2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

Biscuit: A Framework for Near-Data Processing of Big Data Workloads

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, Duckhyun Chang
Memory Business, Samsung Electronics Co., Ltd.

***Abstract*—Data-intensive queries are common in business intelligence, data warehousing and analytics applications. Typically, processing a query involves full inspection of large in-storage data sets by CPUs. An intuitive way to speed up such queries is to reduce the volume of data transferred over the storage network to a host system. This can be achieved by filtering out extraneous data within the storage, motivating a form of near-data processing. This work presents Biscuit, a novel near-data processing framework designed for modern solid-state drives. It allows programmers to write a data-intensive application to run on the host system and the storage system in a distributed, yet seamless manner. In order to offer a high-level programming model, Biscuit builds on the concept of data flow. Data processing tasks communicate through typed and data-ordered ports. Biscuit does not distinguish tasks that run on the host system and the storage system. As the result, Biscuit has desirable traits like generality and expressiveness, while promoting code reuse and naturally exposing concurrency. We implement Biscuit on a host system that runs the Linux OS and a high-performance solid-state drive. We demonstrate the effectiveness of our approach and implementation with experimental results. When data filtering is done by hardware in the solid-state drive, the average speed-up obtained for the top five queries of TPC-H is over 15×.**

***Keywords*-near-data processing; in-storage computing; SSD;**

I. Introduction

Increasingly more applications deal with sizable data sets collected through large-scale interactions [1, 2], from web page ranking to log analysis to customer data mining to social graph processing [3–6]. Common data processing patterns include *data filtering*, where CPUs fetch and inspect the entire dataset to derive useful information before further processing. A popular processing model is MapReduce [7] (or Hadoop [8]), in which a data-parallel map function filters data. Another approach builds on traditional database (DB) systems and utilizes select and project operations in filtering structured records.

Researchers have long recognized the inefficiency of traditional CPU-centric processing of large data sets. In order to make data-intensive processing performance and energy efficient, consequently, prior work explored alternative, *near-data processing* (NDP) strategies that take computation to storage (i.e., data source) rather than data to CPUs [9–17]; these studies argue that excess compute resources within "active disks" could be leveraged to locally run data processing tasks. As storage bandwidth increases significantly with the introduction of solid-state drives (SSDs) and data-intensive applications proliferate, the concept of user-programmable active disk becomes even more compelling; energy efficiency and performance gains of two to ten were reported [12–15].[1]

Most prior related work aims to quantify the benefits of NDP with prototyping and analytical modeling. For example, Do et al. [12] run a few DB queries on their "Smart SSD" prototype to measure performance and energy gains. Kang et al. [20] evaluate the performance of relatively simple log analysis tasks. Cho et al. [13] and Tiwari et al. [14] use analytical performance models to study a set of data-intensive benchmarks. While these studies lay a foundation and make a case for SSD-based NDP, they remain limitations and areas for further investigation. First, prior work focuses primarily on proving the concept of NDP and pays little attention to designing and realizing a practical framework on which a full data processing system can be built. Common to prior prototypes, critical functionalities like dynamic loading and unloading of user tasks, standard libraries and support for a high-level language, have not been pursued. As a result, realistic large application studies were omitted. Second, the hardware used in some prior work is already outdated (e.g., 3Gbps SATA SSDs) and the corresponding results may not hold for future systems. Indeed, we were unable to reproduce reported performance advantages of in-storage data scanning in software on a state-of-the-art SSD. We feel that there is a strong need in the technical community for realistic system design examples and solid application level results.

This work describes *Biscuit*, a user-programmable NDP framework designed specifically for fast solid-state storage devices and data-intensive applications. We portray in detail its design and system realization. In designing Biscuit, our primary focus has been ensuring a high level of programmability, generality (expressiveness) and usability. We make the following key design choices:

Programming model: Biscuit is inspired by flow-based programming models [21]. A Biscuit application is constructed of tasks and data pipes connecting the tasks. Tasks may run on a host computer or an SSD. Adoption of the flow-based programming model greatly simplifies programming as users

[1]Another approach to NDP could take place in the context of main memory (e.g., intelligent DRAM [18]). Our work specifically targets NDP within the secondary storage and does not consider such main memory level processing. See Balasubramonian et al. [19] for a list of recent work spanning both approaches.

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.23

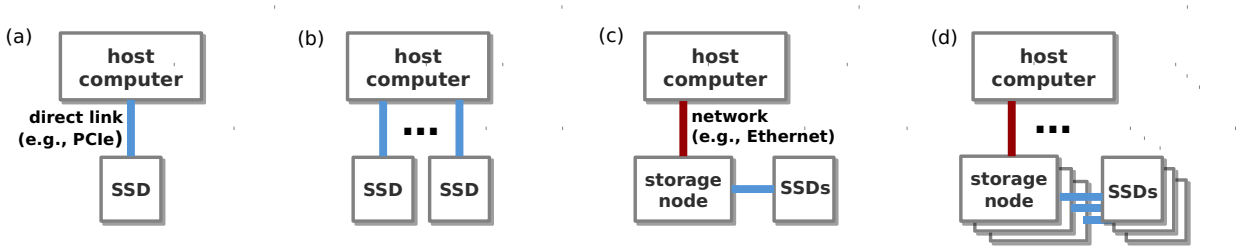

Figure 1. Various host computer and storage organizations. (a) Simple: A host computer with a single SSD. (b) Scale-up: A host computer with multiple SSDs. (c) Networked: A host computer with a networked storage node (e.g., shared SAN). (d) Scale-out: A host computer with multiple networked storage nodes (e.g., Hadoop cluster).

are freed of orchestrating communication or manually enforcing synchronization.

Dynamic task loading/unloading: Biscuit allows the user to dynamically load user tasks to run on the SSD. Resources needed to run user tasks are allocated at run time. This feature decouples user application development and SSD firmware development, making NDP deployment practical.

Language support: Biscuit supports (with few exceptions) full C++11 features and standard libraries. By programming at a high level, users are expected to write their applications with ease and experience fewer errors.

Thread and multi-core support: Biscuit implements lightweight fiber multithreading and comes natively with multi-core support. Moreover, it allows programmers to seamlessly utilize available hardware IPs (e.g., DB scanner [22]) by encapsulating them as built-in tasks.

As such, the main contribution of this paper is the design and implementation of Biscuit itself. Biscuit currently runs on a Linux host and a state-of-the-art enterprise-class NVMe SSD connected to the host through fast PCIe Gen.3 links. To date, Biscuit is the first NDP system reported, that runs on a commodity NVMe SSD. This paper also discusses challenges met during the course of system realization and key design trade-offs made along the way. Because it is unlikely for a single framework to prevail in all possible applications, future NDP systems will likely require multiple frameworks. We believe that the work described in this paper sets an important NDP system design example. We also make other major contributions in this work:

- We report through measurement performance of key operations of NDP on a real, high-performance SSD. For example, it sustains sequential read bandwidth in excess of 3GB/s using PCIe Gen.3 $\times$4 links. The SSD-internal bandwidth (that an NDP program can tap) is shown to be higher than this bandwidth by more than 30%.
- On top of Biscuit, we successfully ported a version of MySQL [23], a widely deployed DB engine. We modified its query planner to automatically identify and offload certain data scan operations to the SSD. Portions of its storage engine are rewritten so that an offloaded operation is passed to the SSD at run time and data are exchanged with the SSD using Biscuit APIs. Our revised MySQL runs all 22 TPC-H queries. No prior work reports a full port of a major DB engine to an NDP system or runs all TPC-H queries.
- Our SSD hardware incorporates a pattern matcher IP designed for NDP. We write NDP codes that take advantage of this IP. When this hardware IP is applied, our MySQL significantly improves TPC-H performance. The average end-to-end speed-up for the top five queries is 15.4$\times$. Also, the total execution time of all TPC-H queries is reduced by 3.6$\times$.

