Towards Memory-Efficient Allocation of CNNs on Processing-in-Memory Architecture

Yi Wang, Weixuan Chen, Jing Yang, and Tao Li, *Fellow, IEEE*

Abstract—Convolutional neural networks (CNNs) have been successfully applied in artificial intelligent systems to perform sensory processing, sequence learning, and image processing. In contrast to conventional computing-centric applications, CNNs are known to be both computationally and memory intensive. The computational and memory resources of CNN applications are mixed together in the network weights. This incurs a significant amount of data movement, especially for high-dimensional convolutions. The emerging Processing-in-Memory (PIM) alleviates this memory bottleneck by integrating both processing elements and memory into a 3D-stacked architecture. Although this architecture can offer fast near-data processing to reduce data movement, memory is still a limiting factor of the entire system. We observe that an unsolved key challenge is how to efficiently allocate convolutions to 3D-stacked PIM to combine the advantages of both neural and computational processing. This paper presents *MemoNet*, a memory-efficient data allocation strategy for convolutional neural *net*works on 3D PIM architecture. MemoNet offers fine-grained parallelism that can fully exploit the computational power of PIM architecture. The objective is to capture the characteristics of neural network applications and perfectly match the underlining hardware resources provided by PIM, resulting in a hardware-independent design to transparently allocate data. We formulate the target problem as a dynamic programming model and present an optimal solution. To demonstrate the viability of the proposed MemoNet, we conduct a set of experiments using a variety of realistic convolutional neural network applications. The extensive evaluations show that, MemoNet can significantly improve the performance and the cache utilization compared to representative schemes.

Index Terms—Processing-in-memory, neuromorphic computing, non-volatile memory, scheduling, data allocation, parallel computing

1 Introduction

Convolutional neural networks (CNNs) are widely applied in many deep learning application domains such as image processing, speech recognition, and natural language processing. Different from traditional computing-centric applications, CNNs require intensive access to both computational and memory resources. Each convolution layer of a CNN requires iterative use of the available processing engines. This produces a large amount of intermediate data that will be gradually streamed out to memory as the computation progresses. The intermediate data will be streamed back to the same processing engine upon the completion of each layer [1]. The data movement between computation and memory directly impacts performance and makes memory become the major bottleneck of the system.

Recent 3D-stacked processing-in-memory (PIM) architecture offers a promising solution to solve the data movement challenge. State-of-the-art PIM architecture stacks multiple layers of embedded DRAM (eDRAM) and allows processing engine close to the data in memory. This three-dimensional structure moves computation inside or near memory, reducing the expensive access latency for load and store operations. Although PIM architecture alleviates the issue for the data movement, it has little reconfigurable or adaptive capabilities. The hardware resources could not proactively adapt to the workload of CNN applications. This problem is further exacerbated as the computational and memory resources of CNN applications are mixed together in the network weights and neuron activity [2]. The overall system performance is often degraded due to insufficient memory to store the intermediate processing results (e.g., partial sums). How to efficiently allocate convolutions to 3D-stacked PIM and fully utilize the resource-constrained global memory become critical issues.

In order to solve this problem, many existing hardware-based approaches adopt ASIC (Application Specific Integrated Circuit) or FPGA (Field-Programmable Gate Array) to accelerate the processing of CNNs. Although they can offer application-specific design to fully utilize the computational capability, they cannot provide a general framework that can speed up the processing for *any* CNN application on *any* hardware platform. We believe that a software-based solution is a promising direction, as it can provide hardware-independent support for various types of neural network applications. The applications do not have to be trained for a specific hardware platform. On the other hand, the hardware platform also does not have to know the unique properties of the application. It is the

- *Y. Wang and W. Chen are with the College of Computer Science and Software Engineering, Shenzhen University, Shenzhen 518060, China. E-mail: yiwang@szu.edu.cn, 2014150270@email.szu.edu.cn.*
- *J. Yang is with Experimental and Innovation Practice Center, Harbin Institute of Technology, Shenzhen 518055, China. E-mail: yangjing2016@hit.edu.cn.*
- *T. Li is with Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611. E-mail: taoli@ece.ufl.edu.*

Manuscript received 30 Aug. 2017; accepted 25 Dec. 2017. Date of publication 9 Jan. 2018; date of current version 11 May 2018.
(Corresponding author: Yi Wang.)
Recommended for acceptance by M. Kandemir.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2018.2791440

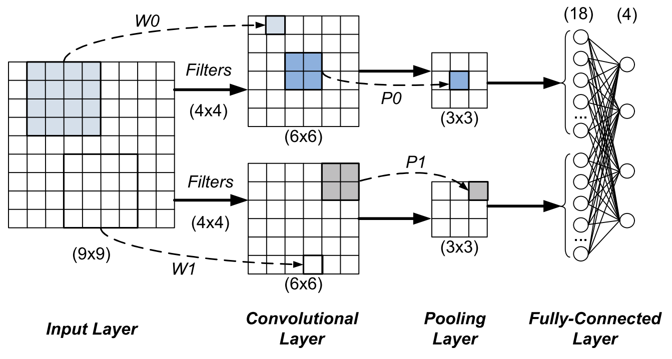


Fig. 1. The standard procedure for CNN with multiple stacked layers.

responsibility of software components (e.g., compiler, device driver) to handle the data allocation of applications onto existing hardware resources.

This paper presents *MemoNet*, a memory-efficient data allocation strategy for convolutional neural networks on PIM architecture. We first conduct a theoretical analysis and show the impact of adjusting mapping of parallelism for hardware resources. This analysis captures the characteristics of neural network applications and determines the memory usage for intermediate processing results. Based on this analysis, MemoNet jointly optimizes the allocation of convolutions and the intermediate processing results. It aims to fully utilize the valuable memory space and reduce the penalty for data movement. The target problem is further transformed and modelled as a dynamic programming problem to assign convolutions to the optimal processing engines.

To demonstrate the viability of the proposed technique, we use a variety of realistic convolutional neural network applications running on Caffe [3], an open-source deep learning framework. We compare MemoNet with the baseline scheme [4] in terms of throughput and off-chip fetching penalty. We also compare MemoNet with our previous study Memolution [5] to evaluate the scalability. Our extensive evaluations show that, MemoNet can significantly improve the throughput (by 22.78 percent on average) and effectively utilize the PIM architecture with the negligible space and timing overhead.

The main contributions of this paper are:

- A hardware-independent data allocation strategy is proposed to enable the transparent mapping for CNN applications onto 3D-stacked PIM architecture.
- A dynamic programming model is proposed to obtain the optimal data allocation for memory.
- As a proof of concept, we compare the proposed technique with representative schemes using a set of real-world CNN applications.

The rest of this paper is organized as follows. Section 2 introduces models and concepts used in this paper. Section 3 presents our proposed technique in detail. Section 4 presents experimental results. Section 5 discusses the related work. Section 6 concludes the paper and discusses future work.

2 Background and Motivation

2.1 Convolutional Neural Networks

CNNs consist of a combination of multiple layers, which can be broadly categorized as convolutional layers, activation layers, pooling layers, and fully-connected layers. Convolutional layers alternate with polling layers to achieve hierarchical neural networks, and convolutional layers dominate the computation time of a CNN. Fig. 1 illustrates the standard processing procedure for CNN. The processing of CNN starts with the input layer (image processing layer). The input layer is an optional pre-processing layer that provides the initial image and defines the training parameters of filters. Given an input feature map (9×9 neurons) in the first layer, the convolutional layer performs an inner product calculation, where pairwise multiplications are performed among the input elements (or neurons) and the filter weights (or synapses).

A convolutional layer applies filters on the input feature maps to extract embedded visual characteristics and generate the output feature maps. Different filters are applied to each feature map with different weights ($W0$ and $W1$). In this example, the convolution kernel with the window size of 4×4 can produce two 6×6 output feature maps.

The pooling layer typically involves a simple maximum or average operation ($P0$ and $P1$) on a small set of input numbers. For example, the max-pooling in Fig. 1 will select the maximum value in each 2×2 elements (i.e., four values). The output of the pooling layer is two 3×3 feature maps. The pooling operation enables position invariance over larger local regions and can effectively downsample the input image.

A fully-connected layer (inner product layer) also applies filters on the input feature maps as in the convolutional layers. It can be treated as a special kind of convolutional layers. The fully-connected layer arranges the output of the pooling layer (i.e., $2 \times 3 \times 3 = 18$ elements). The outputs from pooling operations $P0$ and $P1$ are flattened and concatenated to produce a 1×32 input vector. This one dimensional vector is further multiplied with the weight matrix of size 18×4 to generate an output vector with size 1×4. Each of the convolutional layer or fully-connected layer is also immediately followed by an activation layer, such as ReLu (rectified linear units).

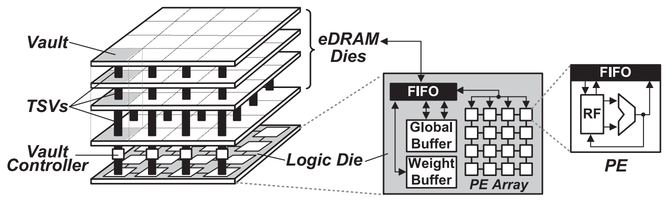


Fig. 2. The emerging three-dimensional PIM architecture for CNN processing.

2.2 3D PIM Architecture

The emerging PIM architecture integrates multiple embedded DRAM (eDRAM) dies and one logic die in a 3D-IC architecture [6]. Fig. 2 illustrates a typical PIM neuromorphic processing architecture. In this architecture, memory is organized into many vaults. Each vault is functionally independent and controlled by a vault controller (memory controller). Processing engines (PEs), a global buffer, and vault controllers are on the logic die. Each PE integrates a PE FIFO, an ALU datapath, and a register file (RF). Multiple PEs can concurrently communicate with multiple eDRAM vaults through high-speed through-silicon via (TSV) technology. Advanced

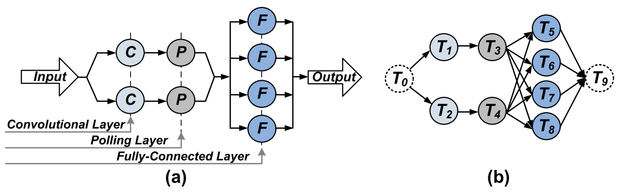


Fig. 3. The data flow graph to model the processing procedure of a CNN.

3D memory standards (e.g., Wide I/O-2 [7] and Hybrid Memory Cube (HMC)[8]) adopt the similar architecture.

On the logic die, two buffers (i.e., global buffer and weight buffer) are used to store the intermediate data and hide the eDRAM access latency. They can communicate with the PE array and eDRAM dies through the input and output FIFO. The global buffer stores the intermediate CNN processing results, such as feature maps. The weight buffer maintains the predefined convolutional weight for convolution layers or fully-connected layers. One major advantage of CNN is the use of shared weight in convolutional layers. The same filter weight is used for each pixel in the same layer. Therefore, a dedicated buffer is allocated for weights to minimize the overhead of accessing weights from the memory.

2.3 Application Model

A CNN application consists of a set of deterministic convolution operations. The functionality of each processing procedure in a CNN can be specified as a number of independent tasks. A set of concurrent convolution or pooling operations can communicate with each other by sending and receiving intermediate processing results via FIFO buffers. The rates at which CNN operations process and generate intermediate data are predefined and known at compile time. Therefore, a CNN can be represented by a synchronous data flow graph or a directed acyclic graph [9], [10].

A weighted graph $G = (V, E, R)$ is used to model the processing procedure of a CNN. $V = \{T_1, T_2, \ldots, T_n\}$ is the vertex set, and each vertex represents a convolution or pooling operation. Each vertex denotes a computing operation, such as inner product, addition, and comparison. One or more vertices can be allocated to one single PE. $E \subseteq V \times V$ is the edge set, and each directed edge, $(T_i, T_j) \in E$ $(T_i, T_j \in V)$, represents the intermediate processing results generated from vertex T_i and requested by vertex T_j. For each directed edge $(T_i, T_j) \in E$, an intermediate processing result $I_{i,j}$ denotes the corresponding data transfer from convolution operation T_i to convolution operation T_j. That is, the outputs (feature maps and partial sums) of a convolution or pooling operation become the inputs of the next convolution or pooling layer. The intermediate processing result does not require computation resources. It only demands storage capacity in cache or eDRAM.

The abstract of the standard procedure for a CNN (shown in Fig. 1) can be modelled as a dataflow graph in Fig. 3a. The nodes with symbols C, P, and F represent a convolution operation, a pooling operation, and a fully-connected operation, respectively. The inputs of a CNN, including initial feature maps and predefined weights, have data dependency relationship with the following convolution operations. It can be treated as a special preprocessing operation. We use a dummy vertex (T_0 in Fig. 3b) to denote the inputs of the CNN. Similarity, the outputs of a CNN can also be represented as a dummy vertex (T_9) in the graph. As a result, the CNN in Fig. 1 can be modelled as a data flow graph in Fig. 3 with 10 vertices and 16 edges.

Each data flow graph represents the processing procedure of one initial input (e.g., a single pixel image). For a CNN application with multiple visual patterns, the same processing procedure will be applied in many times. In CNNs, the number of convolutional sharing is bounded by the width of filter and output feature maps [11]. Therefore, a CNN can be further modelled as an iteratively executed dataflow with deterministic convolutional connections. Let p be the period of each convolution operation T_i and that of each associated intermediate processing result $I_{i,j}$. Convolution operation T_i is associated with three tuples $T_i(s_i, c_i, d_i)$, where s_i is the start time, c_i is the execution time, and d_i is the deadline of T_i. For T_i in the ℓth period, T_i^ℓ, three tuples become $T_i^\ell(s_i^\ell, c_i^\ell, d_i^\ell)$, where $s_i^\ell = s_i + (\ell - 1) \cdot p$, $c_i^\ell = c_i$, and $d_i^\ell = d_i + (\ell - 1) \cdot p$, $\ell \geq 1$. Similarly, intermediate processing result $I_{i,j}$ in the ℓth period can be expressed by $I_{i,j}^\ell(s_{i,j}^\ell, c_{i,j}^\ell, d_{i,j}^\ell)$.

Definition 2.1 (Phase). *Given a graph $G = (V, E, R)$, phase is the process of repeating a set of vertices $T_i \in V$ of the graph G.*

Phase is a computational procedure in which a cycle of convolution or pooling operations is repeated. For a CNN application, a phase will be repeated a specified number of times until a specific learning objective is achieved. The time interval between two successive occurrences of a recurrent phase on the same processing engine is regarded as the period of the graph G.

Definition 2.2 (Iteration). *Given a graph $G = (V, E, R)$, iteration is a set of phases that are concurrently processed by a number of processing engines.*

An iteration consists of at least one phase. The processing time of one iteration is normally the same as that of a phase. In one iteration, convolution or pooling operations from two or more different phases are concatenated to form an iteration. It reflects the repeated execution of phases on multiple processing engines.

Definition 2.3 (Retiming). *Given a graph $G = (V, E, R)$, retiming $R : V \mapsto \mathbb{Z}$ is a function that maps each vertex $T_i \in V$ to an integer $R(i)$. $R(i)$ is initially equal to zero; by retiming T_i once, $R(i) = R(i) + 1$ and one iteration of T_i is rescheduled into the prologue.*

Retiming is originally designed for register allocation problem in a digital circuit. We apply the concept of retiming to describe the delay model of intermediate processing results. For an intermediate processing result $I_{i,j}$, if it can not be allocated between the finishing time of T_i and the release time of T_j, at least one iteration of T_i will be rescheduled into the previous iteration. In this paper, retiming represents the delay of the execution of T_j relative to that of T_i. It can also be interpreted as the newly added iterations (called *prologue*) ahead of the original iterations.

2.4 Motivation

In convolutional neural networks, a convolution or pooling operation needs to perform the same procedure on the feature maps for many times. A significant amount of intermediate processing results are then generated. These intermediate processing results have to be carefully allocated. If an intermediate processing result can be assigned to the PE's on-chip cache, no time delay will be incurred for the coming convolution operation. Otherwise, the load operation from off-PE memory will cost inevitable delay for the following convolution operations.

Memory is critical in the PIM architecture. The data movement in memory system directly impacts performance and power efficiency, two fundamental attributes of the entire system. Neural networks are limited in their ability to store intermediate data over long timescales [2]. The intermediate processing results generated by each convolution or pooling operation will demand fairly large memory capacity. In fact, current PIM architecture can only provide 100-300 KB cache capacity for the entire PE array [10]. How to effectively utilize this limited cache space becomes a critical issue.

Since the behavior of convolutional connections is deterministic, it can be predicted at compile time. A compiler or runtime based approach can help determine the mapping of parallelism onto the underlying hardware. This flexible allocation of convolutions allows no change to existing training process of CNNs, removing the burden of application-specific hardware design. As neural networks are highly parallelizable, both thread-level parallelism and data-level parallelism can be exploited. This fine-grained parallelism helps improve the allocation with efficient usage of memory in 3D PIM architecture. These observations motive us to propose a novel memory-efficient strategy to transparently allocate convolutional connections in PIM architecture.

3 MemoNet: Memory-Efficient Allocation of CNNs on 3D PIM Architecture

3.1 Overview

There are several challenges to enable transparent data allocation and mapping to 3D-stacked PIM architecture. First, a compiler or operating system support is required to identify the characteristics of a specific neural network application. The data flow has to be interpreted and further modelled as fine-grained data structures. Second, a software/hardware co-design framework is required to perform data allocation to either computing or memory components. Since memory is the major bottleneck in 3D-stacked PIM architecture, the solution should take fully advantage of the resource-constrained global buffer to alleviate the memory overhead.

This data allocation problem is not trivial, as the mapping of convolutional operations will determine the occurrence of each intermediate processing result. The actual allocation at on-PE's global buffer or eDRAM may further postpone the execution of the successive convolution operations due to data dependency. The allocation of convolution operations also impacts the preprocessing for the newly introduced prologue, which further influences the processing time of the CNN application.

MemoNet aims to optimize the memory usage of convolutional neural networks on emerging PIM architecture. MemoNet has two major design objectives. First, it fully utilizes the computational power of PIM architecture to speedup the CNN processing. Second, it captures the characteristics of CNN applications and performs convolution allocation to hide memory latency. MemoNet provides a hardware-independent memory efficient data allocation strategy to coordinate between resource producer and resource consumer.

In this section, we first analyze the data allocation for intermediate processing results in Section 3.2. Then we present the convolution allocation to generate an initial schedule in Section 3.3. The rescheduling of the initial schedule to balance cache requirement is presented in Section 3.4. The target problem is further formulated as a dynamic programming method to obtain an optimal solution in Section 3.5.

3.2 The Analysis of Data Allocation

This section analyzes how the allocation of a convolution operation impacts the corresponding intermediate processing results. For the target 3D PIM architecture, fetching data from eDRAM vault costs two times or more access latency than that from the PE array's global buffer. For an intermediate processing result $I_{i,j}$, a convolution or pooling operation T_i produces $I_{i,j}$, and it will be required by T_j. Each intermediate processing result $I_{i,j}$ is associated with two non-negative weights $L_{cache}(I_{i,j})$ and $L_{RAM}(I_{i,j})$, which denote the cost to allocate the intermediate processing result $I_{i,j}$ on the PE's global buffer and the eDRAM in the 3D stacked memory, respectively. The cost L can be considered as the access latency to two different storage media, and $L_{cache}(I_{i,j})$ is much smaller than $L_{RAM}(I_{i,j})$. As c^i_j denotes the execution time of I^i_j, the terms c^i_j and L can be used interchangeably based on the content.

Theorem 3.1. *For a pair of vertices $(T_i, T_j) \in E$ $(T_i, T_j \in V)$ in the ℓth iteration, by retiming T_i for $R(i)$ times and retiming T_j for $R(j)$ times, as long as the retimed vertex $T_{i,\ell-R(i)}$ is rescheduled at most one more iteration ahead of the retimed vertex $T_{j,\ell-R(j)}$, the associated intermediate processing result $I_{i,j}$ is always schedulable on PIM during the time span between the finishing time of $T_{i,\ell-R(i)}$ and the start time of $T_{j,\ell-R(j)}$.*

Proof. The initial computation schedule of a CNN application obeys the data dependency relations in the graph G. In the ℓth iteration of the initial schedule, the finishing time of T_i is no later than the start time of T_j. After retiming T_i for one time relative to the retimed vertex $T_{j,\ell-R(j)}$, T_i will be scheduled in iteration ℓ-$R(j)$-1, which is one iteration ahead of $T_{j,\ell-R(j)}$. Then the time span between the finishing time of $T_{i,\ell-R(i)}$ and the start time of $T_{j,\ell-R(j)}$ is always greater than or equal to the time of one iteration (period p), As all immediate processing results repeatedly execute in each iteration, in one iteration of time, at least one I^i_j has data dependency with T_i and T_j. Let this I^i_j be the immediate processing results of $T_{i,\ell-R(i)}$ and $T_{j,\ell-R(j)}$. Its start time is no earlier than the finishing time of $T_{i,\ell-R(i)}$, and its finishing time is no later than the start time of $T_{j,\ell-R(j)}$. Therefore, I^i_j is always schedulable on PIM during that time span. An example is given in Fig. 4. □

Retiming technique maintains the data dependencies for each pair of vertices and preserves the functionality of the CNN. Theorem 3.1 presents the upper bound of the maximum relative retiming value of each pair of vertices. This tight constraint presents a good property for rescheduling each vertex. This upper bound also implies that the maximum iteration added to the initial schedule is only one

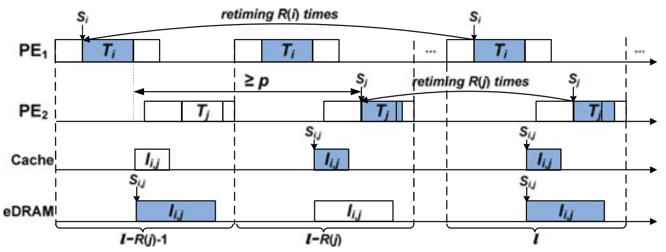


Fig. 4. The exemplary allocation for convolutional connections with data dependency relations.

iteration of time. Let R_{max} be the maximum retiming value among all convolution operations, $R_{max} = \{\max R(T_i), T_i \in V\}$. The prologue time can be obtained by $R_{max} \times p$, and this denotes the preprocessing time before the loop kernel.

Properties 3.1 to 3.3 further classify the allocation to either PE's global buffer or off-PE eDRAM into three cases. For Property 3.1, the time span between the finishing time of T_i and the release time of T_j is longer enough to allocate the intermediate processing result $I_{i,j}$ from eDRAM. Therefore, no extra retiming operations are needed for T_i.

Property 3.1. *For a pair of vertices T_i and T_j, $(T_i, T_j) \in E(T_i, T_j \in V)$, if $S_i + c_i + L_{RAM}(I_{i,j}) \leq S_j$, there is no need to perform extra retiming for T_i. That is, $R(i) = R(j)$.*

For Property 3.2, the time span is not sufficient to accommodate $I_{i,j}$, even though it can be allocated to the global buffer. In this case, T_i has to be retimed. The upper bound provided by Theorem 3.1 further determines that only one extra retiming operation for T_i is needed. For Property 3.3, the difference between $R(i)$ and $R(j)$ becomes zero if I_j^i can be allocated to PE's global buffer; and becomes one if I_j^i has to be allocated to eDRAM. In this case, intermediate processing result I_j^i may postpone the execution of T_j.

Property 3.2. *For a pair of vertices T_i and T_j, $(T_i, T_j) \in E\,(T_i, T_j \in V)$, if $S_i + c_i + L_{cache}(I_{i,j}) \geq S_j$, T_i needs to perform retiming once and exactly once relative to T_j to ensure the schedulability of I_j^i. That is, $R(i) = R(j) + 1$.*

Property 3.3. *For a pair of vertices T_i and T_j, $(T_i, T_j) \in E(T_i, T_j \in V)$, if $S_i + c_i + L_{RAM}(I_{i,j}) > S_j > S_i + c_i + L_{cache}(I_{i,j})$, the value of $R(i)$ depends on the allocation of I_j^i, and $R(i) - R(j) \leq 1$.*

3.3 Generating Initial Schedule with Fine-Grained Parallelism

Based on the analysis in Section 3.2, the allocation of vertices T_i and T_j determines the release time and it may cause extra retiming operations for T_i. As the first step of the proposed solution, MemoNet generates an initial schedule with fine-grained parallelism. The initial schedule takes into account the allocation of immediate processing results and tries to avoid unnecessary retiming operations.

We observe that the fine-grained partition of vertex will increase both the number of vertices and edges in the graph. By appropriately merging several vertices, it can reduce the complexity to process the graph and still preserve the functionality of the application. A vertex T_u that can be merged with another vertex T_v has the following property: First, both T_u and T_v are not dummy vertices; Second, T_u and T_v can be assigned to the same processing engine without causing extra latency in pipelining, i.e., $c_u + c_v \leq max\{c_i\}$, $\{T_u, T_v, T_i\} \in V$.

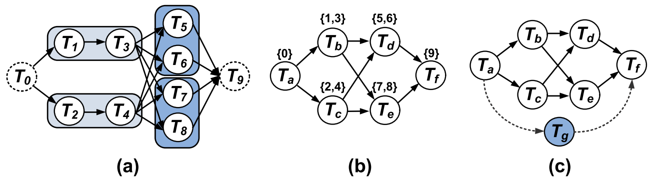


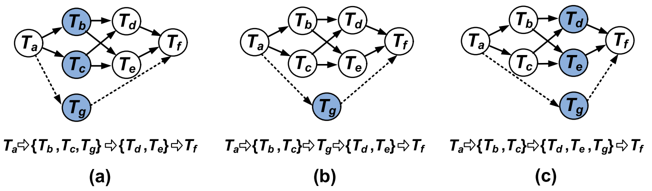
Fig. 5. The merge of vertices and the general data flow graph with asymmetrical structure.

There are two typical cases for vertex merge operations. For the first case, called *sequential merge*, both vertices T_u and T_v, $\{T_u, T_v\} \in V, (T_u, T_v) \in E$, have only one incoming (or outgoing) edge. Vertex T_u is the predecessor of vertex T_v. T_1 and T_3 in Fig. 5a show this case. For the second case, called *parallel merge*, vertex T_u and vertex T_v have the same number of incoming edges and the same number of outgoing edges. Vertex T_u and vertex T_v are in the same layer. T_5 and T_6 in Fig. 5a show this case. After the merge operation, the original data flow graph in Fig. 5a is transformed into a new graph in Fig. 5b.

The merged vertex represents the combination of two computation vertices. Its execution time is the total execution time of the two vertices. For example, vertex T_b in Fig. 5b represents the merged vertex of vertices T_1 and T_3 in Fig. 5a. The merge operation will also merge the edges in the graph. The associated intermediate processing results will denote the total data transfer for incoming edges (or outgoing edges) of vertices T_u and T_v.

The data flow graph of a CNN application normally includes the core subgraph with symmetrical structure. For example, let two dummy vertices (T_a and T_f in Fig. 5b) be the symmetrical axis, the vertices above the axis have the mirrored vertices below the axis. For example, T_b and T_c form the symmetrical structure. They have exactly the same number of incoming (outgoing) edges, and the execution time of these two vertices are the same. Besides this core structure, the data flow graph may also have several individual vertices that could not find mirrored vertex in the graph. We add such an individual vertex (T_g) to the merged graph, and a new data graph is illustrated in Fig. 5c.

The level or layer where a vertex belongs to will affect the allocation order of this vertex on processing engines. For symmetrical vertices (e.g., T_b and T_c in Fig. 6), they get the similar execution time for convolution connections and also get the similar processing time for associated intermediate processing results. They are prone to be assigned to different processing engines at the same time, so they can be concurrently processed. Then the allocation order of vertices can be obtained based on the traverse of vertices (e.g., breadth-first search).



Ta
Tb
Tc
Td
Te
Tf
Tg
Ta⇨{Tb ,Tc,Tg}⇨{Td ,Te}⇨Tf
(a)
Ta⇨{Tb ,Tc}⇨Tg⇨{Td ,Te}⇨Tf
(b)
Ta⇨{Tb ,Tc}⇨{Td ,Te ,Tg}⇨Tf
(c)

Fig. 6. The allocation order of vertices for the graph in Fig. 5c.

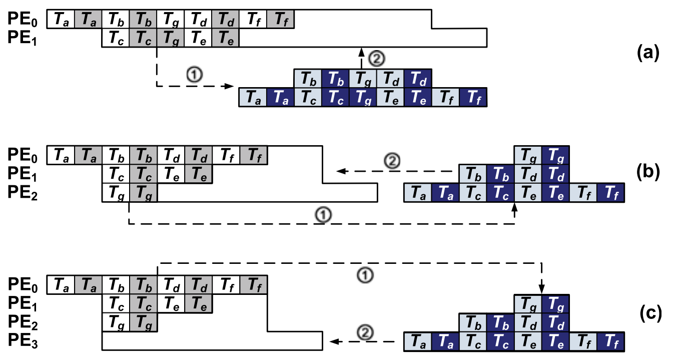


Fig. 7. Generate a phase of convolution connections, and use *rotation* and *insertion* to form an iteration.

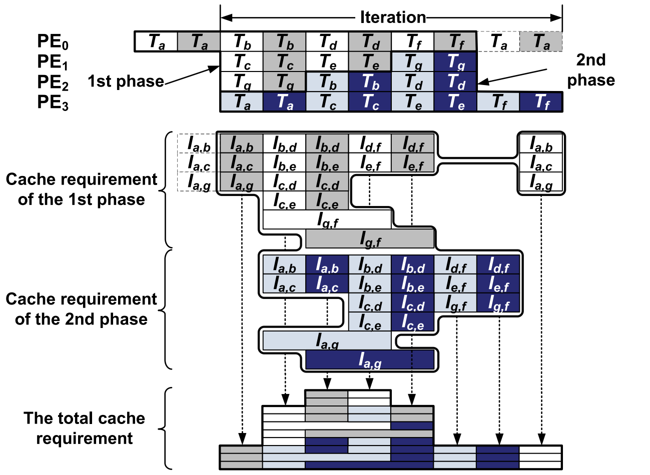
The individual vertex (e.g., T_g) can be allocated to different levels in the graph. (1) As early as possible: the vertex is released at the earliest time. That is, parallel processing with T_b and T_c at the second level in Fig. 6a. (2) A newly created level: the vertex itself becomes the new level of the graph. For example, T_g becomes the third level of the graph in Fig. 6b. (3) As late as possible: the vertex is postponed until its latest time to release. T_g can be released at the same level as T_d and T_e in Fig. 6c. The actual allocation depends on the number of processing engines. For example, if there are only two PEs in the system for the graph in Fig. 6, case 2 is the choice, as the system could not support parallel processing of three vertices. The allocation is also affected by the deadline of the vertex, and the adopted scheduling algorithm will find its appropriate level in the graph.

The allocation of vertices needs to consider the intermediate processing result, which costs the latency $L_{cache}(I_{i,j})$ or $L_{RAM}(I_{i,j})$. In order to hide this latency, MemoNet interleaves the execution of convolution connections from two different processing procedure of CNNs. The dependent computations are separated from one another, increasing the possibility that the successor vertex can be scheduled without stalls. An example is illustrated in Fig. 7, where T_a is immediately followed by another T_a from another processing procedure of the CNN. This step is quite similar to loop unrolling. The major difference is that our technique only repeats symmetrical vertex (e.g., T_a) for once and the individual vertex (e.g., T_g) can be allocated based on the scheduling policy.

The interleaved execution of two processing procedures form the basic execution block, called phase. Fig. 7 illustrates the phase of convolution connections on 2, 3, and 4 PEs for the task graph in Fig. 6. Additionally, by concatenating a set of phases, an iteration can be formed. This involves two steps. The first step, called *rotation*, tries to generate a complement phase that can improve the utilization of PEs. For Fig. 7a with two PEs, T_a, T_b, T_d, and T_f are assigned to PE_0, and they will be assigned to PE_1 in the complement phase.

Once a complement phase is selected, the second step, called *insertion*, allocates each vertex to the newly created complement phase according to the allocation order. The complement phase attempts to move its schedule as early as it can to eliminate any possible idle PE. The initial phase and the complement phase are packed into one iteration. Fig. 8 shows two phases with 4 PEs and the corresponding iteration.

The iteration determines the allocation of vertices, and it will be repeatedly executed in the successive time slot. As a result, the initial schedule of vertices on a number of PEs can be generated. The corresponding cache requirement to allocate each intermediate processing result on PE's global buffer can also be known. Fig. 8 illustrates the life span of each intermediate processing result. Assume that each vertex takes one time unit for processing and each intermediate processing result causes the same amount of buffer space. Then the total cache requirement in each iteration can obtained.


Iteration
PE0
PE1
PE2
PE3
1st phase
2nd phase
Cache requirement of the 1st phase
Cache requirement of the 2nd phase
The total cache requirement
Fig. 8. An iteration of the initial schedule and its cache requirement.

3.4 Balancing of Cache Requirement

As MemoNet adopts fine-grained parallelism, multiple iterations can execute in parallel to scale-up all available processing engines. The current neuromorphic processing architecture normally provides adequate computational resources. MemoNet adopts combinations of multiple iterations to speedup convolution processing. Fig. 9 shows the system with 8 PEs, which can be divided into two groups of PEs. Each group contains 4 PEs. Using the data allocation from Fig. 8, the first and the second iteration can be operated in parallel.

The partition of PEs offers various combinations of iterations. For example, a system with 8 PEs can have two iterations with 4 PEs per iteration. It can also be the combination of four identical iterations with 2 PEs per iteration. There exists the trade-off in terms of PE's utilization versus the time span to maintain intermediate processing results. As a memory-efficient design, MemoNet fully exploits the thread-level parallelism to reduce the latency and cache requirement for intermediate processing results. Assume that each convolutional or pooling layer consists of at most N_v vertices, MemoNet prefers to select the iteration with N_v or $N_v + 1$ PEs. The PE's utilization is compensated by the adoption of rotation and insertion in the creation of an iteration.

Algorithm 3.1 presents how to determine the subschedule of each iteration. Given a weighted graph G to model a CNN application, Algorithm 3.1 first obtains the number of tasks in each layer (Line 1). The maximum one, denoted as $N_{max-count}$, represents the maximum number of PEs that are needed to concurrently process the graph G. Algorithm 3.1 further generates different combinations of schedules for the number of PEs ranging from 2 to $N_{max-count} + 1$ (Line 2). This step will generate the schedules like those in Fig. 7.

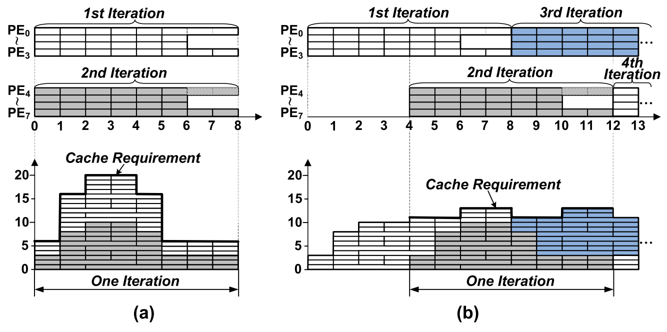


Fig. 9. The allocation of iteration impacts the total cache requirement.

Given N_{pe} processing elements in the PIM architecture, we divide N_{pe} by $N_{max-count}$ and $N_{max-count}+1$ (Lines 3-4). If N_{pe} can be divided without remainder, the quotient becomes the number of subschedules (Lines 5-7). Otherwise, the number of subschedules is determined by both quotient and remainder (Lines 9-17). Then the schedule contains several subschedules, and each subschedule uses different numbers of PEs. The indexes H_{com} and H_{spe} determine the PE's number for these subschedules. Algorithm 3.1 selects the case that divisor and remainder are close to each other. For example, we assume that N_{pe} equals to 17 and it processes the task graph in Fig. 5c where $N_{max-count}$ is equal to 3. N_{pe} (i.e., 17) is divided by 3, the quotient is 5 and the remainder is 2. Similarly, we further divide 17 by 4, and the remainder is 1. Since the case for $N_{max-count}$ (i.e., the remainder is 2) is more closer to the divisor (i.e., 3), the 17 PEs are used to form 6 subschedules. For the first 5 subschedules, each one utilizes 3 PEs. The sixth subschedule will utilize the rest 2 PEs.

Algorithm 3.1. Determine the subschedule of each iteration

```
Input: A weighted graph $G$ to model a CNN application, $N_{pe}$
       processing elements in the PIM architecture.
Output: The combination of subschedules for each iteration
        and the number of PEs in each subschedule.
1:  getTaskNumInEachLayer()
2:  getDiffCombinationSchedule()
3:  $U_1 \leftarrow N_{max-count}$; $U_2 \leftarrow N_{max-count}+1$.
4:  $r_\alpha \leftarrow N_{pe} \% U_1$; $r_\beta \leftarrow N_{pe} \% U_2$.
5:  if either $r_\alpha$ or $r_\beta$ equals to 0 then
6:      $N_g \leftarrow (r_\alpha == 0)?\ (N_{pe}/U_1):(N_{pe}/U_2)$.
7:      $H_{com} \leftarrow (r_\alpha == 0)?\ (U_1-2):(U_2-2)$.
8:  else
9:      if $U_1 - r_\alpha \le U_2 - r_\beta$ then
10:         $N_g \leftarrow N_{pe}/U_1 + 1$.
11:         $H_{com} \leftarrow U_1 - 2$.
12:         $H_{spe} \leftarrow r_\alpha - 2$.
13:     else
14:         $N_g \leftarrow N_{pe}/U_2 + 1$.
15:         $H_{com} \leftarrow U_2 - 2$.
16:         $H_{spe} \leftarrow r_\beta - 2$.
17:     end if
18: end if
```

The cache requirement represents the best-case buffer size, where the PE's global buffer has enough capacity to hold all intermediate processing results. The allocation of iteration impacts the total cache requirement. Fig. 9 shows this behavior graphically. In Fig. 9a, both the first and the second iteration execute in the time span from 0 to 8, and the maximum cache requirement is 20 units. Since the shape of a CNN graph is big in the middle but is small at both ends, the maximum cache requirement occurs at the middle of each iteration.

This unevenly distribution for cache requirement will cause extra latency, as the PE's global buffer may not have enough capacity to concurrently store all intermediate processing results at the time with peak cache requirement (e.g., time span 2-4 in Fig. 9a). In this case, several intermediate processing results have to be allocated to off-PE's eDRAM, which may cause the retiming operation for one iteration. MemoNet solves this problem by rescheduling the start time of the iteration. As shown in Fig. 9b, the start time of iterations on PE4-PE7 is reallocated to time unit 4. Thus, the peak cache requirement is significantly reduced to 13 units, making the balanced requirement for PE's global buffer in each iteration.

To facilitate the analysis of cache requirement, MemoNet partitions the PEs into N_g identical groups, and each group adopts the same allocation of vertices in each iteration. Algorithm 3.2 presents the procedure to obtain the start time for each group of iterations. It handles three cases. For the case that the PE's global buffer can hold all intermediate processing results (Line 1), there is no need to postpone the execution for each group of iteration (Line 2). The second case shows that, the total capacity of the global buffer could not satisfy the requirement no matter how the schedule postpones (Line 4). In this case, several intermediate processing results have to be reallocated to eDRAM (Line 5). For the last case, it is possible to overlap the peak and the valley of cache requirment for different groups of iterations. Both the second and the last case will cause the delay of the execution (Line 7).

Algorithm 3.2. Obtain the start time for each group of iterations of a CNN application

```
Input: An initial schedule $S_{init}$ of an iteration, the period $p$, $N_{ph}$
       phases in each iteration, $N_g$ identical PE groups, the
       capacity of the global buffer $C_t$, the maximum $C_{max}$ and
       the minimum $C_{min}$ cache requirements in one iteration,
       the start time $gs_0$ of the schedule.
Output: The start time $gs_i$ of the $i$th group $i \in [1, N_g-1]$.
1: if $C_{max} \cdot N_g \le C_t$ then
2:     $gs_i \leftarrow gs_0$, $i \in [1, N_g-1]$.
3: else
4:     if $(C_{max}+C_{min}) \cdot N_g > 2C_{total}$ then
5:         Allocate intermediate processing results to eDRAM with
           at least $\frac{1}{2}(C_{max}+C_{min})$-$C_t$ capacity.
6:     end if
7:     $gs_i \leftarrow gs_0 + \frac{1}{N_g} \cdot p \cdot i$, $i \in [1, N_g-1]$.
8: end if
```

3.5 Optimal Convolution Allocation with Prologue Reduction

Based on the analysis of retiming value, to minimize the preprocessing time in the prologue is equivalent to the problem of reducing the maximum retiming value of all convolution operations in the application. In previous sections, we have formed the iteration and obtained the initial schedule. The PEs are further partitioned into several groups, and each group handles one iteration. Then the

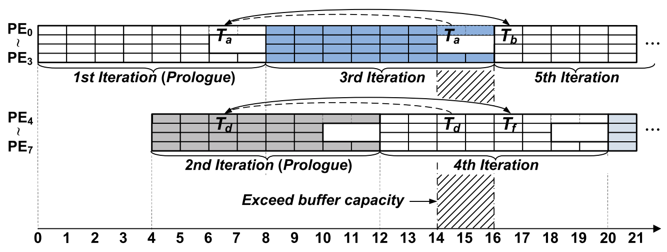


Fig. 10. Retiming of vertices and the extra processing in prologue.

target problem is transformed into the reduction of the maximum retiming value of different groups.

This problem has an optimal substructure property. The optimal solution for the global buffer with capacity C_m is always the subproblem's solution for the buffer with larger capacity C_n, where $C_m \leq C_n$. The target problem can be solved by identifying a collection of subproblems and tackling them one by one, using the allocation with small capacity to help figure out large ones. Therefore, it can be effectively solved by dynamic programming. This section presents the formulation of the optimal allocation of convolution connections with minimum prologue.

Not all intermediate processing results get involved in this prologue reduction process. In Section 3.4, we have generated an initial schedule and obtained the start time of each group of iterations. Only the intermediate processing results that cause the cache requirement beyond the capacity of the global buffer will influence the prologue. These intermediate processing results form a subset of all vertices in the graph.

Algorithm 3.3. Generate an optimal allocation with the shortest prologue

```
Input: A set of $n$ intermediate processing results $V_{sch}$, $V_{sch} \subseteq V$,
       $N_g$ identical PE groups, the intermediate processing
       result $I_{i,j}$ allocated to group $u$, the capacity of the global
       buffer $C_t$, the space requirement $C_\alpha$ to allocate $I_\alpha$ on the
       PE's global buffer, the penalty $\mathbb{P}[C_x, \alpha]$ to place a set of $\alpha$
       intermediate processing results $\{I_1, I_2, \ldots, I_\alpha\} \in V_{sch} \subseteq$
       $V$ on the PE's global buffer with the capacity $C_x$, and a
       queue $Q$.
Output: The allocation and the start time $s_{i,j}$ of each intermedi-
        ate processing result $I_{i,j}$.
 1: for each $I_{i,j} \in V_{sch}$ do
 2:     if $R(i) - R(j) = 0$ then
 3:         $\Delta L(i,j) \leftarrow 0$.
 4:     else
 5:         $\Delta L(i,j) \leftarrow p \cdot (1 - \frac{u}{N_g}) \cdot (R(i) - R(j))$.
 6:     end if
 7:     $Enqueue(Q, I_{i,j})$.
 8: end for
 9: Sort $I_{i,j} \in V_{sch}$ in descending order by $\Delta L(i,j)$.
10: while $Q \neq \emptyset$ do
11:     if $\mathbb{P}[C_x\text{-}C_\alpha, \alpha\text{-}1] < \max\{\mathbb{P}[C_x, \alpha - 1], \Delta L(\alpha)\}$ then
12:         Allocate $I_{i,j}$ on PE's global buffer.
13:     else
14:         Allocate $I_{i,j}$ on eDRAM.
15:     end if
16:     $Dequeue(Q, I_{i,j})$.
17: end while
18: Return $\mathbb{P}[C_t, n]$.
```

Algorithm 3.3 presents the procedure to generate an optimal allocation of intermediate processing results. The objective is to obtain the schedule with the shortest prologue. The first step gets the relative retiming value $(R(i) - R(j))$ for each intermediate processing result $I_{i,j}$ when it is allocated to eDRAM. Theorem 3.1 and three Properties in Section 3.2 can help derive each relative retiming value. If the retiming value is equal to zero, this intermediate processing result will not incur extra retiming operations. For other cases, $I_{i,j}$ will introduce the extra preprocessing with latency $\Delta L(i,j) = p \cdot (1 - \frac{u}{N_g}) \cdot (R(i) - R(j))$, where p is the period, N_g is the number of PE groups, and $I_{i,j}$ is allocated to group u.

The latency $\Delta L(i,j)$ of retiming is affected by the start time of the iteration. The reallocation of different vertices may incur different latency. As shown in Fig. 10, assume that the cache requirement for intermediate processing results exceeds the capacity of the on-PE's global buffer during time units 14-16. PE0-PE3 and PE4-PE7 form two PE groups. The group in PE0-PE3 initially starts at time unit 8 (the third iteration). By retiming T_a once, a new iteration (the first iteration) is added in the prologue, and the schedule will be started at time unit 0. This will cause the latency with 8 time units (from unit 0 to unit 8). On the other hand, the group in PE4-PE7 initially starts at time unit 12 (the fourth iteration). Retiming a vertex in the fourth iteration will only incur the latency with 4 time units (from unit 4 to unit 8).

MemoNet sorts these n intermediate processing results V_{sch}, $V_{sch} \subseteq V$, according to $\Delta L(i,j)$, such that the intermediate processing result $I_{i,j}$ with the maximum extra latency would be scheduled first. In the sorted order, an intermediate processing result $I_{i,j}$ has the αth shortest latency, $\alpha \in [1, n]$. In the dynamic programming model, $I_{i,j}$ and I_α are used interchangeably. Intermediate processing results that have the same latency are recommended to share the same choice of allocation. This precomputation can be done in time $O(n \log n)$.

The second step of Algorithm 3.3 involves in the dynamic programming model, where $\mathbb{P}[C_x\text{-}C_\alpha, \alpha\text{-}1]$ is compared with $\max\{\mathbb{P}[C_x, \alpha - 1], \Delta L(\alpha)\}$. Based on these two values, Algorithm 3.3 determines whether allocating $I_{i,j}$ on eDRAM or not. The dynamic programming model is initialized as $\Delta L(1)$ if C_α is greater than C_x. For the case that C_x is greater than or equal to C_α and $n \geq 2$, $\mathbb{P}[C_x, \alpha]$ is set as $\Delta L(2)$.

An optimal convolution allocation can be obtained by returning the values of $\mathbb{P}[C_t, n]$. A $n \times C_t$ matrix $\mathbb{P}$ is used to record the recursive results of the dynamic programming model. This table reflects the allocation of each intermediate processing result, either at eDRAM or at on-PE's global buffer. Each entry of the matrix takes $O(1)$ time to compute. Therefore, the running time of the dynamic programming procedure takes $O(n \cdot C_t)$.

4 Evaluation

4.1 Experimental Setup

We conduct experiments using a set of benchmarks from real-life CNN applications. The widely used deep learning framework Caffe [3] is used to abstract graph representation. The neural network model obtained from the training phase is sent to Caffe, and we added notations and variables to track the timing and other parameters of each

TABLE 1
Total Execution Time of SPARTA [4], Memolution [5], and MemoNet on 32, 64, and 128 Processing Elements

	32-PE			64-PE			128-PE		
	SPARTA	Memolution	MemoNet	SPARTA	Memolution	MemoNet	SPARTA	Memolution	MemoNet
speech-1	80,224	71,416	60,385	53,664	35,990	31,007	24,960	18,220	17,438
speech-2	130,208	119,756	100,354	87,100	60,068	55,740	43,784	30,224	28,014
character	90,180	88,406	77,154	72,360	44,344	40,147	33,120	22,312	20,847
image-processing-1	145,406	127,686	104,721	97,498	64,248	53,140	49,184	32,530	28,049
image-processing-2	130,364	116,734	101,951	87,308	58,736	51,814	40,664	29,738	26,520
image-processing-3	90,288	85,952	78,842	72,504	43,248	39,971	33,264	21,896	20,001
path-planning	130,416	123,556	109,987	102,714	62,560	55,237	52,728	31,694	29,074
statistical-modeling	165,462	147,968	121,781	109,265	74,452	63,472	51,744	37,694	33,145
video	150,660	147,022	119,712	129,690	73,976	60,738	61,110	37,454	31,029
remote-sensing	120,384	120,124	105,784	103,608	60,594	59,014	48,744	30,678	29,871
nlp-1	69,334	57,012	51,493	56,913	48,028	42,172	50,177	37,481	30,078
nlp-2	83,496	75,102	72,336	69,445	54,978	50,781	57,105	45,315	42,845
sequence-learning-1	296,455	257,194	202,781	195,721	133,479	114,087	113,584	75,012	70,117
sequence-learning-2	131,482	102,575	93,110	84,577	72,485	57,865	75,418	66,747	51,874
social-network	280,937	240,172	185,147	230,048	198,743	144,788	134,750	114,559	98,563

convolution or pooling operation. Among these benchmarks, *speech-1* and *speech-2* are voice recognition applications to identify the speech of a dedicated speaker; *character* is a character recognition application to recognize the bitmap pattern of handwritten characters; *image-processing-1* to *image-processing-3* are image processing applications to identify the major object in the images; *path-planning* is an path planning application that finds the shortest path in the map; *statistical-modeling* models the pattern in the stock market; *video* is a video recognition application to track the moving object in the movie; *remote-sensing* is an application to process the infrared remote sensing images; *nlp-1* and *nlp-2* are natural language processing applications; *sequence-learning-1* and *sequence-learning-2* are CNN applications that adopt sequence learning technique to perform optical character recognition; *social-network* is an application to analyze the structure and global patterns of social networks.

Our experiments are conducted based on the PIM architecture of Neurocube [6]. The Neurocube neuromorphic architecture is an extension of Micron's Hybrid Memory Cube to support neural computing. It provides high-density 3D stacked memory to integrate multiple tiers of DRAM and a number of processing engines. In our experiments, the architecture is configured to contain up to 128 processing engines with cross-bar interconnection.

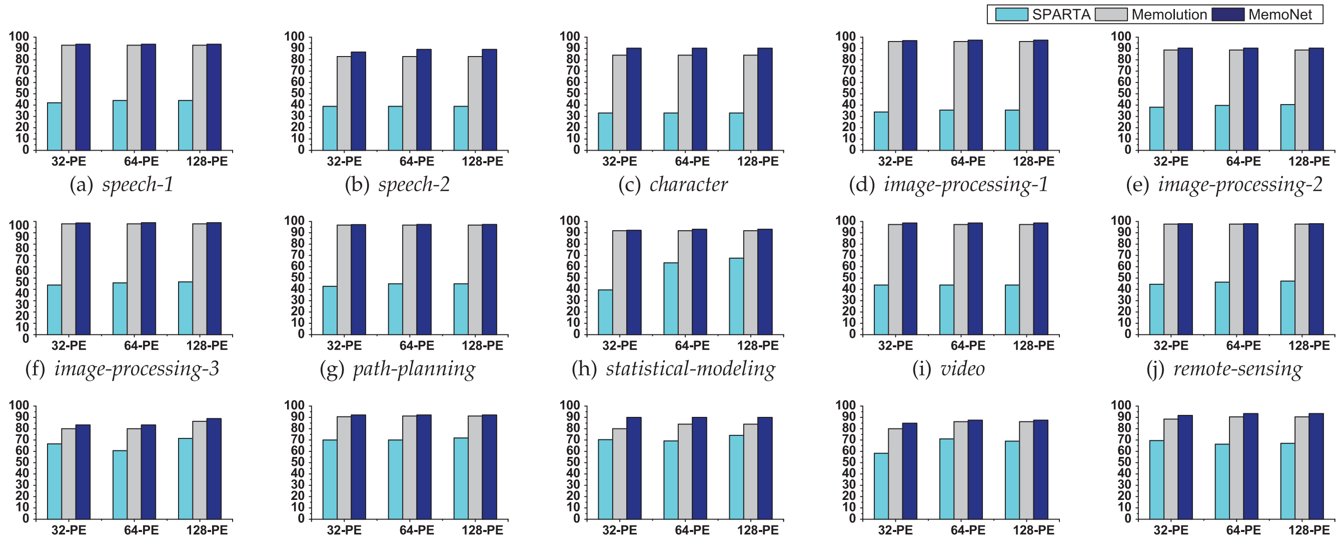
4.2 Experimental Results

1) Processing Time: Table 1 presents the processing time of SPARTA [4], Memolution [5], and and the proposed MemoNet under 32, 64, and 128 PEs. SPARTA is a throughput-aware task allocation approach for many-core platforms. SPARTA collects sensor data to characterize tasks and uses this information to prioritize tasks when performing allocation. Therefore, it is selected as the baseline scheme. In order to make a fair comparison, SPARTA is combined with the retiming technique to preserve the data dependencies. Memolution is our previous work, which adopts a heuristic-based approach to solve the convolution allocation problem. The proposed MemoNet further enhances it and presents a dynamic programming model to obtain the optimal solution. The schemes SPARTA, Memolution, and MemoNet adopt the same input graphs. The experimental results are specified in microseconds. Columns "IMP [4] (%)" and "IMP [5] (%)" represent the reduction of the processing time compared to SPARTA and Memolution, respectively.

From the experimental results, MemoNet can achieve average reductions of 33.01% and 12.54% compared to SPARTA and Memolution, respectively. These reductions come primarily from fine-grained parallelism. In Memolution, the interleaved scheduling of two processing procedures can provide enough time span to allocate intermediate processing results without incurring extra timing overhead of the entire schedule. Therefore, Memolution can achieve better results compared to SPARTA. MemoNet can generate a shorter schedule by capturing the properties of individual tasks. The optimal allocation of intermediate processing results also reduces the processing time in each iteration.

2) Utilization Ratio of PEs: Fig. 11 presents the utilization ratio of processing elements. As expected, MemoNet can achieve the highest utilization ratio. Memolution adopts the rotation and insertion process to generate the initial schedule. An iteration can be generated from two or more complementary phases. Although SPARTA is combined with retiming to further reschedule some iterations into the prologue, the fine-grained reallocation of intermediate processing results are not performed in SPARTA. Therefore, it can not generate a good initial schedule. MemoNet further optimizes the combinations of subschedules of each iteration. The initial schedule is generated based on the subschedules with $N_{max-count}$ or $N_{max-count}+1$, where $N_{max-count}$ is the maximum number of PEs that are needed to concurrently process the CNN application. Therefore, in MemoNet, the subschedule with good utilization ratio will be used to form the initial schedule.

3) Cache Efficiency: As a memory-efficient optimization technique, MemoNet aims to provide an optimal allocation of intermediate processing results. Table 2 presents the number of intermediate processing results that are allocated to the on-PE's shared buffer. The impact of cache efficiency is highly correlated to the utilization ratio of PEs. The higher utilization ratio of PEs denotes the better usage of the on-PE's shared buffer. Due to the nature of the shared buffer,



(a) *speech-1* (b) *speech-2* (c) *character* (d) *image-processing-1* (e) *image-processing-2*
(f) *image-processing-3* (g) *path-planning* (h) *statistical-modeling* (i) *video* (j) *remote-sensing*
(k) *nlp-1* (l) *nlp-2* (m) *sequence-learning-1* (n) *sequence-learning-2* (o) *social-network*

Fig. 11. The utilization ratio of processing elements for SPARTA [4], Memolution [5], and MemoNet on 32, 64, and 128 processing elements.

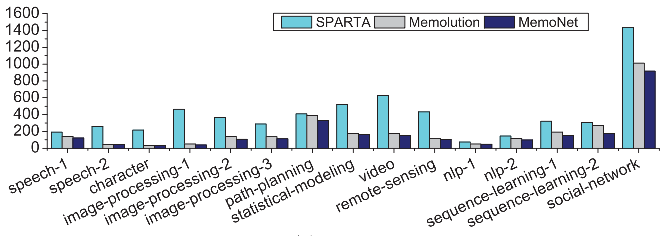
its capacity should be fully utilized for all kinds of schemes. However, some intermediate processing results may occupy the valuable space, which may prevent the efficient usage of the cache. MemoNet provides the analysis on the start time and finishing time of intermediate processing results. The dynamic programming model aims to find the optimal usage of the valuable global buffer. Therefore, it can efficiently use both computational and memory resources in the PIM architecture. From the experimental results, MemoNet can allocate 3.27x and 1.12x of intermediate processing results on the PE's shared buffer.

4) Extra Processing for Retiming: Retiming operations will introduce preprocessing stage of convolutional connections, which will impact the total processing time of the application. This overhead is quantified, and the experimental results are illustrated in Fig. 12. We observe that the retiming value of SPARTA is not sensitive to the number of PEs, while both Memolution and MemoNet can get better scalability with a larger number of PEs. This is due to the fact that, MemoNet uses a set of optimization techniques to generate the initial schedule and improve the utilization of PEs. The retiming operation is combined with these techniques to jointly optimize the allocation of both convolutional connections and intermediate processing results. The fine-grained parallelism also helps reduce this retiming overhead.

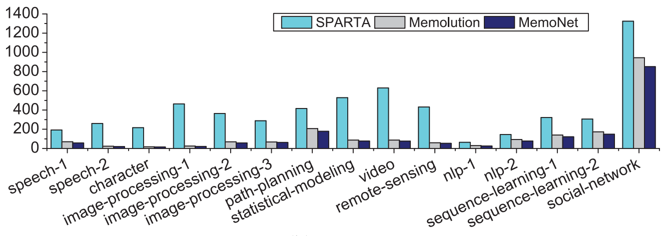
For the baseline scheme SPARTA, it combines the simple retiming technique without applying any further optimization techniques. Therefore, SPARTA will introduce more retiming operations. Compared to Memolution, the proposed MemoNet can further reduce the number of retiming operations. This is mainly due to the adoption of dynamic programming based approach, which can obtain better allocation of intermediate processing results. Although MemoNet may introduce multiple iterations of prologue, compared to the benefit gained from the significant reduction of processing time in each iteration, this overhead is acceptable and can be rectified at the compile time.

TABLE 2
The Number of Intermediate Processing Results that are Allocated to on-PE's Shared Buffer for SPARTA [4], Memolution [5], and MemoNet

	32-PE			64-PE			128-PE		
	SPARTA	Memolution	MemoNet	SPARTA	Memolution	MemoNet	SPARTA	Memolution	MemoNet
speech-1	44	86	91	23	86	91	19	86	91
speech-2	23	114	127	16	115	131	16	115	131
character	22	114	125	18	115	125	18	115	125
image-processing-1	27	70	76	19	70	79	19	70	79
image-processing-2	27	67	71	20	67	71	19	67	71
image-processing-3	32	84	92	20	84	94	15	84	97
path-planning	61	111	120	48	111	126	48	111	126
statistical-modeling	51	60	69	30	60	71	27	60	71
video	16	45	55	14	50	57	15	55	57
remote-sensing	27	63	73	23	63	73	21	63	73
nlp-1	189	306	348	186	321	360	177	369	408
nlp-2	198	264	273	192	270	279	192	270	279
sequence-learning-1	201	267	312	195	273	312	174	276	312
sequence-learning-2	165	228	282	138	237	291	111	240	293
social-network	831	945	998	816	972	1,005	813	993	1,041



(a) *64-PE*



(b) *128-PE*

Fig. 12. The number of retiming operations for SPARTA [4], Memolution [5], and MemoNet on 64 and 128 processing elements.

5) Time Cost for Preprocessing: Both Memolution and MemoNet require joint scheduling of intermediate processing results and convolution operations. The joint scheduling takes extra time cost for preprocessing. The experimental results are specified in microseconds and presented in Table 3. From the results, MemoNet takes much more time to find the optimal allocation compared to Memolution. Memolution adopts the heuristic based approach, while MemoNet uses the dynamic programming model to obtain the optimal solution. The dynamic programming model iteratively searches the best solution for optimal substructures. Therefore, the ranges of time cost are of different orders of magnitude.

We observe that the results for MemoNet are sensitive to the number of PEs. The larger number of PEs indicates that there will be more intermediate processing results used as the inputs of the dynamic programming model. As a heuristic based approach, Memolution does not sensitive to the number of PEs. They could generate the initial schedule within several hundreds of microseconds. Although MemoNet may experience about 3 days of execution for the large scale benchmark (i.e., *social-network*), this preprocessing procedure can be done offline. For the system that requires fast prototyping, the heuristic approach Memolution can obtain near-optimal solutions. To generate an optimal solution for a general CNN application, MemoNet is recommended, which can further improve system throughput and reduce off-chip fetching penalty.

5 Related Work

FPGA/ASIC accelerators for CNNs. FPGA and ASIC accelerators provide hardware-based neuromorphic platforms to speedup the data processing of CNN applications. There have been a number of FPGA implementations [1], [12], [13], [14] and ASIC accelerators [15], [16]. Compared to general purpose graphics processing unit (GPGPU), FPGA platforms can achieve much higher power and computational efficiency [17]. However, for FPGA platforms, once a neural network engine is synthesized, it cannot be programmed on-line [6]. ASIC accelerators can improve the training process and obtain even better performance for specific CNN applications [18]. It is limited by the scalability of on-chip memory or interface with external memory.

The proposed MemoNet is different from these works. It is a software-based technique, which provides a hardware-independent task allocation interface. MemoNet is not designed for a specific hardware platform or a specific CNN application. The underlying hardware resources are abstracted as computing units. Therefore, MemoNet can still enjoy the accuracy and computational efficiency of ASIC or FPGA platforms and provide scalability of parallelism.

PIM for data-intensive workloads. Processing in memory architecture integrates computing units within or near memory. The resurgence of PIM and near data processing is motivated by new technologies such as 3D-stacked memory, accelerator use in specific domains, and data-intensive workloads with high degrees of parallelism. The hardware implementation of PIM can be the stacked memory modules like Hybrid Memory Cubes [8], flash-based memory-channel NVDIMM like eXFlash [19], or emerging memory technology like metal-oxide resistive random access memory (ReRAM) [20], Memristor [21], or other types of persistent memory [22].

TABLE 3
The Time Cost to Execute Memolution [5] and MemoNet on 32, 64 and 128 Processing Elements

	Memolution			MemoNet		
	32-PE	64-PE	128-PE	32-PE	64-PE	128-PE
speech-1	19	26	34	2,078	4,673	6,179
speech-2	131	127	131	11,500	12,318	13,265
character	215	226	218	21,180	25,897	29,370
image-processing-1	74	74	78	1,194	1,552	1,909
image-processing-2	49	79	65	8,782	9,544	10,272
image-processing-3	53	77	75	6,471	9,214	24,873
path-planning	328	308	339	421,432	727,653	1,021,432
statistical-modeling	66	69	62	1,045	5,703	22,941
video	32	32	32	10,156	50,345	70,500
remote-sensing	78	80	94	37,188	88,950	148,820
nlp-1	29	35	32	7,006	8,357	10,137
nlp-2	31	31	46	78	156	515
sequence-learning-1	71	82	79	254	363	909
sequence-learning-2	46	62	78	21,980	85,485	200,526
social-network	310	421	538	996,895	3,279,835	10,493,217

There have been a number of studies that focus on the performance improvement of PIM for data-intensive workloads [23], [24], [25], [26]. These works offer solutions to the problem of mapping workloads to PIM computing units. As a general data allocation strategy, our strategy can be applied to different PIM architectures to fully utilize their computing resources. Our strategy can be combined with these PIM frameworks to further capture the characteristics of workloads and obtain a better initial schedule.

System software support for PIM. System software support for emerging PIM architecture have been investigated in the literature. Some techniques are proposed to improve the throughput and reduce the access latency of the I/O [27]. Some other system software techniques target at the architecture that places computation resources near storage [28]. Such PIM architectures also refer to as in-storage computing or near-data processing. There also have been several works for the optimization of memory in the PIM architecture [29]. The design objective is to improve the utilization of main memory and reduce the miss penalty of cache. The optimization techniques for such PIM architectures are normally called in-memory computing. There have been studies for utilizing non-volatile memory in PIM architectures with system software support [30], [31], [32]. The non-volatile memory could be attached to memory bus as persistent memory, or could be incorporated in the architecture as the hybrid memory system.

The above works seek to provide system software support to achieve fast and stable performance. MemoNet focuses on the 3D-stacked PIM architecture that incorporates processing elements and memory. The benefit of this architecture is the fast access to main memory without transferring data from memory bus. MemoNet can be built on top of the such in-memory computing architectures. The major change lies in the modelling of the memory system, which needs to consider access pattern and special characteristics of memory. Therefore, the system abstractions and modelling of applications can still work on top of the device interface and scheduling framework provided by MemoNet.

Scheduling for dataflow applications. Streaming applications and other data-intensive applications are widely found in embedded systems. Streaming applications are commonly modelled as synchronous data flow graphs or directed acyclic graphs [33], [34]. Previous work on scheduling streaming applications onto multicore systems aims to exploit the parallelism of architecture to improve the performance, memory management, energy efficiency, and reliability [35], [36], [37], [38]. Some other techniques focus on the co-optimization of both computation and communication, considering the allocation and synchronization of shared resources [39], [40], [41], [42], [43]. Even though a CNN application can also be represented as a directed acyclic graph, it has many special properties. The scheduling techniques for general directed acyclic graphs may not be applicable to solve the data allocation problem in CNNs.

Chen et al. proposed a CNN accelerator called Eyeriss [10], in which the CNN application is modelled as a directed acyclic graph. It exploits zero valued neurons by using run length coding for memory compression and data gating zero neuron computations to save power. Our MemoNet aims at the utilization of cache for intermediate processing results. Fine-grained parallelism is adopted to hide the latency caused by intermediate processing results, which could also take advantage of the acceleration framework of Eyeriss to further improve energy efficiency.

6 Conclusion

This paper presents a novel data processing framework, called MemoNet, to optimize the memory usage of convolutional neural networks on emerging processing-in-memory architecture. MemoNet exploits fine-grained parallelism to fully utilize the processing engines in PIM and minimizes the data movement for fetching intermediate processing results from off-PE external memory. The data processing framework first generates an initial allocation of convolution operations that jointly considers computing resource and memory usage. An optimal allocation of intermediate processing results is then obtained based on a dynamic programming method. We demonstrate the effectiveness of our approach by using a set of CNN applications on Caffe, an open-source deep learning framework. Experimental results show that the proposed approach can effectively improve processing speed of CNNs and minimize the off-PE data transfers.

In the future, we plan to work on the parameterized compiler design space exploration and obtain accurate models of CNN applications. We also plan to investigate new architectural features for virtualization environments in PIM, which may improve the predictability of memory usage and further boost the processing speed of deep learning applications.

Acknowledgments

The work described in this paper is partially supported by the grants from National Natural Science Foundation of China (61502309), Guangdong Natural Science Foundation (2016A030313045 and 2017B030314073), Shenzhen Science and Technology Foundation (JCYJ20150529164656096 and JCYJ20170302153955969), Natural Science Foundation of SZU (803/000026080154 and 827-000073), and State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences (CARCH201608). This version is a revised version. A preliminary version of this work appears in the Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2017) [5].

References

[1] M. Alwani, H. Chen, M. Ferdman, and P. Milder, “Fused-layer CNN accelerators,” in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit.*, Oct. 2016, pp. 1–12.
[2] A. Graves, et al., “Hybrid computing using a neural network with dynamic external memory,” *Nature*, vol. 538, no. 7626, pp. 1–6, Oct. 2016.
[3] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding,” *ACMMultimedia*, pp. 675–678, 2014.
[4] B. Donyanavard, T. Mück, S. Sarma, and N. Dutt, “SPARTA: Runtime task allocation for energy efficient heterogeneous manycores,” in *Proc. 11th IEEE/ACM/IFIP Int. Conf. Hardw./Softw. Codesign Syst. Synthesis*, 2016, pp. 27:1–27:10.
[5] Y. Wang, M. Zhang, and J. Yang, “Towards memory-efficient processing-in-memory architecture for convolutional neural networks,” in *Proc. 18th ACM SIGPLAN/SIGBED Conf. Languages Compilers Tools Embedded Syst.*, 2017, pp. 81–90.
[6] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, “Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory,” in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit.*, Jun. 2016, pp. 380–392.

[7] JEDEC Wide I/O-2 Standard, 2018. [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd229–2

[8] J. T. Pawlowski, "Hybrid memory cube (HMC)," in *Proc. IEEE Hot Chips 23rd Symp.*, Aug. 2011, pp. 1–24.

[9] Y. LeCun, K. Kavukcuoglu, and C. Farabet, "Convolutional networks and applications in vision," in *Proc. IEEE Int. Symp. Circuits Syst.*, May 2010, pp. 253–256.

[10] Y. H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit.*, Jun. 2016, pp. 367–379.

[11] W. Wen, et al., "A new learning method for inference accuracy, core occupation, and performance co-optimization on TrueNorth chip," in *Proc. 53nd ACM/EDAC/IEEE Des. Autom. Conf.*, 2016, pp. 1–6.

[12] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2015, pp. 161–170.

[13] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "DeepBurning: Automatic generation of FPGA-based learning accelerators for the neural network family," in *Proc. 53nd ACM/EDAC/IEEE Des. Autom. Conf.*, Jun. 2016, pp. 1–6.

[14] K. Guo, et al., "Angel-Eye: A complete design flow for mapping CNN onto embedded FPGA," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 37, no. 1, pp. 35–47, 2018.

[15] T. Chen, et al., "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. 19th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2014, pp. 269–284.

[16] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "YodaNN: An architecture for ultra-low power binary-weight CNN acceleration," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 37, no. 1, pp. 48–60, Jan. 2018.

[17] A. Rahman, J. Lee, and K. Choi, "Efficient FPGA acceleration of convolutional neural networks using logical-3D compute array," in *Proc. Des. Autom. Test Eur. Conf. Exhibition*, Mar. 2016, pp. 1393–1398.

[18] F. Conti and L. Benini, "A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters," in *Proc. Des., Autom. Test Eur. Conf. Exhibition*, Mar. 2015, pp. 683–688.

[19] Lenovo Group Ltd., 2018. [Online]. Available: http://www.lenovo.com/images/products/system-x/pdfs/datasheets/exflash_memory_channel_storage.pdf

[20] P. Chi, et al., "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit.*, Jun. 2016, pp. 27–39.

[21] L. Xia, B. Li, T. Tang, P. Gu, P. Y. Chen, S. Yu, Y. Cao, Y. Wang, Y. Xie, and H. Yang, "MNSIM: Simulation platform for memristor-based neuromorphic computing system," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. PP, no. 99, p. 1, 2017, doi: 10.1109/TCAD.2017.2729466.

[22] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Improving performance and endurance of persistent memory with loose-ordering consistency," *IEEE Trans. Parallel Distrib. Syst.*, vol. PP, no. 99, p. 1, 2017, doi: 10.1109/TPDS.2017.2701364.

[23] B. Akin, F. Franchetti, and J. C. Hoe, "Data reorganization in memory using 3D-stacked DRAM," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Archit.*, Jun. 2015, pp. 131–143.

[24] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *Proc. Int. Conf. Parallel Archit. Compilation*, Oct. 2015, pp. 113–124.

[25] K. Hsieh, et al., "Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit.*, Jun. 2016, pp. 204–216.

[26] M. Gao and C. Kozyrakis, "HRL: Efficient and flexible reconfigurable logic for near-data processing," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, Mar. 2016, pp. 126–137.

[27] R. Chen, Y. Wang, J. Hu, D. Liu, Z. Shao, and Y. Guan, "vFlash: Virtualized flash for optimizing the I/O performance in mobile devices," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 36, no. 7, pp. 1203–1214, Jul. 2017.

[28] J. Wang, D. Park, Y. Papakonstantinou, and S. Swanson, "SSD in-storage computing for search engines," *IEEE Trans. Comput.*, vol. PP, no. 99, p. 1, 2016, doi: 10.1109/TC.2016.2608818.

[29] D. Liu, Y. Lin, P. C. Huang, X. Zhu, and L. Liang, "Durable and energy efficient in-memory frequent pattern mining," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 36, no. 12, pp. 2003–2016, Dec. 2017.

[30] D. Liu, K. Zhong, X. Zhu, Y. Li, L. Long, and Z. Shao, "Non-volatile memory based page swapping for building high-performance mobile devices," *IEEE Trans. Comput.*, vol. 66, no. 11, pp. 1918–1931, Nov. 2017.

[31] F. R. Poursafaei, M. Bazzaz, and A. Ejlali, "NPAM: NVM-aware page allocation for multi-core embedded systems," *IEEE Trans. Comput.*, vol. 66, no. 10, pp. 1703–1716, Oct. 2017.

[32] J. Choi and G. H. Park, "NVM way allocation scheme to reduce NVM writes for hybrid cache architecture in chip-multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 10, pp. 2896–2910, Oct. 2017.

[33] M. H. Foroozannejad, M. Hashemi, T. L. Hodges, and S. Ghiasi, "Look into details: The benefits of fine-grain streaming buffer analysis," in *Proc. ACM SIGPLAN/SIGBED Conf. Languages Compilers Tools Embedded Syst.*, 2010, pp. 27–36.

[34] Y. Wang, Z. Shao, H. C. B. Chan, D. Liu, and Y. Guan, "Memory-aware task scheduling with communication overhead minimization for streaming applications on bus-based multiprocessor system-on-chips," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 7, pp. 1797–1807, Jul. 2014.

[35] J. J. Han, X. Tao, D. Zhu, H. Aydin, Z. Shao, and L. T. Yang, "Multicore mixed-criticality systems: Partitioned scheduling and utilization bound," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 37, no. 1, pp. 21–34, Jan. 2018.

[36] K. Chronaki, et al., "Task scheduling techniques for asymmetric multi-core systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 7, pp. 2074–2087, Jul. 2017.

[37] P.-J. Micolet, A. Smith, and C. Dubach, "A machine learning approach to mapping streaming workloads to dynamic multicore processors," in *Proc. 17th ACM SIGPLAN/SIGBED Conf. Languages Compilers Tools Theory Embedded Syst.*, 2016, pp. 113–122.

[38] K. M. Barijough, M. Hashemi, V. Khibin, and S. Ghiasi, "Implementation-aware model analysis: The case of buffer-throughput tradeoff in streaming applications," in *Proc. 16th ACM SIGPLAN/SIGBED Conf. Languages Compilers Tools Embedded Syst.*, 2015, pp. 11:1–11:10.

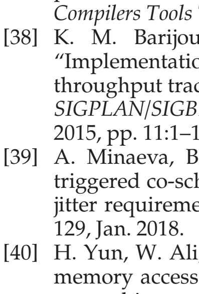
[39] A. Minaeva, B. Akesson, Z. Hanzalek, and D. Dasari, "Time-triggered co-scheduling of computation and communication with jitter requirements," *IEEE Trans. Comput.*, vol. 67, no. 1, pp. 115–129, Jan. 2018.

[40] H. Yun, W. Ali, S. Gondi, and S. Biswas, "BWLOCK: A dynamic memory access control framework for soft real-time applications on multicore platforms," *IEEE Trans. Comput.*, vol. 66, no. 7, pp. 1247–1252, Jul. 2017.

[41] J. Han, X. Tao, D. Zhu, and L. Yang, "Resource sharing in multicore mixed-criticality systems: Utilization bound and blocking overhead," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 12, pp. 3626–3641, Dec. 2017.

[42] J. Huang, R. Li, J. An, D. Ntalasha, F. Yang, and K. Li, "Energy-efficient resource utilization for heterogeneous embedded computing systems," *IEEE Trans. Comput.*, vol. 66, no. 9, pp. 1518–1531, Sep. 2017.

[43] X. Zhu, J. Wu, and T. Li, "Leveraging time prediction and error compensation to enhance the scalability of parallel multi-core simulations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 9, pp. 2553–2566, Sep. 2017.



Yi Wang received the BE and ME degrees in electrical engineering from the Harbin Institute of Technology, Harbin, China, in 2005 and 2008, respectively, and the PhD degree in computer science from the Department of Computing, the Hong Kong Polytechnic University, in 2011. He is currently an associate professor in the College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China. His research interests include embedded systems, non-volatile memory, and real-time scheduling for multi-core systems. He won the best paper award in LCTES 2017 and the best paper award nominee in ASP-DAC 2015.

Weixuan Chen received the BE degree in computer science from Shenzhen University, in 2018. He is currently working toward the master's degree in the College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China. His research interests include system-level design and implementation, real-time systems, and emerging memory techniques for embedded systems.

Jing Yang received the BE and ME degrees in electrical engineering from the Harbin Institute of Technology, Harbin, China, in 2006 and 2008, respectively. She currently an experimentalist at Experimental and Innovation Practice Center, Harbin Institute of Technology, Shenzhen. Before joining Harbin Institute of Technology, she was a senior engineer with State Grid Electric Power Research Institute, from 2008 to 2016. Her research interests include cyber-physical systems, power electronics and drive control techniques, and artificial intelligence in industrial systems.

Tao Li received the graduate degree in computer engineering from Northwestern Polytechnical University, China, in 1993 and the PhD degree in computer engineering from the University of Texas at Austin, in 2004. He is currently a full professor in the Department of Electrical and Computer Engineering, University of Florida. His research interests include computer architecture, microprocessor, memory and storage system design, virtualization technologies, energy-efficient, sustainable/dependable data center, cloud/big data computing platforms, the impacts of emerging technologies and applications on computing, and evaluation of computer systems. He has published more than 90 refereed papers in different journals and conferences in these fields. He co-authored two papers that won the Best Paper Awards in ICCD 2016, HPCA 2011 and six papers that were nominated for the Best Paper Awards in HPCA 2017, ICPP 2015, CGO 2014, DSN 2011, MICRO 2008 and MASCOTS 2006. He is a fellow of the IEEE.