TUPIM: A Transparent and Universal Processing-in-Memory Architecture for Unmodified Binaries

Sheng Xu\textsuperscript{1}, Xiaoming Chen\textsuperscript{2,3*}, Xuehai Qian\textsuperscript{4}, Yinhe Han\textsuperscript{2,3*}

\textsuperscript{1}Department of Computer Science and Technology, Anhui Normal University, Anhui, China
\textsuperscript{2}Center for Intelligent Computing Systems, State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
\textsuperscript{3}University of Chinese Academy of Sciences, Beijing, China
\textsuperscript{4}Department of Computer Science, University of Southern California, CA, USA

*Corresponding authors: chenxiaoming@ict.ac.cn, yinhes@ict.ac.cn

ABSTRACT

Recently, processing-in-memory (PIM) is gaining much attention because it could minimize data movement by conducting computation in memory. Existing PIM approaches require a number of additional procedures during the setup-time, including code re-writing and re-compiling, code annotations, and detailed program profiling, etc. These requirements, however, potentially prevent existing executable binaries benefiting from PIM architectures. For old binary legacies without any source code, it is impossible to run them on existing PIM architectures. To solve these challenges, in this paper we propose a transparent and universal PIM (TUPIM), a novel PIM architecture that can execute unmodified binaries and at the same time take advantages of PIM. TUPIM is a significant advance over the state-of-the-art because it transparently expands the scope of PIM to deploy all applications without any source code, programming models, or compiler modifications. Experiments show that TUPIM can get 2.2x speedup on average (up to 3.67x) and 15.7% energy reduction, compared with conventional CPU-only executions.

1 INTRODUCTION

In the conventional computer architecture, the widening gap between the computational and data transfer speeds is calling for an unconventional computing paradigm. Processing-in-memory (PIM), which integrates some computing logic in the main memory, is a promising solution to migrate the memory wall challenge by conducting part of computation in memory. Recently, there are a number of PIM-related studies, including application-specific PIM architecture design \cite{1-7}, PIM circuit design \cite{8}, PIM coherence \cite{9,10}, PIM task selection \cite{5,11-13}, PIM simulation \cite{14}, etc.

State-of-the-art PIM solutions fail to enable transparent PIM architectures because of two reasons. First, they need detailed workload profiling to determine the PIM kernels (i.e., instructions executed by the in-memory logic). Inappropriate selection of PIM kernels may even degrade performance. For example, Ref. \cite{11} showed that it performs 50% slower than CPU-only executions. To avoid such situations, detailed offline profiling is needed. Second, after PIM kernels are decided, existing PIM solutions more or less need some modifications to the source code and the compiler \cite{11}, by either applying PIM-enabled programming models or designating PIM kernels by manual annotations. For example, the performance and energy improvements of Tesseract \cite{1} on some graph processing applications are obtained by rewriting applications using the dedicated programming model for graph processing, which is not applicable to other PIM architectures or applications. Moreover, the requirements of modifying the source code and the compiler heavily prevent existing binaries without source code benefiting from PIM.

Furthermore, extended PIM programming models introduce many impacts on the host CPUs. For example, to avoid accessing stale data, the in-memory logic should wait until all dirty data of the same data region are written back to the memory (i.e., a data flushing), including all queued writes requests. In such circumstances, it will not increase the parallelism as the CPUs are suspended. These hardware-software cooperative issues make PIM a highly customized architecture and bring obstacles to PIM architectural development, calling for a transparent and universal PIM architecture.

To overcome the aforementioned challenges, this paper proposes TUPIM, standing for a transparent and universal PIM, a PIM architecture that could execute unmodified binaries, virtually allowing all applications getting benefits from PIM if possible. On TUPIM, people just write and compile programs like on conventional architectures. No special factors or details of TUPIM need to be considered. This implies that the architecture is completely “transparent” to software programmers. Such an ambitious goal has never been achieved by existing PIM architectures. This paper makes the following contributions.

1) We introduce TUPIM, the first systematic PIM architecture that can execute unmodified binaries. The architecture
is elaborated to be completely transparent to software developers. It greatly expands the applicability scope of PIM for any workloads without any modifications on the source code, compiler, or programming models.