In the remainder of this paper, we first give the background of this work in Section II by discussing system organizations for NDP and describing what we believe are important for a successful NDP framework. Section III presents Biscuit, including its overall architecture and key design decisions made. Details of our Biscuit implementation are described in Section IV, followed by experimental results in Section V. We discuss our research findings in Section VI and related work in Section VII. Finally, Section VIII concludes.

II. Background

A. System Organizations for NDP

The concept of NDP may be exercised under various system configurations and scenarios. In one embodiment, an entire task may be executed within storage; consider how map functions are dispatched to distributed storage nodes [7, 8]. In another case, a particular task or kernel of an application (like database scan) could be offloaded to run in storage (e.g., Oracle Exadata [24]). In yet another example, a large dataset could be partitioned onto and taken care of by the host computer and storage cooperatively [13]. Fig. 1 shows various system organizations on which an NDP architecture and data processing scenarios may be defined.

In a "direct-attached storage" (DAS) organization (Fig. 1(a) and (b)), there are one or more SSDs attached to a host computer. Connection between a host computer and DAS storage is typically low latency ($<$ 10µs), and when bundled (i.e., multiple ports or lanes form a connection), high bandwidth ($>$ 3GB/s). Most recent work on active SSD assumes a DAS organization [12–15, 20]. Compared to Simple (Fig. 1(a)), Scale-up (Fig. 1(b)) has more aggregate compute resources (in SSDs) as well as internal media bandwidth.

NDP could be done in a networked system organization (depicted in Fig. 1(c) and (d)). In both cases, a host computer may offload work to a storage node (like in the Exadata example and Hadoop). In turn, each storage node (essentially a server) may offload work to its local SSD(s). Optimally partitioning and mapping a given data processing job to heterogeneous resources is a hard challenge, and addressing the problem is beyond the scope of this paper. Nonetheless, a programming and runtime framework to seamlessly integrate host and device resources in an NDP system remains a critical element when deploying such a system.

B. Constructing a Framework for NDP

Then, what are key ingredients needed for building a desirable programming framework for NDP? We list in the following five factors that we find are essential:

Ability to run user-written code on a device. A practical NDP system would allow users to specify a user binary to be dynamically loaded into a device and executed. Ideally, this procedure should be as effortless as running a user binary on a host system, with no need for recompiling existing SSD firmware or dealing with low-level SSD protocols.

Abstract, yet efficient communication between host application and storage-side tasks. Major challenges arise if application programmers have to deal with low-level communication between a host and an SSD. Therefore, a desirable NDP framework must offer an abstraction of data communication between the parties with well-defined semantics.

Efficient resource utilization in runtime. Runtime on an NDP device must utilize resources in an efficient manner. In many SSDs, resources that could be appropriated toward near-data computation are limited. To ensure performance and wider applications, the runtime must carefully conserve its use of compute resources, on-chip/off-chip memory capacity, DMA channels, and memory bandwidth.

Intuitive, high-level programming. Support for high-level language as well as a complete application programming interface are desired, to increase programming productivity and reduce errors. This requirement may complicate runtime implementation on an embedded device, however.

Safety. Lastly, ensuring safety of the overall system is important. For example, ill-behaving user code must not adversely affect the overall operation of an SSD or compromise the integrity of data that belong to a different user. Lack of certain hardware features within an SSD may present challenges to enforcing safety policies.

III. Biscuit

A. System Architecture and Overview

A basic Biscuit system is comprised of at least two physical components: A host system and one or more SSDs (like Simple and Scale-up of Fig. 1). It equally applies to storage nodes that connect to SSD devices in a networked system (Networked and Scale-out). On such physical platforms, Biscuit provides an environment and programming model for host-side and SSD-side tasks to work together.

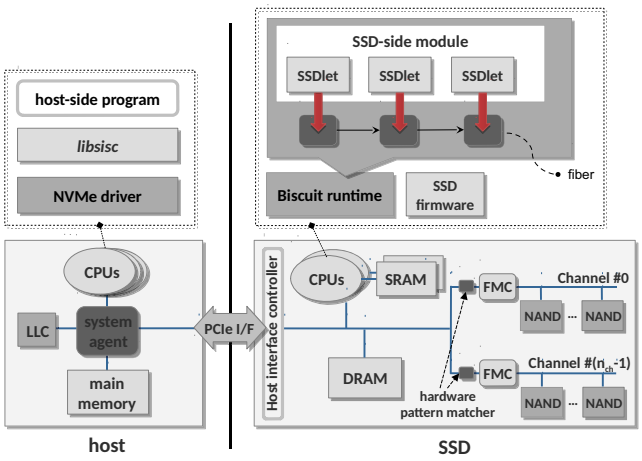

Figure 2. Overall architecture of a Biscuit system.

Fig. 2 depicts the overall Biscuit system architecture. A Biscuit system has distributed software modules and a hardware platform—a host computer and an SSD. Biscuit turns the host and the SSD into a seamless application execution environment with the help of a dedicated runtime on the SSD and a set of high-level APIs offered by Biscuit libraries. These APIs collectively reflect the programming model. An important principle in the Biscuit programming model is separation of *computation model* and *coordination model*, as shown in Fig. 3. The computation model governs how to describe a task unit by specifying computation using its input and output data. On the other hand, the coordination model has to do with creating and managing tasks, and establishing producer and consumer relationship among tasks by associating their inputs and outputs. Biscuit follows the flow-based programming (FBP) model [21] for its coordination model, but allows more general data types for data communication with C++11 move semantics, in addition to the *information packet* type of FBP. Biscuit is designed as an in-storage compute framework. Compared to a host system, SSDs have high internal I/O bandwidth but

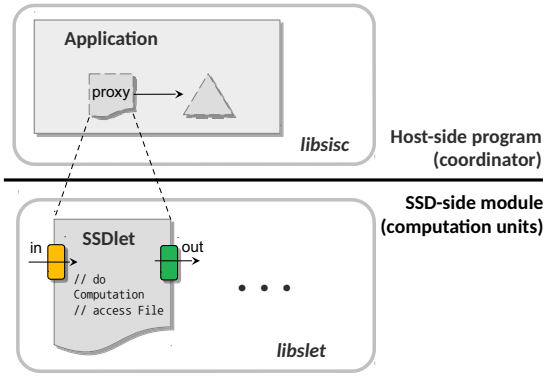

Figure 3. Biscuit programming model.

API是一些封装好的函数，编译好的程序，这些预先定义的函数供人使用，目的是透明得提供应用程序与开发人员基于某软件或硬件得以访问一组例程的能力，而又无需访问源码，或理解内部工作机制的细节。

C语言 API 以函数的形式呈现，例如 printf()、scanf()、fopen() 等。

Java API 主要以类的形式呈现，例如 String、Thread、Date 等。

C++ 是在C语言的基础上进行的扩展，所以 C++ API 既包含函数也包含类。

```
class Filter : public SSDLet<IN_TYPE<int32_t>,
        OUT_TYPE<int32_t, bool>, ARG_TYPE<double>> {
public:
    void run() override {
        auto in = getInputPort<0>();
        auto out0 = getOutputPort<0>();
        auto out1 = getOutputPort<1>();
        double& value = getArgument<0>();

        // do some computation
}};
```

API

Code 1. An example to illustrate SSDLet interface.

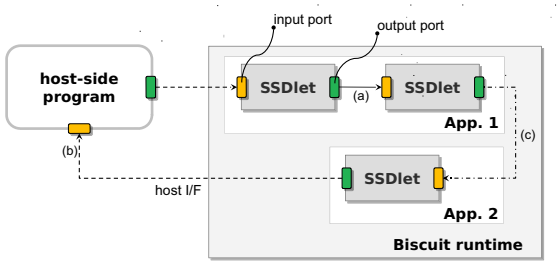

Figure 4. InputPort and OutputPort.

relatively low compute power. Thus, Biscuit fits perfectly into data-centric workloads and can achieve maximum performance by offloading a set of simple data-intensive tasks to SSDs and retrieving intermediate/final computational results only. This programming model is well represented by FBP. Additionally, Biscuit employs more aggressive type checking at compile and run time.

