PIMProf: An Automated Program Profiler for Processing-in-Memory Offloading Decisions

Yizhou Wei*, Minxuan Zhou†, Sihang Liu*, Korakit Seemakhupt*, Tajana Rosing†, and Samira Khan*
*University of Virginia, †University of California San Diego
Email: {yizhouwei, sihangliu, korakit, samirakhan}@virginia.edu, {miz087, tajana}@ucsd.edu

***Abstract*—Processing-in-memory (PIM) architectures reduce the data movement overhead by bringing computation closer to the memory. However, a key challenge is to decide which code regions of a program should be offloaded to PIM for the best performance. The goal of this work is to help programmers leverage PIM architectures by automatically profiling legacy workloads to find PIM-friendly code regions for offloading. We propose PIMProf[1], an *automated* profiling and offloading tool to determine PIM offloading regions for CPU-PIM hybrid architectures. PIMProf efficiently models the comprehensive cost related to PIM offloading and makes the offloading decision by an effective and computational-tractable algorithm. We demonstrate the effectiveness of PIMProf by evaluating the GAP graph benchmark suite and the PARSEC benchmark suite under different PIM and CPU configurations. Our evaluation shows that, compared to the CPU baseline and a PIM-only configuration, the offloading decisions by PIMProf provides 5.33× and 1.39× speedup in the GAP graph workloads, respectively; 2.22× and 1.74× speedup in the PARSEC benchmarks, respectively.**

I. Introduction

Modern workloads, such as graph processing, machine learning, and big data analytics, have increasingly higher demand on memory. Therefore, recent works move computation closer to memory and design different *processing-in-memory* (PIM) architectures to relieve the pressure on main memory bandwidth. For example, some works implement a large number of simple and low-power processors in memory to accelerate general-purpose workloads [1]–[3]; some other works design specialized cores to accelerate certain workloads or computation kernels [4]; it is also viable to employ in-situ bulk logic inside memory arrays, where the in-memory logic only supports simple operations (e.g., bitwise operators) but can utilize the massive internal memory bandwidth [4]–[6]. Figure 1 shows a typical PIM architecture [1, 2, 4], where we integrate processing elements near the memory (e.g., logic die of Micron's hybrid-memory cube (HMC) [7]). The processing units are able to process complex operations without communicating with the host CPU. With the different execution patterns enabled by PIM operations in conventional systems, the immediate question arises—how can one determine which code region to offload to the memory side in order to fully exploit the benefits of PIM?

A common solution in previous works is to offload predetermined procedures that can be accelerated by PIM, such as special instructions [3, 8, 9] or compute kernels [2]. However, a more generic solution is to have general code regions offloaded to PIM,

[1]The source code of PIMProf can be found at https://github.com/Systems-ShiftLab/PIMProf

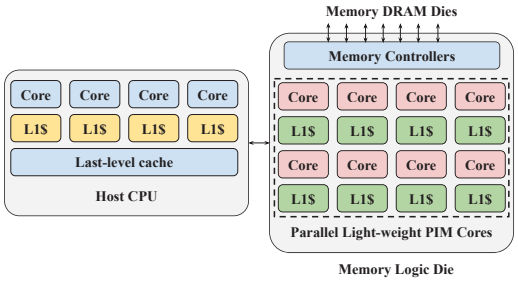

Fig. 1. High-level structure of a processing-in-memory (PIM) architecture.

in order to reap the benefits from PIM, such as high bandwidth and parallelism. Such a general offloading scheme, nonetheless, adds complexities to the offloading decision, as there are extra switching costs between PIM and CPU, such as data movement between PIM and CPU due to data dependency and OS-level context switching. To decide how a general program can best benefit from PIM offloading, there are two major challenges. First, we need a comprehensive model of the costs due to different offloading decisions, where modeling the extra cost of switching between CPU and PIM during runtime is the key. Second, even with a cost model, it is still hard to make the offloading decision for multiple regions when the switching cost is involved, as the switching cost of one region depends on others. Thus, the challenge is how we can efficiently explore offloading decisions. In this work, we implement PIMProf, an automated PIM profiling and offloading tool for general programs running on CPU-PIM hybrid architectures. The contributions of this work are the following:

