Prometheus: Processing-in-memory Heterogeneous Architecture Design From a Multi-layer Network Theoretic Strategy

Yao Xiao, Shahin Nazarian, Paul Bogdan
Department of Electrical Engineering
University of Southern California, Los Angeles, CA, USA
{*xiaoyao, shahin.nazarian, pbogdan*}*@usc.edu*

***Abstract*—With increasing demand for distributed intelligent physical systems performing big data analytics on the field and in real-time, processing-in-memory (PIM) architectures integrating 3D-stacked memory and logic layers could provide higher performance and energy efficiency. Towards this end, the PIM design requires principled and rigorous optimization strategies to identify interactions and manage data movement across different vaults.**

In this paper, we introduce Prometheus, a novel PIM-based framework that constructs a comprehensive model of computation and communication (MoCC) based on a static and dynamic compilation of an application. Firstly, by adopting a low level virtual machine (LLVM) intermediate representation (IR), an input application is modeled as a two-layered graph consisting of (i) a computation layer in which the nodes denote computation IR instructions and edges denote data dependencies among instructions, and (ii) a communication layer in which the nodes denote memory operations (e.g., load/store) and edges represent memory dependencies detected by alias analysis. Secondly, we develop an optimization framework that partitions the multi-layer network into processing communities within which the computational workload is maximized while balancing the load among computational clusters. Thirdly, we propose a community-to-vault mapping algorithm for designing a scalable hybrid memory cube (HMC)-based system where vaults are interconnected through a network-on-chip (NoC) approach rather than a crossbar architecture. This ensures scalability to hundreds of vaults in each cube. Experimental results demonstrate that Prometheus consisting of 64 HMC-based vaults improves system performance by 9.8x and achieves 2.3x energy reduction, compared to conventional systems.

I. Introduction

The era of big data enables programmers to write memory intensive applications. However, traditional systems are unable to handle big volume of data with fast response as they are designed to execute computations. Therefore, once last level cache miss is generated, data has to be fetched from the main memory via off-chip links. Memory bandwidth becomes a bottleneck for those applications. One technique to address this issue is to bring processing units close to main memory [8]. This was proposed a decated ago, but never succeeded due to design complexity. Nowadays, processing-in-memory (PIM) regains its popularity because 3D-stacking technologies allow memory layers stacked upon each other and connected via TSVs (through-silicon vias). Hybrid memory cube (HMC) provided by Micro [6] is an example of the commercial PIM systems. As shown in Figure 2, according to HMC 2.1 specification, inside one cube, there are eight memory layers and one logic layer stacked on top of each other with 32 partitions, which are also called *vaults*.

However, there are two key challenges required to be addressed to exploit the benefits of PIM systems: (1) *Where should data reside among different vaults to reduce data movement and utilize internal memory bandwidth?* [1] reported that performing 512-way multi-constraint graph partitioning improves performance of the PIM accelerator due to reduced off-chip network traffic. (2) *How to scale up future PIM systems to have hundreds of vaults?* Adopting a data-center-on-a-chip paradigm [3], we address the above-mentioned challenges by (1) formulating the first question as an optimization problem and partitioning the graph to have minimal inter-vault communications; (2) designing a scalable PIM system with NoC to efficiently route packets to the destination vault.

The **goal** of this paper is to find an approach to wisely partition data across different vaults in HMC-based systems to exploit high intra-vault memory bandwidth while improving performance and reducing energy consumption. Therefore, we propose the Prometheus framework taking into account the interactions among computations and communication. First, by adopting an LLVM intermediate representation, dynamic trace generation, reduction, code profiling and graph representation, we describe a C/C++ application as an interdependent two-layer weighted graph, where nodes denote LLVM IR instructions and edges represent the data and control dependencies among LLVM instructions. Moreover, the weights associated with the edges represent the amount of time required for specific computational processes to wait for one another to complete their work. Consequently, one layer represents a model of computation where nodes denote computation operations such as *add* and *mul* while the other layer represents a model of communication where nodes denote memory operations, i.e., load and store. Second, we propose an optimization framework that partitions the two-layer network into highly interacting groups of nodes (clusters) such that the energy consumption required for data movement and accesses is minimized. Third, we introduce a community-to-vault mapping strategy which maps each highly interconnected cluster onto a vault while exploiting the NoC communication infrastructure and the high