The computation model and the coordination model are realized in the *libslet* and *libsisc* libraries with corresponding interfaces provided to users. libslet is used to build SSD-side modules, which may contain multiple binary images of SSD-side tasks. A host-side program is composed of code to invoke and coordinate execution of SSD-side tasks using libsisc, and code for host-side computation.

B. SSDlet and Application

An *SSDlet* is a simple C++ program written with Biscuit APIs, and is typically embedded inside an *SSDlet module*. It is a unit of execution independently scheduled, and represented by the SSDLet class. Both libslet and libsisc have this class. SSDLet of libslet is used to define an SSDlet. As shown in Code 1, SSDLet::run is the core method that describes the execution of an SSDlet, and is called when a host-side program orders SSDlets to begin execution. In contrast, SSDLet of libsisc is a proxy class acting as an interface to SSDLet instances on an SSD.

In addition to deriving their own SSDlets from SSDLet and overriding SSDLet::run, programmers should parameterize the SSDLet class to add I/O ports to the SSDlet (as will be explained further in Section III-C), or pass initial arguments to it. The SSDLet class is a template class with three type parameters, as listed in Code 1, IN_TYPE, OUT_TYPE, and ARG_TYPE. Biscuit provides underlying facilities for the programmer to easily declare parameter and retrieve arguments, as well as to inject and eject data through I/O ports.

A group of SSDlets that run cooperatively (e.g., Mappers, Shuffler, and Reducers in Fig. 5) are represented by *Application*. With this class, a host-side program coordinates operations on SSDLet instances by asking them to establish connections among them and start execution. Biscuit can load multiple applications and run them together, either cooperatively or independently.

C. I/O Ports

Upon successful creation of all SSDlet instances, data transmission among them is achieved by simply linking input and output ports as shown in Fig. 4. Biscuit has three different port types:

(a) Inter-SSDlet ports transfer data among SSDlet instances belonging to a single Application instance. They support almost all data types except pointer and array types. *Single producer multiple consumer* (SPMC) and *multiple producer single consumer* (MPSC) connections are allowed among SSDlet instances in addition to *single producer single consumer* (SPSC) connection.

(b) Host-to-device ports transfer data between an SSDlet instance and a host program. They support a sole data type, namely, Packet. To transfer data of other types, data should be convertible to/from Packet by explicit serialization/deserialization functions. This port type only allows the SPSC connection.

(c) Inter-application ports transfer data between two SSDlets from different Application instances. Similar to the host-to-device ports, they only support the Packet data type, or require that data be serializable and deserializable. Again, this port type only allows the SPSC connection.

Biscuit API is strongly typed and implicit type conversion is not allowed. Thus, users must connect different ports of identical data type. For example, they cannot connect a string output to a numeric input. Every data transmitted between a host-side program and an SSDlet or between SSDlets of different applications must be (de)serializable. Our design choices promote type checking during compile time, giving programmers opportunities to fix errors early.

D. File I/O

One of the key strengths of Biscuit is simplicity in handling files. File access APIs are nearly identical to ones in standard libraries, and thus provide users with a flexible interface in familiar semantics. Note that Biscuit prohibits SSDlets from directly using low-level, logical block addresses and forces the SSD to operate under a file system when SSDlets read and write data.

Similar to an SSDlet, files are represented as File classes in both libsisc and libslet. A host-side program creates file instances with File of libsisc and passes them to an SSDlet either as arguments or via ports. Using File of libslet, the

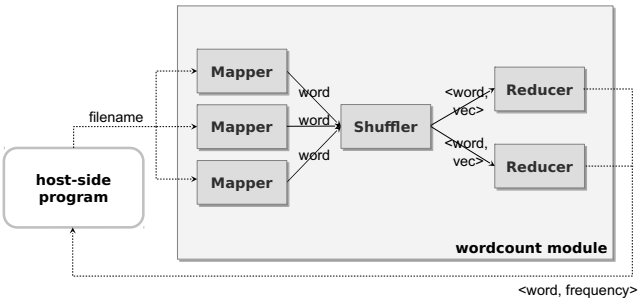

Figure 5. Biscuit programming example: Wordcount.

SSDlet can read/write passed file instances. This separation of functionality ensures that access permission of an SSDlet to files is inherited from that of the associated host program.

The File class supports both synchronous and asynchronous APIs for reading, while an asynchronous write API and a synchronous flush API are provided for writing data. If an application program requires high bandwidth file I/O, asynchronous APIs are recommended.

E. Working Example: Wordcount

Before closing this section, let us consider a wordcount application as a working example. Wordcount is a canonical MapReduce example where Mapper(s), Shuffler(s), and Reducer(s) count the frequency of each word in a given input file in a parallel, distributed manner. As shown in Fig. 5, it is composed of a host-side program and a wordcount module with Mapper, Shuffler, and Reducer SSDlets.

The Mapper class in Code 2 is an implementation of Mapper. It takes a file as an argument and passes tokenized words to Shuffler. Lastly, it is registered in its container module by RegisterSSDLet so that a host-side program can create proxies of this class with a registered identifier.

Code 3 is a host-side implementation of wordcount. First, it specifies the target device with an SSD instance, and loads the wordcount module on it. Second, it creates an Application instance and proxy SSDLet instances. Each SSDLet instance is instantiated by an identifier (e.g., idMapper), and acts as an interface for an SSDLet instance whose class is registered with this identifier. Then, it establishes connections among tasks with Application::connect and Application::connectTo. Calling Application::start makes sure that all SSDlets begin execution after their communication channels are completely set up. Then, by calling the InputPort::get, it gets the word-frequency pairs from Reducer and prints them out.

IV. Implementation

A. SSD Hardware

The target SSD used in this work is a state-of-the-art commercial product, depicted in Fig. 6. Specification of our SSD is summarized in Table I. Additional details that pertain to our design and implementation trade-offs are:

```
class Mapper : public SSDLet<OUT_TYPE<std::pair<std::string, uint32_t>>,
                             ARG_TYPE<File>> {
public:
    void run() {
        auto& file = getArgument<0>();
        FileStream fs(std::move(file));
        auto output = getOutputPort<0>();
        while (true) {
            ...
            if (!readline(fs, line)) break;
            line.tokenize();
            while ((word = line.next_token()) != line.cend()) {
                // put output (i.e., each word) to the output port
                if (!output.put({std::string(word), 1})) return;
}}}};

RegisterSSDLet(idMapper, Mapper) // register class in its container module
```

Code 2. An example SSDLet class: Mapper.

```
int main(int argc, char *argv[]) {
    SSD ssd("/dev/nvme0n1");
    auto mid = ssd.loadModule(File(ssd, "/var/isc/slets/wordcount.slet"));

    // create an Application instance and proxy SSDLet instances
    Application wc(ssd);
    SSDLet mapper1(wc, mid, "idMapper", make_tuple(File(ssd, filename)));
    SSDLet shuffler(wc, mid, "idShuffler");
    SSDLet reducer1(wc, mid, "idReducer");
    ...
    // make connections between SSDlets and from Reducers back to the host
    wc.connect(mapper1.out(0), shuffler.in(0));
    wc.connect(shuffler.out(0), reducer1.in(0));
    auto port1 = wc.connectTo<pair<string, uint32_t>>(reducer1.out(0));
    ...
    // start application so that all SSDlets would begin execution
    wc.start();
    pair<string, uint32_t> value;
    while (port1.get(value) || port2.get(value)) // print out <word,freq> pairs
        cout << value.first << "\t" << value.second << endl;

    // wait until all SSDlets stop execution and unload the wordcount module
    wc.wait();
    ssd.unloadModule(mid);
    return 0;
}
```

Code 3. An example host-side program: Wordcount.

