Two Reconfigurable NDP Servers: Understanding the Impact of Near-Data Processing on Data Center Applications

XIAOJIA SONG and TAO XIE, San Diego State University
STEPHEN FISCHER, Samsung Semiconductor, Inc.

Existing near-data processing (NDP)-powered architectures have demonstrated their strength for some data-intensive applications. Data center servers, however, have to serve not only data-intensive but also compute-intensive applications. An in-depth understanding of the impact of NDP on various data center applications is still needed. For example, can a compute-intensive application also benefit from NDP? In addition, current NDP techniques focus on maximizing the data processing rate by always utilizing all computing resources at all times. Is this "always running in full gear" strategy consistently beneficial for an application? To answer these questions, we first propose two reconfigurable NDP-powered servers called RANS (**R**econfigurable **A**RM-based **N**DP **S**erver) and RFNS (**R**econfigurable **F**PGA-based **N**DP **S**erver). Next, we implement a single-engine prototype for each of them based on a conventional data center and then evaluate their effectiveness. Experimental results measured from the two prototypes are then extrapolated to estimate the properties of the two full-size reconfigurable NDP servers. Finally, several new findings are presented. For example, we find that while RANS can only benefit data-intensive applications, RFNS can offer benefits for both data-intensive and compute-intensive applications. Moreover, we find that for certain applications the reconfigurability of RANS/RFNS can deliver noticeable energy efficiency without any performance degradation.

CCS Concepts: • **Computer systems organization → Processors and memory architectures**; *Secondary storage organization*; **Reconfigurable computing**; **Heterogeneous (hybrid) systems;**

Additional Key Words and Phrases: Near data processing, FPGA, ARM, NDP server, reconfigurability, data center applications, data-intensive, compute-intensive

ACM Reference format:
Xiaojia Song, Tao Xie, and Stephen Fischer. 2021. Two Reconfigurable NDP Servers: Understanding the Impact of Near-Data Processing on Data Center Applications. *ACM Trans. Storage* 17, 4, Article 31 (October 2021), 27 pages.
https://doi.org/10.1145/3460201

1 INTRODUCTION

Contemporary data center servers are normally designed aiming at satisfying the needs of a wide range of applications, from data-intensive applications to compute-intensive applications [27]. However, this goal is becoming increasingly difficult to achieve because these servers still rely

In addition, this work is sponsored in part by National Science Foundation under grant CNS-1813485.
Authors' addresses: X. Song and T. Xie (Corresponding author), San Diego State University, Department of Computer Science, 5500 Campanile Drive, San Diego, CA, 92182, USA; emails: {xsong2, txie}@sdsu.edu; S. Fischer, Samsung Semiconductor, Inc., 3655 N 1st Street, San Jose, CA, 95134, USA; email: sg.fischer@samsung.com.

1553-3077/2021/10-ART31 $15.00
https://doi.org/10.1145/3460201

on the conventional compute-centric model, which always assumes that data stays in secondary storage and moves to host CPUs across the memory hierarchy when needed [5]. As a result, the performance of a data center application may be hindered by a *performance imbalance* between data transfer and data processing. For a data-intensive application, the data transfer bandwidth is usually lower than the data processing bandwidth provided by the server, which makes data transfer its performance bottleneck. A compute-intensive application is in an opposite situation as its performance bottleneck lies in its data processing side.

The performance imbalance of current data center servers causes two problems. First, it limits the performance of a data center application due to the existence of a performance bottleneck. Second, it leads to lower energy efficiency as either memory devices (e.g., DRAM-based main memory or flash SSDs) or host CPUs still consume energy when they are in an idle state waiting for data or instructions [11]. The energy consumed by these idle components in a data center server is a pure waste, which simply increases the energy budget of a data center without any benefit. Hence, it becomes too costly to increase the number of servers in a data center [7]. The advent of big data analytics makes these two problems even worse. On the one hand, big data analytics normally need to process large-size datasets to uncover useful information such as hidden patterns, unknown correlations, market trends, and customer preferences [9]. Thus, a massive amount of data needs to be transferred from secondary storage devices to host CPUs, which aggravates the problem of performance bottleneck at the data transfer side [9]. On the other hand, big data analytics usually employ various machine learning algorithms, which are compute intensive in nature [17]. Their high data processing complexity could shift the performance bottleneck to the data processing side.

A promising solution to the two problems is to design a data center server based on a data-centric model, which advocates processing data in situ so that the necessity of moving data across the memory hierarchy can be minimized [5]. Inspired by this model, a spectrum of **near-data processing (NDP)** techniques [2, 8, 10, 14, 19, 20, 24–26, 28, 40, 42, 47, 48] have been proposed recently. Although they target data at different levels of the memory hierarchy, they share a common methodology: deploying some hardware data processing accelerators (hereafter, NDP engines) such as FPGAs and embedded processors in or near memory devices to process data locally. In fact, NDP is a one-stone-two-birds approach. First, it alleviates the burden of host CPUs by offloading part of or all computations to NDP engines. Second, it largely reduces the pressure of data transfer as the size of processed data is normally much smaller than that of raw data. The major accomplishments of existing NDP techniques include significant improvements in performance and energy efficiency for an array of data-intensive applications [20, 24–26].

Existing NDP techniques mainly focus on data-intensive applications such as databases [20, 24, 26, 42], word count [10], linear regression [10], and scan [10]. This is understandable because after all, NDP is introduced to reduce data movement across the memory hierarchy so that applications can break the memory wall as closely as possible [6, 45]. To understand the impact of NDP on compute-intensive applications, our prior work [36] proposed and evaluated an NDP server architecture for data center applications. Results from [36] demonstrated that the proposed NDP architecture could benefit not only data-intensive applications but also compute-intensive applications in terms of the execution time, energy efficiency, and cost-effectiveness.

The results from the work of [36] are inspiring. However, its proposed NDP server architecture adopted an "always running in full gear" strategy, which maximizes the data processing rate by always utilizing all the computing resources (i.e., NDP engines) at all times. At first glance, this strategy looks reasonable because users invariably seek to execute an application "as fast as possible." Intuitively, it seems that all computing resources are needed all the time in order to achieve the highest execution speed. However, our prior work, **Reconfigurable In-Storage Processing**

(RISP) [37] demonstrated that for some applications the *reconfigurability* of an architecture could actually reduce computing resource utilization while still delivering the highest data processing throughput [37]. Here reconfigurability is defined as the capability of judiciously reconfiguring the resources of an NDP server to better serve an application based on its characteristics. The concept of reconfigurability is borrowed from RISP [37] and its effectiveness is evaluated at the server level in Section 6 to answer the second open question mentioned in the abstract. Unlike this research, the RISP framework [37] focuses on near-data processing within a single SSD (solid-state drive).

In this article, we propose two reconfigurable NDP-powered servers (hereafter, RNDP servers): **Reconfigurable ARM-based NDP Server (RANS)** and **Reconfigurable FPGA-based NDP Server (RFNS)**. In both RANS and RFNS, there are an array of SSDs, with each having a dedicated NDP engine. Compared with the NDP server architecture developed in our prior work [36], the main advantage of RANS and RFNS is that their hardware computing resources (i.e., the number of NDP engines) can be dynamically reconfigured based on the characteristics of an application. Next, we implement a single-engine prototype for each of the two servers based on a conventional data center server (hereafter, conventional server). While the **Single-engine ARM-based NDP Server (SANS)** utilizes an ARM embedded processor as its NDP engine, the **Single-engine FPGA-based NDP Server (SFNS)** employs FPGA as its NDP engine. In particular, SANS is extended from a conventional server [34] by inserting a Fidus Sidewinder-100 board [18] on the path between an SSD and the host CPUs (see Figure 5(a)). Note that SANS only utilizes the Fidus board's ARM Cortex-53 processor as its NDP engine. Similarly, SFNS is extended from the same server by adding a Xilinx VCU1525 FPGA board [46] into the same location (see Figure 5(b)). Further, we measure performance (i.e., data processing bandwidth in terms of MB/second) and energy efficiency (i.e., the amount of data that can be processed per joule in terms of MB/joule) for six data center applications from data intensive to compute intensive (i.e., "Linear Classifier (LC)," "Histogram Equalization (HE)," "k-NN_2," "k-NN_6," "k-NN_8," and "FFT") (see Section 4.4) running on the two single-engine server prototypes and the conventional server, respectively.

Experimental results measured from the two prototypes are then extrapolated to estimate the properties of the two full-size RNDP servers (i.e., RANS and RFNS). Several observations are made. For example, RANS can provide a decent performance and energy efficiency improvement for an application that is not compute intensive (e.g., Linear Classifier and Histogram Equalization). However, for a compute-intensive application (e.g., k-NN_2, k-NN_6, k-NN_8, and FFT) RANS only delivers limited benefits in energy efficiency and no benefit in terms of performance. On the other hand, RFNS offers a very good improvement in both performance and energy efficiency for both data-intensive and compute-intensive applications. In addition, RFNS is prone to provide more benefits in terms of performance and energy efficiency for applications with a higher data processing complexity. We discover that the "always running in full gear" strategy is not wise. For instance, an application with a low data processing complexity may only require part of the NDP engines in a server running at the same time to satisfy its computational needs. Running more NDP engines could only waste energy without any further performance gains. For some applications, the reconfigurability of RANS and RFNS can deliver a noticeable energy efficiency improvement (up to 47.99%) without any performance degradation. We also find that an application whose input size can be reduced by the NDP engines and whose data processing throughput on NDP engines is high is prone to achieve a high energy efficiency from both RANS and RFNS (see Section 6).

This research has the following contributions: (1) We propose two reconfigurable NDP servers to alleviate the performance imbalance between data transfer and data processing for both data-intensive and compute-intensive data center applications. The computing resources of the two proposed servers can be reconfigured so that they are adaptive to the characteristics of an application. (2) We build two single-engine NDP server prototypes by using FPGA and ARM

processor as an NDP engine, respectively. (3) We quantitatively analyze the impact of the two RNDP servers on a range of data center applications in terms of performance and energy efficiency.

The rest of the article is organized as follows. Related work is discussed in Section 2. The architecture of the two RNDP servers is presented in Section 3. Section 4 explains the evaluation methodology. Section 5 evaluates six data center applications running on the two servers. Section 6 evaluates the reconfigurability of the two RNDP servers. Section 8 concludes the article.

2 RELATED WORK

NDP is an umbrella term referring to a broad collection of techniques that move computation close to data. According to the positions of NDP engines in the memory hierarchy, existing NDP techniques can be generally divided into three groups: (1) **in-storage computing (ISC)**, (2) **in-memory computing (IMC)**, and (3) **near-storage computing (NSC)**.

Enterprise-grade flash SSDs now have substantial compute power and very high internal bandwidth [10]. To exploit these resources, a spectrum of ISC techniques including Active Flash [40], **intelligent SSD (iSSD)** [10], Smart SSD [16, 33], Biscuit [20], Caribou [24], BlueDBM [26], Summarizer [28], and YourSQL [25] have been proposed. Essentially, the only difference between a traditional SSD and an Active Flash [40] is that the latter has an enhanced version of **flash translation layer (FTL)** with some extra data analysis functions. Along the same line, iSSD [10] and Smart SSD [16, 33] allow execution of limited application functions (e.g., data filtering or list intersection) on an intelligent SSD [10]. On the other hand, Biscuit [20], Caribou [24], BlueDBM [26], Summarizer [28], and YourSQL [25] all focus on how to integrate ISC in a database system. All existing ISC techniques show that executing some functions in situ on an SSD can improve performance and energy efficiency for some data-intensive applications. Similar to the motivation of ISC, to exploit the high internal bandwidth of DRAM, multiple IMC techniques [2, 8, 19, 48] have been developed. Leveraging the 3D-stacking technology, they normally propose to integrate accelerator logic into custom 3D-DRAM devices to reap the performance and energy efficiency benefits of both accelerators and in-memory processing. Specifically, they focus on either an accelerator architecture where its memory system is separated from the host main memory system or the integration and interaction between the two memory systems using proprietary interfaces [8].