- This is the first work that designs a profiler to automatically determine PIM candidates in a general program for PIM architectures, with different practical offloading overheads taken into account. PIMProf tackles the two challenges with: (1) an efficient cost modeling for PIM-offloaded programs, and (2) an effective and computational-tractable heuristic-based algorithm for offloading decision-making.
- We demonstrate the effectiveness of PIMProf by evaluating a graph benchmark suite, GAP [10], and another more general benchmark, PARSEC [11]), under different PIM and CPU configurations.
- In GAP benchmarks, which are dominated by memory-intensive PIM-friendly kernels, PIMProf shows that many CPU-friendly regions are offloaded to PIM along with PIM-friendly regions to reduce data transfer overhead between them. PIMProf provides 5.33×/1.39× speedup over

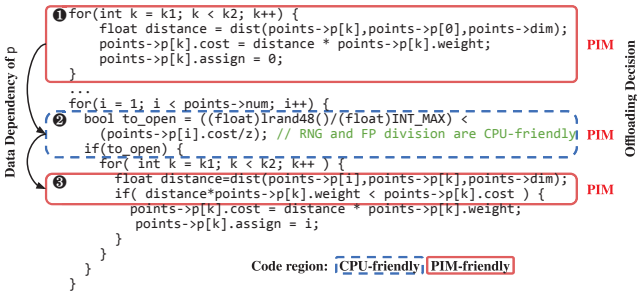

Fig. 2. Streamcluster offloading decision with and without data dependency.

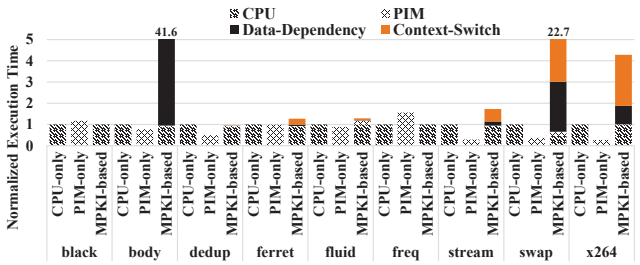

Fig. 3. Offloading performance based on MPKI and parallelism for PARSEC benchmarks, as compared to CPU-only and PIM-only offloading.

CPU/PIM-only configuration for GAP on average.

- Our evaluation on PARSEC shows that only a few workloads have enough memory intensity and parallelism to be able to exploit the benefits of PIM. PIMProf provides 2.22×/1.74× benefit over CPU/PIM-only configurations for PARSEC workloads on average, and shows major speedup for four out of nine workloads we experimented with.

II. Challenges in PIM Offloading

Processing-in-memory (PIM) architectures overcome the memory wall problem [12] by placing computation units close to or within the memory device. Because of the abundant internal bandwidth and parallelism, PIM architectures are efficient in performing memory-intensive and highly-parallel procedures (e.g., specialized instructions or computation kernels). Therefore, a simple strategy of exploiting advantages of PIM can be offloading code regions that meet these characteristics. For example, code regions with a high cache miss rate—typically measured as misses per kilo instructions (MPKI)—can be accelerated by leveraging the high internal bandwidth in PIM. However, in practice, hybrid CPU-PIM execution involves two main categories of additional switching costs between CPU and PIM: the cost from extra memory movement (*data dependency cost*) and the cost from extra context switch (*context switch cost*). Therefore, a naive MPKI- or parallelism-based offloading solution may not perform well, especially in real-life workloads with complicated dependencies.

Figure 2 shows a snippet of code from *Streamcluster*, a workload in the PARSEC benchmark suite [11]. The three code regions are part of a function for computing the approximated *k*-Median. Based on the MPKI and parallelism, both region ❶ and ❸ are PIM-friendly, whereas region ❷, which involves random number generation and floating-point division, is CPU-friendly. However, the data dependency across regions can affect the overall execution time, as switching between CPU and PIM processors incurs data writeback from CPU cache to PIM. In this example, since region ❷ has data dependency with both region ❶ and ❸, executing ❷ along with ❶ and ❸ on PIM minimizes the data transfer overhead between CPU and PIM, therefore, offering better performance.