- The SSD has two types of memory: A small fast memory (on-chip SRAM) and a large slow memory (DRAM).
- The SSD has memory protection units (MPUs) but no memory management units (MMUs).
- The SSD partially supports LDREX and STREX instructions [25], which introduces restriction on using some synchronization primitives.
- Each flash memory channel of the target SSD has a *hardware pattern matcher*. If an SSDlet requests a read operation on a large data chunk with the pattern matcher turned on, the request is distributed to multiple flash memory channels, and the data will flow through the pattern matchers on those channels in parallel. The raw pattern matching throughput corresponds to the throughput of the flash memory channels.

Hardware limitations—less-than-ideal compute power, no cache coherence, a small amount of fast memory, no MMU, and restrictive synchronization primitives—pose challenges to efficient realization of our runtime on this platform while we strive to meet all requirements of the programming model.

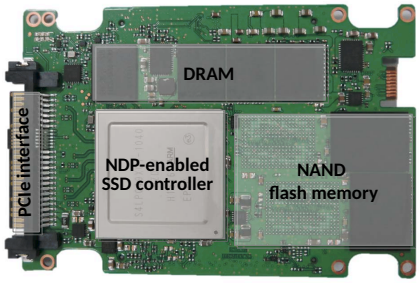

Figure 6. The SSD hardware. Only one side is shown.

B. Biscuit Runtime

Biscuit system components span a host system and an SSD (see Fig. 2). The primary features of Biscuit are multithreading support, efficient data communication, dynamic module loading, and dynamic memory allocation. The Biscuit runtime centrally mediates access to SSD resources and has complete control over all events occurring in the framework.

Cooperative Multithreading. At the request of a host-side program, the Biscuit runtime loads modules on SSDs, instantiates SSDlets, and coordinates their execution. To execute multiple SSDlet instances concurrently, we employ *cooperative multithreading* [26]. Whenever an SSDlet instance is created, it is assigned a fiber (a scheduling unit) and then the fiber executes SSDLet::run of the SSDlet instance.

Cooperative multithreading is a limited form of general multithreading in that context switching occurs only in explicit *yield* calls or blocking I/O function calls. This aspect has several advantages over fully preemptive multithreading, such as very low context switching overhead and resource sharing without locking.

For multi-core support, Biscuit uses applications as a unit of multi-core scheduling instead of SSDlets. This means that all SSDlets in an application are guaranteed to run on the same processor. It has a considerable influence on the implementation of I/O ports, which will be explained next.

I/O Ports as Bounded Queues. As explained in Section III-C, all data transmission except file I/O is done through ports. Biscuit extensively uses bounded queues to implement ports. When the host program makes a connection between an output port and an input port, Biscuit creates a bounded queue and associates each port to both end-points of the queue. Sending and receiving data is achieved by simple enqueue and dequeue operations on the queue.

Complex connections like SPMC and MPSC are elegantly realized with a shared queue. No lock is needed in implementing SPMC and MPSC, because they are supported only in inter-SSDlet ports and the fibers corresponding to those SSDlets are executed on the same processor.

Unlike inter-SSDlet ports, other ports are allowed only an SPSC connection. This is not only a design policy but also a result from the limitation of the target SSD; the SSD does not provide a complete set of synchronization primitives needed to implement multiple producer and multiple consumer queues. In case of host-to-device ports and inter-application ports, producers or consumers may run on different processors and simple bounded queues are not thread-safe any longer.

The host-side and device-side *channel managers* are a mediator for data transfer between a host program and SSDlets. Bounded queues for host-to-device communication are encapsulated in channels. Each channel has three queues: One for incoming data and two for outgoing data. To maintain channels in memory for efficient re-use as well as to limit the total number of channels simultaneously used, channel managers create their own channel pool. All requests from a host-side program to SSDlets and the corresponding responses are encapsulated in channels and transmitted by a host interface protocol.

Dynamic Module Loading. SSDlets must be compiled and linked with libslet into an SSD-side module. Libslet is in charge of bridging the Biscuit runtime and SSDlet instances. When an SSDlet instance is created, it gets a function table from the runtime. This table contains one pointer entry for a function or a software interrupt. SSDlets require a collection of functions to perform their tasks, including key functions like memory allocation and I/O. On the other side, Biscuit also needs functions to create and manage SSDlet instances.

Even though our target SSD does not support MMU or virtual memory, Biscuit can create multiple SSDlet instances from one SSDlet binary because it performs symbol relocation and locates each one in a separate address space when creating an SSDlet instance.

Dynamic Memory Allocation. Biscuit supports dynamic memory allocation. Our current implementation builds on Doug Lea's memory allocator [27]. Biscuit maintains two kinds of memory allocators by default, *system memory allocator* and *user memory allocator*. The system memory allocator is used when the runtime allocates memory space and instantiates dynamic objects. Memory spaces used by the system memory allocator are restricted to Biscuit, and SSDlet instances are prohibited from accessing them. The user memory allocator is for securing memory spaces accessible by SSDlet instances.

Table I
SSD SPECIFICATION.

Item	Description
Host interface	PCIe Gen.3 ×4 (3.2GB/s max. throughput)
Protocol	NVMe 1.1
Device density	1 TB
SSD architecture	Multiple channels/ways/cores
Storage medium	Multi-bit NAND flash memory
Compute resources for Biscuit	Two ARM Cortex R7 cores @750MHz with L1 cache, no cache coherence
Hardware IP	Key-based pattern matcher per channel

C. Host-side Library

As explained in Section III, host programs are linked with the libsisc library. It has a layered architecture; at the bottom of libsisc is a host-to-device communication layer, comprised of a channel manager and its components specialized for different host interface protocols (like NVMe or Ethernet). The channel manager is in charge of managing multiple independent communication channels between libsisc and the Biscuit runtime. Using the channel manager, libsisc maintains a single control channel and creates one or more data channels on demand. The classes provided for programmers, such as SSD, Application, and SSDLet, reside on top and are implemented using the control channel. Data channels are given to input/output port instances for data communication.

V. Experiments

This section presents experimental results to illustrate potential benefits of Biscuit in data-intensive workloads. Results are presented in two parts, basic performance and application level performance. Note that our main goal in presenting performance numbers is not to uncover theoretical maximum performance gains NDP can bring per se. Rather, we present measured numbers on a real system, and the presented numbers reveal what design aspects affect overall system performance and how. Our goal is then to obtain insights into how to derive better NDP architectures and better system design strategies.

A. Experimental Setup

All experiments are performed on a target platform comprised of a Dell PowerEdge R720 server system [28] and our target SSD (Section IV-A). The server system sports two Intel Xeon(R) CPU E5-2640 (12 threads per socket) @2.50GHz and 64 GiB DRAM. 64-bit Ubuntu 15.04 is chosen as the system OS. Relevant details of our target SSD are listed in Table I. We note that the SSD is equipped with a hardware pattern matcher, designed specifically for NDP. Given at most three keywords, each of which is up to 16 bytes long, the pattern matcher looks them up in a configurable amount of data retrieved from the storage medium. In the rest of this section, we call the system configuration with a default, conventional SSD as Conv and the system configuration with the Biscuit framework on the SSD as Biscuit. Unless specified, measurements are made on an unloaded system.

B. Basic Performance Results

The basic performance is measured in latency of port communication and read operation. It is also measured in bandwidth of read operation. We measure the communication latency involved in transmitting data in the Packet type via three different I/O port types of Biscuit (Section III-C). The read latency simply measures the amount of time taken for a single read request to be completed. While it is a round-trip time from the host system back to the host system for Conv, it is a round trip time from and to an SSDlet in the case of Biscuit (i.e., "internal read"). The read bandwidth measures the amount of data transferred in a given time during read I/O operations. Again, it is internal read in the case of Biscuit. It is measured and presented as a function of read request size.

Table II
Measured latency for different I/O port types.

