SSD In-Storage Computing for List Intersection

Jianguo Wang† Dongchul Park§ Yang-Suk Kee§
Yannis Papakonstantinou† Steven Swanson†

†University of California, San Diego §Samsung Electronics Corp.
†{csjgwang, yannis, swanson}@cs.ucsd.edu §{dongchul.p1, yangseok.ki}@ssi.samsung.com

ABSTRACT

Recently, there has been a renewed interest of in-storage computing in the context of solid state drives (SSDs), called "Smart SSDs". Smart SSDs allow application-specific code to execute inside SSDs. This allows applications to take advantage of the high internal bandwidth that Smart SSDs provide. This work studies the offloading of list intersection into Smart SSDs, because intersection is prominent in both search engines and in analytics queries. Furthermore, intersection is interesting because the algorithms are more complex than plain scans; they are affected by multiple parameters, as we show, and provide lessons that can be used in other operations also.

We are interested to know whether Smart SSDs can accelerate the processing of list intersection and reduce the consumed energy. Intuitively, the answer is yes. However, the performance tradeoffs on real devices are complex. We implement list intersection into a real Samsung Smart SSD research prototype. We also provide an analytical model to understand the key factors to the overall performance, and when list intersection can benefit from Smart SSDs. Finally, we conduct experiments on the Samsung Smart SSD. Based on the results (both analytical and experimental), we provide many suggestions for both SSD vendors on how to manufacture powerful Smart SSDs and for applications on how to make full use of the functionalities that Smart SSDs provide.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems – Query Processing; H.3.3 [**Information Search and Retrieval**]: Search Process

Keywords

In-storage Computing, Smart SSD, List Intersection

1. INTRODUCTION

DaMoN'16, June 27, 2016, San Francisco, CA, USA

 ISBN 978-1-4503-3638-3/15/06…$15.00

DOI: http://dx.doi.org/10.1145/2933349.2933353

In a conventional database architecture, solid state drives (SSDs) are treated as storage-*only* units. As a result, data storage and computation are strictly *separated* from one another: Data is stored on SSDs, while computation is performed at the host CPU. If the database wishes to examine a piece of data to make a decision or execute a query, it must pull the data from SSDs through a (relatively) slow interface (such as SATA, SAS, or PCIe) into main memory where the CPU has access to it.

However, recent studies indicate this "*move data closer to code*" paradigm cannot make full use of SSDs [11, 24, 33], for two major reasons. (1) Modern SSDs are usually manufactured with a higher (2-4×) internal bandwidth than the host interface bandwidth. This bandwidth over-provisioning is crucial to execute many complicated FTL (Flash Translation Layer) tasks efficiently. As an example, for the Samsung SSD used in our experiments, the internal bandwidth is 1.5 GB/s while the host interface (SAS) bandwidth is 550 MB/s. Then the limited external bandwidth will squander the high internal bandwidth that SSDs provide. (2) Modern SSDs usually embrace energy-efficient processors (like ARM series 32-bit processors) to execute FTL tasks. The computing capabilities are generally ignored by the conventional database architecture where SSDs are treated as storage-*only* devices.

To fully exploit SSD potential, SSD in-storage computing (a.k.a Smart SSD) was recently proposed [11, 24, 33]. The main idea is to treat an SSD as a small machine (with ARM processors) to execute some programs (e.g., C++ code) directly inside the SSDs. Upon receiving a query, unlike conventional computing architectures that have the host machine execute the query, now the host sends the query (or some query operations) to the Smart SSD. The Smart SSD reads necessary data from flash chips to its device DRAM, and executes the query (or query steps) on its internal processors. Then, only the results (expected to be much smaller than the raw data) are returned to the host machine through the relatively slow host interface. In this way, Smart SSDs change the traditional computing paradigm to "move code closer to data" (a.k.a near-data processing [3, 34]).

Although the (ARM) CPU within the Smart SSD is less powerful (e.g., lower clock speed and higher memory access latency) than the host (Intel) CPU, the Smart SSD has two advantages which make it compelling for some I/O-intensive and computationally-simple applications. (1) The I/O time of accessing data is much less because of the SSD's high internal bandwidth. (2) More importantly, energy can be reduced since ARM processors consume much less power (3-4×) than host CPUs. The energy saving is dramatically

important in today's data centers because energy can take 42% of the total monthly operating cost in data centers [18]; this explains why enterprises like Google and Facebook recently revealed plans to replace their Intel-based servers with ARM-based servers to save energy and cooling cost [20, 25].

A major research question regarding offloading application-specific code within Smart SSDs is *what kinds of applications can benefit from Smart SSDs*? On the one hand, the high internal I/O bandwidth opens up new opportunities; on the other hand, the processing capabilities of Smart SSDs are still very limited. Researchers have explored the offloading of some operations like scan [11] and group-by [33] into Smart SSDs.

This paper explores the offloading of another important operation – *list intersection* into Smart SSDs, which has not been covered before. Intersection is at the core of many applications. For instance, finding documents that contain all the query terms in search engines requires the intersection of several inverted lists;[1] evaluating conjunctive predicates in analytical queries requires the intersection of several columns.