NSC techniques usually insert one **system-on-a-chip (SoC)** or FPGA on the path between a storage device (e.g., an SSD) and host CPUs to accelerate data processing so that the burden of host CPUs can be alleviated. Ibex [42] is developed as an FPGA-based SQL engine that accelerates relational database management systems by reducing both the energy consumption of the processor load operation and latency caused by memory accesses. IBM released Netezza [13], which integrates a considerable number of blade servers with a data warehouse appliance. Each server is equipped with one FPGA between main memory and storage to extract useful data without increasing the host CPU's load. Firebox is a next-generation computer architecture for data centers [3]. It consists of many fine-grained components of SoCs and memory modules connected with high-radix switches. A recent NSC strategy proposes to build a computer system with one FPGA-based accelerator called interconnected-FPGAs to accelerate join operations in a relational database [47]. Although various ISC and IMC techniques have shown their strength in the laboratory, so far only a few of them have become publicly available (e.g., Samsung SmartSSD [16]) based on our knowledge. NSC, however, is more practical as one can develop an NSC-based computer using some commodity products (e.g., a server, FPGA, and SoC). Therefore, in this research we employ NSC to study the impact of NDP on data center applications.

Table 1 qualitatively compares the architectural differences between existing NDP techniques and our two proposed RNDP servers. The first column shows the name of each NDP technique. The second column indicates whether it is ISC or NSC and the number of NDPEs per server if it

Table 1. Comparisons of NDP Techniques

Name	Type/NDPEs	Accelerator	Reconfigurability
iSSD [10]	ISC	General-purpose CPU	No
Active Flash [40]	ISC	ARM processor	No
Smart SSD [16, 33]	ISC	ARM processor	No
Caribou [24]	NSC/5	FPGA	No
BlueDBM [26]	ISC	FPGA	No
Biscuit [20]	ISC	ARM processor	No
Summarizer [28]	ISC	ARM processor	No
YourSQL [25]	ISC	ARM processor	No
Ibex [42]	NSC/1	FPGA	No
Netezza [13]	NSC/1	FPGA	No
Firebox [3]	NSC/unknown	System-on-a-chip (SoC)	No
NSC [47]	NSC/1	FPGA	No
RNDP servers [this work]	NSC/n	ARM processor and FPGA	Yes

is an NSC technique. The main architectural differences lie in two aspects. First, all existing NDP techniques used only one type of hardware accelerator as their NDP engine (e.g., an FPGA board, an embedded CPU, or a SoC). However, our work is the only one that evaluates two types of hardware accelerators (i.e., both FPGA and ARM processor) and then compares their efficacy. Besides, while existing NSC techniques only utilize one accelerator in a server [13, 47], RANS/RFNS proposes to employ one NDP engine for each SSD of an SSD array to fully exploit the parallelism among the SSDs (see Figure 1(b)). Second, our proposed NDP servers are reconfigurable in the sense that the number of NDPEs can be dynamically reconfigured to save energy without any performance loss for some applications. On the contrary, all existing NDP techniques do not have this capability.

3 RNDP SERVER ARCHITECTURE

In this section, we first briefly introduce the general architecture of a conventional server. Next, we present the architecture of our proposed two RNDP servers (i.e., RANS and RFNS).

3.1 The Architecture of a Conventional Server

The most cost-effective way to perform big data analytics is to employ machine learning on cloud computing infrastructure [43]. This approach requires a data center to be able to not only store a huge volume of data but also support a wide range of workloads including both data-intensive and compute-intensive applications.

Figure 1(a) illustrates the architecture of a conventional server with all flash storage. n is the number of SSDs in the server, k is the number of PCIe lanes that each SSD has, and m is the number of PCIe lanes between the PCIe switch and the host CPUs. Its main components include one or multiple multi-core CPUs supporting hyper-threading, DRAM, PCIe bus, PCIe switch, an array of SSDs, and network interface. The PCIe bus provides a high-bandwidth data path between the CPUs and SSDs. There are k PCIe lanes ($k \geqslant 1$) for each SSD. A PCIe switch is in charge of the data path between CPUs and SSDs. In particular, it controls multiple point-to-point serial connections, which fan out from the PCIe switch and then dispatch data directly to their destination devices. The host CPUs can concurrently access all the SSDs through the PCIe switch. A typical data center server [34] that we used in this article owns 36 SSDs with each having four PCIe lanes (i.e., $n = 36$,

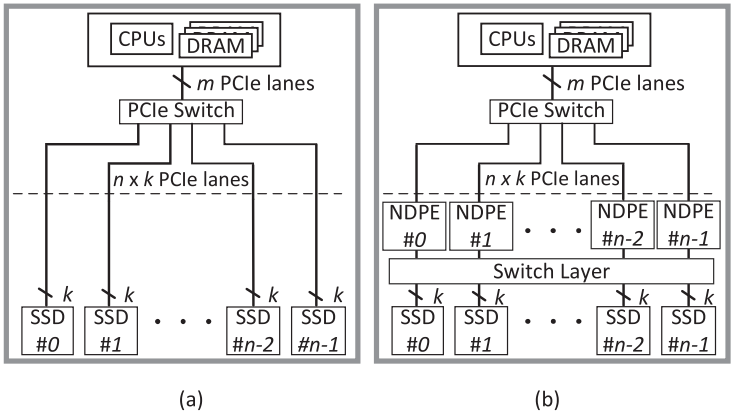

Fig. 1. (a) Architecture of a conventional server. (b) Architecture of RANS/RFNS.

$k = 4$) [34]. Its number of PCIe lanes between the PCIe switch and CPUs is 48 (i.e., $m = 48$) [34]. The two RNDP server prototypes are all extended from this server.

One observation from Figure 1(a) is that the data transfer bandwidth provided by an array of SSDs is underutilized because the number of PCIe lanes from SSDs to the PCIe switch (i.e., $n \times k$) is usually much larger than that number from the PCIe switch to CPUs (i.e., m). The underutilized bandwidth of the SSD array causes two problems. One is that for a data-intensive application, data transfer may become its performance bottleneck because the full bandwidth of the SSD array cannot be exposed to the CPUs. Another problem is that energy consumed by idle SSDs is simply wasted as only part of the SSDs is sending data through the PCIe switch at any given time while all the rest of the SSDs are in a standby state.

To solve these two problems, we propose to deploy an NDP engine layer between the PCIe switch and the SSD array so that the number of NDP engines (i.e., NDPE shown in Figure 1(b)) is equal to the number of SSDs. In theory, the assignment of NDP engines to SSDs is dynamic in the sense that an NDP engine can be assigned to any SSD due to the support of the switch layer (see Section 3.2). The proposed two RNDP servers can not only fully exploit the bandwidth of the entire SSD array to solve the data transfer bottleneck problem but also avoid energy waste caused by idle SSDs. Their architecture will be explained next.

3.2 The Architecture of the Two RNDP Servers

The main architectural difference between a conventional server and a proposed RNDP server is that the latter has two additional layers: a **near-data processing engine (NDPE)** layer and a switch layer (see Figure 1(b)). Although various computing units such as ARM processors, FPGAs, and streaming processors have been used as NDP engines, in this research we only consider the ARM processor and FPGA as they are two mainstream NDP engines.

Each NDP engine consists of four key components: a **processing element (PE)**, DRAM, an interface to host, and an interface to SSD. For a RANS, a PE is simply an ARM processor like an ARM Cortex-A53 [18] (see Figure 2(a)) as the price of an ARM processor is low [12]. For an RFNS, since an FPGA chip is much more expensive than an ARM processor, deploying one FPGA chip for each SSD is not practical. A realistic solution is to let multiple SSDs partition the resources (e.g., DSP slices, lookup tables, RAM, etc.) of one FPGA chip so that multiple application kernels can be instantiated from the same chip. As a result, each partition of an FPGA chip serves as an independent NDP engine, which is dedicated to a particular SSD. In such an NDP engine, the FPGA

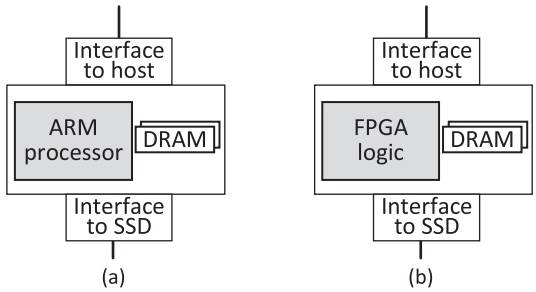

Fig. 2. (a) ARM-based NDP engine. (b) FPGA-based NDP engine.

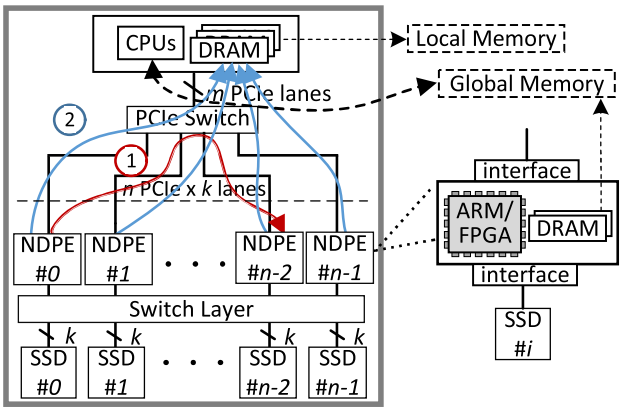

Fig. 3. Data transfer in an RNDP server; data path 1: data transfer from a source NDP engine to a destination NDP engine; data path 2: intermediary results transferred from each NDP engine to the host-DRAM.

logic (see Figure 2(b)) used by an application kernel serves as a PE. Note that all application kernels generated from one FPGA chip can concurrently process data from distinctive SSDs. The number of SSDs that can share one FPGA chip depends on the data processing complexity of an application and the amount of resources that an FPGA chip can offer. In fact, we find that for an application like k-NN, the FPGA chip on a Xilinx VCU1525 FPGA board [46] can simultaneously serve as 36 independent NDP engines for the 36 SSDs. The DRAM in an NDP engine is used to store metadata. Also, it works as a buffer for data movement among an SSD, an NDP engine, and the host CPUs. Intermediate results generated during the execution of an application are also stored in the DRAM. Note that the DRAM is accessible to host CPUs but cannot be accessed by other NDPEs. In this article we call this type of DRAM global memory. The DRAM on the host CPU side can only be accessed by the CPU, which works as local memory (see Figure 3).

An NDP engine cannot autonomously trigger a near-data processing procedure including reading data from an SSD and then processing it. Rather, a data processing procedure is always launched by host CPUs, which are in charge of the following tasks: (1) managing the operating system of the server, (2) monitoring the status of all NDP engines, (3) executing the host-side application, (4) offloading the application kernel to all NDP engines, and (5) writing the arguments to an application kernel in an NDP engine and then enabling it to read and process data from its corresponding SSD. The communications between host CPUs and NDP engines are carried out on a data path represented by the dashed double arrow shown in Figure 3.