	Host-to-device H2D	Host-to-device D2H	Inter-SSDlet	Inter-app.
Latency (µs)	301.6	130.1	31.0	10.7

Communication latency. The measured communication performance is shown in Table II for the three I/O port types. The communication performance between the host and the device is further divided into two categories depending on the direction of transmission, namely, host-to-device (H2D) and device-to-host (D2H). All reported latencies include a scheduling latency involved in the context switching of fiber (Section IV-B) in common, which is dominantly shown in the case of inter-application latency. The inter-SSDlet latency is longer than the inter-application latency by 20.3 µs, which is due to the type abstraction and de-abstraction process involved in the inter-SSDlet port. The host-to-device communication latency includes not only the latency caused by queuing and dequeuing in the channel manager, but also the latency caused by low-level hardware (PCIe) and software (device driver). With these latencies accumulated, the host-to-device communication latency reaches 130.1 µs and 301.6 µs, respectively for D2H and H2D. Our result shows that the H2D latency is longer than the D2H latency; in our implementation, the channel manager has about twice the work to do in the receiver side compared to the sender side. Because CPUs in the target SSD are limited in compute power, the H2D latency is noticeably longer.

While we find that there is room for architecture and software optimizations (e.g., clock frequency scaling, protocol optimization), the basic performance of our framework is adequate for a range of applications we examine.

Data read latency. The read latency of Conv is measured using Linux `pread` I/O primitive. As for Biscuit, we use its internal data read API. Table III shows the measured read latency for completing a single 4-KiB read request. There is a difference of 14.1 µs, yielding about 18% shorter latency in the case of Biscuit. Part of the difference is attributed to the shorter round-trip "path"—there is no need for transmitting data from the device to the host over a host interface. Note that the relative latency difference is expected to grow if the

Table III
Measured data read latency.

	Conv	Biscuit
Latency (µs)	90.0	75.9

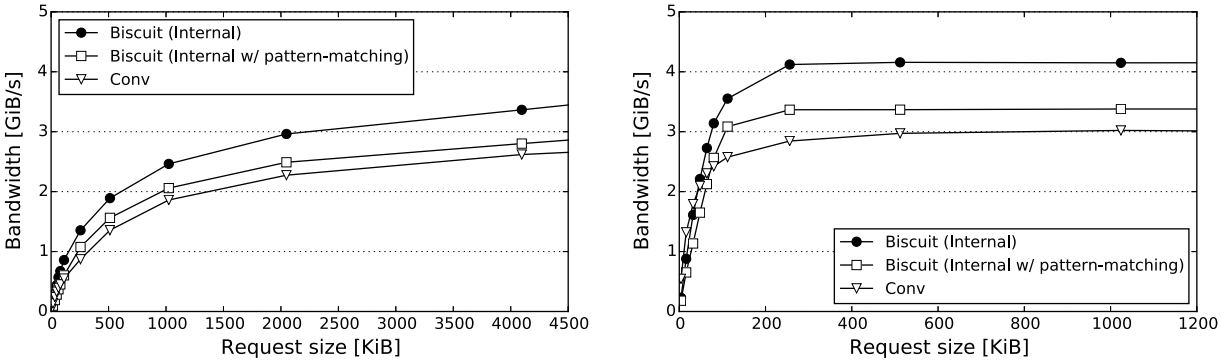

Figure 7. Bandwidth of synchronous (left) and asynchronous (right) read operations as a function of request size.

storage medium latency decreases, e.g., with a single-level cell NAND flash memory or emerging storage-class memory like phase-change RAM or resistive RAM. For example, if the storage medium latency approaches 1 µs, the relative latency difference between Conv and Biscuit will grow to over 40% even if everything else is kept same. As will be shown later, the read latency serves as an important measure for the understanding of pointer chasing performance.

Data read bandwidth. In this experiment, we measure the bandwidth of *synchronous* and *asynchronous* read operations. Synchronous read processes read requests one at a time, while asynchronous read overlaps processing of multiple read requests. Because asynchronous read eliminates the effect of idle time between successive read requests, it reaches maximum possible bandwidth more quickly.

Fig. 7 plots the measured bandwidth of synchronous read (left) and asynchronous read (right). For asynchronous read, up to 32 concurrent (or outstanding) read requests are used. As the read request size is increased, the achieved bandwidth increases rather steeply at the beginning and starts to level off gradually. Notice that the maximum bandwidth is achieved more quickly with asynchronous read; the bandwidth limit is reached as early as at a request size of about 500 KiB. In the case of synchronous read, bandwidth appears to still increase even when requests are 4 MiB or larger.

It is distinctively shown that maximum achievable bandwidth is limited by the host interface for Conv (at around 3.2 GiB/s with PCIe Gen.3 $\times$4). However, such limitation is not present in the case of Biscuit and the underutilized internal bandwidth is fully exploited, outperforming Conv by about 1 GiB/s at a request size of 256 KiB or larger in the case asynchronous read. The gap between Biscuit bandwidth and Conv bandwidth can grow if there are other interferences in play, such as fabric bottleneck (e.g., there are many SSDs on a switched PCIe fabric) or a heavy CPU load.

The plot also shows Biscuit's internal bandwidth when the hardware pattern matcher (per flash memory channel) is enabled. It lies between Conv and pure Biscuit; the pattern matching bandwidth is lower than without pattern matching because the configuration incurs software overheads for controlling hardware IP. On the other hand, this bandwidth is still higher than Conv, and underscores the capability and advantage of having a hardware IP for fast data processing.

C. Application Level Results

Pointer Chasing. Pointer chasing is a key primitive operation of data-dependent logic. Graph traversal, for example, is essentially done by numerous repeated pointer chasing operations [29]. In such scenarios, read latency is critical since performing each data-dependent logic requires a round-trip operation between the host and the storage system. With NDP, a full host-to-device-to-host round-trip delay as well as corresponding host CPU cycles are avoided by performing data-dependent logic entirely within a storage device. To evaluate the potential of this approach, we write and experiment with a simple pointer chasing benchmark on a relational DB built with Neo4j [30]. The DB is derived from the Twitter datasets [31] that contain 42 million vertices of user profiles and 1.5 billion edges of social relations, out of which we randomly choose 100 starting nodes for traversal. The generated dataset is approximately 20 GiB. In order to reveal the benefit of NDP more realistically, we repeat experiments under various system load levels generated by simultaneously running multiple threads of StreamBench [32].

Table IV gives the measured time for performing the pointer chasing benchmark on Conv and Biscuit. With Biscuit, the execution time is reduced from 138.6 to 124.4 in seconds, achieving at least an 11% performance gain. This gain is comparable to the improvement in read latency with Biscuit. In fact, as alluded to the point earlier, pointer chasing performance is directly related to read latency since the execution time is essentially the sum of individual time needed

Table IV
EXECUTION TIME FOR POINTER CHASING.

	#threads	0	6	12	18	24
Exec. time (s)	**Conv**	138.6	143.5	152.5	154.9	155.0
	Biscuit	124.4	124.0	123.3	123.9	123.5

for subsequent read operations. Hence, the relative gain will grow if the storage medium latency becomes smaller. It is also shown that the Conv performance is degraded as we increase the system load due to higher contention in the memory hierarchy. As expected, the Biscuit performance is fairly insensitive to the system load, possibly reducing tail latency and benefiting the system responsiveness.

Simple String Search. We now turn to cases where we can leverage the hardware pattern matcher in the target SSD. To demonstrate the effect of hardware accelerated matching, we write a utility application that performs simple string search on a large compilation of web-log amounting to 7.8 GiB. For Conv, we use Linux `grep`, which implements the Boyer-Moore string search algorithm [33].

As seen in Table V, there is a minimum of 5.3× improvement with Biscuit. It is shown that the performance of Conv is highly affected by the system load, whereas Biscuit is robust. As a result, the relative improvement becomes larger as we increase the system load. The maximum improvement reaches 8.3× when 24 StreamBench threads are running in the background. This large improvement comes from the fact that Biscuit sustains high, on-the-fly search throughput within the SSD thanks to the hardware pattern matcher. While the presented string search application is simple, the idea can easily be extended to many other applications in big data analytics that involve scanning and searching a large corpus of data (e.g., grep, html/xml parsing, table scan), with great potential for significant performance gains.