2) We propose a low-cost and effective PIM kernel selection method to determine whether the instructions are suitable to be executed in the memory.

3) We propose an OoO instruction-level PIM parallelization method based on speculative execution, which ensures correct executions of instructions in PIM and also reduces the impact on the host CPUs.

### 2. TUPIM ARCHITECTURE

#### 2.1 Overview

Here we explain the key ideas of TUPIM and its working flow. TUPIM is able to automatically extract proper PIM-friendly instructions (PFIs) and offload them to the memory at runtime. In order to make our idea easy to follow, we insert an abstracted pipeline stage named Extract to explain the mechanism. Fig. 1 shows the top-level pipeline diagram of TUPIM. When a binary is being executed and decoded in TUPIM, the decoded instructions are stored in an instruction buffer. TUPIM detects PFIs using a sliding window approach. TUPIM identifies PFIs and classifies them into two groups. For instructions to be executed on the host CPUs, dubbed non-PFIs, TUPIM will execute them on the host CPUs as the same as ever. For instructions which may get benefits from PIM, dubbed candidate PFIs (cPFIs), TUPIM will execute them on the in-memory cores speculatively. After the cPFIs are identified, TUPIM invokes another round of check which adjusts whether the cPFIs are really suitable for PIM executions based on some run-time factors. If not, they are then sent back to the host CPUs. Otherwise, TUPIM starts a speculative execution for the PFIs in the memory.

The key idea of TUPIM relies on four features: 1) what PFIs are and how to detect them; 2) how to decide if a detected cPFI is really suitable for in-memory executions; 3) how to execute PFIs and at the same time ensure that all instructions are committed in the correct order with maximized performance and minimized overhead; and 4) how to provide data coherence in the system. In the following subsections, we will introduce our solutions in detail.

#### 2.2 PIM Kernel Selection

##### 2.2.1 PIM-friendly Instructions

To implement a PIM system which automatically detects PIM kernels without any code or compiler modification, the PIM architecture must automatically extract PFIs. According to the prior works [1, 5, 7, 9, 11, 12], PFIs should meet the following three requirements: 1) The selection of PFIs should minimize unnecessary intermediate data movements (a.k.a., data ping-pong) between the on-chip caches and the main memory; 2) PFIs cannot be well served by the caches, and 3) PFIs should be repeatedly executed. We explain these requirements below.

The first requirement indicates that the instructions should be relatively complete with no interleaving partitions. Since the in-memory cores are typically much weaker than CPUs, the benefits of PIM come from the avoidance of unnecessary intermediate data movements in the memory hierarchy. Fig. 2(a) shows two examples of PFI selections. In Fig. 2(a), the instructions I₂ to I₄ form a read-modify-write (RMW) operation. It is obvious that offloading instructions I₂ to I₄ to PIM can minimize intermediate data movements. Otherwise, data will ping-pong between the host and the memory, which increases the off-chip traffic and brings contention stalls to the PIM system. The prior works [5, 7, 11, 12] only offload atomic operations to the memory. An atomic operation is an RMW operation that targets at the same address like the instructions I₂ to I₄. From the point of view of avoiding intermediate data movements, this concept can be extended to a load-store chain (LSC), which is an instruction sequence starting from a load instruction and ending up with the nearest store instruction without branches between them (so there is no off-chip memory traffic between them). In this paper, we extend offloading atomic operations to offloading LSCs, because offloading appropriate LSCs can also get benefits from PIM by avoiding intermediate data movements.

The second requirement indicates that PFIs should involve poor data locality. Locality has been proved to be a low-cost and effective factor to determine if it is necessary to start a PIM operation [5, 11]. Intuitively, if some instructions involve high cache hit, it is hard to get performance benefit from the weak PIM logic because the data movement overhead is eliminated by the CPUs’ caches. Furthermore, it is not enough to offload instructions meeting only the first two requirements into the memory, because PIM offloading introduces additional off-chip overhead. The control flow and the required data of the PIM instructions will travel through a narrow off-chip bus. Thus, it is not wise to offload infrequently executed instructions to the memory. To partly hide the overhead of...
PIM offloading, we observe that good PIM kernels are usually contained in loops. Therefore, it is necessary to keep tracking of the execution count of the dispatched PFIs.