Figure 3 shows the performance of PIM offloading based on MPKI and parallelism for nine PARSEC workloads (detailed methodology in Section IV). We use configurations of CPU-only and PIM-only execution as the baselines and normalize all results to CPU-only. This experiment shows that MPKI-based and parallelism-based offloading schemes cannot improve the performance over the baselines in many cases. Even worse, MPKI-based offloading significantly degrades the performance of CPU-only offloading for several workloads (e.g., 41.6× slower for *Bodytrack* and 22.7× slower for *Swaptions*). The reason behind such performance degradation is the extremely large overhead of data dependency and context switch caused by offloading consecutive code regions to different platforms.

The motivational results demonstrate that PIM offloading should consider not only memory access cost and parallelism but also the data dependency cost. However, making offloading decisions based on the cost of data dependency is challenging, which highly depends on the execution of prior code regions. Therefore, it is extremely hard for programmers to manually find out PIM-friendly code regions, even with tools that profile the cache miss status (e.g., Intel VTune [13] that reads performance counters and Pin tool [14] that simulates the cache hierarchy).

III. PIMProf Design

The goal of this work is to tackle the PIM-offloading problem: For a given program, what regions should be offloaded to the PIM side in a CPU-PIM heterogeneous architecture. There are two significant challenges: First, comprehensive cost modeling of offloading decisions can be complicated due to an extremely large number of data interactions in general programs. Furthermore, it is hard to design a computational-tractable algorithm to explore all possible offloading candidates to minimize the holistic cost. To overcome these challenges, we provide PIMProf, an end-to-end tool that automatically generates offloading decisions for general programs running on CPU-PIM architectures. The workflow of PIMProf is shown in Figure 4.

A. Program Instrumentation

PIMProf statically instruments the program using an LLVM [15] compiler pass. It divides the entire program into small regions to enable fine-grained profiling by inserting lightweight marker functions, and the program regions are used as the basic unit for profiling and offloading. The granularity of program regions is configurable so that applications that vary in size, parallelism and data dependency patterns may all benefit from offloading. In our evaluation, we choose two granularities: the basic-block-grained [16] offloading, which usually works better for programs whose PIM-friendliness changes frequently;

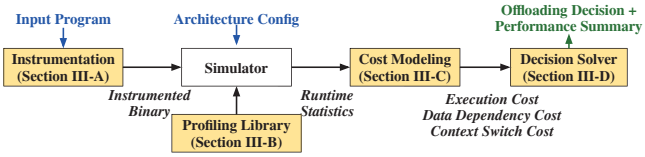

Fig. 4. PIMProf overview.

and the function-grained offloading, which is suitable for larger-scale programs whose PIM-friendliness changes less frequently, by sacrificing some potential optimization opportunities.

B. Runtime Profiling

Based on the automatically-instrumented region boundaries (e.g., basic block or function), PIMProf collects the runtime statistics for each region, including execution time, cache hit rate of memory locations accessed by this code region, and number of instructions in this region. As real PIM hardware is mostly under academic and industry research, PIMProf collects the runtime information from a simulator (Intel's Sniper [17]). To compare the offloading benefits, PIMProf simulates the execution on both the PIM and CPU architectures. With the runtime statistics for both processor types, the next step is to model the cost.

C. Cost Modeling

In order to provide good offloading decisions for a program, it is necessary to fully understand the cost model of different program executions for minimizing the total cost. Although the execution time depends on multiple factors, it is possible to consider those factors separately and then generate the overall cost. We identify two major sources of cost: the execution cost which is due to the execution of the code region (either on CPU or PIM) and the switching cost which is the overhead for maintaining the consistency of data and the program context when switching between CPU and PIM.

1) Execution cost: As PIMProf has collected the statistics about the runtime information of executing the code region both on CPU and PIM, the execution cost directly comes from the runtime profile. In Section IV-A, we elaborate on the details of simulation used for modeling both CPU and PIM architectures.

2) Switching cost: The switching cost comes from two sources. The first source is the data dependency between code regions that are placed on different processing units, e.g., one region on CPU and another on PIM. The second source is the context switch, which mainly includes the overhead of saving and restoring the processor states [18]. Different from the data dependency cost, context switch has a more or less constant cost, which is determined by the operating system.

