktop CPU performance requires much higher effort than doubling a wimpy core performance. And the wimpy cores in the context of Summarizer are performing simpler operations such as filtering than a complex x86 CPU. However, the purpose of this section is to demonstrate the cost effectiveness of achieving higher system performance with cheaper in-SSD processors.

External bandwidth: Another way to improve the performance of the storage system is increasing the bandwidth between the host machine and the SSD since higher external bandwidth may alleviate data transfer congestion in the SSD. One approach for higher external bandwidth is increasing the serial link speed of the PCIe interconnect using higher clock frequency. However, this approach demands significant advances of serial data communication technology, and PCIe's data transfer rate has not changed since the PCIe version 3.0 which was released in 2010. Another approach is to assign more PCIe lanes to the SSD. It requires more I/O pins on the SSD controller SoC (64 pins for PCIe ×4 and 98 pins for PCIe ×8 connections) and more complex wiring on the SSD board, which will cause significant cost increase [13]. In addition, the SSD should occupy more PCIe lanes on the host machine, which are limited resources of the system. System cost also increases if the host CPU and the motherboard support more PCIe lanes.

On the other hand Summarizer can reduce the data congestion by consuming page data with in-SSD computation. Hence, Summarizer not only releases the computation burden of the host CPU but reduces the data traffic from the SSD. Note that the data I/O time is also reduced when Summarizer is applied as shown in Figure 5. Consequently with Summarizer the external bandwidth is effectively improved since more pages are responded to the host within the same period. This improvement is achieved without the cost of increasing the PCIe bus bandwidth.

7 RELATED WORK

Decades ago, projects including ActiveDisks, IDISKS, SearchProcessor and RAP [1, 3, 17, 19, 20, 33] have explored the idea of pushing computation to magnetic storage devices. However, due to long magnetic disk latency and relatively small input/output size, the cost-effectiveness was limited.

With the growth of dataset sizes, data movement becomes an increasingly significant overhead when executing applications [11, 21, 43]. With improvements of non-volatile memory technologies enabling rich bandwidth inside data storage devices, recent research projects, including Summarizer, started to revisit the idea of pushing computation near storage.

Similar to Summarizer, Active Flash[4, 35, 36], SmartSSD[8, 16], Active Disks Meets Flash [7] and Biscuit[12] also try to utilize the embedded cores in a modern SSD to reduce the redundant data movement to free-up the host CPUs and main memory. Active Flash[35] for instance presents an analytic model for evaluating the potential for in-SSD computation. But the implementation and operational details were not presented. Summarizer presents a detailed description of the application development to SSD offloading frameworks. SmartSSD[8] focuses on how to improve specific database operations, such as aggregation, using in-SSD computation. Biscuit[12] states that the approach is based on flow-based programming model. Hence, the applications running on Biscuit are similar to task graphs with data pipes to enable inter-task communication. Summarizer presents a set of general purpose NVMe commands and a programming model that can be used across different application domains to show the full potential of in-SSD computing. Summarizer presents an automated approach to determine when to offload computations for applications written in imperative languages without any restrictions on the code structure.

Summarizer employees general-purpose ARM-based cores that are popular in SSD controllers. Therefore, the system design can implement most architectural supports that Summarizer needs through updating the firmware. On the other hand, Active Disks Meets Flash[7], Ibex[41] or BlueDBM[15] leverages re-configurable hardware or specialized processor architectures to achieve the same goal, limiting the flexibility of applications but increasing the cost of devices.

To expose hidden processing power in SSDs to applications, Summarizer, Biscuit[12], Morpheus[39], SmartSSD[8], and KAML[14] all extended standard NVMe or SATA protocols for applications to describe the desired computation. Unlike KAML or SmartSSD which extended the protocols specifically for database related workloads, Summarizer's NVMe command set provides more flexibility in using the SSD processors.

In terms of programming models, Summarizer leverages the matured development tools in ARM platforms to compose and generate code running on storage devices, without needing application designers to deal with very low-level hardware details or significantly changing existing code. Biscuit's data-flow inspired programming model [12] or the limited API support in Morpheus [39] are more appropriate for specific application scenarios.

Summarizer utilizes existing processors inside flash-based SSDs that originally are used for FTL processing but are also idle most of the time, thus minimizing the additional hardware costs. Processors-in-memory [11], Computational RAMs [9, 18, 22, 29, 37], Moneta [5] and Willow [34] require additional processors in the corresponding data storage units, decreasing the cost-efficiency of proposed designs.

8 CONCLUSION

Big data analytics are hobbled by the limited bandwidth and long latency of accessing data on storage devices. With the advent of SSDs there are new opportunities to use the embedded processors in SSDs to enable processing near storage. However, these processors have limited compute capability and hence there is trade-off between the bandwidth saved from near storage processing and the computing latency. In this paper we present Summarizer a near-storage processing architecture that provides a set of APIs for the application programmer to offload data intensive computations to the SSD processor. The SSD processor interprets these API calls and dynamically determines whether a particular computation can be executed near storage. We implemented Summarizer on a fully functional SSD evaluation platform and evaluated the performance of several data analytics applications. Even with a severely restrictive SSD platform we show that when compared to the baseline that performs all the computations at the host processor, Summarizer improves the performance by up to 20% for TPC-H queries. When using more powerful cores within the SSD, we show that this performance can be boosted significantly thereby providing a compelling argument for higher near-SSD compute capability.