Taking all these factors into consideration, TUPIM performs a three-step check for PFI offloading. First, TUPIM detects LSCs as cPFIs. Then it checks the repetitiveness of them and marks them dispatchable if they have been executed repeatedly. Finally, TUPIM gives offloading decisions according to the predicted locality. After that, the offloaded instructions get ready for executions in the memory.

2.2.2 Candidate PFI Detection. The detection of cPFIs (LSCs in fact) determines the operations that TUPIM can be aware of. Due to the compiling feature that tends to group operations on the same operands into the same basic block [15], cPFIs can be easily detected through pattern matching. Taking the example of Fig. 2(b), the add-self operation (line 6 in the source code) will be compiled and placed in order in the binary. Therefore, by taking advantage of the in-order prefetching feature in the decode stage of the pileine, TUPIM are able to be aware of such operations by walking through the decoding buffer using a sliding window (see ① in Fig. 1).

cPFI detection starts with finding a load instruction in the sliding window. It then looks ahead to find a store instruction. During this process, if TUPIM detects a branch or reaches the bound of the sliding window, it gives up the current matching and starts a new search. Otherwise, it further checks if the found store instruction is related to the current cPFI.

2.2.3 Repetitiveness Detection. While offloading cPFIs has met the first requirement that minimizes intermediate data transfers, cPFIs should meet the other two criteria – repetitiveness and bad locality. For repetitiveness, TUPIM records the history information of the detected cPFIs and generates dispatchable PFIs. If the cPFIs involve good data locality, they will not be offloaded to the memory. These two tasks are undertaken by two modules, a PFI History Table and a PFI Locality Predictor in TUPIM.

The PFI History Table records the repetitiveness of cPFIs. It contains two sub-tables: a Pending PFIs Table (P-Table) and a Recorded PFIs Table (R-Table). The P-Table records the execution status of the latest cPFIs, including the instruction patterns and execution counts. If the execution count of a PFI exceeds a threshold, TUPIM will invalidate the corresponding entry in the P-Table and create a new entry in the R-Table for further PFI matching. The two tables are updated by the Least Recently Used (LRU) cache replacement policy [16]. To reduce the hardware overhead, TUPIM only records the instruction and addressing types for PFI matching but discards the precise address and the register ID, which is shown in the right corner of Fig. 3.

We use Fig. 3 as an example to show how the two PFI History Tables work. The left part of Fig. 3 shows the instruction sequence to be executed. Initially, the R-Table is empty and the P-Table consists of two entries (Pattern1 and Pattern2). When a new cPFI (cPFI2) is detected in the sliding window, it is first compared with the entries in the R-Table to check if it matches a recorded PFI. It returns a miss as the R-Table is empty ①. Then TUPIM checks the entries in the P-Table. Because the P-Table already contains the entry of cPFI2 (Pattern2), it will increase the corresponding counter ②. If the counter exceeds a threshold, TUPIM invalidates the entry of cPFI2 in the P-Table and creates an entry in the R-Table to mark it dispatchable as it has been executed repeatedly ③. It works the same when cPFI3 and cPFI4 are detected ④ ⑤. Next time when the same cPFI is detected again (say, cPFI2 is found in the R-Table) ⑥, TUPIM will send it to the PFI Locality Predictor for further locality checking ⑦, to check whether it is really suitable for PIM executions.

2.2.4 Locality Prediction. The PFI Locality Predictor gives a locality prediction before the dispatchable PFIs are offloaded to the memory. Because PFIs may contain indirect addressing (e.g., LD r13, [rax+0xff]) that needs further address calculation, TUPIM takes advantages of the high performance of the host CPUs to calculate the indirect addressing (see ① in Fig. 1). Note that this operation will not block the pipeline or incur additional latency because calculating the indirect addressing is a necessary operation for such instructions regardless of where they will be executed. After calculating the indirect addressing, the locality information can be given by a tag-array-based locality monitor described in the work named PIM-enabled instruction (PEI) [11]. The locality monitor is a tag array with the same number of sets/ways as that of the last-level cache. If the required address of the PFI is not predicted in the last-level cache, TUPIM will start offloading and executing it in the memory. Otherwise, it implies that the cPFIs involve cache hits so the cPFIs will be demoted as non-PFIs and issued on the host CPUs.