Data dependency Cost. We analyze the data dependency between program regions at *cache line granularity*, as memory transfer is cache-line-grained. When the same cache line of data is shared by two program regions that execute in different places (PIM and CPU), we model a single data transfer as the total cost of one cache line flush issued by the source, plus one cache line fetch issued by the destination. Figure 5a shows a code example with multiple program regions. Assume that we execute region 0 on CPU and region 1 on PIM, and variables `a` and `b` are stored in different cache lines, then when we switch

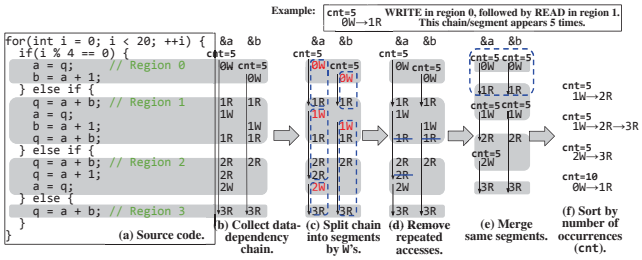

Fig. 5. Optimizations to the data dependency chain.

between them, the memory locations of `a` and `b` each incur a data dependency cost. Because the values of `a` and `b` are both updated by a WRITE in region 0, the CPU needs to flush the updates back to memory and then the PIM unit fetches the updated data from memory. The number of data dependency instances increases if there are multiple shared locations. To compute the total data dependency cost of a certain offloading decision, a naive way is to go over all memory accesses, and increase the cost wherever a data transfer happens. However, this is not feasible for a real-world program as we need to iterate over all possible decisions to find the best one. As a solution, we apply a few optimizations to provide a good offloading decision with reasonable overhead, as described in Section III-D.

Context Switch Cost. The context switch cost appears when two neighboring regions are executed in different places. This cost is usually constant, depending on the operating system. To compute this type of cost, PIMProf keeps track of how many times the program goes across the boundary of one region to another when executing the program by using a *weighted directed graph*. The weight of each edge is the number of times the execution goes from one region to another. Figure 6a shows an example of the context switch graph.

D. Solver for offloading decisions

The target of the solver is to find an offloading decision for each region that minimizes the sum of the execution cost, the data dependency cost, and the context switch cost. The solver generates offline decisions so each region will not change its place of execution during the runtime. Without the data dependency cost, finding an offloading decision that minimizes the other costs is straightforward. However, it becomes challenging when the data dependency cost is considered. This section discusses how we optimize the computation of data dependency cost and incorporate this method into the solver of PIMProf.

Model data-dependency as chains. To formalize the data dependency of all memory accesses, we model them as the *data dependency chains*, shown in Figure 5b. For each cache line, its data dependency chain records the information of all memory accesses to it, including the region ID that the access occurs and the access type (READ/WRITE). An access to different addresses but within the same cache line is recorded to the same chain. Created in this way, all accesses to a cache line are logged in a single data dependency chain. Though functioning, this method is not storage-efficient. Next, we perform several optimizations to reduce its storage overhead.

Split data dependency chains into segments. There are a few observations we can use to remove redundant information

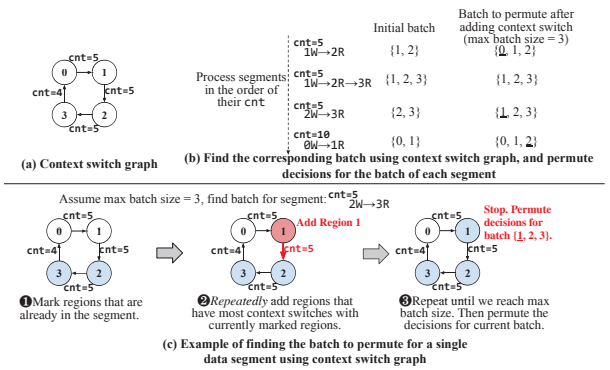

Fig. 6. Permute small batches of regions created from dependency segments and context switch to find offloading decision that minimizes total cost.