We are interested to know whether list intersection can benefit from Smart SSDs. Intuitively we think the answer is yes, because (1) list intersection is I/O-intensive especially when the lists are very long. Running it inside SSDs can leverage the high internal bandwidth. (2) List intersection is computationally-simple because efficient algorithms (e.g., [7]) only evaluate a small portion of every list to find results. Thus, it does not impose too much burden on the computationally weak cores running inside SSDs. (3) The intersection results (output) can be orders of magnitude smaller than the original lists (input) [10]. Therefore, much less data will be transferred though the slow interfaces.

However, the performance tradeoffs on real devices are complex. We implement list intersection into a real Samsung Smart SSD research prototype. We find that many factors can affect performance. In this paper, we share our experience in accelerating list intersection and reducing energy consumption with Smart SSDs. The main contributions of this work are summarized as follows:

- We study the offloading of list intersection into Smart SSDs. Our results have some implications for both SSD vendors on how to manufacture powerful Smart SSDs and for applications on how to make full use of the functionalities that Smart SSDs provide.
- We implement the intersection operation into a real Samsung Smart SSD.
- We show the key factors that affect the overall performance and energy consumption analytically and experimentally.

The rest of this paper is organized as follows. Section 2 provides an overview of the Samsung Smart SSD. Section 3 presents the design and implementation details of offloading list intersection within Smart SSDs. Section 4 provides an analytical model to analyze the performance tradeoffs. Section 5 shows the experimental results. Section 6 discusses some related studies of this paper. Section 7 concludes the paper.

[1]Note that, many search engines including Google and Bing store the inverted index on SSDs (instead of memory) due to the factors such as cost, scalability, and energy [6, 21, 30, 31].

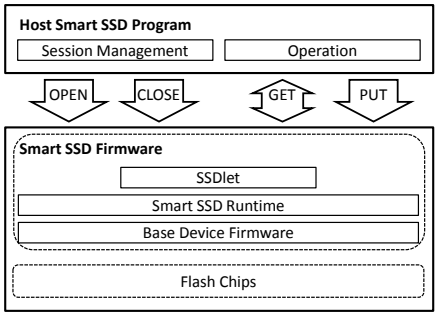


Figure 1: Smart SSD architecture

2. BACKGROUND OF SAMSUNG SMART SSD

The key of the Smart SSD is its programming capability such that user defined programs (e.g., C++ code) can execute within the SSD. However, commercially available SSDs do not have this feature. Thus, we rely on the Samsung Smart SSD platform. In this section, we present the background of the Samsung Smart SSD. Figure 1 describes the architecture which consists of two major components: *Smart SSD firmware* and *host Smart SSD program*.

The Smart SSD firmware is divided into three components: SSDlet, Smart SSD runtime, and base device firmware. An SSDlet is a Smart SSD program running inside the SSD. It implements application logic and responds to a Smart SSD host program. The SSDlet is executed in an event-driven manner by the Smart SSD runtime system, which implements the libraries of Smart SSD APIs. The base device firmware is developed to support normal I/O operations (like read and write) of a storage device.

The host Smart SSD program communicates with the Smart SSD firmware through APIs. There are four important APIs, namely, `OPEN`, `CLOSE`, `GET` and `PUT`. Out of which, `OPEN` and `CLOSE` are session-related APIs while `GET` and `PUT` are operation-related APIs. (1) `OPEN`: as the name suggests, `OPEN` starts a session. Once a session is created, runtime resources such as memory and threads are assigned to run the SSDlet. (2) `CLOSE`: it terminates a session, and all the resources will be reclaimed. (3) `GET`: it allows the host Smart SSD program interact with an SSDlet. `GET` is used to check the status of the SSDlet and receive the output results once they are ready. It implements the polling mechanism (instead of the interrupt mechanism) for SAS/SATA such that the host program has to keep monitoring the status of the Smart SSD. That is because traditional block devices cannot initiate a request to the host using interrupts. (4) `PUT`: it is used to internally write data to the Smart SSD device without help from any local file system.

3. DESIGN AND IMPLEMENTATION

In this section, we provide the system design of offloading intersection to Smart SSDs (Section 3.1), as well as the implementation details (Section 3.2).

As the first investigation of list intersection on Smart SSDs, we focus on the fundamental aspects of intersection. In particular, we consider only two lists for intersection since multi-list intersection can be performed efficiently by intersecting two lists in turn [7].