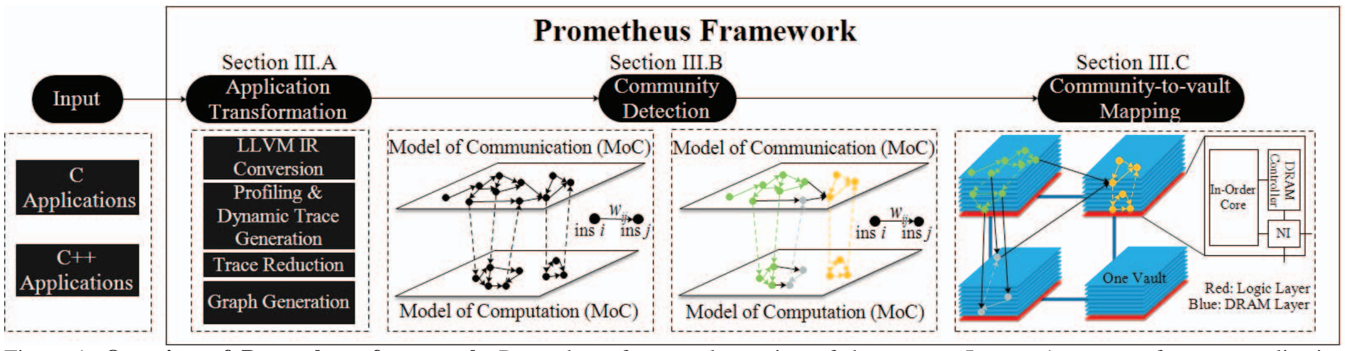


Figure 1: **Overview of Prometheus framework**. Prometheus framework consists of three steps: In step 1, we transform an application into a two-layered graph, one representing model of communication where nodes denote memory operations, a.k.a, load and store, and the other representing model of computation where nodes denote non-memory operations such as *xor* and *zext*. This transformation is performed through code modification, LLVM IR conversion, dynamic trace generation, reduction, profiling, and graph generation. In step 2, we propose an optimization model to better partition the graph into highly connected communities to minimize the energy consumption caused by data access to another community. In step 3, we add a router into the logic layer to form a scalable and efficient NoC substrate and perform community-to-vault mapping.

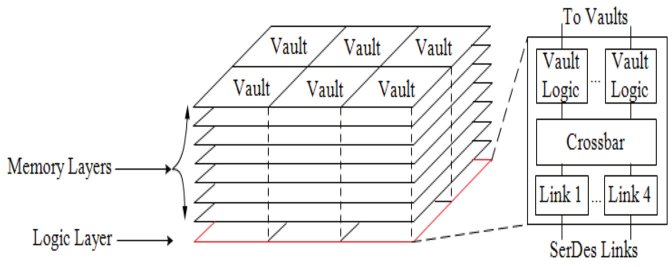


Figure 2: HMC Architecture

internal memory bandwidth provided by TSVs.

The rest of the paper is organized as follows. Section II presents the related work on state-of-the-art PIM and near data processing (NDP). Section III describe the Prometheus framework including application transformation, community detection, and community-to-vault mapping. Section IV summerize several experimental results on performance, NoC traffic, and energy consumption improvement of our framework. Section V concludes for the paper.

II. Related Work

A significant amount of research [1][9][10][11] [12][13][19][26] on PIM and NDP has generated specialized systems for some applications such as graph processing or neural networks. Ahn et al. [1] propose a scalable PIM accelerator Tesseract for parallel graph processing by utilizing the maximum memory bandwidth, communicating between different memory partitions efficiently, and designing a new programming interface to utilize the new hardware design. Chi et al. [5] propose PRIME, a PIM architecture to speed up neural network (NN) applications using ReRAM main memory. PRIME partitions a ReRAM bank into memory/full function (FF)/buffer subarrays. While memory subarrays are only able to store data, FF subarrays have both storage and computation capabilities to calculate forward propagation. Nai et al. [19] present GraphPIM to offload 18 atomic instructions supported by HMC 2.0 into HMC memory systems for graph computing. The approach of offloading is to define a PIM memory region as uncacheable and map host atomic instructions into this region to bypass the cache system. Gu et al. [11] propose a framework for NDP by filtering out extraneous data transferred between CPUs and storage devices. It allows programmers to write applications in a distributed fashion by providing an abstract data communication between the host and storage. Similarly to [23][25], they try to construct dynamic data dependency graphs from input applications but lack a way to differentiate between computations and communications.