in data dependency chains. First, data transfer will only occur when the program performs a READ in one place but the data was previously updated by a WRITE in the other place, i.e., read-after-write (RAW) in different places. There will be no data transfer if all READs between two consecutive WRITEs are executed in the same place as the first WRITE. And, the second WRITE will overwrite the old value, breaking the dependency chain. Thus, the follow-up READs may only depend on the second WRITE. Therefore, we divide the dependency chain into shorter segments that start with a WRITE and end with the READ before the next WRITE (Figure 5c). In this way, the total data dependency cost is equal to the sum of the data dependency cost of all segments. Each segment incurs a cost of at most one cache line flush and one cache line fetch, when the segment contains regions offloaded to different places. Second, subsequent READs from the same region as the first READ/WRITE do not change the data dependency cost of the segment. This is because the place (PIM or CPU) that has executed the first READ/WRITE already holds the latest data. Thus, subsequent READs from the same region will not trigger extra data transfer and can be removed from the segment (Figure 5d). Third, since PIMProf tracks the data dependency at cache line granularity, RAW to a cache line in different places will always trigger one extra cache line flush and one cache line fetch. By assuming that this cost is the same for all cache lines, PIMProf merges the same segments (segment with the same sequence of memory operations and region IDs) from different cache lines (Figure 5e), and tracks the number of occurrences using a counter (`cnt` in Figure 5).

Heuristic decision-solving algorithm. When finding the minimal cost, changing the decision of one offloaded region would cause the dependency cost with other regions to change. Thus, a trivial algorithm that permutes all possible decisions will have a complexity of $O(2^n)$, where n is the number of regions. In a practical program, this complexity cannot be solved within a reasonable amount of time[2]. Therefore, instead of using general-purpose solvers for optimization problems, we create a heuristic algorithm based on our observation: The data dependency segments are usually short (most segments only have 2-3 regions in our experiment) because data remain in cache for a limited time before it is evicted or flushed to memory, such that it is feasible to test all decision combinations for regions in a single segment. Based on this observation, PIMProf uses a heuristic method that permutes the decisions in small batches of regions[3], while keeping the other decisions the same, and check if any of those changes in the decisions reduces the total cost. During permutation, the same region can appear in multiple batches. PIMProf starts from segments with fewer occurrences so that the decisions of more important segments can have a chance to overwrite previous decisions (sort the segments as shown in Figure 5f and 6b). As a result, subsequent decisions that reduce the total cost can have a chance to overwrite previous ones. When creating the batch, PIMProf needs to consider both the data dependency and the context switch. PIMProf starts by initializing each batch with all regions in one segment that have data dependencies. Then, to take related context switch into account (Figure 6a), it keeps adding new regions that have a context switch to/from the existing regions in the batch, until the current batch reaches a pre-set threshold (15 regions in our setup). Figure 6b shows the initial batches and the resulting batches after adding related context switches; Figure 6c demonstrates an example that creates the batch for segment `2W→3R`. After creating all batches, PIMProf iterates over them by permuting the decisions of the regions in each batch. Every iteration, it finds the decision for each region that yields the lowest total cost. Eventually, this heuristic algorithm will find near-optimal offloading decisions for the program.

[2]A special case of our problem where we only consider the execution cost and context switch cost (and set all data dependency cost to 0), is equivalent to the 0-1 quadratic programming problem, which is already NP-complete [19].

TABLE I
SYSTEM CONFIGURATION.

Out-of-Order CPU (baseline)
1/2/4 General purpose processors 3GHz, 4-way superscalar 32kB L1I, 32kB L1D, 256kB L2, 2MB L3
General-purpose in-order (PIM) [2]
16/**32**/64 general purpose cores 32kB L1I, 32kB L1D
Switching Cost
Cache line fetch/flush on CPU: 60 ns, on PIM: 30 ns Context Switch: 2 μs [18]

IV. EVALUATION

In this section, we first describe our evaluation methodology, and then present the results of our evaluated workloads.

A. Methodology

Evaluated Configurations. We model several CPU and PIM architectures on the Sniper simulator [17], as listed in Table I. The baseline configuration consists of out-of-order CPU cores similar to high-performance server processors. And, the configuration of PIM contains Atom-like in-order general-purpose cores [2]. We also provide sensitivity analysis by varying the number of cores on both CPU and PIM.

Evaluated Workloads. We use two widely-used benchmark suites that contain a variety of workloads to demonstrate the

[3]PIMProf will only consider a segment when its occurrence exceeds a threshold (at least 0.01% of the total execution time in our setup) to have enough impact on the overall performance.