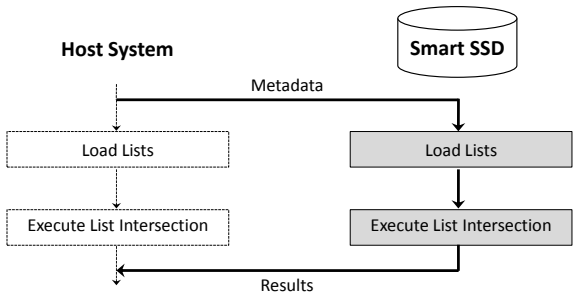


Figure 2: The interaction of the host system and the Smart SSD

3.1 Design

Figure 2 shows how the Smart SSD interacts with the host system. The host system sends some metadata (via `OPEN`) that is necessary to perform list intersection, for example the addresses and lengths of the lists. The Smart SSD then starts to load data (lists) from flash chips to the device memory (DRAM) in pages (of 8 KB). We secure 150 MB DRAM space (but is extensible) for list intersection. Note that, if the lists cannot fit in the device DRAM, we can either increase the DRAM size, or partition the lists like [27] such that each partition fits in the device DRAM. Out of the 150 MB space, 256 KB is a special DRAM area called *core memory*, which can directly communicate with flash chips. That being said, every page will be transferred from flash chips to the core memory first, and copied to the (relatively) big device DRAM afterwards. When all the lists are loaded to the device DRAM, the Smart SSD executes list intersection. Once it is done, the results will be put to an output buffer and then returned to the host side. The host system keeps monitoring the status of the Smart SSD to receive the intersection results in a heart-beat manner (via `GET`). We set the polling interval to be 1 ms (for performance reasons).

3.2 Implementation

We next present the list format and intersection algorithm used in this work.

List format. Each entry in the list has the form $\langle ID, payload \rangle$, where *ID* (4-byte integer) is the field for intersection and *payload* represents arbitrary information about that ID. This data format is generic to represent data in many applications such as databases (both row-stored and column-stored) and search engines. In row-stored databases, the payload represents other fields in a row; in column-stored databases, the payload is empty and hence only the ID is left; in search engines, the payload represents document frequency and positional information of a posting [19]. In addition, each list is sorted in ascending order of the ID.

Intersection algorithm. There are many intersection algorithms in the literature, e.g., [7, 9, 10]. We choose the SvS algorithm [7] because (1) it is simple and does not need any preprocessing; and (2) it is effective in practice [7,10] and thus implemented in many open-source systems like Apache Lucene.[2]

The SvS algorithm works as follows [7]. Let L_1 and L_2 be two lists ($|L_1| \leq |L_2|$). For every element e in L_1, check whether e appears in L_2 (membership checking). If so, e is a result; otherwise, probe the next element in L_1. The membership checking is usually implemented in binary search such that many elements can be skipped; however, if the lists are of similar sizes, linear search is more efficient.[3] In our experiments, we implement both, but only display the faster one as the results (in Section 5).

[2]https://lucene.apache.org/

Symbol	Meaning	Value
D	total data size (of L_1 and L_2)	100 MB
s ($s \geq 4$)	entry size	256 bytes
θ ($\theta \geq 1$)	list size ratio ($\frac{\|L_2\|}{\|L_1\|}$)	1000
$\|L_1\|$	number of entries in L_1	$\frac{D}{s(1+\theta)}$
$\|L_2\|$	number of entries in L_2	$\frac{D\theta}{s(1+\theta)}$
r ($r \leq 1$)	intersection ratio ($\frac{\|L_1 \cap L_2\|}{\|L_1\|}$)	1%
t	host DRAM read time (4 bytes)	5 ns
t'	device DRAM read time (4 bytes)	18 ns
τ	device DRAM copy time (4 bytes)	150 ns
b	host interface bandwidth	550 MB/s
b'	internal bandwidth of SSD	1500 MB/s
f	host CPU frequency	3.4 GHz
f'	device CPU frequency	0.4 GHz
c	instructions per cycle of host CPU	4
c'	instructions per cycle of device CPU	2

Table 1: A list of parameters for the analysis. In which, t, t', τ, b and b' are obtained by our implemented programs; f, f', c and c' are obtained from processor manuals.

4. ANALYTICAL MODEL

In this section, we provide an analytical model to analyze the overall performance and energy consumption when carrying out list intersection on regular and Smart SSDs. Table 1 shows a list of parameters used in the analysis.

We focus on the energy consumption first. Let T and T' be the execution time for executing list intersection on the regular SSD and the Smart SSD. Also, let P and P' be the power (in Watts) of the host (Intel) CPU and the device (ARM) CPU running within the Smart SSD. Then, the energy E consumed by the regular SSD is $E = T \times P$, and the energy E' consumed by the Smart SSD is $E' = T' \times P'$. Since P' is 3-4× less than P, it is more likely that E' is less than E (although the results depend on different queries).

Next, we focus on estimating the execution time T and T'. We divide the processing of list intersection into four main stages and analyze the cost of each stage individually. Thus, T (T') will be a summation of the cost of each individual stage.

(1) **Load data** (Section 4.1). Load each list individually from flash chips to memory (Section 4.1). On Smart SSDs, the memory refers to the device DRAM while on regular SSDs, it refers to the host DRAM.

(2) **Memory copy** (Section 4.2). Copy data from the core memory to the device memory. This stage is only applicable to Smart SSDs, because data can only be loaded from flash chips to the small core memory (256 KB). This is an implementation issue of the Samsung Smart SSD, and we will remove such a constraint in the future. However, on

[3]The time complexity of intersection is $O(|L_1| \cdot \log |L_2|)$ and $O(|L_1| + |L_2|)$ using binary search and linear search, respectively. When $|L_1| \approx |L_2|$, the latter cost is cheaper.

regular SSDs, data can directly go from flash chips to the host DRAM, without such an extra data copy.