ACKNOWLEDGMENT

This work was supported by DARPA-PERFECT-HR0011-12-2-0020 and NSF-CAREER-0954211. We also would like to thank Dell EMC for their support in developing the SSD evaluation platform.

REFERENCES

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. 1998. Active Disks: Programming Model, Algorithms and Evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '98)*. ACM, New York, NY, USA, 81–91. https://doi.org/10.1145/291069.291026

[2] Amber Huffman. 2012. NVM Express Revision 1.1. http://nvmexpress.org/wp-content/uploads/2013/05/NVM_Express_1_1.pdf. (2012).

[3] J. Banerjee, D. K. Hsiao, and K. Kannan. 1979. DBC A Database Computer for Very Large Databases. Vol. 28. IEEE Computer Society, Washington, DC, USA, 414–429. https://doi.org/10.1109/TC.1979.1675381

[4] Simona. Boboila, Youngjae Kim, Sudharshan S. Vazhkudai, Peter Desnoyers, and Galen M. Shipman. 2012. Active Flash: Out-of-core data analytics on flash storage. In *Mass Storage Systems and Technologies, 2012 IEEE 28th Symposium on (MSST '12)*. 1–12. https://doi.org/10.1109/MSST.2012.6232366

[5] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. 2010. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '10)*. IEEE Computer Society, Washington, DC, USA, 385–395. https://doi.org/10.1109/MICRO.2010.33

[6] Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011. Essential Roles of Exploiting Internal Parallelism of Flash Memory Based Solid State Drives in High-speed Data Processing. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*. IEEE Computer Society, Washington, DC, USA, 266–277. http://dl.acm.org/citation.cfm?id=2014698.2014864

[7] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R. Ganger. 2013. Active Disk Meets Flash: A Case for Intelligent SSDs. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS '13)*. ACM, New York, NY, USA, 91–102. https://doi.org/10.1145/2464996.2465003

[8] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 1221–1230. https://doi.org/10.1145/2463676.2465295

[9] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, Ihn Kim, and Gokhan Daglikoca. 2002. The Architecture of the DIVA Processing-in-memory Chip. In *Proceedings of the 16th International Conference on Supercomputing (ICS '02)*. 14–25.

[10] EEMBC. 2017. CoreMark Scores. http://www.eembc.org/coremark. (2017).

[11] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. 2015. Practical Near-Data Processing for In-Memory Analytics Frameworks. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT '15)*. IEEE Computer Society, Washington, DC, USA, 113–124. https://doi.org/10.1109/PACT.2015.22

[12] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon

Jeong, and Duckhyun Chang. 2016. Biscuit: A Framework for Near-data Processing of Big Data Workloads. (2016), 153–165. https://doi.org/10.1109/ISCA.2016.23

[13] ITRS. 2009. International Technology Roadmap for Semiconductors 2009 Edition: Assembly and Packaging. http://www.itrs2.net/itrs-reports.html. (2009).

[14] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. 2017. KAML: A Flexible, High-Performance Key-Value SSD. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA '17)*. 373–384. https://doi.org/10.1109/HPCA.2017.15

[15] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. 2015. BlueDBM: An Appliance for Big Data Analytics. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/2749469.2750412

[16] Yangwook Kang, Yang-Suk Kee, Ethan L. Miller, and Chanik Park. 2013. Enabling cost-effective data processing with smart SSD. In *Mass Storage Systems and Technologies (MSST '13)*.

[17] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. 1998. A Case for Intelligent Disks (IDISKs). *SIGMOD Rec.* 27, 3 (Sept. 1998), 42–52. https://doi.org/10.1145/290593.290602

[18] Peter M. Kogge. 1994. EXECUBE-A New Architecture for Scaleable MPPs. In *Proceedings of the 1994 International Conference on Parallel Processing - Volume 01 (ICPP '94)*. IEEE Computer Society, Washington, DC, USA, 77–84. https://doi.org/10.1109/ICPP.1994.108

[19] Hans-Otto Leilich, Günther Stiege, and Hans Christoph Zeidler. 1978. A Search Processor for Data Base Management Systems. In *Proceedings of the Fourth International Conference on Very Large Data Bases - Volume 4 (VLDB '78)*. VLDB Endowment, 280–287. http://dl.acm.org/citation.cfm?id=1286643.1286682

[20] Chyuan Shiun Lin, Diane C. P. Smith, and John Miles Smith. 1976. The Design of a Rotating Associative Memory for Relational Database Applications. *ACM Trans. Database Syst.* 1, 1 (March 1976), 53–65.