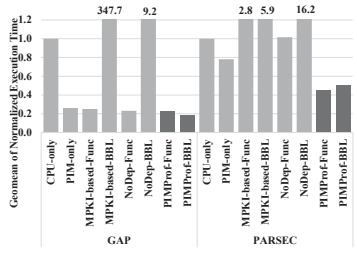

Fig. 7. Geometric mean of the execution time of all design points, PIMProf performance highlighted. (Execution time normalized to CPU-only).

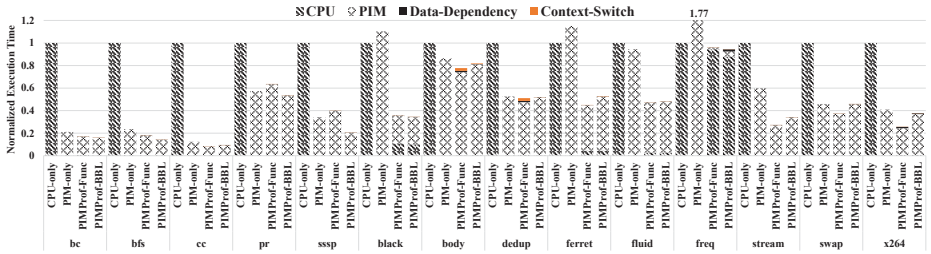

Fig. 8. Execution time breakdown of GAP and PARSEC workloads using PIMProf offloading decisions. We include four categories of costs: execution cost on CPU, execution cost on PIM, data-dependency cost and context-switch cost. The data dependency and context switch cost in this graph are not significant because PIMProf decisions remove most of them. (Execution time normalized to CPU-only).

flexibility of PIMProf: (1) graph benchmark suite (GAP) [10]—high memory intensity and parallelism. (2) PARSEC benchmark suite [11]—irregular workloads that are harder for manual offloading.

System Configurations. We evaluate four system configurations as the baselines compared to PIMProf.

- **CPU-only** runs the whole program on CPU.
- **PIM-only** runs the whole program on PIM.
- **MPKI-based (MPKI & parallelism-only)** detects the MPKI and parallelism of the program and offloads a region to PIM if both the MPKI and parallelism of the region exceed the threshold (5 for MPKI [20] and 16 for parallelism).
- **NoDep (Execution-cost-aware-only)** decides where to execute a region based on simply whether the execution cost is lower on CPU or PIM. It is usually suitable for applications with multiple independent kernels.
- **PIMProf (Data-dependency & context-switch-aware)** decides whether to execute a region on CPU or PIM based on the execution, data dependency, and context switch cost.

All configurations including the baselines and PIMProf will be evaluated with two different offloading granularities: *function-level* (Func) and *basic-block-level* (BBL).

B. Performance Analysis

We first demonstrate the performance of two different benchmark suites when running on a CPU-PIM hybrid system with a 1-core CPU and a 32-core HMC system. Figure 8 further breaks down the performance of PIMProf (basic-block-grained).

Performance of Graph Workloads. We evaluate the performance of graph workloads with in-order PIM cores, as prior work demonstrated that these kernels provide a significant performance benefit when offloaded to PIM [8]. Figure 8 shows the latency breakdown of each cost category when running these workloads on CPU, PIM, and CPU-PIM (under different strategies). We draw three conclusions from the results. First, offloading graph kernels as a whole to PIM (PIM-only) provides on average 1.89× speedup as compared to the CPU-only execution. Second, the MPKI-aware offloading method (MPKI-based-Func) is only 3.9% better than the PIM-only method because the switching overhead (data dependency and context switch) offsets the benefits from PIM offloading. Third, PIMProf (with awareness of switching overhead) provides 5.33× speedup over CPU-only (39% and 34% faster than PIM-only and MPKI-aware offloading, respectively). We also show that using PIMProf heuristics on function-level granularity (PIMProf-Func) provides a 13% improvement over PIM-only, since the GAP workloads have simpler data dependency compared to PARSEC. However, due to the coarser granularity, PIMProf-Func misses some offloading opportunities. Thus, it is slower than PIMProf-BBL (19% slower). We conclude that the offloading decision made by PIMProf reduces the switching cost, while still exploiting the benefits of PIM architecture.