(3) **Do intersection** (Section 4.3). When all the lists are ready, run the SvS algorithm for intersection on both Smart SSDs and regular SSDs.

(4) **Send results** (Section 4.4). After the computation is finished, send results back to the host. This stage is only applicable to Smart SSDs.

4.1 Cost of Loading Data

Let T_1 and T_1' be the time (in ms) of reading lists from the regular SSD and the Smart SSD, then we have,

- Regular SSD:

$$T_1 = \frac{D}{b} = \frac{100}{550} \text{ s} \approx 182 \text{ ms}$$

- Smart SSD:

$$T_1 = \frac{D}{b'} = \frac{100}{1500} \text{ s} \approx 67 \text{ ms}$$

To verify the accuracy of the model, Figure 3a shows the actual time for loading the same amount of data. On the Smart SSD, the actual time is 65 ms (the estimated time is 67 ms); on the regular SSD, the actual time is 198 ms (the estimated time is 182 ms).

Remark. From the analysis, the internal bandwidth (b') is an important factor. The benefit of Smart SSDs comes from the high internal bandwidth. In general, SSDs are manufactured with a higher internal bandwidth to execute many complicated FTL tasks. Based on the current technology trends, there is no sign (at least in the near future) of closing the gap between the two [3, 11].

Note that b'/b is the upper bound of the performance gains that Smart SSDs could potentially achieve. The value is $1500/550 = 2.73\times$ in our case.

4.2 Cost of Memory Copy

This stage is only applicable to Smart SSDs by copying data from the core memory (256 KB) to the (relatively) big device memory (150 MB). Recall the entry of each list has the form $\langle ID, payload \rangle$. However, only the ID field (4 bytes) is used for intersection. Thus, in this stage, we only copy the ID field from the core memory to the device memory. Note that if the payload is empty, the entire data has to be copied.

Let T_2' be the cost of memory copy (for all the entries), since all entries in the lists have to be copied, then,

$$T_2' = \frac{D}{s} \cdot \tau = \frac{100}{256} \cdot 150 \approx 59 \text{ ms}$$

To verify the accuracy of the model for this stage, Figure 3a shows the actual time, which is 62.5 ms while the estimated time is 59 ms. The error rate is only 5.6%.

Remark. Based on the analysis, if D and τ are fixed, then T_2' is inversely proportional to the entry size s. Consider an extreme case where $s = 4$ (the payload is empty), then T_2' can be as high as 3750 ms! That is much higher than the I/O cost saving (which is $182 - 67 = 115$ ms). Thus, Smart SSDs cannot accelerate list intersection when the entry size is very small (e.g., 4 bytes).

4.3 Cost of List Intersection in Memory

We next analyze the cost of carrying out list intersection in memory. It is very challenging to analyze, because it is related to different processors (Intel CPU and ARM CPU) which have different architectures. Many hard-to-model factors like out-of-order execution, deep pipelining and branch prediction can affect the performance.

We estimate the number N_m of memory accesses and the number N_c of CPU operations. Recall that the intersection algorithm works as follows. For each element $e \in L_1$, check whether e appears in L_2. In this algorithm, for every memory access, there are around three associated CPU operations (compare, increment and assignment). Thus we set $N_c = 3N_m$ in the cost model.

Next we estimate N_m, which depends on the approach (either binary search or linear search) for membership checking. For linear search, N_m can be estimated as $2(|L_1|+|L_2|)$. We set the constant factor at 2, because every comparison usually moves one pointer at a time. For binary search, $N_m \approx 2|L_1| \cdot \log_2 |L_2|$.

Let T_3 and T_3' be the time of performing list intersection on the regular SSD and the Smart SSD. Note that each CPU operation takes $\frac{1}{c}$ cycle (on average) to finish because of the superscalar architecture, where c is the number of instructions per cycle. Thus,

$$T_3 = N_m \cdot t + N_c \cdot \frac{1}{f} \cdot \frac{1}{c} = N_m(t + \frac{3}{cf})$$

- Regular SSD:

$$T_3 = N_m(t + \frac{3}{cf}) = N_m(5 + \frac{3}{4 \times 3.4}) = N_m(5 + 0.22)$$
$$= \begin{cases} 2(|L_1| \log_2 |L_2|)(5 + 0.22) & \triangleright \text{ binary search} \\ 2(|L_1| + |L_2|)(5 + 0.22) & \triangleright \text{ linear search} \end{cases}$$