In the NDP architecture proposed by [47], each server (called computing node in [47]) only has one NDP engine and NDP engines belonging to different servers are interconnected in order to reduce the communication cost caused by data exchange between different NDP engines. In our proposed RNDP server architecture, however, NDP engines are not directly connected to each other because doing so will make hardware connection routing very complicated considering that each RNDP server proposed in this research can have dozens of NDP engines. Instead, when an NDP engine has a need to transfer data to one of its peers, it leverages a PCIe **peer-to-peer (P2P)** communication strategy [29], which is a part of the PCIe specification. The PCIe P2P communication enables regular PCIe devices (i.e., NDP engines in our case) to establish direct data transfer without the need to use host memory as a temporary storage or use the host CPU for data movement. Thus, data transfer from a source NDP engine to a destination NDP engine can be accomplished through the PCIe switch in a **Direct Memory Access (DMA)** manner. PCIe P2P communication significantly reduces the communication latency and does not increase hardware design complexity. Data path 1 shown in Figure 3 illustrates this process. After all NDP engines finish their data processing, the results from each NDP engine will be aggregated at the host-DRAM for a further processing in CPU. Data path 2 shown in Figure 3 shows this case. Compared to the NDP architecture proposed in [47], our proposed RNDP server architecture lays a small burden on the host when an application needs to frequently exchange data between different NDP engines. The burden comes from the fact that the host needs to serve the scheduling and interrupt of a DMA data transfer procedure, although it performs other tasks while the transfer is in progress. However, our architecture has three advantages: (1) the design complexity is greatly reduced, (2) it is more compatible with a conventional server and (3) the fine-grained coupling of NDP engines with SSDs (i.e., each SSD has a dedicated NDP engine) leads to a high level of parallelism in data transfer and data processing.

For a specific application, its performance bottleneck on an RNDP server may exist in one of the following four locations shown in Figure 3 (from top to bottom): host CPUs, data path between host CPUs and NDPEs, NDPEs, or data path between NDPEs and SSDs. When a performance bottleneck of an application occurs on the data path between host CPUs and NDPEs (i.e., the aggregate data processing bandwidth of NDPEs is higher than the data transfer bandwidth between PCIe switch and host CPUs), the host CPUs are able to shut down part of NDPEs to reduce the aggregate data processing bandwidth so that it matches the data transfer bandwidth provided by the m PCIe lanes. In this case, the switch layer will connect multiple SSDs to a single active NDPE (i.e., an NDPE in a power-on status) so that data from any SSD can still be accessed and processed by NDPEs. By doing so, the rest of the NDPEs (i.e., inactive NDPEs) are powered off, and thus, energy will be saved.

Figure 4 illustrates how the switch layer works through an extreme NDPE-SSD connection scheme. The switch layer consists of an array of multiplexers and a group of point-to-point connections between the multiplexers and NDPEs. It can be adaptively configured to change the connections between NDPEs and SSDs. At one extreme, it may be configured so that each SSD will be connected to an exclusive NDPE as shown in Figure 1(b). We call this connection scheme *mux1* since each SSD can only connect to one NDPE. At the other extreme, the switch layer may be configured so that each SSD will be able to connect to all NDPEs. We call this connection scheme *muxn* or full connection as each SSD can be accessed by any NDPE. Although the full-connection scheme shown in Figure 4 offers the highest level of flexibility in connection between SSDs and NDPEs, it is not practical for it requires a very complex routing design, especially when the number of SSDs/NDPEs is large. The working mechanism of the switch layer under the *mux2* connection scheme (see Figure 8) and the benefits of reconfigurability will be analyzed in Section 6.

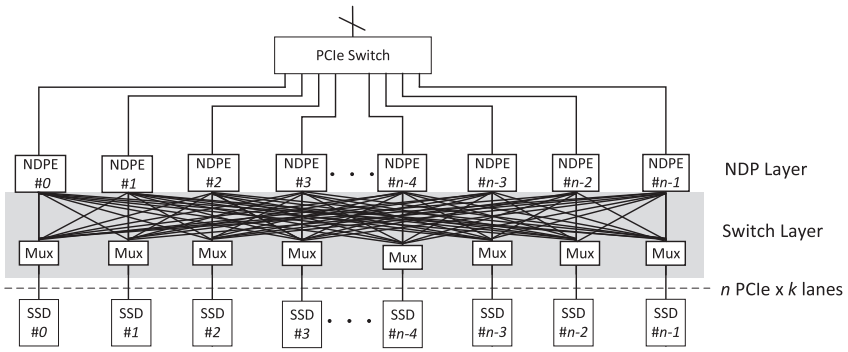

Fig. 4. A *muxn* connection scheme (i.e., full connection) in switch layer.

4 IMPLEMENTATIONS

In this section, we first explain our implementation methodology. Next, we describe how we implement the two RNDP server prototypes, SANS and SFNS, with each only having one NDP engine. The two single-engine RNDP server prototypes are all extended from a conventional server [34] (see Section 3.1) with two 18-core Intel Xeon CPUs and 36 PCIe 500GB SSDs [34]. Figure 1(a) shows its internal structure. For simplicity, hereafter the conventional server is referred to as CNS (**C**onve**N**tional **S**erver). Finally, we provide implementation details of the six data center applications running on the two prototypes and CNS.

4.1 Implementation Methodology

To develop two full-size RNDP servers (i.e., RANS and RFNS) based on our proposed architecture shown in Figure 1(a), 36 Fidus Sidewinder-100 boards [18] and 36 Xilinx VCU1525 FPGA boards [46] are required. In addition, a commodity server normally only provides a few PCIe slots on its motherboard, and most of them have been preoccupied by some indispensable boards like NIC cards. For instance, the CNS used in this article is equipped with four PCIe slots, but only one of them is available for a third party to insert an additional board on the data path between the switch layer and PCIe switch [34] (see Figure 5(b)). Therefore, it is impossible for us to plug 36 Fidus boards or 36 FCU1500 boards into the motherboard of the CNS. To do so, the motherboard needs to be completely redesigned. The very high hardware cost (e.g., 72 accelerator boards) and massive hardware revision (e.g., a complete redesign of a motherboard) are beyond our capacity. Fortunately, the major goal of this research is to understand the impact of NDP on data center applications. Therefore, we build a single-engine prototype for each of the two RNDP servers.

Six applications are executed on the two prototypes, and then their experimental results are extrapolated to estimate their performance and energy efficiency on the two full-size RNDP servers (i.e., RANS and RFNS), respectively. Since 36 SSDs and 36 NDPEs can work in parallel as shown in Figure 1(b), the performance of RANS/RFNS can be estimated as 36 × "performance of SANS/SFNS" (see Table 4 and Table 7). Predicting the behaviors (e.g., disk failure rates [4, 30]) or properties (e.g., performance modeling [44]) of an unavailable infrastructure (e.g., a large disk drive population [4, 30]) or software system (e.g., a parallel application at an arbitrary scale [44]) by extrapolating existing results obtained from a small-scale system is a common technique used by researchers. Similar to [4, 30, 44], our extrapolated results for the two full-size RNDP servers are not absolutely accurate as the overhead caused by a higher software complexity and a more complicated hardware management scheme (e.g., now host CPUs need to manage 36 NDPEs simultaneously) due to

Table 2. Platform Specifications

	Specifications
Server	Two CPU sockets; 36 SSDs m = 48; n = 36; k = 4 (see Figure 1(a))
CPU [23]	Xeon 6154 : 64bit, 3.0 GHz, 18 cores, 36 threads DDR4: 128 GB
PCIe	48 lanes attached to host CPUs (36 GB/s) 144 lanes attached to SSDs (108 GB/s)
SSD	PCIe×4; 3 GB/s
ARM platform	Quad-core Cortex-A53: 64-bit, 1.1 GHz [18]
FPGA platform	Xilinx VCU1525 platform [46]

increasing the number of NDPEs from 1 to 36 is ignored. Nevertheless, one can speculate a significant performance improvement when 36 NDPEs are working in parallel, and the overhead should be minor compared with the improvement. Thus, the extrapolated results are effective to estimate the performance of the two RNDP servers and predict their general trends, which are summarized in our seven new findings.

The execution of an application on an RNDP server can be divided into four stages: (1) SSD: data transfer from SSDs to NDP engines through a switch layer; (2) NDP: data processing in NDP engines; (3) NDP2CPU: data transfer from NDP engines to host-DRAM; and (4) CPU: data processing in host CPU. These four stages are organized in a pipelined fashion. The bandwidth of a stage (i.e., either data transfer bandwidth or data processing bandwidth) is denoted as $\#_{BW}$, where # can be replaced by SSD, NDP, NDP2CPU, or CPU. Thus, the performance of an application running on the server is determined by

$$\text{performance} = \min(SSD_{BW}, NDP_{BW}, NDP2CPU_{BW}{}^{*}\alpha, CPU_{BW}), \tag{1}$$

where α is the ratio of the size of an NDP engine's input data to the size of the NDP engine's output data. The values of α of the six applications used in this article can be found in Table 3. For example, if an application has an α value of 8, this indicates that for this application the size of the data outputted by an NDP engine is only one-eighth of that of its input data. This is the reason the $NDP2CPU_{BW}$ of this application needs to be multiplied by 8 as the effect of shrinking the size of data to be transferred by 8 times is equivalent to enlarging the data transfer bandwidth by 8 times without changing the size of the data. The implication of Equation (1) is that the performance of an application is equal to the bandwidth of the slowest stage of its execution. An example of using this equation can be found in Section 5.1. Based on our experiments on CNS, the read bandwidth of an SSD is approximately 3 GB/s and $NDP2CPU_{BW}$ is about 36 GB/s (see Table 2). When all 36 SSDs work concurrently, the SSD_{BW} is equal to 108 GB/s (i.e., 36 × 3 GB/s). The NDP_{BW} is equal to $NDPE_{BW}{\times}n$, where $NDPE_{BW}$ denotes the data processing bandwidth of one NDP engine and n is the total number of NDP engines. Obviously, the values of $NDPE_{BW}$ and CPU_{BW} depend on the characteristics of an application. These application-dependent values will be measured in Section 5. We will use Equation (1) to calculate the performance of the six applications.

4.2 Implementation of SANS

The PE of the NDP engine in a SANS is a quad-core Cortex-A53 ARM processor embedded in a Fidus Sidewinder-100 SoC board [18]. Table 2 summarizes the specifications of the conventional server that we used and the two accelerator boards. The Fidus board is a Zynq UltraScale+ MPSoC board that features a quad-core 64-bit ARM Cortex-A53 processor running at 1.1 GHz, some

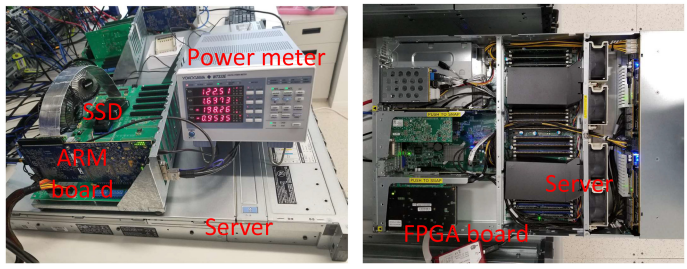

Fig. 5. (a) The SANS prototype. (b) The SFNS prototype.

programmable logic, and 16 GB DDR4 memory @1866MT/s [18]. Its PCIe Gen3 NVMe interfaces enable the ARM cores to directly read data from an attached SSD. Its PCIe × 8 host interface and 1 Gigabit Ethernet interface provide a data movement and communication channel to the host CPUs. The SANS prototype is shown in Figure 5(a).

In our experiments, an application executed on the ARM cores is first compiled by a cross-platform compiler *aarch64-linux-gnu-g++*. Next, the compiled executable file is offloaded as an executable file from host CPUs to the ARM cores in an NDP engine. Finally, a data processing procedure is launched by the ARM cores. After running the applications on SANS, we evaluate their performance and energy efficiency in Section 5.

4.3 Implementation of SFNS

In an SFNS, the PE of an NDP engine is built by FPGA logic (see Figure 2(b)). We use a Xilinx VCU1525 FPGA board (see Table 2) to implement an NDP engine in SFNS. This FPGA board is populated by a Kintex UlterScale FPGA, 16 GB DDR4 memory @2400MT/s, and PCIe Gen3 × 8 interface for the communication with the host [46]. An SDAccel development environment is used for the application design. It includes a system compiler, RTL-level synthesis, placement, routing, and bitstream generation [46]. The system compiler employs underlying tools for **high-level synthesis (HLS)**. The VCU1525 FPGA board is plugged into a PCIe Gen3 × 8 slot of the CNS server. Figure 5(b) shows the SFNS prototype.

In the SDAccel development environment, the **Open Computing Language (OpenCL)** standard is used for parallel programming. It provides a programming language and runtime APIs to support the development of applications on the OpenCL platform model, which includes the host CPUs and FPGAs. Details of SDAccel and OpenCL can be found in [46]. Note that the data flow of SFNS is different from that of the OpenCL framework, which is shown in Figure 6. The primary benefit of NDP comes from reducing data movement by directly reading/processing data from where they are stored (i.e., SSDs in the case of SFNS). Thus, an NDP engine in SFNS is expected to be able to fetch data from an SSD to its DRAM (step 1 in Figure 6(a)). And then the data are transferred to the FPGA to be processed, after which the results are sent back to the DRAM in the NDP engine (step 2 in Figure 6(a)). Finally, the results will be transferred to the host CPU (step 3 in Figure 6(a)). However, the proposed NDP engine in an SFNS is built in the OpenCL framework, which always starts data processing from host CPU. When there is a need to execute an application kernel on the FPGA board, the host CPU first reads data from an SSD to the host-DRAM (step 1 in Figure 6(b)). Next, the host CPU writes the data to the DRAM in an NDP engine (step 2 in Figure 6(b)). After the data have been processed in the NDP engine (step 3 in Figure 6(b)), they are eventually transferred to the host-DRAM (step 4 in Figure 6(b)).

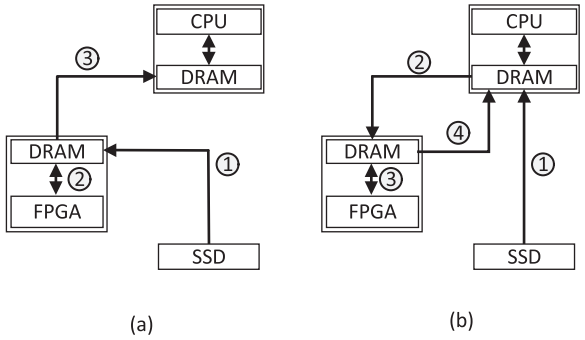

Fig. 6. (a) Data flow in SFNS. (b) Data flow in OpenCL framework.

Since it is very difficult, if not impossible, to change the data flow of the OpenCL framework to the way that an SFNS desires, we find a workaround to bypass this issue. In particular, we use steps 2 to 4 in Figure 6(b) to emulate steps 1 to 3 in Figure 6(a) in our experiments in order to estimate an application's wall time when it is running on an SFNS. The only difference between these two sets of steps lies in where to fetch the raw data. While SFNS is expected to achieve this by reading data from an SSD to the DRAM of an NDP engine (step 1 in Figure 6(a)), the OpenCL framework actually accomplishes this task by transferring raw data from host-DRAM to the DRAM of an NDP engine (step 2 in Figure 6(b)). However, the VCU1525 FPGA board can deliver a 10 GB/s data transfer bandwidth [46] in step 2 shown in Figure 6(b), which is much higher than the 3 GB/s data transfer bandwidth provided by an SSD (see Table 2) in step 1 shown in Figure 6(a). Therefore, a delay is injected to deliberately lower the data transfer bandwidth from 10 GB/s to 3 GB/s, by which we achieve our goal. Note that an OpenCL host runs on a host CPU when an application runs on an NDPE as an OpenCL kernel.

4.4 Implementation of Applications

In this article, six data center applications from data-intensive to compute-intensive are selected to study the impact of NDP on them. We first introduce them one by one, and then we briefly summarize their characteristics. Note that all these six applications are embarrassingly parallel.

4.4.1 Linear Classifier (LC). In the field of machine learning, a linear classifier achieves statistical classification by making a classification decision based on the value of a linear combination of the features [31]. If the input feature vector to the classifier is a real vector $\vec{x}$, then the output score is $y = f(\vec{w} \cdot \vec{x}) = f(\sum_j w_j x_j)$. $\vec{w}$ is the vector of the weights used in the classifier. In our use case j is set to 8, which makes LC a data-intensive application. A parallel implementation of this algorithm can be found from [31]. The size of the dataset used for this application is 37 GB. Since the classification for each data point is independent from the other points, the classifying of each point can be parallelized among 36 NDP engines.

4.4.2 Histogram Equalization (HE). Histogram equalization is a computer image processing technique used to improve contrast in images. It transforms pixel intensities so that the histogram of the resulting image is approximately uniform. This allows for areas of lower local contrast to gain a higher contrast [31]. A parallel implementation of this algorithm can be found from [31]. The size of the dataset used for this application is 3.4 GB. The execution of histogram equalization on different pictures can be done concurrently across 36 NDP engines.

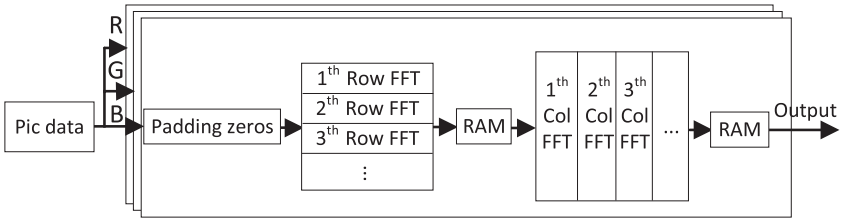

Fig. 7. 2D FFT on FPGA.

4.4.3 k-NN_2, k-NN_6, and k-NN_8. Given a set S of n reference data points in a space and a query point q, the k-NN algorithm [32] returns the k points in S that are closest to point q. The main steps of k-NN include (1) computing n squared Euclidean distances between the query point q $(x_1, x_2, \ldots, x_i)$ and the n reference points of the set S $(s_1, s_2, \ldots, s_i)$:

$$distance = (x_1 - s_1)^2 + (x_2 - s_2)^2 + \cdots + (x_i - s_i)^2,$$

and (2) sorting the n distances while preserving their original indices specified in S. The k nearest neighbors would be the k points from the set S corresponding to the k lowest distances of the sorted distance array. Since the distance calculation for each point in the database is independent, step 1 can be executed concurrently among all 36 NDP engines. In step 2, the calculated distances are aggregated and then sorted in order to discover the k nearest points of the query point. This step is carried out in the host CPUs after all results from step 1 are aggregated to the host-DRAM.

The algorithm of k-NN is straightforward, and its computational complexity depends on the number of features of each data point. We use k-NN_2 to represent a k-NN problem where each data point has two features. Similarly, a k-NN problem with each data point having eight features is denoted as k-NN_8 in this article. A larger number of features per data point indicates a higher computational complexity of a k-NN problem. The CPU and ARM codes start from a parallel implementation of the k-NN algorithm from the Rodinia library [32]. The size of the dataset used for this application is totally 130 GB [32].

4.4.4 FFT. FFT is an algorithm that samples a signal over a period of time (or space) and divides it into its frequency components. It is probably the most ubiquitous algorithm employed to analyze and manipulate digital or discrete data. It is also a well-recognized compute-intensive application [21]. The algorithm consists of two 1D FFTs, i.e., a row-wise FFT and a column-wise FFT. Note that for each picture its three color (i.e., R, G, B) values can be processed in parallel as shown in Figure 7. To obtain the best performance on CNS, we employ **Math Kernel Library (MKL)** [41] for the implementation of 2D FFT on the Xeon CPUs. A 2D FFT implementation on FPGA adopts a 1D FFT IP core from Xilinx, and it mainly consists of a 256×256 size row-wise 1D FFT module, buffer, and column-wise 1D FFT module (see Figure 7). This design is implemented at the RTL level. We run 2D FFT on 800 colorful pictures with total size of 238.54 MB (source: [15]).

4.4.5 Summary of the Six Applications. Among these six applications, **linear classifier (LC)** is the most data-intensive as its CPU_{BW} on CNS is 37.76 GB/s and its $NDP2CPU_{BW}$ is 36 GB/s (see Table 4). From **histogram equalization (HE)** to k-NN_2, k-NN_6, and k-NN_8, applications become more compute intensive. Finally, FFT is the most compute-intensive one. We run the six applications on CNS, SANS, and SFNS separately. The results of these applications on the two prototypes will be extrapolated to estimate the properties (i.e., performance and energy efficiency) of RANS and RFNS, which will be discussed in Section 5. The dataset of each of the six applications is speculated to be manually sharded into 36 equal-sized sub-datasets, which are then distributed

Table 3. Wall Times of the Six Applications

	Wall Time (s)					
	NDP/CPU					
App	LC	HE	FFT	k-NN_2	k-NN_6	k-NN_8
α	8	1	8	8	8	8
CNS	NA/0.98	NA/0.13	NA/2.05	NA/17.22	NA/31.33	NA/47.45
SANS	16.67/0	1.98/0	132.60/0	825.50/0.62	1,843/0.62	2,354/0.62
SFNS	13.45/0	1.14/0	3.02/0	36.31/0.62	34.39/0.62	32.83/0.62
*RANS	0.46/0	0.06/0	3.68/0	22.93/0.62	51.21/0.62	65.40/0.62
*RFNS	0.37/0	0.03/0	0.08/0	1.01/0.62	0.96/0.62	0.91/0.62

*The data in these rows are extrapolated.

manually across the 36 SSDs before an application starts to run. When part of NDP engines can be powered off to save energy for a particular application (see Section 6.2), we calculate offline to obtain the number of active NDP engines based on Equation (3). Assume that the number of active NDP engines is y. Next, the dataset of the application is supposed to be manually re-sharded across the y SSDs that are associated with the y active NDP engines if it has not been evenly distributed across all n SSDs. Finally, the application is re-launched on the server with n active NDP engines. In short, the proposed RNDP servers adopt a static manual-sharding strategy to distribute data across SSDs.

5 EVALUATION OF THE IMPACT OF NDP ON DATA CENTER APPLICATIONS

In this section, we empirically measure the performance and energy efficiency of the six applications running on CNS, SANS, and SFNS, respectively. The experimental results obtained from the two prototypes (i.e., SANS and SFNS) are then extrapolated to estimate the performance and energy efficiency of the two full-size RNDP servers (i.e., RANS and RFNS), respectively. RANS and RFNS are "always running in full gear," where 36 NDPEs and 36 SSDs are working in parallel. Note that the meaning of "always running in full gear" in this article is that all 36 NDP engines of an RNDP server are powered on (i.e., "active" as shown in Figure 8) during the execution of an application, even though they may switch between a busy status and an idle status once in a while.

The benefits of reconfigurability of RANS and RFNS will be analyzed in the next section. Energy efficiency is represented by the amount of data that can be processed per joule (i.e., MB/joule). We first present the results of the six applications in Section 5.1 and Section 5.2, which are followed by a detailed analysis of these results in Section 5.3.

5.1 Evaluation of FFT, LC, and HE

Performance evaluation: Performance of an application is determined by Equation (1), which has been explained in Section 4.1. Table 3 and Table 4 show the wall times and performance of FFT, LC, and HE on the five servers, respectively. "App" shown in Table 3 is a shorthand for "application," and "All" stands for "all three applications" (see Table 4). Since there is no NDP engine in CNS, "NA" (i.e., not applicable) is used for the three applications' "Wall time of NDP" and "BW of NDP" columns. Besides, since the three applications are entirely implemented and executed in NDP engines, there is no computing task for host CPUs. Thus, their values of "Wall time of CPU" and "BW of CPU" are "0" and "+∞," respectively.

The BW of either an NDP engine or host CPUs is equal to the size of the dataset divided by wall time. The wall time of RANS/RFNS is derived by the wall time of SANS/SFNS divided by 36, as 36 NDP engines can work in parallel assuming that the dataset has been evenly distributed among

Table 4. Performance of LC, HE, and FFT

	Bandwidth (GB/s)					Performance (GB/s)		
	NDP/CPU			SSD(s)	NDP2CPU			
App	LC	HE	FFT	All	All	LC	HE	FFT
CNS	NA/37.76	NA/26.15	NA/0.11	108	36	36	26.15	0.11
SANS	2.22/+∞	1.72/+∞	$1.80e^{-3}$/+∞	3	36	2.22	1.72	$1.80e^{-3}$
SFNS	2.75/+∞	2.98/+∞	0.08/+∞	3	36	2.75	2.98	0.08
*RANS	80.43/+∞	56.67/+∞	0.06/+∞	108	36	80.43	36	0.06
*RFNS	100.00/+∞	113.33/+∞	2.91/+∞	108	36	100.00	36	2.91

*The data in these rows are extrapolated.

Table 5. Energy Efficiency of LC, HE, and FFT

	NDP Engines (Watt)			Server Only (Watt)		Energy Consumption (Joule)			Energy Efficiency (MB/Joule)		
				Active	Idle						
App	LC	HE	FFT	All	All	LC	HE	FFT	LC	HE	FFT
CNS	0	0	0	613.70	30.33	601.43	79.78	1258.10	63.00	43.64	0.19
*RANS	207.36	207.36	207.36	613.70	30.33	109.34	14.26	874.7	346.52	244.15	0.27
*RFNS	421.20	743.40	652.68	613.70	30.33	167.07	23.21	54.64	226.78	150.00	4.37

*The data in these rows are extrapolated.

the 36 SSDs. "Performance" (see Table 4) of an application is derived from Equation (1). Take LC for example: its execution wall time on the NDP engine of SFNS is 13.45 seconds (see Table 3). Since the size of its dataset is 37 GB (see Section 4.4.1), its NDP_{BW} is 2.75 (GB/s) (i.e., 37/13.45, see Table 4). Meanwhile, its SSD_{BW}, $NDP2CPU_{BW}$, value of α, and CPU_{BW} are 3 GB/s (only one SSD is used in SFNS), 36 GB/s (see Table 2), 8 (see Table 3), and "+∞," respectively. Based on Equation (1), the performance of LC on SFNS is 2.75 GB/s as its bandwidth of the NDP stage (i.e., NDP_{BW}) is the lowest among the four pipelined stages (see Section 4.1). Since 36 NDPEs are supposed to work in parallel in an RFNS, the wall time of LC running on RFNS is estimated to 0.37 second (i.e., 13.45/36 = 0.37) (see Table 3). Unlike HE, whose size of the dataset is unchanged after the processing of tge NDP engine (i.e., α = 1), the size of the dataset of LC is reduced by 8 times (i.e., α = 8) after the processing of the NDP engine [39]. To saturate $NDP2CPU_{BW}$ (i.e., 36 GB/s), its NDP_{BW} should be at least 36 x 8 = 288 GB/s, which is much higher than 80.43 GB/s and 100 GB/s (i.e., NDP_{BW} of LC running on RANS and RFNS). That is why the performance of LC on RANS and RFNS is NDP_{BW} rather than $NDP2CPU_{BW}$.

Energy efficiency: Table 5 summarizes energy consumption and energy efficiency of the three applications running on the three servers. The "Server Only" column provides the power of the CNS server. Since there is no NDPE in CNS, a "0" shows up in the "NDP Engines (watt)" column for the three applications running on CNS. The power values of CNS in active status and idle status are 613.7 watts and 30.33 watts, respectively. These two values are measured by the power meter shown in Figure 5(a). Since the wall time of LC running on CNS is 0.98 second (see Table 3), its energy consumption on CNS is 601.43 joules (i.e., 613.7 watts × 0.98 second = 601.43 joules). The size of the dataset of LC is 37 GB (see Section 4.4.1). Thus, its energy efficiency is 63 MB/joule (i.e., 37 GB ÷ 601.43 joules = 63 MB/joule). We also use the power meter to measure the power consumptions of the single NDP engine in both SANS and SFNS when the six applications are running on them. The power of the NDP engine of SANS is 5.76 watts for all six applications as they use the same ARM processor (i.e., Quad-core Cortex-A53 [18]). This power value is then multiplied by 36 to represent the power of the 36 NDPEs in RANS, which is 207.36 watts (i.e., 5.76 watts × 36 = 207.36 watts) (see Table 5 and Table 8). The total energy consumption of an application running

Table 6. FPGA Utilization of 36 NDPEs in RFNS

Application	LUT (6-Input)	REG	BRAM (36 Kb)	DSP Slices
FPGA resources [46]	1,182,240	2,364,480	2,160	6,840
LC	108,108 (9.14%)	125,568 (5.31%)	72 (3.33%)	180 (2.63%)
HE	1,673,352 #(140.54%)	1,961,604 (82.96%)	*6300 0	0 0
k-NN_2	129,816 (10.98%)	172,512 (7.30%)	72 (3.33%)	432 (6.32%)
k-NN_6	227,016 (19.20%)	364,140 (15.40%)	288 (13.33%)	1,440 (21.10%)
k-NN_8	272,916 (23.08%)	444,960 (18.81%)	288 (13.33%)	1,944 (28.42%)
FFT	481,932 (40.81%)	643,608 (27.21%)	180 (8.28%)	3,456 (50.50%)

*Off-chip DRAM used for the overfilled BRAM.
#Larger than 100% means more than one FPGA chip needed.

on a server is the sum of its energy consumption of NDPEs, energy consumption of host CPU in active status, and energy consumption of host CPU in idle status. Therefore, the total energy consumption of LC running on RANS is (207.36 + 30.33) watts × 0.46 second (see Table 3) = 109.34 joules. Thus, its energy efficiency is (37 GB × 1024) ÷ 109.34 joules = 346.52 MB/joule.

Unlike SANS, the power consumptions of the NDP engine of SFNS for the six applications are distinct (i.e., 11.7 watts for LC, 20.65 watts for HE, 18.13 watts for FFT, 11.68 watts for k-NN_2, 11.94 watts for k-NN_6, 12 watts for k-NN_8). The reason is that each of these applications utilizes a unique collection of resources from the same FPGA platform [46]. These resources include on-chip memory, **digital signal processing (DSP)** slices, registers (i.e., REGs), programmable logic such as **flip-flops (FFs)**, and **lookup tables (LUTs)**. On-chip memory consists of distributed RAM, **block RAM (BRAM)**, and UltraRAM. Together, they provide a customized high-speed storage to buffer data between tasks. LUT is a table that determines what the output is for any given inputs. BRAM stores data inside FPGA, and DSP slices implement signal processing functions. Table 6 presents the number of various resources provided by the FPGA chip on a Xilinx VCU1525 board [46] (see the first row) and the FPGA utilizations of the six applications when they are running on RFNS (see the rest of the rows). In particular, the capacity of each BRAM is 36 Kb and the FPGA chip has totally 2,160 BRAM [46]. Also, each LUT has six inputs. The amount of power consumed by an application running on the NDPEs of RFNS is correlated with its FPGA utilization. This is the reason HE consumes the largest amount of power in the NDP engines of RFNS (see Table 5 and Table 8) as its overall FPGA utilization is the highest among the six applications (see Table 6).

5.2 Evaluation of the Three K-NN Applications

Performance evaluation: Table 3 and Table 7 show the wall times and performance of k-NN_2, k-NN_6, and k-NN_8 on the five servers, respectively. Unlike FFT, LC, and HE, an execution of a k-NN application on an RNDP server (i.e., RANS or RFNS) involves both host CPUs and NDPEs. There are two steps in an execution of a k-NN application. While the first step (i.e., computing n squared Euclidean distances between the query point q and the n reference points of the set S) is performed in NDPEs, the second step (i.e., sorting the n distances while preserving their original indices specified in S) is carried out in host CPUs. Note that in the first step distance calculations

Table 7. Performance of k-NN_2, k-NN_6, and k-NN_8

	Bandwidth (GB/s)					Performance (GB/s)		
	NDP/CPU			SSD(s)	NDP2CPU			
k-NN_	2	6	8	All	All	2	6	8
CNS	NA/7.55	NA/4.15	NA/2.74	108	36	7.55	4.15	2.74
SANS	0.16/209.68	0.07/209.68	0.06/209.68	3	36	0.16	0.07	0.06
SFNS	3.58/209.68	3.78/209.68	3.96/209.68	3	36	3	3	3
*RANS	5.67/209.68	2.54/209.68	1.99/209.68	108	36	5.67	2.54	1.99
*RFNS	128.71/209.68	136.42/209.68	142.86/209.68	108	36	108	108	108

*The data in these rows are extrapolated.

Table 8. Energy Efficiency of k-NN_2, k-NN_6, and k-NN_8

	NDP Engines (Watt)			Server Only (Watt)		Energy Consumption (Joule)			Energy Efficiency (MB/Joule)		
				active	idle						
KNN_	2	6	8	All	All	2	6	8	2	6	8
CNS	0	0	0	613.70	30.33	10,568	19,227	29,120	12.60	6.92	4.57
*RANS	207.36	207.36	207.36	613.70	30.33	5,830.72	12,541	15,916	22.83	10.61	8.36
*RFNS	420.48	429.84	432	613.70	30.33	817.06	803.46	782.29	162.93	165.68	170.17

*The data in these rows are extrapolated.

can be performed in parallel as there is no data dependency among the distances. The values in the "NDP" column show the performance of step 1 in NDP engines, and the values in the "CPU" column demonstrate the performance of step 2 in host CPUs (see Table 7). The only difference among the three k-NN applications is the number of features per data point used for the distance calculation in step1.

Results in Table 7 demonstrate that when running on an RANS, the performance of k-NN_2 is 2.23× and 2.85× of that of k-NN_6 and k-NN_8, respectively. This is mainly because each data point of k-NN_2 only has two features, and thus, the time complexity of its first step is much lower than that of the other two k-NN applications. However, when each of them is running on an RFNS, they end up with the same performance because the SSD bandwidth now becomes the performance bottleneck for all three k-NN applications.

Energy efficiency: Table 8 summarizes the energy efficiency of the three k-NN applications. Take k-NN_2 running on RANS, for example. Its wall times on NDP engines and host CPU are 22.93 seconds and 0.62 second (see Table 3), respectively. Thus, its total energy consumption is equal to (207.36 watts + 30.33 watts) × 22.93 seconds + 613.70 watts × 0.62 second = 5830.72 joules. Since the size of the dataset of k-NN_2 is 130 GB (see Section 4.4.3), its energy efficiency is therefore equal to 22.83 MB/joule (i.e., 130 GB ÷ 5830.72 joules = 22.83 MB/joule).

Results in Table 8 demonstrate that when running on an RANS, the energy efficiency of k-NN_2 is 2.15× and 2.73× of that of k-NN_6 and k-NN_8, respectively. The reason is that each data point of k-NN_2 only has two features, and thus, its first step can be executed in NDPEs much faster than the other two k-NN applications. Therefore, it consumes much less energy. However, when the three k-NN applications are running on an RFNS, they deliver a very similar level of energy efficiency because they need to run a similar amount of time. Again, this is due to the fact that the SSD bandwidth now becomes the performance bottleneck for all of them.

5.3 Impact of NDP on Data Center Applications

A quantitative comparison between CNS and the two RNDP servers (i.e., RANS and RFNS) is also given in Table 9. Based on Table 9, several new findings on the impact of NDP on data center applications can be obtained.

Table 9. Improvements of RANS and RFNS

Comparison with CNS		LC	HE	k-NN_2	k-NN_6	k-NN_8	FFT
*RANS	Performance	2.23x	1.38x	0.75x	0.61x	0.73x	0.55x
	Energy efficiency	5.49x	5.59x	1.81x	1.53x	1.83x	1.42x
*RFNS	Performance	2.78x	1.38x	14.30x	26.02x	39.42x	26.45x
	Energy efficiency	3.6x	3.44x	12.93x	23.94x	37.23x	23.00x

*The data in these rows is extrapolated.

Table 9 shows that in terms of performance, RANS outperforms CNS by 2.23× and 1.38× for LC and HE, respectively. **Finding 1:** *For data-intensive but compute-light applications, RANS can provide performance benefits by offloading computation from host CPUs to NDP engines that are close to data.* The performance benefits stem from RANS's capability of exploiting the full bandwidth of the SSD array, and thus avoiding the data transfer bottleneck on the data path between PCIe switch and host-DRAM (see Figure 1(a)). Table 9 also shows that RANS is inferior to CNS in terms of performance for k-NN_2 (0.75×), k-NN_6 (0.61×), k-NN_8 (0.73×), and FFT (0.55×), which all have a data processing complexity higher than that of LC and HE. **Finding 2:** *RANS cannot benefit compute-intensive applications in terms of performance.* Although offloading computation to near-data processing engines enables RANS to enjoy the high data throughput bandwidth of the SSD array, for compute-intensive applications these benefits cannot offset the significant discrepancy in computational capacity between a 1.1 GHz embedded processor and a 3.0 GHz Xeon CPU.

Can a compute-intensive application also reap the benefits of NDP? The answer is yes, which is confirmed by the Performance row of RFNS in Table 9. **Finding 3:** *RFNS can offer performance benefits not only for data-intensive applications but also for compute-intensive applications.* The FPGA's hardware-level acceleration capability in RFNS remedies the weakness of an embedded processor in RANS. This advantage of RFNS and the benefits brought by NDP (i.e., fully exploiting the high data throughput of the SSD array, and thus reducing data movement) together explain this finding. The performance benefits cannot be gained by simply putting data processing engines at the host CPU side because data transfer from PCIe switch to host CPUs could become a system performance bottleneck if doing so. As we can see from Table 7, for k-NN_8 running on RFNS, its NDP bandwidth (i.e., 142.86 GB/s) is much higher than its NDP2CPU bandwidth (i.e., 36 GB/s).

From Table 9, we obtain the following findings. **Finding 4:** *RFNS offers more benefits in terms of performance and energy efficiency for applications with a higher data processing complexity, which is contrary to RANS.* **Finding 5:** *Compared with CNS, both RANS and RFNS can deliver a higher energy efficiency for all six applications.* The reason is that host CPUs are not energy efficient in nature. Offloading all or partial computational load to an NDP engine not only relieves the computational burden of the host CPU but also reduces the data movement, which can better improve the energy efficiency of the entire system.

6 EVALUATION OF THE RECONFIGURABILITY OF TWO RNDP SERVERS

In this section, we first present the motivation of reconfigurability. Next, we explain the working mechanism of the switch layer under the *mux2* connection scheme through which the reconfigurability of the two RNDP servers can be achieved. Finally, we quantitatively analyze the energy savings due to the reconfigurability of the two servers.

6.1 Motivation of Reconfigurability

Although the two RNDP servers (i.e., RANS and RFNS) are proposed to be a reconfigurable NDP server, they have not been reconfigured in any sense so far. All experimental results from the two

servers so far are obtained when they are "always running in full gear" (i.e., 36 NDP engines are working concurrently all the time). After inspecting the performance results of the six applications shown in Table 4 and Table 7, we observe, however, that during the execution of each application there always exist some noticeable bandwidth dependencies among the four pipelined stages. The implication is that for a specific application, its performance bottleneck could occur in any of the four stages. Take HE for example: its SSD_{BW}, NDP_{BW}, $NDP2CPU_{BW}$, and CPU_{BW} are 108 GB/s (i.e., 3 GB/s × 36 = 108 GB/s), 113.33 GB/s, 36 GB/s, and $+\infty$ (see Table 4), respectively. Clearly, data transfer between NDPEs to the host CPU becomes its performance bottleneck as its performance is limited to 36 GB/s even though NDPEs can process its data at a much higher rate (i.e., 113.33 GB/s). The performance bottleneck not only degrades the performance of HE but also leaves devices like SSDs, NDPEs, and host CPUs in an idle status when they wait for data transfer between NDPEs and the host CPU to finish. When these devices work in an idle status, they merely waste energy without any performance gains.

This observation inspires us to propose a reconfigurable NDP server where its resources can be reconfigured to reduce potential energy waste for some applications (e.g., HE) without degrading their performance. A reconfigurable NDP server is an NDP server with reconfigurability, which is defined as the capability of judiciously reconfiguring the resources of an NDP server to better serve an application based on its characteristics (e.g., computational complexity). In theory, each member of all major types of devices (e.g., SSDs, NDPEs, and host CPUs) in an RNDP server (see Figure 1(b)) could be powered on/off in order to reconfigure the resources of the server to meet the computational requirements of an application while reducing unnecessary energy consumption. However, we find out that current architecture of a data center server like CNS does not allow us to power on/off either an SSD or a host CPU. We discover that it is practical to power on/off an individual NDPE in an RNDP server. In this research, the reconfigurability of an RNDP server concentrates on reducing energy waste by powering off partial NDPEs without affecting the performance of an application. Through our prior performance bottleneck analysis of HE, we understand that part of NDPEs should be powered off to save energy when the performance bottleneck of an application occurs either on the data path between NDPEs and host CPUs or on the data path between SSDs and NDPEs. Now, the question becomes how many of the 36 NDPEs should be powered off once such a performance bottleneck occurs. In other words, we need to know how many active NDPEs are needed for a particular application. The number of active **NDP engines (NANEs)** can be calculated by the following equation:

$$nane = \text{ceil}(\min(SSD_{BW}, NDP_{BW}, NDP2CPU_{BW}{}^{*}\alpha, CPU_{BW})/SNDP_{BW}), \tag{2}$$

where α is the ratio of the size of an NDP engine's input data to the size of the NDP engine's output data (see Table 3), and $SNDP_{BW}$ is the data processing bandwidth of a single NDPE. All other variables have been explained in Equation (1), and ceil(x) is the ceiling function.

6.2 Working Mechanism of the Switch Layer

Considering that the dataset of an application is stored across all 36 SSDs, we need to ensure that after powering off, part of the NDPE's data from any of the 36 SSDs can still be accessed and processed by at least one active NDPE. One approach to achieving this goal is to configure the switch layer so that all SSDs are still reachable by at least one active NDPE. In Section 3.2, we explained how the switch layer works through a connection scheme called *muxn* (see Figure 4). We understand that *mux1* and *muxn* are two extreme cases. While the former eliminates the possibility of reconfigurability of an RNDP server, the latter leads to an extremely complex routing design in the switch layer. Thus, these two schemes are no longer considered. Instead, we find that a *mux2* connection scheme is able to provide us an opportunity to demonstrate the benefits of

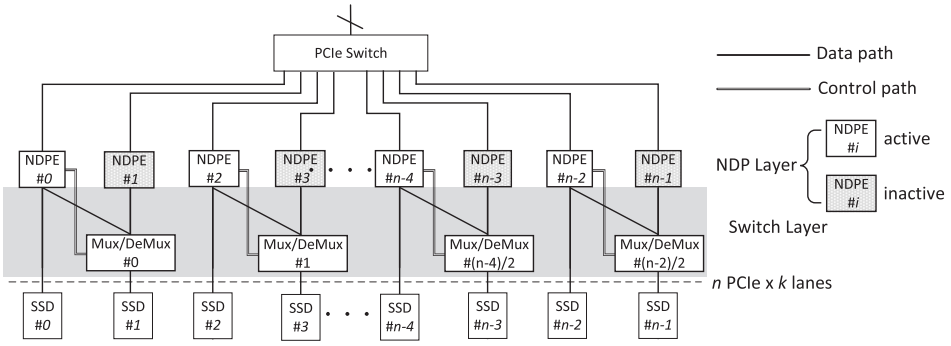

Fig. 8. A *mux2* connection scheme in switch layer.

reconfigurability while leading to a simple routing design. Thus, in this section the *mux2* scheme is adopted. Although this scheme might not be the best player that can offer the most energy savings for an application, its simple routing design makes it easy to present a reconfigurable NDP server.

Figure 8 illustrates the *mux2* scheme where an active NDPE is able to access two neighboring SSDs. The switch layer shown in this figure could be implemented by using an array of 2:1 Mux/DeMux switches. Each switch is a PI3DBS16212 module [22], which is a high-performance, passive signal multiplexer/demultiplexer that switches multiple serial protocols including USB3.1, SAS3.0, and PCIe4. The connection protocol used in this research is PCIe4. The nominal bandwidth of each PI3DBS16212 switch is 20 Gbps [22]. More detailed information of this product can be found at [22]. Now, we explain how data sharing is achieved by using the switch layer when parts of the NDPEs are powered off. In the *mux2* connection scheme, each active NDPE is only responsible for accessing two neighboring SSDs. Take *NDPE#0* for example: it only needs to manage data from/to both *SSD#0* and *SSD#1* when half of NDPEs have been powered off to save energy. When raw data is reading from *SSD#1* while *NDPE#1* has been powered off (see Figure 8), *Mux/DeMux#0* serves as a 1:2 DeMux. In this scenario, *NDPE#0* issues a selecting signal (i.e., logical 0) to *Mux/DeMux#0* along the control path between them (see Figure 8), which enables the raw data fetched from *SSD#1* to flow to *NDPE#0* along the data path. When some final results need to be written back from *NDPE#0* to *SSD#1*, the *Mux/DeMux#0* switch now serves as a 2:1 Mux. At this moment, *NDPE#0* again issues a selecting signal (i.e., logical 0) to the switch so that the final results can be sent back from *NDPE#0* to *SSD#1*.

Compared with an NDP server architecture without the switch layer, the extra work caused by the switch layer for an active NDPE includes (1) issuing a 1-bit binary selecting signal along the control path to a Mux/DeMux switch when accessing a neighboring SSD and (2) inserting an SSD ID number for each data package that is fetched from the two neighboring SSDs (e.g., *SSD#0* and *SSD#1* for *NDPE#0*) to distinguish the data sources. Since both actions are simple operations, they consume very little resources in terms of computation (e.g., cycles of an ARM processor or FPGA in an NDPE) and storage (e.g., the DRAM within an NDPE shown in Figure 2) within an active NDPE. Thus, the resource consumption on the NDP engines caused by the use of the switch layer is minimal when a *mux2* connection scheme is employed. One obvious limitation in data sharing is that when its dedicated NDPE is powered off (e.g., *NDPE#1*), the data from an SSD (e.g., *SSD#1*) can only be shared by the NDPE that belongs to its left-neighbor SSD (e.g., *SSD#0*). If *Mux/DeMux#0* fails, the data from *SSD#1* can no longer be shared by any other NDPEs. In the *mux2* connection scheme, the two neighboring SSDs and the two neighboring NDPEs as well as one Mux/DeMux switch can be logically combined into one NDP module. For example, *SSD#0*,

Table 10. An Example of Activating NDPEs When $n = 8$

nane_mux2	IDs of Activated NDPEs
4	0 2 4 6
5	0 1 2 4 6
6	0 1 2 3 4 6
7	0 1 2 3 4 5 6
8	0 1 2 3 4 5 6 7

SSD#1, *NDPE#0*, *NDPE#1*, and *Mux/DeMux#0* can be logically combined into one NDP module (see Figure 8). Theoretically, the switch layer shown in Figure 8 can be readily scaled up by just adding more NDP modules in an NDP server. However, in reality, the scalability of the switch layer is restricted by the capacity of an NDP server (see Figure 3) that has a limited physical space, a limited power budget, and limited routing design options. After one of these limitations is reached, adding more NDP modules becomes infeasible. In this research, our NDP server can support up to 18 such NDP modules (i.e., 36 SSDs, 36 NDPEs, and 18 Mux/DeMux switches [22]).

From Figure 8, we can see that the maximal number of NDPEs that can be powered off is 18 (i.e., $n/2$, where n is 36 in our case). Thus, Equation (2) can be revised to the following one in order to calculate the number of active NDPEs needed under the *mux2* connection scheme:

$$nane\_mux2 = \max(\text{ceil}(\min(SSD_{BW}, NDP_{BW}, NDP2CPU_{BW}{}^{*}\alpha, CPU_{BW})/SNDP_{BW}), 18). \quad (3)$$

Based on Equation (3), under the *mux2* scheme an application needs at least half of n NDPEs to be activated ($n = 36$ in our case). Under the *mux2* scheme, each SSD can be connected to one of the following two NDPEs: either its corresponding NDPE (e.g., *SSD#0* connecting to *NDPE#0* shown in Figure 8) or its left-neighbor SSD's corresponding NDPE (e.g., *SSD#1* connecting to *NDPE#0* shown in Figure 8). Active NDPEs are selected as follows: (1) all NDPEs with an even ID number are set to active; (2) if more active NDPEs are needed for an application, NDPEs with an odd ID number will be activated in an ascending order till the number of total active NDPEs reaches *nane_mux2*; (3) if an NDPE is active, its associated SSD connects to it; otherwise, its associated SSD connects to the NDPE belonging to its left neighbor. Table 10 shows an example of how active NDPEs are selected when the total number of NDPEs is equal to 8. In summary, an RNDP server needs to be reconfigured before an application starts to run on it. In other words, each time a new application starts to run, the server needs to be reconfigured in order to fit its needs. Before an application starts to run, the number of active NDPEs that it needs can be calculated by Equation (3). This is because CPU_{BW} (i.e., the data processing bandwidth of the host CPU) and $SNDP_{BW}$ (i.e., the data processing bandwidth of a single NDPE) can be obtained by profiling a benchmark on the server before the application starts to run. As for SSD_{BW} (i.e., data transfer bandwidth of SSDs, which is 3 GB/s per SSD) and $NDP2CPU_{BW}$ (i.e., data transfer bandwidth from NDP engines to host-DRAM, which is 36 GB/s), they are known performance parameters of an RNDP server as shown in Table 2. Next, part of NDPEs are selected to be activated, and then, each SSD is connected to a particular active NDPE under the *mux2* scheme.

6.3 Energy Savings Due to Reconfigurability

Now, we will use the HE application as an example to demonstrate how to obtain the minimal number of active NDPEs needed for a specific application running on RFNS. Next, we will show how much energy benefits can be reaped by exploiting the reconfigurability of RANS/RFNS. For HE, its values of SSD_{BW}, NDP_{BW}, $NDP2CPU_{BW}$, CPU_{BW}, and $SNDP_{BW}$ are 108 GB/s, 113.33 GB/s, 36 GB/s, $+\infty$, and 2.98 GB/s, respectively (see Table 4). Its α is 1 (see Table 3). Based on Equation (2),

Table 11. Energy Efficiency Gains from Reconfigurability of RANS and RFNS

		LC	HE	k-NN_2	k-NN_6	k-NN_8	FFT
*RANS	*36 - nane_mux2*	0	15	0	0	0	0
	Saved energy (joule)	−0.017	5.18	−0.41	−0.92	−1.18	−0.066
	Improvement (%)	−0.016	36.33	−0.007	−0.007	−0.007	−0.0075
*RFNS	*36 - nane_mux2*	0	18	5	7	8	0
	Saved energy (joule)	−0.013	11.14	58.96	80.22	87.33	−0.0014
	Improvement (%)	−0.003	47.99	7.21	9.98	11.16	−0.0026

*The data in these rows is extrapolated.

we know that the number of active NDPEs needed by HE running on RFNS is 13 (i.e., ceil(36 ÷ 2.98) = 13). In other words, up to 23 NDPEs on RFNS can be powered down to improve the energy efficiency of HE without affecting its performance. Limited by the *mux2* connection scheme, the maximum number of NDPEs that can be powered down is 18 because the value of *nane_mux2* for HE is 18 based on Equation (3). Thus, half of the 36 NDPEs in RFNS will be powered off when the HE application is running on RFNS under a *mux2* connection scheme (see Figure 8).

Since there is no performance degradation when only 18 out of 36 NDPEs are working, the wall time of HE execution on RFNS does not change. The power of NDPEs in RFNS for HE is now reduced from 743.40 watts (see Table 5) to 743.40 × (18/36) = 371.7 watts. Hence, the total energy saved by HE on RFNS is 371.7 watts × wall time (on NDP) = 371.7 watts × 0.03 second (see Table 3) = 11.15 joules. Since the switch layer is not needed when an NDP server does not have reconfigurability and using this switch layer does introduce energy overhead, to make comparisons fair we choose an NDP server without the switch layer as a baseline system when we calculate energy savings and energy efficiency improvements for each application. The power of each Mux/DeMux switch shown in Figure 8 is only 0.99 milliwatt (i.e., supply voltage 3.3 V × 300 μA = 0.99 mW [22]). Thus, the total power consumption of 18 such switches used in Figure 8 is less than 18 milliwatts. When HE is running on RFNS, the energy consumption of the switch layer is no more than 18 mW × wall time (on NDP) = 18 mW × 0.03 second (see Table 3) = 0.0054 joule. Therefore, the final energy saving due to reconfigurability for HE on RFNS is at least 11.14 joules (i.e., 11.15 joule - 0.0054 joule ≈ 11.14 joules). Compared to the energy consumed by HE on FNS (FPGA-based NDP Server) without the switch layer, HE gains 11.14 joules ÷ 23.21 (see Table 5) joules = 47.99% energy efficiency improvement.

Using the same methodology, we can obtain the values of "36-*nane_mux2*" (i.e., the number of inactive NDPEs), "saved energy," and "energy efficiency improvement" for all six applications running on RANS/RFNS. Table 11 summarizes energy efficiency gains from reconfigurability of the two RNDP servers for the six applications. From Table 11, we can see that not all six applications can benefit from reconfigurability by judiciously adjusting the number of NDPEs while maintaining a highest performance. Among the six applications, HE is the only application that can benefit from reconfigurability on both RANS and RFNS. The reason is that on both servers, its NDPE bandwidth is higher than its overall performance (see Table 4). In other words, its performance bottleneck on RANS/RFNS does not occur on NDPEs. Therefore, there is an opportunity for it to enjoy the benefits of reconfigurability by powering off part of NDPEs without any performance loss. As for the three k-NN applications, they can only benefit from reconfigurability on RFNS as they can power off five, seven, and eight NDPEs, respectively (see Table 11). Like LC and FFT, the performance bottleneck of the three k-NN applications on RANS occurs on NDPEs. This is the reason they cannot benefit from the reconfigurability of RANS. Obviously, when NDPE

itself becomes a performance bottleneck of an application, there is no room for the application to gain an energy efficiency improvement from reconfigurability. In this scenario, energy saving and energy efficiency improvement of an application are both negative values shown in Table 11 because the switch layer still needs to consume energy while it cannot provide any benefits. For example, the wall time of k-NN_8 is 65.4 seconds when it is running on RANS (see Table 3). The energy consumption caused by the switch layer for K-NN_8 would be at most 1.18 joules (i.e., 65.4×18 milliwatt ≈ 1.18 joules). Therefore, the energy saving of k-NN_8 due to reconfigurability is -1.18 joules and the energy efficiency improvement is -0.007% (see Table 11).

Now, we present an in-depth analysis of the impact of α and NDPE bandwidth of an application on the benefits that it can obtain from the reconfigurability of RFNS/RANS. First, a low value of α indicates that after NDPEs process the data fetched from SSDs, the size of the results to be sent from NDPEs to host CPUs is still large. In this situation, a high bandwidth of NDPEs is prone to saturate the bandwidth of the data path between NDPEs and host CPUs or the data processing bandwidth of host CPUs. When a "saturation" happens, we know that the data processing bandwidth of NDPEs is higher than necessary as the performance of an application is now limited to either data transfer bandwidth between NDPEs and host CPUs or data processing bandwidth of host CPUs. Thus, it is wise to power off part of NDPEs to lower the bandwidth of NDPEs so that it matches the relatively lower data transfer bandwidth between NDPEs and host CPUs. In this way, energy saving from reconfigurability can be achieved. A low α value (i.e., 1) and a high bandwidth of NDPEs of HE explain why it gains the highest energy efficiency improvements on both RANS and RFNS (see Table 11). Similarly, a high bandwidth of NDPEs of k-NN_2, k-NN_4, and k-NN_8 on RFNS provides them an opportunity to benefit from the reconfigurability of RFNS. However, their high values of α (i.e., 8, see Table 3) greatly reduce the benefits. This is the reason the energy efficiency improvements for the three k-NN applications (i.e., 7.21%, 9.98%, and 11.16%) are lower than that of HE (i.e., 47.99%). Second, when we compare the energy efficiency gains derived from reconfigurability of RANS and RFNS, it is clear that RFNS can deliver more benefits than RANS for the same application (e.g., HE). This observation can also be explained by the fact that RFNS can always deliver a higher bandwidth of NDPEs than RANS for the same application (see the "NDP/CPU" column in Table 4 and Table 7) due to its hardware-level acceleration of FPGA. The conclusion is that a high bandwidth of NDPEs is prone to obtain more benefits from the reconfigurability of an RNDP server.

In summary, the "always running in full gear" strategy for a reconfigurable NDP server may not be wise for an application with a high NDPE bandwidth and a low value of α. Reconfigurability of RANS and RFNS can be exploited to achieve an additional energy efficiency gain without any performance degradation. In this section, we obtain the following findings. **Finding 6:** *An application with a high NDPE bandwidth and low α might be able to achieve an energy efficiency improvement by powering off partial NDPEs without any performance degradation.* **Finding 7:** *RFNS can deliver more benefits from the reconfigurability than RANS for the same application due to its hardware-level acceleration of FPGA.*

7 DISCUSSIONS

In this section, we discuss the limitations of this research, which could inspire researchers to develop an enhanced reconfigurable NDP server framework for data centers in the future.

Applicability to complex parallelism patterns: As an initial step toward understanding the impact of NDP on data center applications, in this research we only consider embarrassingly parallel applications. In these applications, data processing time, data transfer bandwidth, and storage bandwidth are all linear, which makes our models simple. Still, there are many non-linear parallel

applications running in data centers. In these scenarios, an application consists of multiple non-identical tasks that need to communicate with each other, CPUs may have distinct power modes invoked at different utilizations, start-up and shut-down overhead are non-trivial, and SSDs need to perform housekeeping operations (e.g., garbage collection) at unexpected times. All these characteristics need to be factored into future data center NDP computing models. Nevertheless, we hope that the insights on the impact of NDP on embarrassingly parallel applications provided by this research would motivate researchers to further investigate this topic on both fine-grained parallel applications and coarse-grained parallel applications.