DB Scan and Filtering. The last application we examine is data analytics with a real DB engine. We chose MariaDB 5.5.42 [23] (an enhanced fork of MySQL, known to be in use at Google), with its default storage engine called XtraDB. In order to accelerate data analytics on this platform, we rewrite portions of MariaDB using Biscuit APIs. Specifically, we modified the query planner of MariaDB to (1) automatically identify within a given query a candidate table with filter predicates amenable for offloading; (2) perform quick check on the table to estimate selectivity using a sampling method, (3) determine whether the candidate table is indeed a good target (based on a selectivity threshold), and finally (4) offload the identified filter to the SSD. We also reorganized XtraDB's datapath for passing an offloaded operation to the SSD and for exchanging data between the host system and the SSD. Here, *selectivity* of filter predicates against a base table is calculated as a fraction of pages that satisfy filter conditions from the total number of pages in the table. The value of selectivity ranges from zero to one, with zero being the highest selectivity value. This implies that no pages

Table V
EXECUTION TIME FOR STRING MATCHING.

	#threads	0	6	12	18	24
Exec. time (s)	**Conv**	12.2	14.8	16.3	18.8	19.9
	Biscuit	2.3	2.3	2.3	2.3	2.4

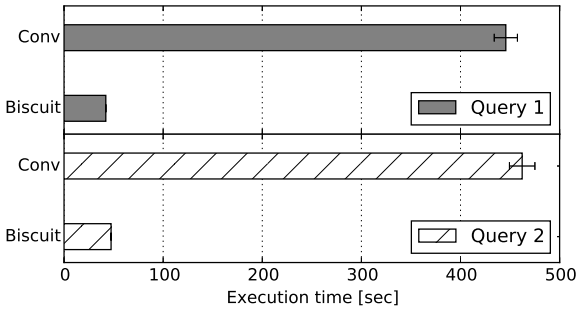

Figure 8. Performance of SQL queries on lineitem table.

will be selected from the table (best case for NDP). As a benchmark dataset, we populated TPC-H dataset [34] at a scale factor of 100. Once loaded into the DB, it becomes nearly 160 GiB in total. In the first experiment, we choose to run two filtering queries from a recent related work [35] for illustration:

```
<Query 1>
SELECT l_orderkey, l_shipdate, l_linenumber
FROM lineitem
WHERE l_shipdate = '1995-1-17'

<Query 2>
SELECT l_orderkey, l_shipdate, l_linenumber
FROM lineitem
WHERE (l_shipdate = '1995-1-17' OR l_shipdate = '1995-1-18')
      AND (l_linenumber = 1 OR l_linenumber = 2)
```

The first is a simple query with a single filter predicate, and the second has a more complex WHERE clause. Above queries have a selectivity of 0.02 and 0.04, respectively, for the l_shipdate predicate(s). Thus, both queries are characterized by a high filtering ratio with a low computational complexity.

Fig. 8 gives the result. The error bars indicate a 95% confidence interval. In the case of Conv, execution time varied significantly across ten repeated experiments, depending on CPU and cache utilization in the system. On the other hand, execution times on Biscuit were very consistent. As shown, Biscuit achieves the speed-up of about 11× and 10× for the two queries studied. Query 1 sees a slightly greater performance gain as the selectivity is higher (i.e., higher filtering ratio). It is important to point out that performance gain would depend highly on the selectivity in a given query, which is expected since the lower the rate of device-to-host transmission is, the higher the benefit of NDP can be. Similar to the simple string search application, the performance gain mainly comes from the large internal bandwidth of Biscuit, as well as the rapid scanning of the dataset by the hardware pattern matcher. Together with the high filtering ratios, Biscuit reduces the amount of data to transfer over a host interface significantly. The result clearly demonstrates the potential of the Biscuit approach to substantially improve query processing in modern big data workloads.

Power Consumption. Fig. 9 shows the measured power consumption in Watts during the execution of Query 1 as

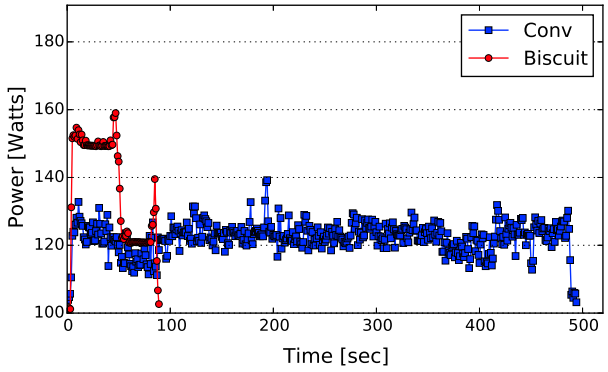

Figure 9. Measured system power consumption during the execution of Query 1.

Table VI
OVERALL ENERGY CONSUMPTION.

	Conv	Biscuit
Total Energy (kJ)	60.5	12.2

a function of time. This power is of the whole system including the server and the target SSD. As already shown in Fig 8, the execution time of Biscuit is markedly shorter.[2] The power consumption in both cases grows rapidly as the query processing begins. The average system power consumption with Conv and Biscuit are measured as 122 Watts and 136 Watts, respectively during query execution, whereas the system's idle power is 103 Watts. Biscuit consumes more power because it keeps SSD busy consuming nearly full bandwidth during query processing. Comparatively, Conv does not utilize the full bandwidth of the target SSD due to inefficiency in query processing logic and I/O. Biscuit achieves substantially lower energy consumption than Conv thanks to its significantly reduced execution time. Table VI shows that Biscuit consumes nearly 5 times less energy for Query 1.

TPC-H Results. Finally, let us present and discuss the full TPC-H execution results collected on the MariaDB system. We ran all 22 queries in the TPC-H suite [34]. TPC-H is the de facto industry standard for OLAP performance.

Fig. 10 gives our results, sorted by speed-up. Among all queries, there are eight queries that MariaDB does not attempt to leverage NDP because there is no proper filter predicate in the query (Q1, Q7, Q11, Q13, Q18, Q19, Q21, and Q22). In the case of Q1, Q7, Q11, and Q21, the query planner gives up NDP because it expects the selectivity to be very low (e.g., predicate is a single character) or the target table size is too small. Other queries are considered not a fit due to limitations of our hardware pattern matcher (e.g., it can't handle the "NOT LIKE" predicate condition) or there is

[2]The plot shows that the system does not go back immediately to the idle mode after query execution is finished. There is extra work performed (e.g., synchronizing the buffer cache). To be fair, we include this period in calculating power/energy consumption.

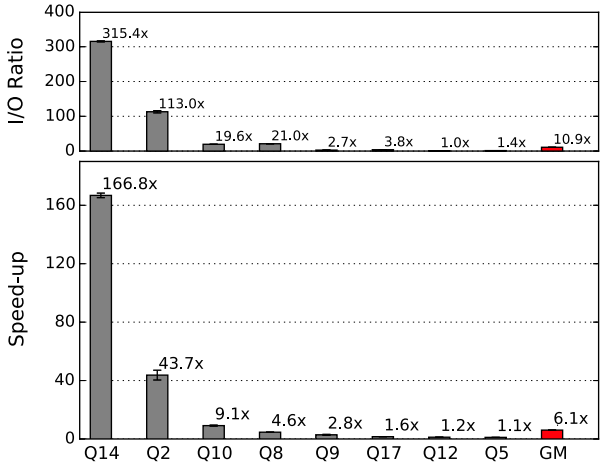

Figure 10. TPC-H results

no filter predicate at all in one case (Q18). There are also six queries for which our sampling heuristic advises MariaDB not to offload filtering because of low selectivity. Hence, relative performance of these 14 queries is simply 1.0.