However, none of these approaches analyze the impact of data partitioning on performance and energy consumption. In this paper, we formulate an optimization model to partition data and use higher memory bandwidths of PIM (minimize data movement) while balancing loads across vaults.

III. The Prometheus Framework

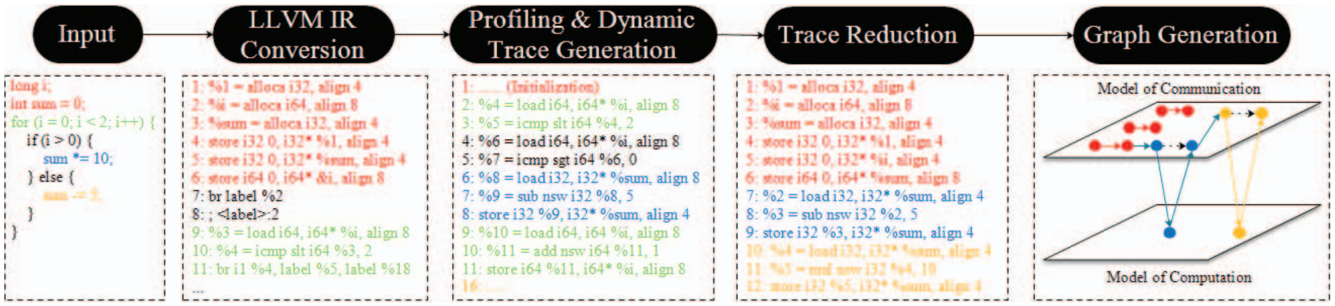
In this section, as shown in Figure 1, we present our Prometheus framework consisting of three steps: (A) Application Transformation; (B) Community Detection; and (C) Community-to-vault Mapping.

A. Application Transformation

Figure 3 shows a high-level logic diagram on how an input C/C++ application is transformed into a two-layer network. First, each input C/C++ application is converted to LLVM IR instructions. We then modify and use Contech [21] to collect dynamic traces of the application and latency for memory operations. Next, we remove all IR instructions corresponding to control statements in C such as *if-else*, *for*, and *while*. Finally, we automatically construct a two-layer network by analyzing data and control dependencies to preserve strict program order and functionality.

1) LLVM IR Conversion: We transform each C/C++ application to its corresponding LLVM IR instructions using the *Clang* compiler: *Clang -emit-llvm -S*.

2) Profiling: We profile the program by instrumenting the lightweight function *rdtsc()* and some inline code before and after each memory operations to get the amount of clock cycles (T) and data size (D). The weights associated with



```
long i;
int sum = 0;
for (i = 0; i < 2; i++) {
    if (i > 0) {
        sum *= 10;
    } else {
        sum -= 5;
    }
}
```

```
1: %1 = alloca i32, align 4
2: %i = alloca i64, align 8
3: %sum = alloca i32, align 4
4: store i32 0, i32* %1, align 4
5: store i32 0, i32* %sum, align 4
6: store i64 0, i64* &i, align 8
7: br label %2
8: ; <label>:2
9: %3 = load i64, i64* %i, align 8
10: %4 = icmp slt i64 %3, 2
11: br i1 %4, label %5, label %18
...
```

```
1: ...... (Initialization)
2: %4 = load i64, i64* %i, align 8
3: %5 = icmp slt i64 %4, 2
4: %6 = load i64, i64* %i, align 8
5: %7 = icmp sgt i64 %6, 0
6: %8 = load i32, i32* %sum, align 8
7: %9 = sub nsw i32 %8, 5
8: store i32 %9, i32* %sum, align 4
9: %10 = load i64, i64 %i, align 8
10: %11 = add nsw i64 %11, 1
11: store i64 %11, i64* %i, align 8
16: ......
```

```
1: %1 = alloca i32, align 4
2: %i = alloca i64, align 8
3: %sum = alloca i32, align 4
4: store i32 0, i32* %1, align 4
5: store i32 0, i32* %i, align 4
6: store i64 0, i64* %sum, align 8
7: %2 = load i32, i32* %sum, align 4
8: %3 = sub nsw i32 %2, 5
9: store i32 %3, i32* %sum, align 4
10: %4 = load i32, i32* %sum, align 4
11: %5 = mul nsw i32 %4, 10
12: store i32 %5, i32* %sum, align 4
```