The need for application reconfiguration in RFNS: It is understandable that both OpenCL hosts and OpenCL kernels need to be reconfigured when an RFNS switches to serve a new application. However, for a particular application that has already run on an RFNS, when the number of SSDs processed by an NDP engine changes due to powering off part of NDPEs to save energy, the RFNS needs to reconfigure both OpenCL hosts and OpenCL kernels in an offline mode so that each OpenCL kernel can have the knowledge of its new data sources. The need for reconfiguration of a particular application (i.e., configuring it more than once) is one limitation of the proposed RFNS. To avoid it, one approach is to dynamically lower the resource usage or clock frequency of FPGA to save energy rather than powering off part of NDPEs. We will explore this option in the future.

A static manual-sharding strategy: For the six applications used in this research, a static manual-sharding strategy is sufficient for two reasons. First, in all datasets of the six applications, each data element (e.g., a data point in LC or a picture in HE) is independent of other data elements, which makes partitioning a dataset in a horizontal way feasible. Second, the datasets of all six applications are static as (1) their sizes are fixed, (2) the popularity of a data element is unchanged over time, and (3) all data elements in a dataset are equally popular. However, when a proposed RNDP server is employed for an application with a dynamic workload, new challenges such as load fluctuation, horizontal skew (i.e., some data elements are accessed more frequently than others), and temporal skew (i.e., the skew distribution changes over time) have to be addressed. Thus, a smart auto-sharding mechanism that can dynamically re-distribute data elements of a dataset across all active SSDs is required. Fortunately, existing auto-sharding techniques for datacenter applications [1] and specific databases [35, 38] shed light on how to develop such a strategy for an enhanced version of RNDP servers.

8 CONCLUSIONS

Existing NDP techniques mainly focus on developing an intelligent storage/memory device with computational capabilities (e.g., Sumsung SmartSSD [33]) and they normally target data-intensive applications. A conventional data center server, however, needs to serve not only data-intensive but also compute-intensive applications. Besides, it is challenged by a performance imbalance between data transfer and data processing. An NDP-powered data center server seems a promising solution to solve the problem of performance imbalance while effectively serving a wide spectrum of applications. Moreover, an NDP sever can provide an in-depth understanding of the impact of NDP on data center applications.

In this article, we first implement a single-engine prototype for each of the two proposed reconfigurable NDP servers. Next, we measure the performance and energy efficiency of six data center applications running on the two prototypes. The measured experimental results are then extrapolated to estimate and predict the performance and general trends of RANS and RFNS. In particular, we demonstrate that the "always running in full gear" strategy for a reconfigurable NDP server may not be wise for certain applications. Finally, we provide seven new findings, which shed light on the development of a full-fledged reconfigurable NDP server for data centers.

ACKNOWLEDGMENTS

This research was supported by Samsung Memory Solutions Laboratory (MSL). We thank our colleagues from MSL who provided insights and expertise that greatly assisted the research.

REFERENCES

[1] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, R. Peon, Larry Kai, A. Shraer, Arif Merchant, and Kfir Lev-Ari. 2016. Slicer: Auto-sharding for datacenter applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 739–753.

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2016. A scalable processing-in-memory accelerator for parallel graph processing. *ACM SIGARCH Computer Architecture News* 43, 3 (2016), 105–117.

[3] Krste Asanovic and David Patterson. 2014. Firebox: A hardware building block for 2020 warehouse-scale computers. In *12th USENIX Conference on File and Storage Technologies (FAST'14)*. Keynote presentation.

[4] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. 2007. An analysis of latent sector errors in disk drives. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'07)*. ACM, 289–300.

[5] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H. Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. 2014. Near-data processing: Insights from a micro-46 workshop. *IEEE Micro* 34, 4 (2014), 36–42.

[6] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. 2017. It's time to think about an operating system for near data processing architectures. In *Workshop on Hot Topics in Operating Systems*. ACM, 56–61.

[7] Jing Bi, Haitao Yuan, Wei Tan, MengChu Zhou, Yushun Fan, Jia Zhang, and Jianqiang Li. 2017. Application-aware dynamic fine-grained resource provisioning in a virtualized cloud data center. *IEEE Transactions on Automation Science and Engineering* 14, 2 (2017), 1172–1184.

[8] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Kevin Hsieh, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu. 2017. LazyPIM: An efficient cache coherence mechanism for processing-in-memory. *IEEE Computer Architecture Letters* 16, 1 (2017), 46–50.

[9] C. L. Philip Chen and Chun-Yang Zhang. 2014. Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information Sciences* 275 (2014), 314–347.

[10] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R. Ganger. 2013. Active disk meets flash: A case for intelligent SSDs. In *27th International ACM Conference on International Conference on Supercomputing (ICS'13)*. ACM, 91–102.

[11] Seokhei Cho, Changhyun Park, Youjip Won, Sooyong Kang, Jaehyuk Cha, Sungroh Yoon, and Jongmoo Choi. 2015. Design tradeoffs of SSDs: From energy consumption's perspective. *ACM Transactions on Storage (TOS)* 11, 2 (2015), 8.

[12] CNXSoft. 2015. AllWinner A64 a quad core 64-bit ARM cortex A53 SoC for tablets. https://www.cnx-software.com/2015/01/08/allwinner-a64-is-a-5-quad-core-64-bit-arm-cortex-a53-soc-for-tablets/z.

[13] George S. Davidson, Jim R. Cowie, Stephen C. Helmreich, Ron A. Zacharski, and Kevin W. Boyack. 2006. *Data-centric Computing with the Netezza Architecture*. Technical Report. Sandia National Laboratories.

[14] Arup De, Maya Gokhale, Rajesh Gupta, and Steven Swanson. 2013. Minerva: Accelerating data analysis in next-generation SSDs. In *Field-Programmable Custom Computing Machines (FCCM'13)*. IEEE, 9–16.

[15] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition, 2009 (CVPR'09)*. IEEE, 248–255.

[16] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query processing on smart SSDs: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 1221–1230.

[17] Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox. [n. d.]. Mapreduce for data intensive scientific analyses. In *eScience'08*. IEEE, 277–284.

[18] Fidus Systems Inc. 2017. Fidus Sidewinder-100. https://www.xilinx.com/products/boards-and-kits/1-o1x8yv.html.

[19] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. 2015. Practical near-data processing for in-memory analytics frameworks. In *2015 International Conference on Parallel Architecture and Compilation (PACT'15)*. IEEE, 113–124.

[20] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: A framework for near-data processing of big data workloads. In *43rd Annual International Symposium on Computer Architecture (ISCA'16)*. ACM/IEEE, 153–165.

[21] Hongjiang He and Hui Guo. 2008. The realization of FFT algorithm based on FPGA co-processor. In *Second International Symposium on Intelligent Information Technology Application, 2008 (IITA'08)*. Vol. 3. IEEE, 239–243.

[22] DIODES Incorporated. 2018. PI3DBS16212, 2:1 Mux/De-Mux Switch. https://www.diodes.com/assets/Databriefs/PI3DBS16212-Product-Brief.pdf.

[23] Intel. 2017. Intel® Xeon® Gold 6154 Processor. https://ark.intel.com/products/120495/Intel-Xeon-Gold-6154-Processor-24_75M-Cache-3_00-GHz.
[24] Zsolt István, David Sidler, and Gustavo Alonso. 2017. Caribou: Intelligent distributed storage. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1202–1213.
[25] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. 2016. YourSQL: A high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment* 9, 12 (2016), 924–935.
[26] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. 2015. BlueDBM: An appliance for big data analytics. In *42nd Annual International Symposium on Computer Architecture (ISCA'15)*. ACM/IEEE, 1–13.
[27] Dzmitry Kliazovich, Pascal Bouvry, and Samee Ullah Khan. 2013. Simulation and performance analysis of data intensive and workload intensive cloud computing data centers. In *Optical Interconnects for Future Data Center Networks*. Springer, 47–63.
[28] Gunjae Koo, Kiran Kumar Matam, H. V. Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, Murali Annavaram, et al. 2017. Summarizer: Trading communication with computing near storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 219–231.
[29] David Mayhew and Venkata Krishnan. 2003. PCI express and advanced switching: Evolutionary path to building next generation interconnects. In *Proceedings of the 11th Symposium on High Performance Interconnects, 2003.* IEEE, 21–29.
[30] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. 2007. Failure trends in a large disk drive population. In *5th USENIX Conference on File and Storage Technologies (FAST'07)*. USENIX Association.
[31] David Putzolu, Sanjay Bakshi, Satyendra Yadav, and Raj Yavatkar. 2000. The phoenix framework: A practical architecture for programmable networks. *IEEE Communications Magazine* 38, 3 (2000), 160–165.
[32] Rodinia. 2009. Rodinia: Accelerating compute-intensive applications with accelerators. http://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Rodinia:Accelerating_Compute-Intensive_Applications_with_Acceleratorsz.
[33] Samsung. 2016. SmartSSD® Computational Storage Drive. https://samsungsemiconductor-us.com/smartssd//.
[34] Samsung. 2017. Mission Peak NGSFF All Flash NVMe Reference Design. http://www.samsung.com/semiconductor/insights/tech-leadership/mission-peak-ngsff-all-flash-nvme-reference-design/.
[35] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. 2014. Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1035–1046.
[36] Xiaojia Song, Tao Xie, and Stephen Fischer. 2019. A near-data processing server architecture and its impact on data center applications. In *International Conference on High Performance Computing*. Springer, 81–98.
[37] Xiaojia Song, Tao Xie, and Wen Pan. 2018. RISP: A reconfigurable in-storage processing framework with energy-awareness. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID'18)*. IEEE, 193–202.
[38] Rebecca Taft, E. Mansour, M. Serafini, J. Duggan, Aaron J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. 2014. E-store: Fine-grained elastic partitioning for distributed transaction processing. *Proceedings of the VLDB Endowment* 8 (2014), 245–256.
[39] Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. 2011. Phoenix++: Modular mapreduce for shared-memory systems. In *Proceedings of the 2nd International Workshop on MapReduce and Its Applications*. ACM, 9–16.
[40] Devesh Tiwari, Simona Boboila, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. 2013. Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *11th USENIX Conference on File and Storage Technologies (FAST'13)*. 119–132.
[41] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188.
[42] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: An intelligent storage engine with support for advanced SQL offloading. *Proceedings of the VLDB Endowment* 7, 11 (2014), 963–974.
[43] Caesar Wu, Rajkumar Buyya, and Kotagiri Ramamohanarao. 2016. Big data analytics = machine learning + cloud computing. *arXiv preprint:1601.03115* (2016).
[44] Xing Wu and Frank Mueller. 2011. ScalaExtrap: Trace-based communication extrapolation for spmd programs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*. ACM, 113–122.
[45] Wm A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News* 23, 1 (1995), 20–24.
[46] Xilinx. 2017. Xilinx Xilinx Virtex UltraScale+ FPGA VCU1525. https://www.xilinx.com/products/boards-and-kits/vcu1525-a.html.

[47] Masato Yoshimi, Yasin Oge, and Tsutomu Yoshinaga. 2017. Pipelined parallel join and its FPGA-based acceleration. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 10, 4 (2017), 28.

[48] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: Throughput-oriented programmable processing in memory. In *the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC'14)*. ACM, 85–98.

Received July 2020; revised March 2021; accepted April 2021