Performance of PARSEC Workloads. PARSEC workloads have higher irregularity than the simpler graph workloads. Therefore, it is usually hard to find out PIM-friendly regions in PARSEC workloads by directly analyzing the MPKI and parallelism. In this experiment, we examine them with PIMProf to determine if PARSEC workloads contain PIM-friendly regions. Figure 7 demonstrates the overall speedup with various decision-making strategies for these workloads. First, PIMProf-Func provides 2.22× speedup over CPU-only and 1.74× over PIM-only. Second, we also notice that PIMProf-BBL is not performing as well as PIMProf-Func. We found that the heuristic search algorithm is limited by the maximum number of regions. Because the number of functions is much less than that of basic blocks, the function-grained scheme is less likely to be constrained by the limit. Nonetheless, it provides 1.99× speedup over CPU-only and 1.55× over PIM-only. Third, due to the large switching cost of PARSEC workloads, MPKI-based and NoDep methods are not performing well. Figure 8 shows the PIMProf decision execution time breakdown of these workloads. We make the following observations: First, PIMProf decides to offload most regions to CPU for PIM-unfriendly workloads (e.g., *Ferret*) to minimize the overhead but can only provide marginal improvement over CPU-only. Second, PIMProf-Func provides a 38% improvement on average for PIM-friendly workloads (e.g., *Bodytrack*, *Dedup*, *Streamcluster*, *Swaptions* and *X264*). Third, *Blackscholes*, *Ferret*, and *Fluidanimate* do not show good performance with a naive PIM-only strategy, but PIMProf-Func is able to figure out PIM/CPU-friendly regions in these workloads and provides 2.50× speedup over the PIM-only configuration. We conclude that PIMProf is effective in determining the offloading decisions for irregular workloads and can be used to profile real-world workloads to estimate their expected performance improvements from PIM architectures.

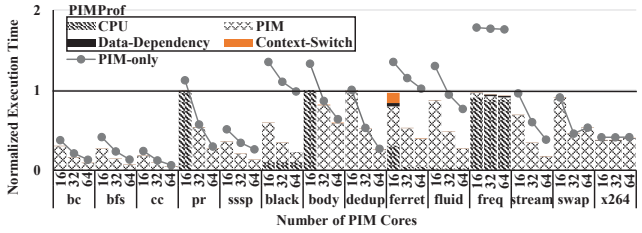

Fig. 9. PIMProf-BBL execution time breakdown when fixing CPU core number while varying PIM core number. (Normalized to CPU-only)

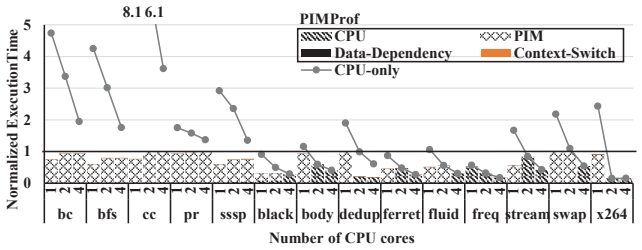

Fig. 10. PIMProf-BBL execution time breakdown when fixing PIM core number while varying CPU core number. (Normalized to PIM-only)

C. Sensitivity Analysis

We next perform a sensitivity study on different CPU/PIM core configurations.

Parallelism. For multi-threading workloads, the performance tightly depends on the parallelism provided by the underlying hardware, including both the CPU and PIM. Therefore, we test PIMProf-BBL offloading decisions for GAP and PARSEC workloads on architectures with variable numbers of CPU and PIM cores. We first fix CPU core number to 1 and vary the number of PIM cores, as shown in Figure 9. In general, PIMProf-BBL provides better performance than CPU-only and PIM-only in all configurations. However, the offloading benefit is marginal in some scenarios (e.g., 16-core *Ferret*), as the CPU performance is comparable with PIM and the benefit does not compensate for the switching overhead. We also observe that the decision of benchmarks, e.g., *Pagerank* and *Bodytrack*, moves from "mainly execute on CPU" to "mainly execute on PIM" as the number of PIM cores increases. Likewise, in workloads such as *Bodytrack* and *Dedup*, with PIM cores fixed to 32, the decision moves from "mainly execute on PIM" to "mainly execute on CPU" as the number of CPU cores increases (shown in Figure 10).