Figure 3: **Overview of application transformation**. First, we convert a C program to LLVM IR instructions. Second, we profile and execute the instructions in order to collect dynamic traces including computations, the amount of time and data size for CPUs to finish each memory operation. Third, we remove control IR statements by identifying a series of patterns. Fourth, we analyze data and control dependencies between instructions and construct a two-layered graph. Black dotted lines represent memory dependencies detected by alias analysis.

edges in the two-layered network is the product of T and D. The rationale for considering this weighted two-layer network representation is motivated by our goal to partition dependent memory operations into the same vault in order to minimize data movement. This profiling is architecture independent but results can indicate the underlying memory hierarchy: The larger the T and D are, the further away is the data from cores (possibly in LLC or main memory) as data in memory have to be fetched via off-chip links, which is time-consuming. Therefore, we encode data storage and memory hierarchy into weights used in the graph.

3) Dynamic Trace Generation & Trace Reduction: We utilize *Contech* to collect dynamic IR traces. Like full loop unrolling, dynamic traces are aware of how many iterations loops have, leading to fine-grained load balancing when traces are partitioned into clusters. Furthermore, due to the nature of dynamic traces, we are aware of the execution flow of the application and there is no need to store IR instruction corresponding to control statements in C such as *if-else*, *for*, and *while*. Therefore, we perform trace reduction to lower execution overhead by identifying some patterns associated with control statements and removing them. For example, *if* statements have the following structure: the type of the first instruction is *load* and the second instruction dependent on the first one is *icmp*. As long as we find such pattern **in a basic block consisting only of two instructions**, we remove this basic block. As illustrated in Figure 3, lines 2 and 3 in the third file correspond to the *for* statement. We check whether this basic block has only two instructions in which one represents *load* while the other denotes *icmp* that depends on the first *load* instruction. If all requirements are satisfied, we remove lines 2 and 3 as indicated in the fourth file without green colored texts.

4) Graph Generation: We encode communications, computations, and their interconnected dependencies by constructing two-layered graphs where one layer represents the model of communication, and the other one denotes the model of computation. Nodes in one layer denote computations whereas nodes in the other layer denote communications. Edges in the communication layer are detected by alias analysis[1]

[1]In LLVM, we perform alias analysis using -basicaa -aa-eval -print-all-alias-modref-info.

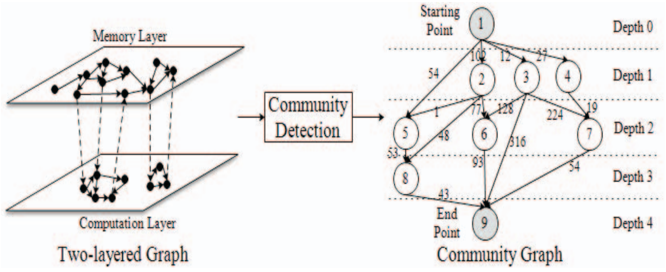


Figure 4: Building on the two-layered graph we generated in Section A, we partition the graph into interdependent communities representing a set of IR instructions to be executed in sequential.

whereas the rest of edges are analyzed by data and control dependencies. A formal description of the two-layer network representation of an application is provided in section III.B (see definition 1). As shown in Figure 3, except lines 8 and 11, which are computation nodes, the rest are nodes in the space of the communication layer. We analyze dependencies in two phases: During the first phase, we analyze data and control dependencies among instructions, which corresponds to non-dotted edges in Figure 3. During the second phase, we perform alias analysis to connect different subcomponents in the communication layer into one graph. Black dotted edges in Figure 3 demonstrate this connection.

B. Identification of Processing Communities

Community detection in networks is a technique to find groups of vertices which have higher probability of connection with each other than vertices in other groups [24]. Therefore, in this paper, we adopt the community detection idea and partition the two-layered graph into interconnected processing communities which are next mapped to vaults. Thus, we build a community graph, which is similar to task graph, where nodes represent communities including a series of instructions executed sequentially while edges and their weights represent dependencies and communication cost between communities and encoding concurrent interactions. Therefore, the goal of this section is to formulate a mathematical optimization model and partition the graph into communities while balancing the load among communities.