2.3 Speculative Execution of PFIs

When PFIs are offloaded to the memory and about to execute, the execution is controlled and monitored by a hardware unit called Speculative Execution Scheduler in TUPIM, which allows PIM instruction executing remotely and speculatively like normal instructions executed on OoO CPUs. The speculative execution of PFIs does not block the CPUs’ pipeline, which hides the overhead of data flushing caused by PIM coherence. Since TUPIM detects PFIs to be executed in the near feature in the decoding buffer, the execution sequence may be incorrect. Therefore, TUPIM allows PFIs to be speculatively executed to partly alleviate the overhead.

In modern computer architectures, the instruction committing order is guaranteed by holding a global sequential tag for each instruction. The tag is used in the reorder buffer in an
OoO core and acts as an index for misprediction. It is assigned during the decode stage of the pipeline. Therefore, the committing order can easily be guaranteed by requesting the host CPUs for the global sequential tags whenever a PFI is completed in the memory. However, this implementation may involve significant ordering stalls and unnecessary off-chip communication to the system as each PFI should wait for either the host CPUs or the latest offloaded PFI for exclusive data access. We propose a simple-yet-effective approach to solve this problem. Due to the look-ahead feature of TUPIM, it knows exactly which PFI may lead to fault data. In TUPIM, there are three types of faults that may lead to data inconsistency: 1) instruction dependency causing register value change, 2) misprediction of the branch instructions at the host processors, and 3) stale data copy access. Since the last fault can be solved by coherence management which is discussed in the next subsection, we will introduce our solutions to the first two faults. Thus, TUPIM performs a speculative execution by invalidating or re-executing PFIs according to the fault type. Meanwhile, TUPIM broadcasts the sequential tag to the memory periodically to commit executed PFIs in the memory. This mechanism ensures that the offloaded PFIs can be pre-computed and in-order committed.

In TUPIM, the execution control is ensured by holding a hardware unit called Offloaded PFI Station (OPS). Each entry of OPS records the status of an offloaded PFI, which contains four sub-regions: Seq, InsCount, Ready and Register Modify Bits. Seq is the sequential tag of the first instruction of the PFI. Combining with InsCount which records the instruction count of a PFI, TUPIM is aware of the exact order of a PFI. The Ready bit indicates if a PFI has its addresses calculated and is ready for offloading. We use the Tomasulo algorithm [17], which laid the groundwork for modern OoO processor designs, to show how TUPIM interacts with the existing OoO cores in Fig. 4. All data are transferred through a bus called Common Data Bus (CDB). By snooping on the CDB, TUPIM is able to detect all register value changes and the results of the current execution, which lies on the basis of dependency checking and misprediction apperceiving, which are described in the following content.

2.3.1 Dependency Check. The dependencies among instructions are usually detected in the rename stage. Since PFIs are filtered before renaming, no dependency can be detected between PFIs. Therefore, TUPIM should detect instruction dependencies and register value changes for offloaded PFIs. The Register Modify Bits in the OPS are used to track the modifications to the registers and instruction dependencies during the PFI executions. Due to the speculative execution feature of TUPIM, only read-after-write (RAW, true dependency) should be detected among different PFIs. Write-after-read (WAR) and write-after-write (WAW, false dependency) could benefit from speculative execution. Therefore, for each architectural register in the instruction set architecture (ISA), it has two encoded bits in the Register Modify Bits. As shown in Fig. 5, when two PFIs are offloaded sequentially Q, TUPIM detects their dependency through bit checking. At point Q, TUPIM finds no conflict because no inter-PFI RAW happens. When the next PFI fills in at point Q, TUPIM detects inter-PFI RAW conflicts and thus stalls to wait for results. This implementation brings at least two benefits. First, TUPIM is able to detect any dependency, either between the CPUs and offloaded PFIs or inter-PFIs. