Triple Engine Processor (TEP): A Heterogeneous Near-Memory Processor for Diverse Kernel Operations

HONGYEOL LIM and GIHO PARK, Sejong University

49

The advent of 3D memory stacking technology, which integrates a logic layer and stacked memories, is expected to be one of the most promising memory technologies to mitigate the memory wall problem by leveraging the concept of near-memory processing (NMP). With the ability to process data locally within the logic layer of stacked memory, a variety of emerging big data applications can achieve significant performance and energy-efficiency benefits. Various approaches to the NMP logic layer architecture have been studied to utilize the advantage of stacked memory. While significant acceleration of specific kernel operations has been derived from previous NMP studies, an NMP-based system using an NMP logic architecture capable of handling some specific kernel operations can suffer from performance and energy efficiency degradation caused by a significant communication overhead between the host processor and NMP stack.

In this article, we first analyze the kernel operations that can greatly improve the performance of NMP-based systems in diverse emerging applications, and then we analyze the architecture to efficiently process the extracted kernel operations. This analysis confirms that three categories of processing engines for NMP logic are required for efficient processing of a variety of emerging applications, and thus we propose a Triple Engine Processor (TEP), a heterogeneous near-memory processor with three types of computing engines. These three types of engines are an in-order core, a coarse-grain reconfigurable processor (CGRA), and dedicated hardware. The proposed TEP provides about 3.4 times higher performance and 33% greater energy savings than the baseline 3D memory system.

CCS Concepts: • **Computer systems organization** → *Multicore architectures*; Heterogeneous (hybrid) systems;

Additional Key Words and Phrases: 3D stacked memory, near-memory processing (NMP), big data application, heterogeneous architecture

ACM Reference format:

Hongyeol Lim and Giho Park. 2017. Triple Engine Processor (TEP): A Heterogeneous Near-Memory Processor for Diverse Kernel Operations. *ACM Trans. Archit. Code Optim.* 14, 4, Article 49 (December 2017), 25 pages. <https://doi.org/10.1145/3155920>

1 INTRODUCTION

Recently, emerging applications like machine learning (ML), high-performance computing (HPC), natural language processing (NLP), and big data analytics algorithms have diffused explosively toward various domains. However, the tremendous cost of data movement through a deep memory hierarchy of state-of-the-art computer systems has become a major issue that threatens both

New article, not an extension of a conference paper.

This work is supported by National Research Foundation of Korea (NRF), under grant 2015-R1D1A1A01059711.

Authors' addresses: H. Lim and G. Park (Corresponding author), Computer Science and Engineering, Sejong University, 209, Neungdong-ro, Gwangjin-gu, Seoul, Korea; emails: hylim@sju.ac.kr, ghpark@sejong.ac.kr.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2017 ACM 1544-3566/2017/12-ART49 \$15.00

<https://doi.org/10.1145/3155920>

performance and energy efficiency. The advent of 3D memory stacking technology, which integrates a logic layer and stacked memory, is expected to be the most promising memory technology to mitigate memory wall problems by utilizing the concept of near-memory processing (NMP). As localized data processing within the logic layer of the stacked memory is available, various studies have been investigated for an NMP logic layer architecture capable of acceleration and offloading near the stacked memory.

In prior works, various studies have focused on accelerating the specific kernel operations involved in applications, and the NMP logic architecture based on dedicated accelerators or multiple programmable cores has been introduced as a solution (Thanh-Hoang et al. 2016; Ahn et al. 2015; Kim et al. 2015; Ahn et al. 2015; Akin et al. 2014; Zhu et al. 2013). However, the big data applications containing various kernel operations are expected to be one of the primary target applications to be executed on a system with the concept of NMP (Balasubramonian et al. 2014; Guz et al. 2015). While a significant acceleration of the specific kernel operation has resulted from the previous studies, the data included in several unsupportable kernel operations within the NMP logic layer should be transferred to the host processor through a deep memory hierarchy. As a result, previous NMP logic layer architectures can suffer from a significant communication overhead between the host processor and the NMP stack. This overhead leads to degraded performance and energy efficiency of the NMP-based system. Therefore, the NMP logic layer architecture, which can reduce communication overhead by handling the various kernel operations contained in an application and accelerate these kernel operations, thus having a significant impact on performance, is of critical importance to fully exploit the benefits of an NMP-based system.

In this study, we propose a Triple Engine Processor (TEP), which is a heterogeneous near-memory processor for processing various kernel operations included in applications. Our TEP consists of three different data processing engines: an energy-efficient programmable core for control-flow and memory reference operations, reconfigurable processors for data-parallel computing operations, and a dedicated hardware engine (or a set of dedicated hardware engines) for specific kernel operation that can result in much higher performance than processing in the programmable core. In the evaluation result, the proposed TEP achieves significant performance improvements, about 3.4 times and 1.9 times better on average, compared to the baseline 3D memory system and an NMP engine consisting of 16 in-order cores, respectively. In addition, our TEP provides 33% lower energy consumption and reduces 52% of data traffic compared to the baseline 3D memory system.

In summary, the key contributions of our work are as follows:

- To design the proposed TEP, we first extract the NMP kernel from various applications that can benefit from performance and energy efficiency by allocating them to the NMP stack and provide a detailed analysis of their characteristics.
- We analyze the key operations contained in the extracted NMP kernels and classify them into three operation types according to the characteristics of the key operation. Based on the classified operation types, we explore several representative architectures models with different features required to handle them efficiently, and derive three architectures for the NMP logic layer.
- We propose TEP, a heterogeneous near-memory processor designed with three architecture models derived from the analysis of applications. We also provide the operating mechanism, communication interface, and programming model to enable efficient data processing in our TEP-based system.
- Finally, we evaluate various NMP-based models introduced in previous studies. We demonstrate the effectiveness of a heterogeneous architecture model in the logic layer of the

Fig. 1. A baseline NMP system and a block diagram of the NMP logic base.

NMP-based system by evaluating the proposed TEP and previous NMP-based models in terms of performance, energy, and communication overhead.

The remainder of this article is organized as follows. Section 2 explains the background for this study. Section 3 introduces the target applications used in this study. Section 4 classifies the operation types and explores diverse architectures for NMP logic. Section 5 presents our proposed TEP. Section 6 explains our evaluation methodology. Section 7 demonstrates the simulation results. Section 8 presents prior works. Finally, Section 9 concludes this study.

2 BACKGROUND

In this section, we provide background on the baseline NMP architecture and Hybrid Memory Cube (HMC).

2.1 Baseline NMP Architecture

Figure 1 illustrates an overview of our baseline NMP architecture. In general, an NMP-based system is composed with a high-performance host processor and an NMP stack. The host processor communicates with the NMP stack using high-speed I/O links, as shown in Figure 1. The NMP stack is constructed with a logic layer and DRAM dies based on 3D integration technology like HMC (Jeddeloh et al. 2012). An NMP stack can be daisy-chained to other NMP stacks, or many NMP stacks can constitute processor/memory-centric networks to utilize the processor I/O bandwidth in a system. The stacked DRAM dies are divided into vertically aligned partitions, called *vaults* in HMC, and each vault has a memory controller (aka *vault controller*) in the logic layer, as shown in Figure 1. The vault controller is responsible for ensuring DRAM-specific timing constraints, DRAM sequencing, error detection and correction, and so on.

2.2 Application Acceleration in NMP-Based System

In an NMP-based system, an NMP logic is responsible for the data acceleration/offloading by locating closely with the vertically stacked memory. Like SIMD/GPU engines, the NMP logic accelerates some specific parts of an application, such as specific operations, functions, code segments, or threads (Balasubramonian et al. 2014). The host processor, on the other hand, executes the remaining segments that are not allocated to the NMP stack and processes the output data produced by the NMP logic.

Therefore, the kernel allocation policy for the NMP engine can have a decisive impact on the performance and energy efficiency of NMP-based systems. In addition, the characteristics of the kernel to be allocated to the NMP engine can be utilized as one of the key factors for designing

Table 1. List of Target Applications Used in this Study

Domain	Application name	Brief description	Input Dataset	Reference
Machine Learning (ML)	K-means	K-means clustering algorithm.	262,144 pairs of random (r, g, b) values.	[A. Yazdanbakhsh et al. 2016]
	Restricted boltzmann machine (RBM)	A stochastic neural network algorithm with applications in collaborative filtering, feature learning, and topic modeling.	Netflix dataset in [S. Thomas et al. 2014]	[S. Thomas et al. 2014]
	Liblinear	A machine learning library for large-scale linear classification and regression.	FBI crime statistics dataset in [S. Thomas et al. 2014].	[S. Thomas et al. 2014]
	Stemmer	A word stemming algorithm based on [M. F. Porter 1980] to expose the root of a word by matching and truncating common worked endings.	Dataset constructed of 4M words list.	[J. Hauswald et al. 2015]
Natural Language Processing (NLP)	Latent Dirichlet allocation (LDA)	A topic modeling algorithm to discover topics from unordered documents.	20 news groups dataset, topics - 4, 8, 16.	[S. Thomas et al. 2014]
High Performance Computing (HPC)	MiniMD	Parallel molecular dynamics (MD) simulation package used in supercomputers.	Default parameters with a serial mode.	[M. A. Heroux et al. 2009]
	HPCCG	A simple conjugate gradient benchmark code for a 3D chimney domain.	Default parameters with a serial mode.	[M. A. Heroux et al. 2009]
In-Memory DB	Memcached	A KEY-VALUE dataset caching system. In this study, we use standard Memcached (version 1.4.25).	Twitter dataset (around 10GB) with fraction of 0.9:0.1 (GET/SET command).	[Memcached], [T. Palit et al. 2016]
Computer Vision	Super-resolution Reconstruction (SRR)	An image resolution recover algorithm from a series of low resolution (LR) image to one high resolution (HR) image.	100 compressed color frames of size 240 x 320.	[S. Thomas et al. 2014]
	Motion Estimation (ME)	The process of finding motion vectors that describe the transformation of one 2D image.	100 compressed color frames of size 240 x 320.	[S. Thomas et al. 2014]
Automotive	3D-LiDAR	[Alempijevic et al. 2006] based a signal processing algorithm for automatic extrinsic calibration of a 3D laser scanner and optical camera system.	Sample datasets gathered from 3D laser scanner and omnidirectional camera system mounted on the roof of a vehicle.	[G. Pandey et al. 2012]

the architecture of the NMP logic layer (Balasubramonian et al. 2014; Loh et al. 2013). For example, an NMP engine with limited functionality and careless kernel allocation for that engine can cause inefficient (or useless) data movement between the host processor and the NMP stack, which increases the communication overhead.

3 TARGET APPLICATIONS AND PROFILING

3.1 Target Applications

Table 1 shows 11 different target applications used in this study. We select diverse applications from various application domains, such as ML, NLP, HPC, in-memory DB, computer vision, and automotive systems, which have been considered in previous NMP studies (Farmahini-Farahani et al. 2015; Ahn et al. 2015; Kim et al. 2015).

3.2 Profiling and Analysis Methodology

In this study, we use GPROF (Graham et al. 1982) and KCacheGrind (2013) profiling tools to extract the major functions with a large contribution on program execution time. To extract the major function, we use a high-performance multi-core system with a 6-core CPU (Intel i7-5930K @3.50GHz) and 32GB total system memory. A last-level cache (LLC) has 15MB and is shared across all cores. In this article, except for Memcached, 3D-LiDAR, and K-means applications, the major functions are extracted until the sum of the execution times of the functions included in the target application exceeds 95%, because these applications contain functions that perform file or network I/O event handling. The misses per kilo instruction (MPKI) of the last-level cache (LLC) of the target applications and memory intensity, meaning the memory traffic caused by certain functions, are measured in the simulation framework. (The detailed simulation configuration will be presented in Section 6.)

Table 2. Analysis Results of Target Applications

Application name	Application profiling result				Operation types % time in kernel (# of code block)		
	Name of major functions (NMP Kernels)	% time	Global MPKI	Memory Intensity	DCO	MMO	CFO
K-means	euclideanDistance	35.2%	3.41	80.3%	89.9% (1)	10.1% (1)	0.0% (0)
	assignCluster	23.1%			99.8% (1)	0.0% (0)	0.2% (1)
	segmentImage	12.5%			90.1% (3)	9.8% (2)	0.1% (1)
RBM	train	44.0%	10.28	92.1%	82.1% (3)	17.6% (1)	0.3% (3)
	activateVisibleUnits	30.5%			94.1% (3)	5.2% (2)	0.7% (2)
	activateHiddenUnits	25.1%			99.2% (2)	0% (0)	0.8% (3)
Liblinear	solve l2r l1l2 svc	99.6%	41.58	98.7%	82.8% (6)	14% (4)	3.2% (8)
Stemmer	m	49.9%	24.97	99.1%	98.8% (3)	0% (0)	1.2% (6)
	cons	25.0%			0.0% (0)	98.9% (1)	1.1% (1)
	setto	25.0%			2.20% (1)	97.8% (1)	0% (0)
LDA	digamma	70.8%	31.68	95.8%	100% (1)	0% (0)	0% (0)
	compute likelihood	12.0%			99.9% (3)	0% (0)	0.1% (1)
	lda inference	12.0%			92.2% (3)	7.4% (1)	0.4% (1)
MiniMD	compute halfneigh	77.1%	4.71	81.1%	94.5% (1)	4.5% (1)	1.0% (2)
	build	17.2%			93.2% (1)	5.4% (4)	1.4% (4)
	initialIntegrate	3.2%			100% (1)	0% (0)	0% (0)
HPCCG	HPC sparsemv	83.3%	31.21	87.5%	96.7% (1)	3.3% (2)	0.0% (0)
	ddot	11.1%			99.7% (2)	0.2% (1)	0.1% (0)
	waxpby	3.7%			99.9% (3)	0% (0)	0.1% (1)
Memcached	assoc find	43.8%	78.36	86.9%	2.4% (1)	97.5% (1)	0.1% (2)
	jenkins hash	20.2%			94.8% (1)	0% (0)	5.2% (4)
SRR	MatMul	57.7%	16.37	93.6%	100% (1)	0% (0)	0% (0)
	GaussSeidel	27.1%			92.7% (2)	7.1% (1)	0.2% (2)
	get_g	9.4%			92.9% (1)	6.3% (2)	0.8% (6)
ME	median	62.5%	11.34	96%	5.2% (1)	94.7% (1)	0.1% (1)
	Motion Est	35.5%			100% (1)	0% (0)	0% (0)
3D-LiDAR	get histogram	48.9%	31.28	91.5%	98.8% (4)	0% (0)	1.2% (6)
	matrix vector multiply 3x3 3d	29.1%			100% (1)	0% (0)	0% (0)
	vector add 3d	12.5%			100% (1)	0% (0)	0% (0)

3.3 Profiling Results

Table 2 summarizes the analysis results of the target applications. As shown in Table 2, most applications have many functions, but only some of them contribute significantly to the total execution time (see the “% time” column in Table 2) and generate almost all of the memory traffic generated by the application (see the “memory intensity” column in Table 2). Thus, we can expect two significant benefits in terms of performance and energy efficiency by adopting the concept of NMP to the application. In this study, we focus on designing the architecture of an NMP engine by considering the extracted major functions as target NMP kernels to be executed in NMP logic and analyzing the characteristics of key operations included in NMP kernels.

4 ARCHITECTURE DESIGN BASED ON CLASSIFICATION OF OPERATION TYPES

In this section, we analyze the key operations included in the extracted NMP kernels and classify them into three operation types according to the characteristics of the key operations. We also analyze the architectural features required to efficiently support the classified operation types and explore several representative architecture models. Finally, we derive three architectures for the NMP logic used in the proposed baseline architecture.

```

1 void MatMul (double** mat1, double mat2[], double result[]) {
2   for(int i = 0; i < (I*I); i++) {
3     for (int j = 0; j < (j*j); j++) {
4       result[i] += mat1[i][j] * mat2[j];
5     }
6   }
7 }

```

(a) Code example of DCO type (in SRR application).

```

1 void train() {
2   for (int user = 0; user < USERS; user++) {
3     int data[NUM_VISIBLE + 1];
4     memcpy(data, trainingData[user], NUM_VISIBLE * sizeof(int));
5     /* omission */
6     int pos[NUM_VISIBLE + 1][NUM_HIDDEN + 1];
7     for (int v = 0; v < NUM_VISIBLE + 1; v++) {
8       if (data[v] != -1) {
9         for (int h = 0; h < NUM_HIDDEN + 1; h++)
10            pos[v][h] = data[v] * hidden[h];
11       }
12     }
13   }
14   /* omission */
15 }

```

(b) Code example of mixed MMO, CFO, and DCO types (in RBM application).

```

1 item *assoc_find(const char *key, const size_t nkey, const uint32_t hv) {
2   item *it;
3   unsigned int oldbucket;
4
5   if (expanding && (oldbucket = (hv & hashmask(hashpower - 1))) >= expand_bucket) {
6     it = old_hashtable[oldbucket];
7   } else {
8     it = primary_hashtable[hv & hashmask(hashpower)];
9   }
10
11   item *ret = NULL;
12   while (it) {
13     if ((nkey == it->nkey) && (memcmp(key, ITEM_key(it), nkey) == 0)) {
14       ret = it;
15       break;
16     }
17     it = it->h_next;
18   }
19   return ret;
20 }

```

(c) Code example of mixed CFO and MMO types (in Memcached application).

Fig. 2. Code examples of the categorized operation types in this study.

4.1 Three Different Types of Kernel Operations

- (1) **Data-parallel Computing Operation (DCO):** The first type of operation is the DCO, which computes a large amount of data based on high data-level parallelism. Figure 2(a) shows a code example of a DCO type composed of code segments that can be processed in parallel, such as computing blocks with large loop bodies and data-flow computations. Table 2 shows that the target applications (e.g., K-means, RBM, LDA, miniMD, HPCCG,

SRR, and 3D-LiDAR) contain a small number of DCO types, but they are a large part of execution time. Thus, we can expect performance and energy efficiency benefits by accelerating the DCO type at the NMP logic layer.

- (2) **Memory Manipulation Operation (MMO)**: The second type of operation is MMO, which refers to a large amount of data stored in the memory. The target applications contain a large number of memory reference operations such as data movement, initialization, copying, scatter/gather, and comparison. Since the MMO type causes large amounts of data traffic in memory hierarchy by referencing bulk-scaled data, the MMO type has been visited as one of the suitable operations to offload in the NMP logic layer from many previous studies (Loh et al. 2013; Akin et al. 2015). Figure 2(b) presents a code example of the MMO type. The MMO type handles large-scale data with very simple address calculations (or no calculations). In Table 2, the MMO type makes a significant contribution to the total execution time of Stemmer, ME, and Memcached applications.
- (3) **Control-Flow Operation (CFO)**: In our target applications, we can easily identify multiple code segments, which include control-flow (e.g., if/else or switch/case statements) and low (or no) data-level parallelism operations. In this study, these operations are called the control-flow operations (CFO) type. As shown in Figures 2(b) and 2(c), the CFO type performs logical operations or simple calculations using the data mainly from the DCO or MMO type, and the results of the CFO type is often used to determine the next program flow. As shown in Table 2, most NMP kernels extracted from the target applications contain a large number of the CFO type, but the CFO type has a smaller impact on execution time than other types.

4.2 Architecture Design for Processing Three Operation Types

- **Architecture model for the DCO type**: The DCO type is calculated with limited operators such as addition, subtraction, multiplication, division, and square root, and the order of the operators that are executed for each NMP kernel is different. Therefore, it is necessary to have an architecture with both the programmability and the ability to process various types of operations in parallel to efficiently process the DCO type. Several architectural models such as fine-grain reconfigurable arrays (FPGAs), coarse-grain reconfigurable arrays (CGRAs), and SIMD/GPU-based accelerators can be considered to meet the requirements for the DCO type. While these models are structures that can handle DCO types by utilizing resources that allow for parallel processing, CGRA can be considered an appropriate architectural model for performing the DCO types that involve most parallel workloads. This decision is made based on the results that CGRAs provide higher performance and lower energy consumption for parallel workload processing than the SIMD/GPU engine and have a much smaller configuration data than FPGAs, as demonstrated in previous studies (Gao et al. 2016; Huang et al. 2013; Govindaraju et al. 2012; Tripp et al. 2007). In addition, low energy is important, because the near-memory processor has more stringent power/thermal constraints. Thus, this article focused on CGRAs to support the DCO types that involve most parallel workloads.
- **Architecture model for MMO type**: To handle the MMO in the NMP logic layer, we can approach them in two ways, similar to previous studies. First, because most MMO types have fixed-function operations such as memory read-manipulation-write, a dedicated hardware accelerator (Akin et al. 2015; Seshadri et al. 2013) can be designed for the operations. Dedicated hardware accelerators have the advantages of providing much higher performance and energy efficiency than programmable architecture models. However, because it is designed for specific operations, it is very difficult to apply other operations (or

applications). On the other hand, we can consider a programmable architecture model such as the energy efficient in-order core used in previous NMP research (Kim et al. 2015; Ahn et al. 2015). Programmable engines can overcome the disadvantages of dedicated hardware accelerators, but they have the disadvantage of relatively low performance. Given the trade-offs in both architectures, we can use an energy-efficient in-order core or a dedicated hardware accelerator (or a set of dedicated hardware accelerators) to process the MMO type. If some specific tasks that have a significant impact on the overall application performance, then a hardware accelerator or set of accelerators will be designed to accelerate the tasks. The target kernel operation and the architecture of the accelerator will be discussed in Section 5.

- **Architecture model for CFO type:** As shown in Figures 2(b) and 2(c), the data consumed by CFO is shared with the DCO and MMO types and is often used to determine the next program flow. To handle the CFO type including control-flow operations, an architecture model with flexibility and programmability is required. Thus, we can consider a low-energy programmable core (e.g., in-order core) employed in a NMP logic layer like prior studies (Kim et al. 2015; Ahn et al. 2015)

Previous studies have focused on the NMP engine architecture to accelerate the DCO or MMO type rather than the CFO, but the CFO type can be a major cause of increased communication overhead between the host processor and the NMP stack in terms of performance and energy efficiency. In a NMP-based system with a NMP logic layer that cannot handle the CFO, the data consumed by the CFO type should be transferred to the host processor through a deep memory hierarchy. For instance, CGRAs-based accelerators specialized for the DCO type (Gao et al. 2016; Farmahini-Farahani et al. 2015) or dedicated hardware logics designed for the MMO type (Akin et al. 2015; Seshadri et al. 2013) should rely on host-processing to handle the data included in the CFO type. In other words, the data consumed by operations not supported in the NMP engine should be passed to the host processor and the NMP engine must wait until the host processing is completed and the data is stored in the main memory (i.e., until the data to be used is updated). As a result, it causes increments of global data transfer through printed circuit boards (PCBs) in proportion to the amount of the shared data and this can diminish the benefits obtained from an NMP-based system. According to Jeddeloh et al. (2012); Pugsley et al. (2014), the global data transfer through a PCB, which is located between the host processor and the NMP stack, consumes more than 60 times the energy of a TSV transfer.

5 TRIPLE ENGINE PROCESSOR (TEP)

This section introduces TEP, a heterogeneous near-memory processor that can efficiently handle a variety of kernel operations. The proposed TEP consists of three computing engines determined through application analysis and architectural exploration. Finally, this section presents a programming model and operating mechanism for the proposed TEP.

5.1 Overall Architecture

As shown in Figure 3(a), our TEP is integrated in a memory control interface of the NMP logic layer. Because the proposed TEP is integrated into the memory control interface, the TEP can easily access data distributed across multiple vaults. In addition, we can use the logics already implemented in the standard HMC's memory interface, such as refresh controllers, DRAM sequencers, read/write buffers, and bolt routers. Our TEP can issue memory requests into target vault controllers through a vault router and each vault controller schedules the memory requests.

Fig. 3. Memory control interface and overall architecture of TEP.

Figure 3(b) shows the overall architecture of the TEP proposed in this study. Our TEP consists of three main data processing units, each of which is designed with an energy-efficient in-order core for the CFO and MMO types, and CGRA-based accelerators for the DCO type, and a dedicated hardware accelerator to accelerate the memory compare operation. The architecture details of each component are as follows:

- **Control Computing Unit (CCU):** In this study, the CCU is designed as a single-issue in-order core like the ARM Cortex-A5 processor (ARM [CortexA5 Processor CortexA5](#)). The CCU manages the other two accelerators and is responsible for communication with the host processor, and it can merge/retrieve the output data produced from the two accelerators. In addition, the CCU is responsible for the processing of the CFO and MMO types. In this study, the memory compare operation of the MMO type is accelerated by designing a dedicated hardware accelerator (the architecture is introduced below).

The CCU has a translation lookaside buffer (TLB) and an input/output memory management unit (IOMMU) to support the virtual addresses translation and a direct memory access (DMA) to access the required data from the stacked memory. The virtual memory-related issues will be discussed in Section 5.4. Our CCU has a 16KB single-level instruction cache with a 64byte line size and an 8KB stream buffer to retain a temporal data stream (Jouppi 1990). In this study, we employ a stream buffer to utilize the sequential memory access patterns of the MMO type, such as memmove, memcpy, memset, and swap. A stream buffer consists of a simple FIFO queue that maintains the prefetched data streams, and the CCU releases the modified data for writing to the main memory through a vault router. In our architecture, the CCU issues the prefetching request with successive lines starting at the first memory request, and the stream buffer is flushed when a miss occurs. Finally, the CCU manages the register files to enable communication with the host processor and the other two accelerators. The communication model of our TEP will be explained in Section 5.3.

- **Reconfigurable Processing Unit (RPU):** The RPU is designed with CGRAs and accelerates the DCO type, including data-parallel computing blocks and data-flow operations. The RPU is constructed with four CGRAs to enable concurrent computations with partitioned data and the computation block can be divided into smaller slices.¹ In this study, we use DySER, which has significant benefits in performance and energy efficiency as the basic

¹According to Govindaraju et al. (2012), since the FUs of the RPU will be inactive during reconfiguration on every invocation, it is reported that a very large computing block, which is required to the multiple reconfiguration of CGRA, lowers the overall utilization of the RPU.

Table 3. Simulation Configuration

(a) Host processor	
Cores	x86-64 quad cores, 4GHz, 4-wide OoO, 256-entry RoB
Cache line size	64 Byte
L1 I-Cache	32KB, 4-way, 3-cycle latency
L1 D-Cache	32KB, 8-way, 4-cycle latency
L2 Cache	Private, 256KB, 8-way, 15-cycle latency
L3 Cache (LLC)	Shared, 20MB, 8 banks, 20-way, 30-cycle latency
TLB	32 entries, 2MB page, 200-cycle miss penalty
Memory latency	TLB + L1, L2, L3 + Interconnection + PCB/TSV + DRAM

(b) Control Computing Unit (CCU)	
Core	Single-issue, in-order core, 1GHz.
Cache memory	16KB I-Cache, 4-way, 3-cycle latency, 8KB Stream buffer, 2-cycle latency.
TLB	16 entries, 2MB, 120-cycle miss penalty.

(c) Reconfigurable Processing Unit (RPU)			
Basic Parameters	Frequency	800MHz	
	# FUs	64	
	Configuration latency	64-cycle	
	Configuration memory	64-entry, 4-cycle	
Functional Units	Unit	Count	Latency (cycle)
	INT-ALU	20	1
	FP-ALU		5
	INT-MUL	35	4
	FP-MUL		7
	INT-DIV	6	12
	FP-DIV		12
	SQRT	3	10
Switch	81	1	

(d) Memory Compare Unit (MCU)	
Comparator	Compares 32Byte at once, 2 cycle latency per 4byte. (8 comparators with 4Byte compare capability)
TLB	16 entries, 2MB, 120-cycle miss penalty

(e) NMP with in-order cores	
Cores	Single-issue, 16 in-order cores, 1GHz
L1 I/D-Cache	16KB, 4-way, 3-cycle latency
TLB	16 entries, 2MB, 120-cycle miss penalty
Memory latency	TLB + L1 + TSV + DRAM.

(f) 3D Memory Stack	
Organization	32GB, 8 layers × 16 vaults × 8 stacks
Timing parameters	t _{RP} = 17, t _{CCD} = 6, t _{RCD} = 17, t _{CL} = 17, t _{WR} = 19, t _{RAS} = 34 cycles
Serial links	160GBps, 8-cycle latency
Off-chip links	16Bytes per cycle, four links.

architecture of our RPU (Govindaraju et al. 2012; Farmahini-Farahani et al. 2015). Some modifications of the architecture such as the number of functional units (FUs), placement of FUs, and configuration memory are performed to reflect the analysis results of the target applications.

The RPU is composed of a grid of word-sized FUs that perform various arithmetic and logic operations, which are connected by a configurable interconnect fabric. The RPU also has some internal memory near each FU for temporary results. The FUs of our RPUs are unified dual-mode integer/floating point units, meaning that they can compute in either integer mode or floating point mode (Govindaraju et al. 2012; Farmahini-Farahani et al. 2015). Most FUs can operate addition, subtraction, and multiplication, while a few FUs can perform more complex operations, such as division and square root. In this study, to make efficient routing between FUs, we place FUs according to the arithmetic priority to be used in the computation and the routing distance from the RPU input interface; for example, the FU performing multiplication is placed closer to the input interface than the FU performing addition. Table 3 summarizes the details of the RPU with 64 FUs used in this article (See Section 6).

A configuration context contains the data to be mapped to the RPU for various dataflow graphs and is retained in the memory until it is required and it is cached in the configuration memory local to the CCU. Note that it relies on a compiler to create its configuration and insert instructions in the application to communicate with the CCU. The role of the compiler and programming interface for the proposed TEP will be explained in Section 5.2. The configuration memory consists of a fully associative history table that has a 64-entry and is maintained with the least recently used (LRU) policy.

- Dedicated Hardware Accelerator(s) (Memory Compare Unit (MCU) as an example):** Although most kernel operations included in the application can be handled by employing the CCU and RPU as engines of TEP, a dedicated accelerator (or a set of hardware accelerators) can be designed and used additionally for certain operations to achieve higher performance and energy efficiency than processing in a programmable core. In this work, we present the MCU as an example of dedicated hardware engines for accelerating memory compare operation (i.e., *memcmp()* in the C/C++ standard library suite) with pointer-chasing in a linked-list data structure. According to Saunders (2011), the memory compare operation suffers from very long delays on conventional processor architecture from comparing the memory regions byte-by-byte. In our target applications, we identified both a memory compare operation for pointer-chasing in linked-list structure and a hash function to calculate a hash index, both of which contribute a significant portion of the execution time in the Memcached application (see Table 2). Thus, the MCU accelerates the kernel operation of the Memcached application from among the target applications.

Although this article presents the MCU for only memory compare operation, other additional dedicated accelerators can be also designed and used for other tasks requiring much higher performance or energy efficiency than processing in the programmable core. For instance, another dedicated hardware accelerator for memory swap operation, which accounts for approximately 62% of the total execution time of the ME application, can be designed and used to accelerate the operation as a component of the set of dedicated hardware accelerators in proposed TEP.

To achieve fast memory compare operation with the pointer-chasing, our MCU is designed to have a coarse-grained comparator, simple control logics, and a table buffer, as shown in Figure 3. In our MCU, the comparators have two inputs: one is the KEY value of the table buffer fetched from memory and the other is a KEY value requested by the host processor and is stored in a register. A coarse-grained comparator consists of eight comparators, each of which can compare 4-bytes; thus, it can compare 32-bytes at once.

5.2 Programming Model for the TEP

First, this article assumes that a compiler capable of recognizing the proposed TEP is constructed. Programmers need to make efforts to add several primitives to the program code so that the host processor can recognize and assign tasks to the NMP kernel of the application and each processing engine. Note that the task allocation and implementation methodologies will be explained in Section 6.1. This section describes the programming model for recognizing the NMP kernels, the mapping method of the calculation block in four RPUs, and the application programming interface to trigger the MCU.

- DCO mapping model for CGRAs:** Figure 4(a) shows the partition model to map the computation sub-region into each RPU. A large DCO region can be partitioned into several smaller computation regions like MapReduce to allow concurrent CGRA computations with partitioned data (Farmahini-Farahani et al. 2015; Islam et al. 2014). Some boundary data could be duplicated to enable independent RPU computations. The RPUs output data is processed and merged by the CCU after RPU computations are completed.
- MCU API:** We define an API to trigger MCU as follows.

```
void * mcucmp(const void * bucket_addr, unsigned int offset_key, unsigned int offset_nkey, unsigned int offset_next, const void * target_addr, size_t length);
```

A defined *mcucmp* has six parameters, each of which is stored in the register of the MCU register file. First, the *bucket_addr* parameter is the starting address of the first item that will be searched

Fig. 4. Communication interface between the host processor and TEP.

in the indexed hash bucket (see lines 5–9 in Figure 2(c)). The *offset_key*, *offset_nkey*, and *offset_next* parameters are used to extract the field containing a *KEY* value, length of *KEY*, and next pointer field existing in a linked-list data structure. Finally, the *target_addr* and *length* parameters are an address of a *KEY* value and a length of the *KEY* that are requested for retrieval. The defined API has a very simple and intuitive form, and a programmer can accelerate the memory compare operation by just passing a few parameters without any modification to or re-organization of the data.

5.3 Communication Model for the TEP-based System

The CCU in our TEP is responsible for communication between the host processor and the two acceleration units (i.e., RPUs and MCU). Figure 4(b) presents the communication interface of the proposed TEP.

First, the host processor invokes the proposed TEP by identifying a directive that indicates the NMP kernel inserted by the NMP kernel recognizer, and initializes a finish register (❶). The host processor periodically polls the finish register to check for the completion of kernel operations by the TEP or exceptions. CCU can process the allocated operations (e.g., CFO or MMO type) (❷). When the CCU reaches the RPU instruction inserted by the RPU instruction generator, the CCU initializes a status register and configures the RPUs to run the required configuration for the target kernel (❸). The RPU computes the instructions in a computation sub-region after the input data included in each memory sub-region are ready in each FU. The CCU periodically polls the status register to check for RPU processing completion or errors. The output data produced by RPUs can be processed and merged by the CCU according to the inserted instructions. After the RPU acceleration is completed, the status register is updated (❹). When the CCU reaches MCU's API (i.e., *mcucmp()*), the status register is initialized and the CCU writes the required parameters into the MCU register file (❺). After MCU acceleration is completed, the MCU updates the status register and writes the address of the retrieved item into a return register to be used for informing the host processor (❻). The CCU can process the remaining instructions and stores the output data into the main memory (❼) and notifies the host processor of the completion of the kernel processing by updating a finish register (❽). If the item address retrieved by the MCU is written to the return register, then the host processor may read the address from the return register and perform additional operations.

5.4 Other Issues for TEP Operations

- **Cache coherency and memory consistency:** In our architecture, the host processor and the TEP work serially and do not execute on the same data simultaneously. Therefore, when the TEP is enabled, the host processor waits for the TEP processing to complete. In the proposed TEP, the RPU and MCU are executed serially as well. Note that the effect of performance and energy consumption by the sequential execution model of TEP is reflected in our simulation results in the Section 7.

The TEP has no permission to access the on-chip cache memory of the host processor. Therefore, the data accessed by the TEP should not be retained in the processor's on-chip caches. To avoid inconsistencies between the cache contents of the host processor and main memory, the memory region used by the TEP (i.e., the data consumed in the NMP kernels) is defined as a non-cacheable region. The host processor accesses data in this region by using non-temporal instructions (e.g., *MOVNTQ*, *MOVNTPS*, and *MASKMOVQ* in x86 architecture) that bypass the cache hierarchy.

Even though the MCU performs read-only operations without any manipulation/modification of the data, the cache inconsistencies problem could occur from the SET command (e.g., *add(KEY, VALUE)*), which stores a new KEY-VALUE pair item. Since the new item is stored in the processor's on-chip cache memory, the MCU cannot access the inserted KEY data. To solve this problem, we bypassed the cache memory for the data used in SET commands by using non-temporal instructions.

- **Virtual-to-Physical address translation:** Because the virtual addresses are delivered to our TEP, it should access the stacked memory after translating from the provided virtual addresses to the physical addresses. To allow for address translation, we employ an IOMMU and TLB in the CCU. The TLB misses are served from the OS executed on the host processor, similar to an IOMMU in a conventional system. The TLB has 16 entries with 2MB page size to accelerate address translation. We use large pages for the entire system to minimize TLB misses in big data applications (Navarro et al. 2002).
- **Area overhead estimation:** For the estimation of the area overhead for the proposed architecture, we use McPAT and CACTI 6.5 tools that are assumed 40nm technology (Li et al. 2009; Muralimanohar et al. 2009). We use the logic area of an ARM Cortex-A5, which is the baseline architecture of the CCU, from the reported value in ARM CortexA5 Processor (CortexA5). The CCU has a total area of 1.25mm^2 (0.27mm^2 of logic, 0.13mm^2 of 16KB L1 instruction cache, 0.61mm^2 of stream buffer, and 0.24mm^2 of TLB). To estimate the area of the RPU, we use the area of 64 FUs and 81 switches from Farmahini-Farahani et al. (2015). The area of RPU is 1.07mm^2 (four RPUs occupy a total area of 4.31mm^2). For the MCU, the area of a comparator is estimated from Kumar et al. (2014) and it is about 0.1mm^2 . In total, the area of the proposed TEP is approximately 5.66mm^2 , and it corresponds to only 2.5% of the area of an 8Gb DRAM die area (e.g., 226mm^2 (Shevgoor et al. 2013)).
- **Discussion of the thermal issues:** Unlike a conventional host processing system, the NMP engine has more stringent power/thermal constraints. The proposed TEP consists of three computing engines; those are an in-order core, four CGRAs, and a dedicated hardware unit, and those do not consume much energy as like a high performance processor. We believe that the thermal effect of TEP on stacked DRAM devices will not be significant, because the proposed TEP has a much smaller number of processing engines than some earlier studies, which analyzed the thermal effects of NMP-based processing engines and do not have significant thermal issues (Pugsley et al. 2014; Farmahini-Farahani et al. 2015; Eckert et al. 2014).

6 EXPERIMENTAL METHODOLOGY

6.1 TEP-based Simulation Methodology

To allocate tasks for the TEP, we implement an NMP kernel recognizer that recognizes the NMP kernel, an RPU instruction generator that generates special instructions for the RPU, and an MCU instruction parser that reads information from the defined APIs. In this study, we implement these into PINTool (Luk et al. 2005).

- **NMP kernel recognizer:** First, we implement an NMP kernel recognizer in PINTool that recognizes the NMP kernels to be allocated into the TEP and can mark the starting point of NMP kernels in the list of instructions that will be executed on system. For this study, the modification of applications is required to run them on a TEP-based system that conforms to a TEP-aware programming model including indication of the NMP kernels to be assigned to the TEP, tagging the RPU pragma, and the MCU's API.
- **RPU instruction generator:** We implement an RPU instruction generator into PINTool to create the RPU configuration and insert additional instructions to communicate with the CCU. Similar to Farmahini-Farahani et al. (2015), the generator identifies a pragma directive indicating a DCO region and it partitions the DCO region into a memory sub-region for loads, stores, and address calculations, and a computation sub-region for all other instructions. In this phase, the RPU instruction generator analyzes the data dependency within the marked RPU region and creates the configuration context including computation and memory LD/ST slices. Finally, the generator creates the RPU configuration for the computation sub-region and inserts communication instructions into the memory sub-region.
- **Other instruction parser:** We implement an MCU instruction parser into PINTool that can recognize the defined MCU's API. The parser extracts several pieces of information that are required to process memory compare operations with pointer-chasing based on the MCU's API. Finally, CFO and MMO instructions are also converted to the simulation segments executable in our simulation environment.

Finally, simulation execution contexts including the RPU configuration, MCU instruction, and CCU/MMO instruction are assigned to each processing engine in our TEP, and the host processor invokes the proposed TEP according to the TEP call instruction inserted into the host processor's executable code.

6.2 Simulation Environment

To evaluate our architecture, we use MacSIM simulator (MacSIM 2016), a cycle-accurate x86-64 system simulation tool whose frontend is PINTool. We also use DRAMSim2 for DRAM timing simulation, which has a modeled HMC-based NMP stack (Rosenfeld et al. 2011). Table 3 summarizes the parameters used in this study.

- **Host processors:** Table 3(a) summarizes the architectural parameters of the host processor. We model the Intel Xeon Processor E3-1275, a high-performance host processor with 4GHz and quad-cores, in which each core executes instructions out-of-order with a 256-entry RoB (Intel Xeon Processor 2015). Each core has private L1 and L2 caches, and a last-level L3 cache is shared by all cores.
- **TEP-based system:** The TEP is the proposed architecture, and it is composed of a CCU, four RPUs, and an MCU. The architecture details for the proposed TEP configuration are summarized in Tables 3(b), 3(c), and 3(d).
- **HMC-based memory configuration:** We model HMC-based timing parameters used in Rosenfeld (2014) and Kim et al. (2015) for the stacked memory. We assume the stacked

memory has four links, 16 partitions, two banks per partition, and 32-TSVs per vault. For the simulation, we use 16 vaults organization, because it is reported to have the highest throughput with 256 total banks based on Rosenfeld (2014). We use $t_{RP} = 17$, $t_{CCD} = 6$, $t_{RCD} = 17$, $t_{CL} = 17$, $t_{WR} = 19$, $t_{RAS} = 34$ cycles for DRAM timing parameters (see Table 3(f)).

6.3 Architecture Models for the Evaluation

We compare the performance and energy consumption of the proposed TEP with other architectures. The list of evaluated NMP architecture models is as follows:

- **Host-only system:** This model is the baseline architecture that is comprised of a high-performance host processor and HMC-based memory stacks (Table 3(a)) without any accelerator. In this model, the target application runs on a host processor.
- **Host with RPU+MCU:** This model is an HMC-based system with RPUs and a MCU integrated with the host processor. In other words, this model assumes that the proposed RPUs and MCU are integrated as the on-chip accelerators into the host processor. In this model, the RPUs and MCU accelerate DCO and MMO (i.e., memory compare operation) on the proposed model, respectively, and the data to be consumed in the accelerators is delivered through a LLC of the host processor (Intel's Hybrid CPU-FPGA 2016).
- **NMP with in-order cores:** This model signifies an NMP layer with 16 single-issue in-order cores where each core has a single-level instruction and data cache. In this model, NMP kernels are executed on the NMP with an in-order cores processor, and L1 misses are sent to the DRAM array through direct TSV connections. In other words, we assume that each in-order core is integrated with a vault controller connected to TSVs similar to prior NMP studies (Kim et al. 2015; Ahn et al. 2015). The details of the NMP with in-order cores are summarized in Table 3(e).
- **NMP with RPU+MCU:** This model represents an NMP logic layer that consists of RPUs and an MCU without the CCU. Thus, instead of the CCU, the host processor is responsible for CFO and MMO type processing. To allow communication between the host processor and NMP stack, we model a simple controller including several control logics such as register files, TLB, MMU, DMA, and configuration memory (see Tables 3(c) and 3(d)).
- **TEP:** This model is the proposed TEP architecture with a CCU, RPUs, and MCU (see Tables 3(b), 3(c), and 3(d)).

6.4 Estimation of Energy Consumption

In this study, we basically approximate the energy consumption of the evaluated models by using an analytic model that has been used to investigate the energy consumption of the NMP-based system (Kim et al. 2015). Table 4 shows the major parameters for estimating the energy consumption used in this study. We assume each host core and in-order core, excluding on-chip caches, consumes 10W and 80mW of active power based on Kim et al. (2015) and Pugsley et al. (2014). Because there are four cores in the host processor and 16 cores in the "NMP with in-order cores" model, these consume 40W and 1.28W in total, respectively. The power consumed by each memory component (e.g., on-chip caches, stream buffer, and TLB) in the host processor and TEP is calculated based on the dynamic power of each component and simulation statistics. Thus, the memory access characteristics are reflected in the estimation of overall energy consumption.

In our TEP, the energy consumed by each FU and switch of the proposed RPU are based on Farmahini-Farahani et al. (2015). We assume most energy consumption of the MCU operation is by the comparators and use 22.4pJ for the comparator as reported by Krashinsky (2011). We use

Table 4. Energy Parameters for Estimating the Energy Consumption

(a) Power and energy parameters of host processor, in-order core, and DRAM	
Logic power	Host core: 10W (total 40W), In-order core: 80mW
On-chip cache (Host processor)	L1: 0.415nJ, L2: 3.24nJ, L3: 6.95nJ per access
On-chip cache (in-order core)	L1: 0.415nJ, Stream buffer: 0.16nJ per access
SRAM Leakage Power	4.05nW per cell
TSV transfer	0.078pJ per bit
Global transfer (PCB)	4.7pJ per bit
DRAM access energy	28.034nJ
DRAM background power	0.470W

(b) Energy parameters of each component in TEP		
RPU	INT-ALU	2.2pJ
	FP-ALU	7.1pJ
	INT-MUL	13.1pJ
	FP-MUL	11.3pJ
	INT-DIV	30.1pJ
	FP-DIV	27.7pJ
	Switch	1.11pJ
	Configuration Memory	0.08nJ per access
MCU	Comparator	22.4pJ per compare

CACI 6.5 to obtain the power consumption of the cache memory, RPU's configuration memory, TLB, and stream buffer. In the stacked DRAM-side, we use the energy consumed for each DRAM access based on Muralimanohar et al. (2009). In a stacked DRAM-side, we use the energy consumed for each DRAM access is estimated based on Udipi et al. (2010). PCB and TSV transfer energy values are estimated based on from Pugsley et al. (2014); Udipi et al. (2011); Woo et al. (2010), respectively. Note that the static power of the major components associated with the proposed TEP is reflected in the energy consumption according to the equations defined in Kim et al. (2015). The static power of the RPU (6.8mW) and the MCU (0.64mW) is assumed to be in proportion to the area of the CCU as investigated in Marowka et al. (2012).

In this study, the dynamic power of the proposed TEP is calculated below.

First, total dynamic energy consumption is presented as a summation of the energy consumed by the host processor, TEP, global data transfer, and NMP memory:

$$E_{TOTAL} = E_{HOST} + E_{TEP} + E_{GLOBAL_XFER} + E_{NMP_MEMORY}. \quad (1)$$

In Equation (1), the energy consumption of host processor and TEP can be expanded as follows:

$$E_{HOST} = E_{HOST_LOGIC} + E_{HOST_CACHE}, \quad (2)$$

$$E_{TEP} = E_{CCU} + E_{RPU} + E_{MCU}. \quad (3)$$

In Equation (2), we calculate the energy consumption of the host processor and cache memory based on the equations defined in Kim et al. (2015). The energy consumption of the proposed TEP including CCU, RPU, and MCU can be calculated as follows:

$$E_{CCU} = E_{CCU_LOGIC} + E_{CACHE} + E_{STREAM_BUFFER}, \quad (4)$$

$$E_{RPU} = \sum_{t=1}^N (E_{FU} + E_{SWITCH}) + E_{CONFIGURATION_MEMORY}, \quad (5)$$

$$E_{MCU} = \sum_{t=1}^N (E_{COMPARATOR}). \quad (6)$$

The energy consumed by the stacked memory including the data transfer energy by TSVs can be calculated by Equation (7):

$$E_{NMP_MEMORY} = (E_{ACCESS} + E_{TSV_XFER}) \times N_{ACCESS}. \quad (7)$$

Fig. 5. Performance evaluation result of target applications.

Finally, the global transfer energy is calculated by the energy consumed due to the off-chip data transfer through PCB-based bus:

$$E_{GLOBAL_XFER} = E_{PCB_XFER} \times N_{DRAM_ACCESS}. \quad (8)$$

7 EVALUATION RESULTS

7.1 Performance Results

Figure 5(a) shows the speedup of the NMP kernels extracted from the target applications. In the evaluation result, our TEP achieves about 5.3 times and 2 times speedup on average compared to the “Host-only system” and “NMP with in-order cores” models, respectively. Figure 5(b) shows the overall speedup results of the target applications. Our TEP achieves the highest performance improvement among the evaluated architecture models. On average, the proposed TEP delivers 3.4 times and 1.9 times better performance than the “Host-only system” and “NMP with in-order cores” model, respectively.

The following four major observations can be made based on the performance results.

- (1) First, we can identify that high and mid MPKI applications (e.g., Memcached, Liblinear, LDA, 3D-LiDAR, HPCCG, Stemmer, RBM, SRR, and ME) achieve better performance on every NMP-based model compared to the host-processing models (e.g., “Host-only system” and “Host with RPU+MCU” model). Thus, we infer that a major source of high MPKI of the application resulted from the extracted NMP kernels and performance improvement can be achieved by processing the NMP kernels locally by the NMP-based engine. On the other hand, low MPKI applications (e.g., miniMD and k-means) do not experience significant performance improvement on conventional NMP models (e.g., the “NMP with in-order cores” and “NMP with RPU+MCU” models) compared to the “Host with RPU+MCU” model. Therefore, an LLC MPKI application can be utilized as one of the major metrics to determine whether the application should be allocated into the NMP stack.
- (2) We can identify a distinct performance effect in accordance with the contribution of the three operation types classified in this study (i.e., DCO, MMO, and CFO), based on a comparison of the performance of the “NMP with RPU+MCU” and “NMP with in-order cores” models. In Figure 5(a), applications that have a high contribution of the DCO type (e.g.,

LDA, 3D-LiDAR, HPCCG, Liblinear, RBM, miniMD, and k-means) achieve an outstanding speedup on the “NMP with RPU+MCU” model compared to the “NMP with in-order cores” model.

On the contrary, some applications with a high contribution of MMO or CFO type (e.g., Stemmer, SRR, and ME) experience higher performance on “NMP with in-order cores” than the “NMP with RPU+MCU” model. This result means that “NMP with in-order cores” can handle the MMO and CFO types efficiently, whereas the “NMP with RPU+MCU” model suffers from a significant overhead when processing them on the host processor, because it cannot handle those types. In the case of the Memcached application, a remarkable performance improvement can be achieved from the acceleration of MCU (about 53% improvement with the “NMP with RPU+MCU” over the “NMP with CCU+RPU”), because it is designed for accelerating the memory compare operation, which is one of the major performance bottlenecks of the application. This result shows that the “NMP with in-order cores” and “NMP with RPU+MCU” models can achieve significant performance improvements for applications with suitable operations for each structure. However, performance improvement cannot be obtained for all applications only with each structure as well.

- (3) Our proposed TEP provides an outstanding speedup for all NMP kernels. As shown in Figure 5(a), our TEP achieves about 5.3 times and 2 times on average greater speedup for the NMP kernels than the “Host-only system” and “NMP with in-order cores” models, respectively. In particular, the proposed TEP provides better performance than the “NMP with RPU+MCU” model in LDA and 3D-LiDAR applications with small contributions of MMO and CFO types in total execution time. As a result, Figure 5 shows that the necessity of a heterogeneous NMP engine consisting of the specialized accelerators and a flexible computing engine capable of handling various operations and accelerating certain kernels to effectively utilize the benefits of NMP-based systems.
- (4) Finally, we realized that the data shared between NMP and non-NMP kernels, which sends/receives parameters among the kernels (i.e., functions) within a program, can influence the overall performance of an application significantly (see Figure 5(b)). To analyze the side-effect on the overall performance of applications, we calculate the expected speedup, when the kernel speedup that resulted from the proposed TEP (Figure 5(a)) is reflected in the total NMP kernel contributions, without any performance loss due to communication by the shared data between NMP and non-NMP kernels. For this study, we use Amdahl’s law as shown in Equation (9) (Gustafson 1988). In other words, a legend of *Expected speedup by TEP* in Figure 5(b) means the maximum performance that can be obtained by the system equipped with the proposed TEP:

$$\text{Expected speedup} = \frac{l}{((1 - C) + C/S)},$$

$C = \text{the percentage of execution time of NMP kernels,}$

$S = \text{speedup of the kernel by TEP.}$ (9)

In Figure 5(b), HPCCG, Memcached, Stemmer, RBM, and miniMD applications, which share a large amount of the data between the NMP and non-NMP kernels, experience lower performance compared to the expected performance. On the contrary, LDA, 3D-LiDAR, Liblinear, SRR, ME, and K-means applications, which consume most of the data within the extracted NMP kernels, achieve performance improvements almost similar to the expected speedup by enjoying a relatively low overhead. Therefore, in the case of

Fig. 6. Evaluation result for data traffic and energy consumption.

an application program with a very low execution time of the non-NMP kernel, completely offloading the application to the NMP logic layer can be considered as a method for minimizing the communication overhead due to data sharing between the NMP and the non-NMP kernels.

7.2 Data Traffic

Figure 6(a) shows the overall data traffic from the stacked memory to the on-chip memory of the host processor on the evaluated systems. In Figure 6(a), data traffic (left y-axis) is normalized to that of "Host-only system" model. The proposed TEP can reduce 52% of the overall data traffic on average compared to the "Host-only system" model. In particular, 3D-LiDAR, Liblinear, SRR, ME, and K-means applications, which consume the most data within the NMP kernels, experience a significant reduction of data traffic of about 52% on average compared to the "Host-only system" model. However, these applications suffer from higher data traffic on the "NMP with RPU+MCU" model than the proposed architecture, because they should communicate frequently with the host processor to handle the data included in the MMO and CFO types within the NMP kernels. Thus, this result shows that the necessity of a programmable core for the NMP engine to reduce the communication overhead of NMP-based systems and improve the performance.

Figure 6(a), on the other hand, shows that the "NMP with in-order cores" model is also as effective as the proposed TEP in reducing the data traffic between the host processor and the NMP logic layer by processing all NMP kernels extracted from the target applications. Thus, in terms of the data traffic, Figure 6(a) implies that the kernel coverage capability of the NMP architecture, which can handle the various tasks contained in the application, is important to reduce data traffic on NMP systems.

Furthermore, we measure the data traffic caused by shared data between the NMP and non-NMP kernels on our TEP architecture. The shared data proportion of each application is presented as a line chart in right y-axis in Figure 6(a). As shown in Figure 6(a), we can observe that HPCCG, Stemmer, RBM, and miniMD applications, which showed a large performance gap from the

expected performance (Figure 5(b)), share about 22% of the data between the NMP and non-NMP kernels. This result means that a large amount of data is consumed by the host processor for executing non-NMP kernels even though it has a very small contribution to the total execution time. In addition, Figure 6(a) shows that applications, such as miniMD and K-means application, have low MPKI and experience high traffic overhead on the NMP models evaluated in this article. This result shows that the data sharing between the kernels of the applications has a larger impact on the traffic overhead on the applications with low MPKI, so these applications would not be suitable for the NMP model.

7.3 Energy Consumption

Figure 6(b) shows the energy consumption of each application on the evaluated models. In Figure 6(b), each bar details the energy consumed by each architectural component. DRAM access refers to the energy consumed by DRAM devices, on-chip transfers include consumed energy for on-chip cache memories and PCB transfers, and execution and scheduling involves the consumption of the host processor and TEP engine. The proposed TEP provides a significant energy saving of about 33% compared to the “Host-only system” model on average. Our TEP provides significant energy saving on the on-chip data transfer by processing the kernel operations locally within the NMP logic layer consisting of an energy-efficient core and two accelerators.

Figure 6(b) shows that the proportion of the operation type classified in this study can have a significant effect on the energy consumption as shown in the performance results. First, applications with a dominant contribution of the DCO type can experience high-energy savings in models with RPU (i.e., “Host with RPU+MCU,” “NMP with RPU+MCU,” and “TEP”) that process the DCO type in parallel. On the contrary, applications with a dominant contribution of the MMO or CFO type experience high energy reduction including in the programmable core (i.e., “NMP with in-order cores” and “TEP”). In addition, we can confirm that an NMP logic with a restricted processing ability is unlikely to offer high-energy saving potential compared to the versatile NMP models based on the energy consumption of the “NMP with RPU+MCU” model that cannot handle various operations. Finally, Figure 6(b) shows that the applications sharing a large amount of data between the NMP and non-NMP kernels experience higher energy consumption on the on-chip data transfer.

8 RELATED WORK

In the 1990s, various studies for processing-in-memory (PiM) were performed as early forms of the NMP model. Prior works related to PiM tried to overcome the memory wall problem between the processor and memory by integrating the data processing logics into a single die (Mai et al. 2000; Draper et al. 2002). Although they delivered several outstanding results, the PiM model has been difficult to employ in the industry due to the high cost to make a single chip and lack of killer applications. This section briefly summarizes the prior studies based on the classification of the architecture models that were considered as a primary architecture in an NMP logic layer.

8.1 NMP Engine with Fully Programmable Cores

An NMP engine constructed with fully-programmable cores has been investigated in various NMP-related studies. Ahn et al. (2015) introduced a near-data processing accelerator consisting of a number of in-order cores to process a large-scale graph algorithm, and proposed a low-cost message passing mechanism with specialized prefetchers to support efficient communication between vault domains. Kim et al. (2015) designed an analytical model to estimate the energy consumed in an NMP system consisting of low power in-order cores. Pugsley et al. (2014) and Islam et al. (2014) employed a number of in-order cores as a near-data processing engine to offload

the MapReduce workloads. Hong et al. (2016) proposed an in-memory accelerator constructed with a number of in-order cores to minimize the memory bottleneck caused by pointer chasing operations. In Yitbarek et al. (2016), Intel Atom processors are employed in an NMP logic layer to accelerate the data-intensive kernels, and Scrbak et al. (2015) investigated the number of cores, cache size, and core frequency, in terms of performance and energy efficiency upon the MapReduce workload, to derive the design parameters for an NMP engine with in-order cores.

In previous studies, many NMP engines with fully programmable cores can perform a variety of tasks and are easily adaptable to a variety of applications. However, it is difficult to expect high performance and low power consumption with fully programmable cores over NMP engines designed for specific tasks.

8.2 NMP Engine with Reconfigurable Logics

Reconfigurable architecture has been studied in various domains like multimedia, signal processing, and pattern matching to exploit the spatial parallelism from a large loop work. This research has focused on two categories—CGRA that computes with word granularity (Goldstein et al. 1999; Govindaraju et al. 2012; Huang et al. 2013), and FPGA that computes with bit granularity (Hauser et al. 1997; Hartenstein et al. 2001; Mishra et al. 2006). The reconfigurable architecture showed outstanding benefits in terms of performance and energy efficiency compared to a fully-programmable architecture for many data-parallel applications.

Recently, reconfigurable architectures have been reconsidered in various pieces of NMP research as an efficient near-memory accelerator capable of providing benefits in terms of performance and energy efficiency. Farmahini-Farahani et al. (2015) implemented an acceleration engine with multiple CGRAs into the NMP logic layer to accelerate a large-scale loop body of big data applications. Gao et al. (2016) proposed a heterogeneous reconfigurable logic as near-data processing units to improve power and area efficiency. GPU-based NMP engines have been also studied to accelerate the parallel workloads involved in big data applications. Zhang et al. (2014) proposed a GPGPU execution units with 3D-stacked DRAM for in-memory computing and Hsieh et al. (2016) proposed a compiler-based mechanism to automatically identify the code region to offload and map data to a GPU-based NMP engine.

The reconfigurable/GPU-based NMP engines have resulted in improved performance by accelerating the data-parallel workloads. However, they can experience a high communication overhead between the host processor and NMP logic layer.

8.3 NMP Engine with Dedicated HW Logics

In many NMP-related studies, an architecture with a fixed-function hardware logic has been investigated for the various workloads. Hong et al. (2016) introduced a dedicated hardware logic to accelerate a pointer-chasing operation on a linked-list data structure, and Lee et al. (2015) proposed hardware to minimize the overhead of atomic operations for machine-learning workloads. Seshadri et al. (2013) proposed a hardware logic that supports the copy and initialization operation of the bulk data on the stacked DRAM. Akin et al. (2014) and Akin et al. (2015) proposed a memory accelerator to handle the data reorganization operations, such as shuffle, pack/unpack, and swap with mathematical framework. Thanh-Hoang et al. (2016) introduced dedicated hardware to accelerate the data movement between multiple accelerators implemented in the stacked memory. The NMP engine, which is based on dedicated hardware logic, provides significantly improved performance for a target kernel. However, it can experience high communication overheads with the host processor due to its limited functionality, and is difficult to adapt to other applications.

In summary, most prior studies used a single type of computing engine for the NMP engine. NMP engines, which are designed with one type of computing engine such as dedicated hardware

or fully programmable logics, can experience high communication overheads and low performance for certain tasks. Unlike prior works, this study proposed a heterogeneous NMP architecture with three different types of computing engines, which can accelerate and execute three types of kernel operations extracted from various applications.

9 CONCLUSION

With the advent of 3D memory stacking technology, various approaches have been explored for near-memory processing that can accelerate and offload certain kernel operations included in a variety of big data application areas. While most previous works have delivered improved performance and energy efficiency for some specific kernel operations with a specific type of NMP engine, this study shows that an NMP engine with a single-type of processing engine can have some drawbacks as follows. NMP engines with dedicated hardware logic designed for a specific task can provide fast processing and low power consumption but can suffer a relatively high communication overhead with the host processor compared to NMP engines with programmable cores. On the other hand, while the NMP engines with energy-efficient in-order cores that have been adopted in previous studies can be applied to a variety of applications, it is difficult to expect high performance and low power consumption over dedicated hardware specialized for specific kernels.

In this study, we propose TEP, a near-memory processor with three types of processing engines to execute and accelerate various kernel operations included in emerging applications. To design the TEP, we categorize three different types of kernel operations—CFO, MMO, and DCO—which are extracted from various applications, and then analyze the architectural features to process these three categorized types efficiently. The proposed heterogeneous NMP architecture, TEP, has been designed to have three computing engines; a CCU designed with an energy-efficient in-order core for CFO and MMO types, an RPU designed with CGRAs for DCO type, and a (or a set of) dedicated hardware accelerator(s). A representative hardware accelerator engine, called a MCU for memory compare operations, is designed as an example of a set of dedicated hardware accelerators.

In our evaluation results, the proposed TEP provides about 3.4 times increased speed and about 52% reduction of data traffic (33% energy saving) on average compared to the non-NMP system. Through this study, we have presented the effectiveness of a heterogeneous NMP engine that can handle the diverse kernel operations included in applications by evaluating various NMP-based models. Furthermore, we have observed how the data shared between NMP and non-NMP kernels can interfere with the benefits of the NMP-based system. A heterogeneous NMP architecture with a new combination of different architectural models such as GPU, SIMD, and FPGA, and an NMP architecture that completely offloads applications to minimize communication overhead will be studied in our future work.

REFERENCES

- J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. 2015 A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. ACM, New York, NY, 105–117. DOI: <https://doi.org/10.1145/2749469.2750386>
- J. Ahn, S. Yoo, O. Mutlu, and K. Choi. 2015. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. ACM, New York, NY, 336–348. DOI: <https://doi.org/10.1145/2749469.2750385>
- B. Akin, J. Hoe, and F. Franchetti. 2014. HAMLeT: Hardware accelerated memory layout transform within 3D-stacked DRAM. In *Proceedings of the 2014 High Performance Extreme Computing Conference (HPEC'14)*. IEEE Computer Society, Washington, DC, 241–252. DOI: <https://doi.org/10.1109/HPEC.2014.7040954>
- B. Akin, F. Franchetti, and J. Hoe. 2015. Data reorganization in memory using 3D-stacked DRAM. *SIGARCH Comput. Arch. News* 43, 3 (June 2015), 131–143. DOI: <https://doi.org/10.1145/2872887.2750397>

- R. Balasubramonian, J. Chang, T. Manning, J. Moreno, R. Murphy, R. Nair, and S. Swanson. 2014. Near-data processing: Insights from a MICRO-46 workshop. In *IEEE Micro*. IEEE, Washington, DC, 36–42. DOI: <http://dx.doi.org/10.1109/MM.2014.55>
- Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. 1999. PipeRench: A co/processor for streaming multimedia acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*. IEEE Computer Society, Washington, DC, 28–39. DOI: <http://dx.doi.org/10.1145/300979.300982>
- ARM CortexA5 Processor. Retrieved from www.arm.com/products/processors/cortex-a/cortex-a5.php.
- J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. Kang, I. Kim, and G. Daglikoca. 2002. The architecture of the DIVA processing-in-memory chip. In *Proceedings of the 16th International Conference on Supercomputing (ICS'02)*. ACM, New York, NY, 14–25. DOI: <http://dx.doi.org/10.1145/514191.514197>
- Y. Eckert, N. Jayasena, and G. Loh. 2014. Thermal feasibility of die-stacked processing in memory. In *Proceedings of the 2nd Workshop on Near-Data Processing (Held in conjunction with MICRO'14)*.
- A. Farmahini-Farahani, J. Ahn, K. Morrow, and N. Kim. 2015. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *Proceedings of the IEEE Symposium on High Performance Computer Architecture (HPCA'15)*. IEEE, Washington, DC, 283–295. DOI: <https://doi.org/10.1109/HPCA.2015.7056040>
- Z. Guz, M. Awasthi, V. Balakrishnan, M. Ghosh, A. Shayesteh, and T. Suri. 2015. Realtime-analytics is the killer application for processing-in-memory. In *Proceedings of the 2nd Workshop on Near-Data Processing Conjoined with the 47th IEEE/ACM International Symposium on Microarchitecture (WoNDP'14)*. ACM New York, NY.
- S. Graham, P. Kessler, and M. Mckusick. 1982. gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction (SIGPLAN'82)*. ACM, New York, NY, 120–126.
- V. Govindaraju, C. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. 2012. DySER: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*. 32, 5, 38–51. DOI: <http://dx.doi.org/10.1109/MM.2012.51>
- M. Gao, C. Delimitrou, D. Niu, K. Malladi, H. Zheng, B. Brennan, and C. Kozyrakis. 2016. DRAF: A low-power DRAM-based reconfigurable acceleration fabric. *SIGARCH Comput. Arch. News*. 44, 3, 506–518. DOI: <https://doi.org/10.1145/3007787.3001191>
- M. Gao and C. Kozyrakis. 2016. HRL: Efficient and flexible reconfigurable logic for near-data processing. In *Proceedings of 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. IEEE, Washington, DC, 126–137. DOI: [10.1109/HPCA.2016.7446059](https://doi.org/10.1109/HPCA.2016.7446059)
- T. Granlund. 2014. Instruction latencies and throughput for AMD and Intel x86 Processors. Retrieved from <https://gmplib.org/tege/x86-timing.pdf>.
- J. Gustafson. 1988. Reevaluating Amdahl's law. In *Proceedings Commun. ACM* 31, 5, 532–533. DOI: <http://dx.doi.org/10.1145/42411.42415>
- M. Heroux, D. Doerfler, P. Crozier, J. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. Keiter, H. Thornquist, and R. Numrich. 2009. Improving performance via mini-applications. In *SANDIA REPORT*. Retrieved from <https://mantevo.org/MantevoOverview.pdf>.
- T. Hauswald, M. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars. 2014. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, NY, 223–238. DOI: <http://dx.doi.org/10.1145/2694344.2694347>
- Y. Huang, P. Ienne, O. Temam, Y. Chen, and C. Wu. 2013. Elastic CGRAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'13)*. ACM, New York, NY, 171–180. DOI: <http://dx.doi.org/10.1145/2435264.2435296>
- R. Hartenstein. 2001. Coarse grain reconfigurable architecture (embedded tutorial). In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference (ASP-DAC'01)*. ACM, New York, NY, 564–570. DOI: <http://dx.doi.org/10.1145/370155.370535>
- B. Hong, G. Kim, J. Ahn, Y. Kwon, H. Kim, and J. Kim. 2016. Accelerating linked-list traversal through near-data processing. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. ACM, New York, NY, 113–124. DOI: <https://doi.org/10.1145/2967938.2967958>
- J. Hauser and J. Wawrzyniek. 1997. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM'97)*. IEEE Computer Society, Washington, DC, 12–21. DOI: <http://dx.doi.org/10.1109/FPGA.1997.624600>
- K. Hsieh, E. Ebrahim, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. Keckler. 2016. Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems. In *Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture*. ACM, New York, NY, 18–22. DOI: <https://doi.org/10.1109/ISCA.2016.27>

- Intel's Hybrid CPU-FPGA. Retrieved from <http://www.embeddedintel.com/commentary.php?article=2143>.
- M. Islam, M. Scrbak, K. M. Kavi, M. Ignatowski, and N. Jayasena. 2014. Improving node-level mapreduce performance using processing-in-memory technologies. In *Proceedings of the Euro-Par Workshop 2014*. DOI : http://doi.org/10.1007/978-3-319-14313-2_36
- J. Jeddeloh and B. Keeth. 2012. Hybrid memory cube new DRAM architecture increases density and performance. In *Proceedings of the IEEE International Symposium on VLSI Technology (VLSIT'12)*. IEEE, Los Alamitos, CA, 87–88. DOI : <https://doi.org/10.1109/VLSIT.2012.6242474>
- N. Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'90)*. ACM, New York, NY, 364–373. DOI : <http://dx.doi.org/10.1145/325164.325162>
- H. Kim, H. Kim, S. Yalamanchili, and Arun Rodrigues. 2015. Understanding energy aspect of processing near memory for HPC workloads. In *Proceedings of the International Symposium on Memory Systems (MEMSYS'15)*. ACM Press, New York, NY, 276–282. DOI : <https://doi.org/10.1145/2818950.2818985>
- R. Kumar and S. Hirekhan. 2014. Design of 8-bit comparator using 45nm CMOS technology. *Int. J. Modern Eng. Res. (IJMER)* 4, 6, 59–64.
- D. Keen. 2012. FlexRAM: Toward an advanced Intelligent Memory system. In *Proceedings of the 2012 IEEE 30th International Conference on Computer Design (ICCD'12)*. IEEE Computer Society, Washington, DC, 5–14. DOI : <http://dx.doi.org/10.1109/ICCD.2012.6378608>
- R. Krashinsky. 2011. *Microprocessor Energy Characterization and Optimization through Fast, Accurate, and Flexible Simulation*. Master's thesis. Electrical Engineering and Computer Science, MIT.
- KCacheGrind. 2013. Homepage. Retrieved from <http://kcachegrind.sourceforge.net/html/Home.html>.
- H. Loh, N. Jayasena, M. H. Oski, M. Nutter, D. Roberts, M. Meswani, D. P. Zhang, and M. Ignatowski. 2013. A processing-in-memory taxonomy and a case for studying fixed-function PIM. In *Proceedings of the Workshop on Near-Data Processing (WoNDP'13)*.
- C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *SIGPLAN Not.* 40, 6, 190–200. DOI : <http://dx.doi.org/10.1109/eScience.2008.59>
- S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. ACM, New York, NY, 469–480. DOI : <http://dx.doi.org/10.1145/1669112.1669172>
- J. Lee, J. Sim, and H. Kim. 2015. BSSync: Processing near memory for machine learning workloads with bounded staleness consistency models. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT'15)*. IEEE Computer Society, Washington, DC, 241–252. DOI : <https://doi.org/10.1109/PACT.2015.42>
- N. Muralimanohar, R. Balasubramonian, and N. Jouppi. 2009. CACTI 6.0: A tool to model large caches. Technical Report HPL-2009-85. HP Labs.
- K. Mai, T. Paaske, N. Jayasena, R. Ho, William J. Dally, and M. Horowitz. 2000. Smart memories: A modular reconfigurable architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)* ACM, New York, NY, 161–171. DOI : <http://dx.doi.org/10.1145/339647.339673>
- M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. Goldstein, and M. Budiu. 2006. Tartan: Evaluating spatial computation for whole program execution. In *SIGOPS Oper. Syst. Rev.* 40, 5, 163–174. DOI : <http://dx.doi.org/10.1145/1168917.1168878>
- A. Marowka. 2012. Extending Amdahl's law for heterogeneous computing. In *Proceedings of the IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA'12)*. IEEE Computer Society, Washington, DC, 10–13. DOI : <https://doi.org/10.1109/ISPA.2012.47>
- Memcached V. 1.4.32. Retrieved from <http://memcached.org/>.
- J. Navarro, S. Iyer, P. Druschel, and A. Cox. 2002. Practical, transparent operating system support for superpages. In *SIGOPS Oper. Syst. Rev.* 36, SI, 89–104. DOI : <http://dx.doi.org/10.1145/844128.844138>
- G. Pandey, J. R. McBride, S. Savarese, and R. Eustice. 2012. Automatic targetless extrinsic calibration of a 3d lidar and camera by maximizing mutual information. In *Proceedings of the AAAI National Conference on Artificial Intelligence*. IEEE Computer Society, Washington, DC, 2053–2059.
- S. Pugsley, J. Jests, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li. 2014. NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads. In *Proceedings of the IEEE Symposium on Performance Analysis of Systems and Software (ISPASS'14)*. IEEE, Washington, DC, 190–200. DOI : <https://doi.org/10.1109/ISPASS.2014.6844483>
- P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. DRAMSim2: A cycle accurate memory system simulator. *IEEE Comput. Arch. Lett.* 10, 1, 16–19. DOI : <http://dx.doi.org/10.1109/L-CA.2011.4>

- P. Rosenfeld. 2014. *Performance Exploration of the Hybrid Memory Cube*. Ph.D. dissertation, University of Maryland, College Park.
- V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungrinur, G. Pekhimenko, Y. Luo, O. Mutlu, P. Gibbons, M. Kozuch, and T. Mowry. 2013. RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*. ACM, New York, NY, 185–197. DOI : <http://dx.doi.org/10.1145/2540708.2540725>
- R. Saunders. 2011. *A Study in Memcmp*. Python dev list.
- M. Shevgoor, J. Kim, N. Chatterjee, R. Balasubramonian, A. Davis, and A. Udipi. 2013. Quantifying the relationship between the power delivery network and architectural policies in a 3D-stacked memory device. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*. ACM, New York, NY, 198–209. DOI : <http://dx.doi.org/10.1145/2540708.2540726>
- MacSIM simulator. Retrieved from <http://comparch.gatech.edu/hparch/macsim.html>.
- M. Scrbak, M. Islam, Kr. M. Kavi, M. Ignatowski, and N. Jayasena. 2015. Processing-in-memory: Exploring the design space. In *Proceedings of Architecture of Computing Systems (ARCS'15)*. 43–54. DOI : https://doi.org/10.1007/978-3-319-16086-3_4
- T. Thanh-Hoang, A. Shambayati, and A. Chien. 2016. A data layout transformation (DLT) accelerator: Architectural support for data movement optimization in accelerated-centric heterogeneous systems. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe (DATE'16)*. EDA Consortium, San Jose, CA, 1489–1492.
- S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. Taylor. 2014. CortexSuite: A synthetic brain benchmark suite. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'14)*. IEEE Computer Society, Washington, DC, 76–79. DOI : <http://dx.doi.org/10.1109/IISWC.2014.6983043>
- J. Tripp, J. Frigo, and P. Graham. 2007. A survey of multi-core coarse-grained reconfigurable arrays for embedded applications. In *Proceedings of the 11th Annual Workshop on High Performance Embedded Computing*. Retrieved from https://www.ll.mit.edu/HPEC/agendas/proc07/Day3/08_Tripp_Pres.pdf.
- A. Udipi, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. Jouppi. 2011. Combining memory and a controller with photonics through 3D-stacking to enable scalable and energy-efficient systems. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. ACM, New York, NY, 425–436. DOI : <http://dx.doi.org/10.1145/2000064.2000115>
- A. Udipi, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. Jouppi. 2010. Rethinking DRAM design and organization for energy-constrained multi-cores. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 175–186. DOI : <http://dx.doi.org/10.1145/1815961.1815983>
- D. Woo, N. Seong, D. Lewis, and H. Lee. 2010. An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth. In *Proceedings of the IEEE 16th International Symposium on High Performance Computer Architecture (HPCA'10)*. ACM, New York, NY, 425–436. DOI : <https://doi.org/10.1109/HPCA.2010.5416628>
- Intel Xeon Processor. Retrieved from <https://goo.gl/X7XyJw>.
- A. Yazdanbakhsh, D. Mahajan, P. Lotfi-Kamran, and H. Esmailzadeh. 2016. AxBench: A multiplatform benchmark suite for approximate computing. *IEEE . Design Test*. 34, 2, 60–68. DOI : <https://doi.org/10.1109/MDAT.2016.2630270>
- S. Yitbarek, T. Yang, R. Das, and T. Austin. 2016. Exploring specialized near-memory processing for data intensive operations. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe. (DATE'16)*. EDA Consortium, San Jose, CA, 1449–1452.
- Q. Zhu, B. Akin, H. E. Sumbul, F. Sadi, J. Hoe, L. Pileggi, and F. Franchetti. 2013. A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing. In *Proceedings of the IEEE International Conference on 3D System Integration (3DIC'13)*. IEEE, Washington, DC, 1–8. DOI : <https://doi.org/10.1109/3DIC.2013.6702348>
- D. Zhang, N. Jayasena, A. Lyashevsky, J. Greathouse, L. Xu, and M. Ignatowski. 2015. TOP-PIM: Throughput-oriented programmable processing in memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC'14)*. ACM, New York, NY, 85–98. DOI : <http://doi.acm.org/10.1145/2600212.2600213>

Received April 2017; revised October 2017; accepted October 2017