Before formulating the optimization model, we introduce two formal definitions for input and output graphs:

Definition 1: A two-layered graph TG is a weighted directed graph $\mathcal{G}_1 = TG(n_i, l_i, e_{ij}, w_{ij} | i, j \in \{1, ..., N\}; l_i \in \{1, 2\})$ where nodes n_i in the layer l_i (1 or 2) represent memory or non-memory instructions; edges e_{ij} represent dependencies found by alias analysis in the memory layer ($l_i = 1$) and data/control dependencies; edge weights w_{ij} represent latency times data size for memory instructions.

Definition 2: A community graph CG is a weighted directed graph $\mathcal{G}_2 = CG(V_i, d_i, E_{ij}, W_{ij} | i, j \in \{1, ..., N\})$ where nodes V_i represent a series of IR instructions to be executed in sequential, which are called communities; edges E_{ij} represent dependencies between communities; edge weights W_{ij} represent communication cost from one node i to another j. Depth d_i represents the *largest* number of hops node i takes to the root which is considered as a starting point[2].

Based on these definitions, we formulate the following optimization problem: **Given** a two-layered graph TG, **find** communities which **maximize** the following function

$$\max_{C_i, C_j} \quad F = Q - R \tag{1}$$

$$Q = \frac{1}{2W} \sum_{ij} [W_{ij} - \frac{s_i s_j}{2W}] \delta(C_i, C_j) \tag{2}$$

$$R = \frac{\alpha}{2W} \sum_{\substack{1 \le u \le n_c \\ 1 \le v \le n_c \\ u \ne v}} |W_u - W_v| \delta(d_u, d_v) \tag{3}$$

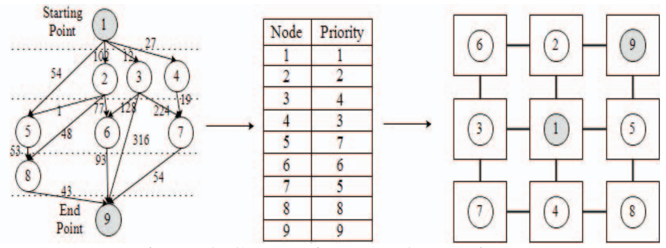
where W is the sum of the total weights in TG; W_i is the sum of weights in the community i; W_{ij} is the weight between nodes i and j; s_i, the strength of a node i, is the sum of the weights of edges adjacent to the node i; C_i is the community to which node i belongs; n_c is the number of communities; d_i is the depth of community i; $\delta(i, j)$ equals 1 when $i = j$; α controls the importance of load balancing.

The function Q measures the difference between the sum of weights within a community $W_{in} = \sum_{ij} W_{ij} \delta(C_i, C_j)$ and that adjacent to the community $W_{adj} = \sum_{ij} \frac{s_i s_j}{2W} \delta(C_i, C_j)$. By maximizing F, Q should also be maximum. Therefore, W_{in}, which represents workload in a community, increases and W_{adj}, representing communication cost decreases. Therefore, data movement is confined almost within each community. The function R quantifies the load balancing at any depth. As shown in Figure 4, communities at the same depth can be executed in parallel. Therefore, we need to make sure the loads in those communities should be balanced to reduce the overhead of core idle waiting. W_u computes the load in the community u and $\delta(d_u, d_v)$ ensures that communities u and v are at the same depth. Hence, in order to maximize F, R should be minimized, enforcing W_u and W_v to be nearly equalized at the same depth ($\delta(d_u, d_v) = 1$).

C. Community-to-vault Mapping

In this section, based on the community graph, we aim to build a scalable PIM system and map communities to vaults.

[2]Note that the depth of node 5 should be 3 rather than 2 because the longest path is $\{1, 2, 5, 8, 9\}$. The depth can be found using levelization.



Node	Priority
1	1
2	2
3	4
4	3
5	7
6	6
7	5
8	8
9	9

Figure 5: Community-to-vault mapping

1) Scalable PIM System: Some memory-intensive applications require more memories to store huge amount of data. Therefore, in order to increase memory capacity in PIM systems, more HMCs are utilized and connected via high-speed Serializer/Deserializer (SerDes) links to provide high memory bandwidth. However, SerDes links consume almost half of HMC's power [1][19][20]. In order to save energy wasted on SerDes links, we propose a scalable PIM system with NoC in the logic layer to efficiently route packets to the destination vault instead of the crossbar used in HMC as shown in Figure 1. Therefore, in order to have more memories, we simply add extra vaults with routers to this design instead of connecting it to HMCs via SerDes links.

2) Mapping: We propose Algorithm 1 to map communities detected in Section B to available vaults in the scalable PIM system. First, we rank the priorities of communities by first assigning higher priorities to communities at the lower depth. For example, in Figure 5, the starting community in the depth 0 gets the highest priority. If communities are at the same depth, assign higher priorities to one with the higher communication cost. For example, communication costs for communities 3, 4, and 5 at the depth 1 are 102, 12, 27 respectively, then the priority order in this depth should be $3 > 5 > 4$. After priority assignment, we map communities onto NoC in a greedy way as more important communities (with higher priorities) should take up the better location, which is the center of the chip as shown in Figure 5.

IV. Evaluation

A. System Configuration

1) DDR3: We utilize 64 in-order cores to model a DDR3-based system. Each core has a 64KB L1 private cache and a 256KB distributed L2 shared cache as shown in Table 1, with a memory controller to connect to memory subsystem, i.e., DDR3. This system is the baseline for our evaluation.

2) HMC: Table 1 shows configuration parameters of our evaluated scalable HMC-based system, which includes 64 vaults with eight memory layers and one logic layer. In the logic layer, one vault consists of the same cores used in the DDR3-based system and NoC to connect them. To further evaluate different data partitioning schemes, we apply METIS [16], a multi-way graph partitioning, and our proposed community detection (CD) into clusters to be mapped onto different vaults in our system.

B. Simulation Configuration

We use Contech [21] as the frontend functional simulator to generate dynamic LLVM traces from C/C++ applications,

Algorithm 1 Community-to-vault Mapping Algorithm

```
Input: The community graph (CG)
Output: A set of tuples $T$ indicating which community maps
    to which vault
1:  /* Priority Assignment */
2:  PriorityQueue = ()
3:  for nodes in each depth do
4:      Sort nodes by comm costs in the descending order
5:      PriorityQueue.append(nodes)
6:  end for
7:  /* Community-to-vault Mapping */
8:  Mapping = ()
9:  for node $\in$ PriorityQueue do
10:     if node is the starting point then
11:         place = the center of the mesh-based NoC
12:     else
13:         place = closest to the parent node it depends (Greedy)
14:     end if
15:     Mapping.append((node, place))
16: end for
```

Table I: Configuration parameters

Processor	Cores	In-order, 16 MSHRs
	L1 private cache	64KB, 4-way associative 32-byte blocks
	L2 shared cache	256KB, distributed
Memory Layer	Configuration	16 GB cube, 64 vaults 8 DRAM layers
	Internal Bandwidth	16 GB/s per vault
	DRAM Timing	DDR3-1600
	Scheduling Policy	FR-FCFS
Network	Topology	Mesh
	Routing Algorithm	XY routing
	Flow Control	Virtual channel flit-based

write a compiler-based parser to construct a two-layered graph and perform community detection to partition the graph into clusters. We model 3D-stacked memory layers that follow the 2.1 specification [6] using ASMSim [22] and NoC communication substrate using Booksim2 [15] as backend timing simulators. Both simulators are cycle-accurate and trace-based. (Booksim2 supports Netrace traces as simulation input.) Table 2 lists the 7 benchmarks we use to validate the system.

For our energy evaluation, we model the energy consumption of caches in cores using CACTI 6.0 [18] and compute the energy of memory layer access, which is 3.7 pJ/bit [20] assuming memory operations dominate. Next, following [14], we derive the total energy consumption of a transaction from node i to node j described as follows: $E_{ij} = N(n_{hops}E_{router} + (n_{hops}-1)E_{flit})$ where N, n_{hops}, E_{router}, and E_{flit} represent

Table II: Benchmarks and descriptions

Benchmark	Description	Source
BS	Calculate European options	PARSEC[2]
SC	Solve the online clustering problem	PARSEC[2]
BP	Back-propagation	Rodinia[4]
KM	K-means	Rodinia[4]
MD	Simulate molecular dynamics	OmpSCR[7]
FFT	Compute Fast Fourier Transform	OmpSCR[7]
MDB	Calculate Mandelbrot Set	OmpSCR[7]

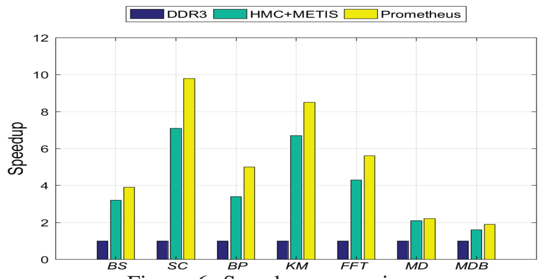


Figure 6: Speedup comparison

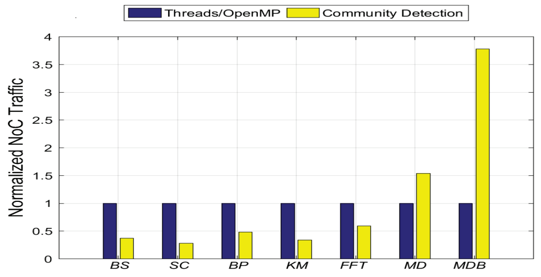


Figure 7: NoC traffic with different parallelism approaches

the number of bits to be transferred, the number of hops, energy consumption of routers and flit transfer respectively. we assume that interconnect consumes 2 pJ/bit for flit transfer E_{flit} and 1.5 pJ/bit for routers to process flits E_{router} [17].

C. Experimental Results

1) Performance: Figure 6 compares the speedup between DDR3 and HMC-based systems. The embarrassingly parallel application (i.e. *MDB*) and applications such as *MD* and *BS* may not benefit too much from PIM due to low off-chip memory bandwidth usage compared to what PIM could provide. Therefore, speedup improves only at most 4x compared to the DDR3-based system. However, if applications such as *SC* require high off-chip memory bandwidth, then compared to DDR3-based systems which could only provide no more than 100 GB/s, our proposed HMC-based system could provide 1TB/s. Distinction is more pronounced if we increase the number of vaults per cube. Therefore, the speedup improvement for *SC* is 9.8x as high as DDR3-based systems.

In HMC, we adopt METIS and CD to partition the graph into interconnected clusters. However, for embarrassingly parallel programs, our graph representation cannot guarantee that clusters after graph partitioning are independent to each other. Therefore, the performance improvement of applications such as *MD* and *MDB* is at most 1x compared to DDR3-based systems where it is easy to parallelize using threads. Nevertheless, our graph partitioning scheme outperforms METIS because this scheme tries to minimize communication while balancing the load.

2) NoC Traffic: Figure 7 illustrates the normalized NoC traffic with respect to threads/OpenMP and our proposed graph partitioning. For applications such as *MD* and *MDB*, there are few data dependencies among threads while clusters are interconnected in our graph representation. Therefore, NoC traffic for those applications degrades somewhat compared to threads running almost independent on cores. However,

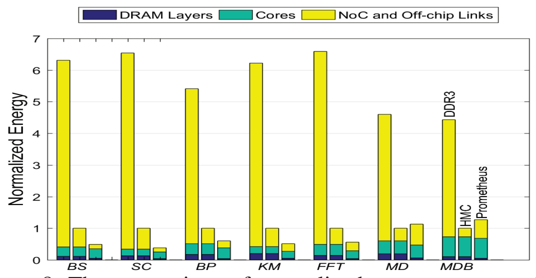


Figure 8: The comparison of normalized energy consumption

for most non-embarrassingly parallel applications, community detection tries its best to confine data movement within a cluster, leading to lower energy.

3) Energy Consumption: Figure 8 shows the comparison of normalized energy consumption among DDR3, HMC, and Prometheus systems. Computations should remain the same (green bar in Figure 8) for all systems, while HMC improves energy consumption regarding off-chip links compared to DDR3 as HMC has higher off-chip memory bandwidth and shorter distance between cores and memory, causing shorter execution time. Prometheus further improves energy consumption compared to HMC as we apply community detection to partition the graph into clusters to minimize data communications between vaults. In other words, NoC traffic is reduced for most applications except *MD* and *MDB*, thus energy consumption for NoC (yellow bar in Figure 8) improves a lot.

V. Conclusion

In this paper, we present *Prometheus*, an optimization framework to find the best data partitioning scheme for PIM systems to improve the performance and energy consumption. *Prometheus* exploits the high memory bandwidth ($\sim$ 1TB/s) of PIM systems by (1) representing each application as a two-layered graph where in the computation layer, nodes denote computation instructions and edges denote data dependencies; in the communication layer, nodes denote load/store instructions and edges are formed by alias analysis. (2) performing community detection to find interconnected clusters ensuring that data movement is almost confined within each cluster and workloads among clusters are balanced. (3) designing a scalable PIM system where vaults are connected via NoC rather than crossbar and mapping clusters to vaults in a greedy fashion. Our evaluation with 64 vaults and one in-order core per vault demonstrates that performance improvement is 9.8x and 1.38x as high as traditional DDR3-based systems and PIM systems with METIS graph partitioning repectively. Energy consumption improvement is 2.3x, compared to PIM system without community detection as *Prometheus* tries to reduce NoC traffic between different vaults.

VI. Acknowledgement

The authors gratefully acknowledge the support by the National Science Foundation under CAREER Award CPS-1453860, the U.S. Army Defense Advanced Research Projects Agency (DARPA) under grant no. W911NF-17-1-0076, DARPA Young Faculty Award under grant no. N66001-17-1-4044, the Okawa Foundation Award and the University of Southern California support.

References

[1] J. Ahn et al. "A scalable processing-in-memory accelerator for parallel graph processing". In: *ISCA*. 2015.
[2] C. Bienia et al. "The PARSEC benchmark suite: Characterization and architectural implications". In: *PACT*. 2008.
[3] P. Bogdan. "Mathematical modeling and control of multifractal workloads for data-center-on-a-chip optimization". In: *NOCS*. 2015.
[4] S. Che et al. "Rodinia: A benchmark suite for heterogeneous computing". In: *IISWC*. 2009.
[5] P. Chi et al. "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory". In: *ISCA*. 2016.
[6] Hybrid Memory Cube Consortium. "Hybrid memory cube specification 2.1". In: 2013.
[7] A. J. Dorta et al. "The OpenMP source code repository". In: *PDP*. 2005.
[8] J. Draper et al. "The architecture of the DIVA processing-in-memory chip". In: *ICS*. 2002.
[9] M. Gao et al. "Practical near-data processing for in-memory analytics frameworks". In: *PACT*. 2015.
[10] M. Gao and C. Kozyrakis. "HRL: efficient and flexible reconfigurable logic for near-data processing". In: *HPCA*. 2016.
[11] B. Gu et al. "Biscuit: A framework for near-data processing of big data workloads". In: *ISCA*. 2016.
[12] K. Hsieh et al. "Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation". In: *ICCD*. 2016.
[13] K. Hsieh et al. "Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems". In: *ISCA*. 2016.
[14] J. Hu and R. Marculescu. "Energy-and performance-aware mapping for regular NoC architectures". In: *IEEE TCAD*. 2005.
[15] N. Jiang et al. "A detailed and flexible cycle-accurate network-on-chip simulator". In: *ISPASS*. 2013.
[16] G. Karypis and V. Kumar. "A fast and high quality multilevel scheme for partitioning irregular graphs". In: *SISC* (1998).
[17] G. Kim et al. "Memory-centric system interconnect design with hybrid memory cubes". In: *PACT*. 2013.
[18] N. Muralimanohar et al. "CACTI 6.0: A tool to model large caches". In: *HP Laboratories*. 2009.
[19] L. Nai et al. "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks". In: *HPCA*. 2017.
[20] S. H. Pugsley et al. "NDC: Analyzing the impact of 3D-stacked memory+ logic devices on MapReduce workloads". In: *ISPASS*. 2014.
[21] B. P. Railing et al. "Contech: Efficiently generating dynamic task graphs for arbitrary parallel programs". In: *TACO*. 2015.
[22] L. Subramanian et al. "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory". In: *MICRO*. 2015.
[23] Y. Xiao et al. "A Load Balancing Inspired Optimization Framework for Exascale Multicore Systems: A Complex Networks Approach". In: *ICCAD*. 2017.
[24] Y. Xue and P. Bogdan. "Reliable Multi-Fractal Characterization of Weighted Complex Networks: Algorithms and Implications". In: *Scientific Reports* (2017).
[25] Y. Xue and P. Bogdan. "Scalable and realistic benchmark synthesis for efficient NoC performance evaluation: A complex network analysis approach". In: *CODES+ISSS*. 2016.
[26] D. Zhang et al. "TOP-PIM: throughput-oriented programmable processing in memory". In: *HPDC*. 2014.