For the eight queries that leverage NDP, we saw meaningful speed-ups, correlated with the I/O reduction ratios shown together. This ratio is calculated by dividing the number of pages read by Conv with that of Biscuit. The average (geometric mean) speed-up of the eight queries is as high as 6.1×. Five of those queries observe a significant improvement of 2.8× or more with the average speed-up of 15.4×. For five queries, we obtained surprisingly high speed-ups; for example, Q14 exhibited a speed-up of 166.8×. The reason for such a large performance gain is because the effect of early filtering is magnified in the case of complex join operations [36]. Unlike the original MariaDB policy of choosing the smallest table first in a nested join operation, our query planning heuristic (implanted in MariaDB) places a filter table that is the NDP target first in the join order. By doing so, our system greatly reduces the number of intermediate row sets that must be read from the SSD and examined by the DB engine (because many rows in the first table have been discarded already). There was a 315.4× reduction of I/O traffic in the case of Q14.

When running all 22 queries, Conv takes nearly two days, while Biscuit finishes in about 13 hours. In terms of total query execution time, Biscuit beats Conv by a factor of 3.6×. The top five queries were found to take more than 70% of the total query execution time.

VI. DISCUSSIONS

Based on our results and experiences, we make several observations, findings and remarks.

Costs of an NDP framework. We find both sound design choices and careful system optimizations play an extremely important role in realizing a successful NDP system. For us, the task of designing Biscuit required examination of

many different programming models and their implementation costs on an embedded platform, given the set of design priorities we had (Section II-B). When it came to system optimization, most metrics we present in Section V-B improved by 2× to over 50×, compared to an early feature-complete implementation. We also managed to keep the Biscuit runtime's footprint small; the binary size of Biscuit is only 313 KiB (single-core config.) or 553 KiB (dual-core config.).

Basic performance of the Biscuit runtime shows that there is no significant performance degradation due to the runtime. This is not surprising because it adds no complications to handling I/O and managing media, albeit running on top of heavily engineered SSD firmware. All I/O requests issued by Biscuit go through the same I/O paths with normal I/O requests, and the underlying SSD firmware takes care of media management tasks such as wear leveling and garbage collection. While there is room for further improvement (e.g., the I/O port communication latency), non-trivial improvements are seen in the read latency and bandwidth, compared to a conventional host-level SSD I/O. Such improvements are the result of tight integration of Biscuit with the existing firmware. Gains in the read operation match well with our expectation for an NDP architecture that both latency and bandwidth benefit from removing data transfer over a host interface.

Is NDP for all? It is logically clear that not all applications would benefit significantly from NDP. There are several conditions that must be met to make a given application a good fit for NDP. First, the application must have data processing tasks that deal with stored data; it makes little sense to migrate a task to an SSD that does not even touch SSD data. Second, the I/O and compute patterns of those tasks must possess certain characteristics; prior and our work identify that tasks having data dependent logic (i.e., I/O processing and light-weight compute alternate) and high-bandwidth I/O with simple scanning (i.e., instructions to process unit data are few) could benefit from NDP. If the application logic can be revised to take advantage of new execution profiles of offloaded tasks (e.g., new DB query planning heuristic), chances are that NDP benefits become even more tangible.

On our current implementation of Biscuit and the target SSD, improvement obtained for the pointer chasing application is nice but not spectacular. On the other hand, the simple string search and TPC-H applications saw a large speed-up (that cannot be matched by straightforward host software optimizations) by fully exploiting the high SSD-internal bandwidth and hardware pattern matcher in the target SSD. As such, we recognize and emphasize the importance of effective architectural support for successful NDP system design. Our results corroborate earlier findings [13, 22] that hardware support is instrumental in boosting the performance of data-intensive workloads. Such support will be even more critical as both external bandwidth and internal SSD bandwidth increase. Software optimizations on embedded processors can't simply keep up. Addressing limitations mentioned in Section IV-A is also highly desirable.

Importance of an NDP programming model. We feel that it is Biscuit's programmability and expressiveness that allowed us to explore versatile applications in this work, from pointer chasing to database scan and filtering with relatively straightforward design and coding. With full C++11 features and standard libraries, the amount of user code in SSDlets for an application barely exceeds 200 lines. Given the complexity of these applications and the level of improvements obtained, the coding effort is unprecedentedly low.

Moreover, none of the applications we wrote required any modification to the SSD firmware. With Biscuit in place, one only needs to worry about the target application itself and writing SSDlets properly in the host side and the device side. This aspect is of great advantage to developers as they do not deal with complex internal structures and algorithms inside SSD firmware (e.g., flash translation layer), while seeking improvement in a data-intensive application.

NDP and RAID. RAID (redundant array of independent disks) techniques are employed to combine and protect multiple SSDs on some host platforms (e.g., Scale-up of Fig. 1). Because RAID remaps logical block addresses among storage devices, it poses challenges and limitation to NDP. RAID 1 (i.e., replication) is less of a concern because data are available in all disks. However, RAID 10 and 5 are harder to deal with. In this case, for successful NDP programming, Biscuit, application, as well as SSDlets must all be aware of data layout. For applications where order in file contents is unimportant, one could have SSDlets filter data and the host task process all partial results. In general, however, RAID makes in-SSD NDP programming substantially more complicated.

Fortunately, in large storage and data processing platforms, software-driven inter-server replication and erasure coding become increasingly popular [7, 8, 38, 39]. Moreover, for emerging NVMe SSDs, no hardware based RAID solutions exist, and most multiple SSD deployments use a software-defined data layout. In such systems, file and metadata semantics are often preserved per disk. For example, a SkimpyStash cluster [40] may allocate dedicated metadata SSDs, and one can leverage Biscuit to accelerate metadata traversal in those SSDs. In another example, typical Hadoop [8] and Ceph [39] deployments don't use RAID. We find ample opportunities for in-SSD NDP in real-world deployments.

VII. Related Work

From Active Disk to Smart SSD, there is rich literature on storage-based NDP [9–15, 20, 41]. However, few studies discuss their framework in usability and real implementation

Table VII
COMPARISON OF PUBLISHED NDP FRAMEWORKS AND THIS WORK.

	Acharya [9]	Riedel [11]	Kang [20]	Do [12]	Tiwari [14]	Seshadri [15]	Biscuit
NDP framework							
programming model	stream-based	X	event-driven	session-based	X	RPC	flow-based
dynamic task loading	N/A	N/A	X	X	N/A	X	O
dynamic mem. alloc.	N/A	N/A	X	X	N/A	X	O
language support	N/A	N/A	C	C	N/A	C	C++11/14
Evaluation							
system impl.	simulator	prototype (emulation)	real SSD	real SSD	research SSD	prototype (FPGA)	real SSD
target storage	hard disk	hard disk arrays	SATA SSD (3 Gb)	SAS SSD (6 Gb)	OpenSSD [37] (Jasmine)	proprietary (PCIe2)	NVMe SSD (PCIe3)
applications	simple data-intensive workloads	simple data-intensive workloads	web-log analyzer, data filter	MS SQL Server (limited)	simple data-intensive workloads	atomic writes, caching, kvstore	MySQL (fully integrated)

perspectives. We show in Table VII a quick summary of their key properties and system realization. In the next, we touch on the most relevant prior work.

Among early active disk work [9–11], Acharya et al. [9] give the most comprehensive programming model description. They adopt a stream-based programming model that is similar to Biscuit's flow-based one. In their model, a user prepares disk-resident code ("disklets") and host peers. Disklets take in-disk streams (specified by file names) as input and generate streams for a host-resident task. "DiskOS" is the disk-side runtime that manages resources for disklets and ensures safety of execution. Acharya et al. establish several key concepts of a storage-based NDP system; unfortunately, they didn't build a real system for evaluation.

There are two independent pieces of recent work comparable to ours [12, 20]. First, Do et al. [12] build an NDP system using a real SSD. They revise a version of Microsoft SQL Server so that selection and aggregation operators can be pushed down to SSDs. However, their implementation is limited and the device-side query processing code is compiled into the SSD firmware. Kang et al. [20] extend the Hadoop MapReduce framework to turn SSDs into distributed data processing nodes. However, they do not provide a full-fledged runtime, primarily limited by the hardware and SSD firmware. Unlike these prototypes, Biscuit fully supports dynamic task loading and unloading and dynamic memory allocation, as well as C++11 constructs and standard libraries.

Lastly, the "Willow SSD" by Seshadri et al. [15] is a PCIe based FPGA emulation platform. Like Biscuit, Willow points to a system where user programmability (in a storage device) is a first-class design goal. The reported Willow prototype has a generic RPC interface, but little detail is disclosed about the APIs used to write SSD-resident code. By comparison, Biscuit is a product-strength NDP system on commercial SSDs. Biscuit allows the user to dynamically load user tasks to run on the SSD and focuses on how easily one can program an NDP system, whereas Willow authors explicitly (in the original paper) say it is not their design goal.

There are other system examples that bear resemblance to this work. Oracle Exadata [24] and IBM Netezza [42] are commercial systems that take the concept of NDP into practice. An Exadata system and Netezza offload certain data-intensive operations like scan to its storage server and special FPGA compute units, respectively. These systems do not take compute to inside SSDs and are complementary to our work. Ibex [35] employs an FPGA between a DB system (MySQL) and SSDs to offload filter and aggregation queries. Authors write a custom storage engine to accommodate their FPGA design, but do not propose or study a general framework where a user can write a new application.

VIII. CONCLUSIONS

This paper described the design and implementation of Biscuit, an NDP framework built for high-speed SSDs. With Biscuit, we pursued achieving high programmability on heterogeneous, distributed resources encompassing host CPUs and device CPUs, as well as realizing an efficient runtime on the device side. Our current target platform is a commodity server and high-performance NVMe SSDs. Biscuit is the first reported product-strength NDP system implementation for such a target. We successfully ported Biscuit on small and large data-intensive applications including MySQL. In principle, there is little reason why Biscuit can't be extended to support task offloading between networked servers in various system organizations (as outlined in Section II-A). In ensuing efforts, we are extending Biscuit to incorporate support for multiple user sessions and developing non-trivial data-intensive applications on Biscuit.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their constructive comments, which helped improve the quality of this paper. Many researchers and individuals have contributed at various stages to this work, including: Woojin Choi, Minwook Jung, Yang Seok Ki, Daniel DG Lee, Chanik Park, and Man-Keun Seo.

REFERENCES

[1] A. Halevy, P. Norvig, and F. Pereira, "The unreasonable effectiveness of data," *IEEE Intelligent Systems*, vol. 24, no. 2, pp. 8–12, 2009.

[2] R. E. Bryant, "Data-intensive supercomputing: The case for DISC," Tech. Rep. CMU-CS-07-128, Carnegie Mellon University, 2007.

[3] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Proceedings of the 7th International Conference on World Wide Web 7*, WWW7, pp. 107–117, 1998.

[4] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pp. 135–146, ACM, 2010.

[5] J. Han and K. Chang, "Data mining for web intelligence," *Computer*, vol. 35, no. 11, pp. 64–70, 2002.

[6] W. H. Inmon, *Building the Data Warehouse*. Wellesley, MA, USA: QED Information Sciences, Inc., 1992.

[7] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pp. 137–150, USENIX, 2004.

[8] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st ed., 2009.

[9] A. Acharya, M. Uysal, and J. Saltz, "Active disks: Programming model, algorithms and evaluation," in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pp. 81–91, ACM, 1998.

[10] K. Keeton, D. A. Patterson, and J. M. Hellerstein, "A case for intelligent disks (IDISKs)," *SIGMOD Rec.*, vol. 27, no. 3, pp. 42–52, 1998.

[11] E. Riedel, G. A. Gibson, and C. Faloutsos, "Active storage for large-scale data mining and multimedia," in *Proceedings of the 24th International Conference on Very Large Data Bases*, VLDB '98, pp. 62–73, Morgan Kaufmann, 1998.

[12] 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 *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pp. 1221–1230, ACM, 2013.

[13] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger, "Active disk meets flash: A case for intelligent SSDs," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pp. 91–102, ACM, 2013.

[14] D. Tiwari, S. Boboila, S. Vazhkudai, Y. Kim, X. Ma, P. Desnoyers, and Y. Solihin, "Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines," in *Presented as part of the 11th USENIX Conference on File and Storage Technologies*, FAST '13, pp. 119–132, USENIX, 2013.

[15] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: A user-programmable SSD," in *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pp. 67–80, USENIX, 2014.

[16] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind, "Bluedbm: An appliance for big data analytics," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 1–13, ACM, 2015.

[17] C. Li, Y. Hu, L. Liu, J. Gu, M. Song, X. Liang, J. Yuan, and T. Li, "Towards sustainable in-situ server systems in the big data era," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 14–26, ACM, 2015.

[18] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, 1997.

[19] 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," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, 2014.

[20] Y. Kang, Y. Kee, E. L. Miller, and C. Park, "Enabling cost-effective data processing with smart SSD," in *IEEE 29th Symposium on Mass Storage Systems and Technologies*, MSST '13, pp. 1–12, 2013.

[21] P. Morrison, *Flow-Based Programming: A New Approach to Application Development*. CreateSpace, 2nd ed., 2010.

[22] S. Kim, H. Oh, C. Park, S. Cho, and S. Lee, "Fast, energy efficient scan inside flash memory," in *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, ADMS '11, pp. 36–43, 2011.

[23] "Mariadb." https://mariadb.org/.

[24] M. Subramaniam, "A technical overview of the oracle exadata database machine and exadata storage server," *An Oracle White Paper*, pp. 1–43, 2013.

[25] "LDREX and STREX." http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0008a/ch01s02s01.html.

[26] "Cooperative threading." http://c2.com/cgi/wiki?CooperativeThreading, 2014.

[27] D. Lea, "A memory allocator." http://g.oswego.edu/dl/html/malloc.html, 2000.

[28] "dell-poweredge-r720-spec-sheet." http://www.dell.com/downloads/global/products/pedge/dell-poweredge-r720-spec-sheet.pdf.

[29] J. E. Hopcroft, J. D. Ullman, and A. V. Aho, *Data structures and algorithms*. Addison-Wesley, 1983.

[30] "Neo4j." http://neo4j.com.

[31] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?," in *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pp. 591–600, ACM, 2010.

[32] "STREAM: Sustainable memory bandwidth in high performance computers." http://www.cs.virginia.edu/stream/.

[33] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[34] "Tpc-h." http://www.tpc.org/tpch/.

[35] L. Woods, Z. Istvan, and G. Alonso, "Ibex - an intelligent storage engine with support for advanced SQL off-loading," *VLDB Endowment*, vol. 7, no. 11, pp. 963–974, 2014.

[36] "Block nested loop." https://en.wikipedia.org/wiki/Block_nested_loop.

[37] "The OpenSSD project." http://www.openssd-project.org.

[38] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows azure storage: A highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 143–157, ACM, 2011.

[39] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, (Berkeley, CA, USA), pp. 307–320, USENIX Association, 2006.

[40] B. Debnath, S. Sengupta, and J. Li, "SkimpyStash: RAM space skimpy key-value store on flash-based storage," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, (New York, NY, USA), pp. 25–36, ACM, 2011.

[41] D.-H. Bae, J.-H. Kim, S.-W. Kim, H. Oh, and C. Park, "Intelligent SSD: a turbo for big data mining," in *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management*, CIKM '13, pp. 1573–1576, ACM, 2013.

[42] P. Francisco, "IBM puredata system for analytics architecture: A platform for high performance data warehousing and analytics," *IBM Redbooks*, pp. 1–16, 2014.