[21] Yang Liu, Hung-Wei Tseng, Mark Gahagan, Jing Li, Yanqin Jin, and Steven Swanson. 2016. Hippogriff: Efficiently moving data in heterogeneous computing systems. In *2016 IEEE 34th International Conference on Computer Design (ICCD '16)*. 376–379. https://doi.org/10.1109/ICCD.2016.7753307

[22] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. 2000. Smart Memories: A Modular Reconfigurable Architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*. ACM, New York, NY, USA, 161–171. https://doi.org/10.1145/339647.339673

[23] Marvell. 2017. High performance PCIE SSD Controller (88SS1093). http://www.marvell.com/storage/ssd/88SS1093. (2017).

[24] Micron. 2017. Micron 3D NAND technology. https://www.micron.com/products/nand-flash/3d-nand. (2017).

[25] Micron. 2017. Micron NAND Flash by Technology. https://www.micron.com/products/nand-flash. (2017).

[26] Microsemi. 2017. Flashtec NVMe Controllers. http://www.microsemi.com/products/storage/flashtec-nvme-controllers/flashtec-nvme-controllers. (2017).

[27] PassMark. 2017. Hard Drive Benchmarks - Solid State Drive (SSD) Chart. http://www.harddrivebenchmark.net/ssd.html. (2017).

[28] PassMark. 2017. PassMark CPU benchmark. http://www.cpubenchmark.net. (2017).

[29] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberley Keeton, Christoforos Kozyrakis, Randi Thomas, and Kathy Yelick. 1997. Intelligent RAM (IRAM): chips that remember and compute. In *Solid-State Circuits Conference, 1997. Digest of Technical Papers. 43rd ISSCC., 1997 IEEE International*. 224–225. https://doi.org/10.1109/ISSCC.1997.585348

[30] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. 2001. Active Disks for Large-Scale Data Processing. *Computer* 34, 6 (June 2001), 68–74. https://doi.org/10.1109/2.928624

[31] Samsung. 2015. Samsung SSD 850 PRO Data Sheet, Rev.2.0 (January, 2015). http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/Samsung_SSD_850_PRO_Data_Sheet_rev_2_0.pdf. (2015).

[32] Mohit Saxena, Michael M. Swift, and Yiying Zhang. 2012. FlashTier: A Lightweight, Consistent and Durable Storage Cache. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, New York, NY, USA, 267–280. https://doi.org/10.1145/2168836.2168863

[33] Stewart A. Schuster, H. B. Nguyen, Esen .A. Ozkarahan, and Kenneth C. Smith. 1979. RAP.2 - An Associative Processor for Databases and Its Applications. *Computers, IEEE Transactions on* C-28, 6 (June 1979), 446–458. https://doi.org/10.1109/TC.1979.1675383

[34] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A User-Programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 67–80. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/seshadri

[35] Devesh Tiwari, Simona Boboila, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter J. Desnoyers, and Yan Solihin. 2013. Active Flash: Towards Energy-efficient, In-situ Data Analytics on Extreme-scale Machines. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. USENIX Association, Berkeley, CA, USA, 119–132. http://dl.acm.org/citation.cfm?id=2591272.2591286

[36] Devesh Tiwari, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Simona Boboila, and Peter J. Desnoyers. 2012. Reducing Data Movement Costs Using Energy Efficient, Active Computation on SSD. In *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems (HotPower'12)*. USENIX Association, Berkeley, CA, USA, 4–4. http://dl.acm.org/citation.cfm?id=2387869.2387873

[37] Josep Torrellas. 2012. FlexRAM: Toward an advanced Intelligent Memory system: A retrospective paper. In *Computer Design, 2012 IEEE 30th International Conference on (ICCD '12)*. 3–4.

[38] Hung-Wei Tseng, Yang Liu, Mark Gahagan, Jing Li, Yanqin Jin, and Steven Swanson. 2015. *Gullfoss: Accelerating and Simplifying Data Movement among Heterogeneous Computing and Storage Resources*. Technical Report CS2015-1015. Department of Computer Science and Engineering, University of California, San Diego technical report. http://csetechrep.ucsd.edu/Dienst/UI/2.0/Describe/ncstrl.ucsd_cse/CS2015-1015

[39] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. 2016. Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 53–65. https://doi.org/10.1109/ISCA.2016.15

[40] Wikipedia. 2017. Raspberry Pi. http://en.wikipedia.org/wiki/Raspberry_Pi. (2017).

[41] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: An Intelligent Storage Engine with Support for Advanced SQL Offloading. *Proc. VLDB Endow.* 7, 11 (July 2014), 963–974. https://doi.org/10.14778/2732967.2732972

[42] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Efficient Similarity Joins for Near Duplicate Detection. In *Proceedings of the 17th International Conference on World Wide Web*. 131–140.

[43] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proc. VLDB Endow.* 6, 10 (Aug. 2013), 817–828. https://doi.org/10.14778/2536206.2536210