- Smart SSD:

$$T_3' = N_m(t' + \frac{3}{c'f'}) = N_m(18 + \frac{3}{2 \times 0.4}) = N_m(18 + 3.75)$$
$$= \begin{cases} 2(|L_1| \log_2 |L_2|)(18 + 3.75) & \triangleright \text{ binary search} \\ 2(|L_1| + |L_2|)(18 + 3.75) & \triangleright \text{ linear search} \end{cases}$$

	s = 4 bytes	s = 256 bytes
$\theta = 1$ (linear search)	$N_m = 5 \times 10^7$ T_3 = 261 ms T_3' = 1087 ms	$N_m = 7.8 \times 10^5$ T_3 = 4.07 ms T_3' = 16.96 ms
$\theta = 1000$ (binary search)	$N_m = 1.2 \times 10^6$ T_3 = 6.41 ms T_3' = 26.72 ms	$N_m = 14,512$ T_3 = 0.075 ms T_3' = 0.31 ms

Table 2: Cost of list intersection, note that $|L_1| = \frac{D}{s(1+\theta)}$ and $|L_2| = \frac{D\theta}{s(1+\theta)}$ ($D = 100$ MB by default)

To verify the accuracy of the model for this stage, Figure 3a shows the actual time, which is 0.47 ms when $s = 256$ and $\theta = 1000$, while our estimated time is 0.31 ms (see Table 2). The gap is due to many other factors such as out-of-order execution, deep pipelining, branch prediction (or even functional calls). However, this is the CPU time of executing list intersection, which is generally smaller than the I/O time in Section 4.1. Thus, our analytical most can still capture the major factors that can affect the performance of list intersection.

Remark. Compared to regular SSDs, Smart SSDs require more time for performing list intersection in this stage. That is not surprising because Smart SSDs suffer from high memory access latencies ($t' > t$) and low-clocked processors ($f' < f$). However, in some cases, the I/O cost saving of the first stage can justify the penalty of this stage. For example, when $\theta = 1000$ and $s = 256$, the Smart SSD introduces an overhead of only $0.31 - 0.075 = 0.235$ ms (Table 2). That is because the intersection algorithm can perform efficient skipping without scanning all the data.

The analysis also implies that, Smart SSDs suffer more from slow memories than slow processors. That can be seen from the two factors: t' (which is 18) and $\frac{3}{c'f'}$ (which is 3.75). That is because list intersection is computationally-simple in the sense that every element needs at most several CPU operations. However, accessing the element from memory is even more expensive than those collected CPU operations. Thus in order for Smart SSDs to accelerate list intersection, we should improve the memory access speed first, e.g., by dedicating more caches.

4.4 Cost of Sending Results to Host

This stage is applicable to the Smart SSD. Let s' be the entry size in the output results. Note that s' may be different from s, where s is the size of the original entry $\langle ID, payload \rangle$, while s' is the size of the projected (result) entry (e.g., ID). Let T_4' be the cost of sending intersection results back to the host, and

$$T_4' = \frac{|L_1| \cdot r \cdot s'}{b} = \frac{\frac{D}{s(1+\theta)} \cdot r \cdot s'}{b} = \frac{D}{b} \cdot \frac{r}{1+\theta} \cdot \frac{s'}{s}$$
$$= 182 \text{ ms} \times \frac{1\%}{1+1000} \times \frac{4}{256} = 2.8 \times 10^{-5} \text{ ms}$$

Remark. The cost of this stage depends on the intersection size, which is $|L_1| \times r$ in our case. In the worst case, the intersection is the *shortest* list, which is at most half of the total data size. That is, $r = 100\%$, $\theta = 1$ and $s' = s$ such that, $\frac{r}{1+\theta} \cdot \frac{s'}{s} = 1/2$. For the general k-list intersection (where k is the number of lists), the intersection size is at most $1/k$ of the total data size. Hence, Smart SSDs can reduce data movement by a factor of at least k (which is 2 in our case) for list intersection.

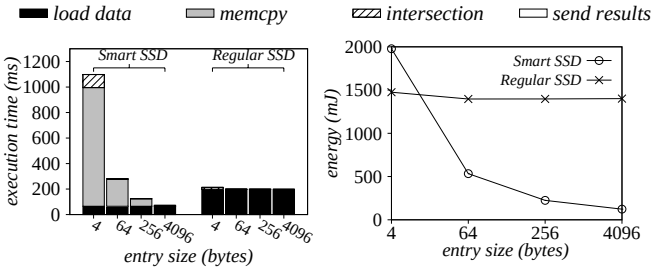
However in some applications, if only the intersection size (rather than the actual intersection results) is concerned, the cost of this stage is negligible, because only a 4-byte integer will be transferred to the host.

5. EXPERIMENTS

In this section, we empirically evaluate whether Smart SSDs can accelerate list intersection. We describe the experimental setup in Section 5.1, and present the experimental results in Section 5.2.

5.1 Experimental Setting

In our experiments, the host machine is a commodity server with Intel i7 processor (3.40 GHz) and 8 GB memory running Windows 7. The Smart SSD is a 400 GB SLC SSD, which is connected to the host machine via a host bus adaptor (HBA). The host interface bandwidth (SAS) is 550 MB/s. The internal bandwidth is 1.5 GB/s. The regular SSD is an identical SSD but without implementing any query offloading. The processor running inside the Smart SSD is ARM Cortex-R4 with a clock speed of 400 MHz.



(a) execution time (ms) (b) energy consumption (mJ)

Figure 3: Effect of the entry size

We measure the performance of list intersection in two aspects: actual execution time and energy consumption. All the values are averaged across three runs for accuracy. We use *WattsUp*[4] to measure the energy consumed. All the programs (running in Smart SSDs and regular SSDs) are coded in C++.

5.2 Results

In this work, we use synthetic data with varying parameters to understand the key factors to the overall performance and energy. We use two lists L_1 and L_2 ($|L_1| \leq |L_2|$), and define the intersection ratio r as $\frac{|L_1 \cap L_2|}{|L_1|}$. As explained in Section 3.2, each entry in the list has the form $\langle ID, payload \rangle$. All the IDs are picked up randomly (uniformly) from the domain $[0, 2^{32} - 1]$. In all the experiments, the total data size of the two lists is around 100 MB in order to fit in the device memory. As explained in Section 3.1, if the lists cannot fit in the device DRAM, we can increase DRAM size or partition the lists such that each partition fits in the device memory.

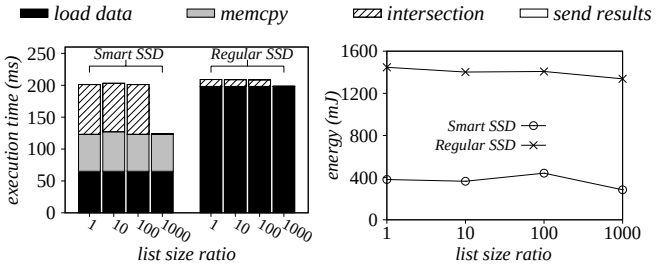
5.2.1 Effect of the Entry Size

Our first set of experiments evaluates the effect of the entry size s, which is a very important parameter. We set the size of L_2 as 100 MB and the size of L_1 as 0.1 MB. That is because in practice, one list is usually longer than the other [10]. We set the intersection ratio as 1%, and vary s from 4 to 4096. Figure 3 plots the results. Both the execution time and energy consumption decrease as s increases, because the lists become shorter.

Figure 3a shows that, Smart SSDs can improve I/O performance. However, when s is small, there is a significant overhead on memory copy. For example when $s = 4$, memcpy accounts for 85% of the total execution time. That is because in the Samsung Smart SSD, there is no cache (neither data-cache nor instruction-cache). Thus, the device DRAM access latency is high. When s increases, the overhead of memcpy (also intersection) decreases because the number of entries decreases. When $s = 256$, it becomes faster to execute intersection within Smart SSDs.

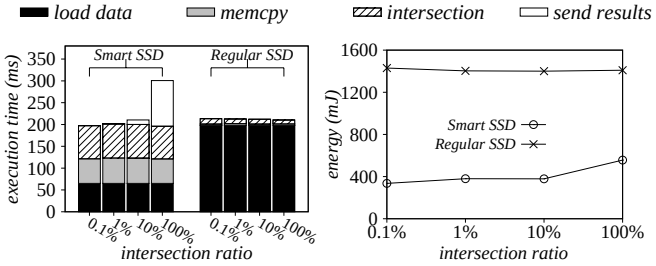
Figure 3b demonstrates the energy consumption, which is roughly proportional to the execution time. But Smart SSDs consume much less power (energy/time). For example, when $s = 4$, even if the Smart SSD incurs 5.1× more execution time, it only consumes 1.3× more energy than the regular SSD. When $s = 64$, the Smart SSD consumes

[4]https://www.wattsupmeters.com



(a) execution time (ms) (b) energy consumption (mJ)

Figure 4: Effect of the list size ratio



(a) execution time (ms) (b) energy consumption (mJ)

Figure 5: Effect of the intersection ratio

2.6× less energy than the regular SSD. That is because the ARM processor running inside the Smart SSD is much more power-efficient. Similar facts can also be seen from Figure 4b and Figure 5b.

5.2.2 Effect of the List Size Ratio

We next evaluate the effect of the list size ratio (θ), which determines to what extent the skipping can occur. We set the total size of L_1 and L_2 to be 100 MB. We vary θ from 1 to 1000, while fixing the entry size s to be 256 bytes and the intersection ratio r to be 1%. Figure 4 shows the results.

It shows that, (1) there is a clear performance improvement when θ increases from 100 to 1000. That is because of effective data skipping (with binary search). Note that the execution time does not change much when θ increases from 1 to 100, that is because linear search is applied. (2) It also show that, when $s = 256$, Smart SSDs can accelerate list intersection for any θ.

5.2.3 Effect of the Intersection Ratio

Finally, we evaluate the effect of the intersection ratio r, which is defined as $\frac{|L_1 \cap L_2|}{|L_1|}$. In this set of experiments, in order to clearly see the effect of r, we set $|L_1| = |L_2|$, $s = 256$ and return the entire entries in the intersection results. Figure 5 plots the results.

It reveals that, when r increases, the cost of sending back results to the host also increases on the Smart SSD. It is not surprising that r does not affect the performance of the regular SSD. When r is 100%, Smart SSDs can no longer accelerate the processing of list intersection, although the total amount of data transferred to the host is reduced by 50%.

However in some applications, if we only care about the intersection size instead of the actual intersection results, the cost of sending results back to the host will be almost free. That is because only a 4-byte integer is needed to be transferred to the host.

6. RELATED WORK

The idea of in-storage computing has been around for decades. Many research efforts have been devoted to making it practical.

As early as in the 1970s, initial work had already proposed to leverage specialized hardware to accelerate query processing within storage devices (i.e., hard disks at that time). As an example, CASSM embedded a processor for each disk track (called "processor-per-track") [26]. Another example is the Ohio State Data Base Computer (DBC) [14], which associated the processing logic with each read/write head of a hard disk (called "processor-per-head"). However, none of the systems turned out to be successful due to high design complexity and manufacturing cost.

In the late 1990s, the bandwidth of the hard disk kept increasing while the cost of embedded processors kept decreasing. These technology trends made it feasible to offload bulk computation to each individual disk. Researchers explored in-storage computing in terms of hard disks (e.g., active disks [1] or intelligent disks [15]). The goal was to offload application-specific query operators inside hard disks to save data movement. They examined the concept in database area by offloading several primitive operators (e.g., selection, group-by). Later on, Erik et al. extended the application to the data mining and multimedia areas [23] (e.g., frequent sets mining and edge detection). Although interesting, few real systems adopted the proposals due to various reasons including limited hard disk bandwidth, limited computing capabilities, and marginal performance gains achieved.

With the advent of SSDs, researchers started to rethink about in-storage computing in the context of SSDs (called "Smart SSDs"). SSDs offer many advantages over HDDs, notably very high internal bandwidths and computing capabilities. More importantly, executing code inside SSDs can save energy due to less data movement and power-efficient embedded ARM processors. These advantages make the concept of in-storage computing on SSDs much more practical and promising.

As a result, Smart SSDs gained much attention from industries. For instance, IBM started to install Smart SSDs to the Blue Gene supercomputers to boost performance [12]. Teradata's Extreme Performance Appliance [28] is another example of integrating SSDs and database functionalities. Oracle's Exadata [22] also started to offload complex query processing into storage servers. Samsung explored the potential of offloading map-reduce functions to Smart SSDs [5, 13]. Smart SSDs were popular in academia too. For example, Kim et al. investigated pushing down the scan operator and join operator to SSDs [16, 17]. Later on, Do et al. [11] worked on a real Smart SSD (developed by Samsung). They integrated the Smart SSD with Microsoft SQL Server by offloading two operators: scan and aggregation. Woods et al. built another Smart SSD prototype with FPGAs [32]. They studied operations including group-by and integrated the prototype with MySQL storage engines. In the data mining area, Bae et al. studied offloading functions like k-means [2]. In the data analytics area, De et al. proposed to push down key-value stores inside SSDs [8]. In the system area, Seshadri et al. built the Willow system [24] and studied the offloading of many applications, e.g., file system,

transactional processing. Tseng et al. built the Morpheus system to offload object deserialization [29].

Our work investigates the potential benefit of Smart SSDs to another important operation – list intersection, which has not been covered before.

7. CONCLUSION

Executing programs within Smart SSDs is a new computing paradigm to make full use of the SSDs' hardware capabilities. Rather than transfer the data to main memory where the CPU can access it, Smart SSDs execute code inside SSDs directly, exploiting the high internal bandwidth that SSDs provide. This work studies the offloading of an important operation – list intersection to Smart SSDs. By working on the Samsung Smart SSD, we find many factors that can affect the performance and energy of executing list intersection within Smart SSDs:

(1) **Device memory speed**. Although low-clocked on-device processors slow down performance, the penalty incurred by high memory access is even higher. This is a new result to the community while existing studies on Smart SSDs did not emphasize [4, 11, 24].

(2) **Internal bandwidth**. The internal bandwidth of SSDs is very important since the benefit of running list intersection within Smart SSDs comes primarily from the high internal bandwidth.

(3) **Entry size**. The entry size is also an important factor to list intersection, large entry sizes reduce memory accesses per page. Smart SSDs can accelerate list intersection when the entry size s is large enough.

(4) **List size ratio**. The list size ratio is also an important factor, which determines to what extent the skipping can occur. The higher the ratio is, the more likely that Smart SSDs can accelerate list intersection.

(5) **Intersection size**. Finally, the intersection size can also affect the performance of Smart SSDs, because it determines the amount of data transferred to the host. Smart SSDs can accelerate list intersection when the intersection size is small.

Our results also have implications for both SSD vendors to design powerful Smart SSDs and applications to fully utilize the capabilities of Smart SSDs. (1) On the SSD vendor side, while it is always good to have faster processors and high internal bandwidths, it is urgent to improve the memory access speed. Sometimes, the memory access time can even dominate the total query processing time (Figure 3). To do so, SSD vendors can introduce more caches (both data-cache and instruction-cache) to SSDs. (2) On the application side, clearly not all applications can benefit from Smart SSDs. They have to be I/O-intensive and computationally-simple. Also, the number of memory accesses should be small and the output size should be (much) smaller than the input size.

8. REFERENCES

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: programming model, algorithms and evaluation. In *ASPLOS*, pages 81–91, 1998.

[2] D. Bae, J. Kim, S. Kim, H. Oh, and C. Park. Intelligent ssd: a turbo for big data mining. In *CIKM*, pages 1573–1576, 2013.

[3] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson. Near-data processing: insights from a micro-46 workshop. *Micro, IEEE*, 34(4):36–42, 2014.

[4] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger. Active disk meets flash: a case for intelligent ssds. In *ICS*, pages 91–102, 2013.

[5] I. S. Choi, W. Yang, and Y. Kee. Early experience with optimizing I/O performance using high-performance ssds for in-memory cluster computing. In *BigData*, pages 1073–1083, 2015.

[6] T. Claburn. Google plans to use intel SSD storage in servers. http://www.networkcomputing.com/storage/google-plans-to-use-intel-ssd-storage-in-servers/d/d-id/1067741, 2008.

[7] J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *TOIS*, 29(1):1–25, 2010.

[8] A. De, M. Gokhale, R. Gupta, and S. Swanson. Minerva: accelerating data analysis in next-generation ssds. In *FCCM*, pages 9–16, 2013.

[9] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *SODA*, pages 743–752, 2000.

[10] B. Ding and A. C. König. Fast set intersection in memory. *PVLDB*, 4(4):255–266, 2011.

[11] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query processing on smart ssds: opportunities and challenges. In *SIGMOD*, pages 1221–1230, 2013.

[12] Jülich Research Center. Blue gene active storage boosts i/o performance at jsc. http://cacm.acm.org/news/169841-blue-gene-active-storage-boosts-i-o-performance-at-jsc, 2013.

[13] Y. Kang, Y. Kee, E. L. Miller, and C. Park. Enabling cost-effective data processing with smart ssd. In *MSST*, pages 1–12, 2013.

[14] K. Kannan. The design of a mass memory for a database computer. In *ISCA*, pages 44–51, 1978.

[15] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Rec.*, 27(3):42–52, 1998.

[16] S. Kim, H. Oh, C. Park, S. Cho, and S. Lee. Fast, energy efficient scan inside flash memory. In *ADMS*, pages 36–43, 2011.

[17] S. Kim, H. Oh, C. Park, S. Cho, S.-W. Lee, and B. Moon. In-storage processing of database scans and joins. *Information Sciences*, 327:183–200, 2016.

[18] W. Lang and J. M. Patel. Energy management for mapreduce clusters. *PVLDB*, 2010.

[19] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[20] R. Merritt. Facebook likes wimpy cores, cpu subscriptions. http://www.eetimes.com/document.asp?doc_id=1261990, 2012.

[21] M. Miller. Bing's new back-end: cosmos and tigers and scope. http://searchenginewatch.com/sew/news/2116057/bings-cosmos-tigers-scope-oh, 2011.

[22] Oracle Corporation. Oracle exadata white paper, 2010.
[23] E. Riedel, G. A. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *VLDB*, pages 62–73, 1998.
[24] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. Willow: a user-programmable ssd. In *OSDI*, pages 67–80, 2014.
[25] M. Smolaks. Google is testing qualcomm's 24-core arm chipset. http://www.datacenterdynamics.com/servers-storage/report-google-is-testing-qualcomms-24-core-arm-chipset/95681.fullarticle, 2016.
[26] S. Y. W. Su and G. J. Lipovski. Cassm: a cellular system for very large data bases. In *VLDB*, pages 456–472, 1975.
[27] S. Tatikonda, B. B. Cambazoglu, and F. P. Junqueira. Posting list intersection on multicore architectures. In *SIGIR*, pages 963–972, 2011.
[28] Teradata Corporation. Teradata extreme performance alliance. http://www.teradata.com/t/extreme-performance-appliance.
[29] H.-W. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, and S. Swanson. Morpheus: creating application objects efficiently for heterogeneous computing. In *ISCA*, 2016.
[30] J. Wang, E. Lo, M. L. Yiu, J. Tong, G. Wang, and X. Liu. The impact of solid state drive on search engine cache management. In *SIGIR*, pages 693–702, 2013.
[31] J. Wang, E. Lo, M. L. Yiu, J. Tong, G. Wang, and X. Liu. Cache design of ssd-based search engine architectures: an experimental study. *TOIS*, 32(4):1–26, 2014.
[32] L. Woods, Z. István, and G. Alonso. Ibex - an intelligent storage engine with support for advanced sql off-loading. *PVLDB*, 7(11):963–974, 2014.
[33] L. Woods, J. Teubner, and G. Alonso. Less watts, more performance: an intelligent storage engine for data appliances. In *SIGMOD*, pages 1073–1076, 2013.
[34] S. L. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos. Beyond the wall: near-data processing for databases. In *DaMoN*, 2015.