PIM Core Frequency. Unlike the CPU that typically contains powerful cores, PIM architectures need to meet a tighter hardware constraint, e.g., only allows light-weight in-order cores. Therefore, we explore the design space of PIM cores and test the efficiency of PIM offloading for a few PIM core configurations that vary in core frequency. The results in Figure 11 shows that PIMProf-BBL also tends to move more execution from CPU to PIM as the frequency of PIM cores increases.

V. Conclusions

In this work, we propose PIMProf, an automated profiling and offloading tool to determine PIM offloading regions for CPU-PIM hybrid architectures. PIMProf tackles the challenges

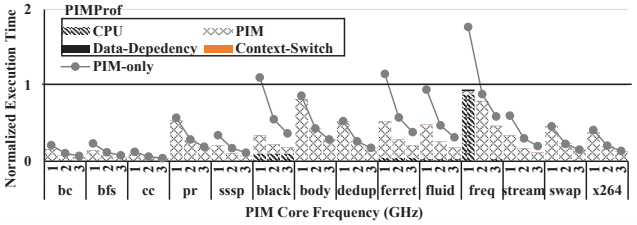

Fig. 11. PIMProf-BBL execution time breakdown when fixing CPU core while increasing PIM core frequency. (Normalized to CPU-only)

of PIM offloading through efficient cost modeling and optimization algorithm. Our evaluation shows that PIMProf provides 5.33×/1.39× speedup over CPU/PIM-only configuration for graph benchmarks, and 2.22×/1.74× speedup over CPU/PIM-only configuration for PARSEC benchmarks. A wide range of future PIM-related research can benefit from PIMProf for automatically profiling the emerging applications running on PIM architectures and quickly generating efficient PIM-based acceleration for general programs.

VI. Acknowledgement

This work was supported in part by JUMP CRISP, an SRC program sponsored by DARPA; and NSF grants (#1730158, #2100237, #1911095, #2112167, #2052809, #1826967).

References

[1] D. P. Zhang *et al.*, "TOP-PIM: Throughput-oriented programmable processing in memory," in *HPDC*, 2014.
[2] J. Ahn *et al.*, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA*, 2015.
[3] K. Hsieh *et al.*, "Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems," in *ISCA*, 2016.
[4] M. Gao *et al.*, "TETRIS: Scalable and efficient neural network acceleration with 3D memory," in *ASPLOS*, 2017.
[5] F. Gao *et al.*, "ComputeDRAM: In-Memory compute using off-the-shelf DRAMs," in *MICRO*, 2019.
[6] V. Seshadri *et al.*, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *MICRO*, 2017.
[7] Hybrid Memory Cube Consortium, "HMC Specification 2.0," 2014.
[8] J. Ahn *et al.*, "PIM-Enabled Instructions: A low-overhead, locality-aware processing-in-memory architecture," in *ISCA*, 2015.
[9] R. Hadidi *et al.*, "Cairo: A compiler-assisted technique for enabling instruction-level offloading of processing-in-memory," *TACO*, 2017.
[10] S. Beamer *et al.*, "The GAP benchmark suite," *CoRR*, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619
[11] C. Bienia *et al.*, "The PARSEC benchmark suite: Characterization and architectural implications," in *PACT*, 2008.
[12] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.
[13] Intel, "Intel vtune profiler," https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html, 2021.
[14] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.
[15] "The llvm compiler infrastructure," https://llvm.org, 2021.
[16] F. E. Allen, "Control flow analysis," in *Proceedings of a Symposium on Compiler Optimization*, 1970.
[17] T. E. Carlson *et al.*, "An evaluation of high-level mechanistic core models," *TACO*, 2014.
[18] J. Litton *et al.*, "Light-weight contexts: An OS abstraction for safety and performance," in *OSDI*, 2016.
[19] A. Caprara, "Constrained 0–1 quadratic programming: Basic approaches and extensions," *European Journal of Operational Research*, 2008.
[20] A. Boroumand *et al.*, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *ASPLOS*, 2018.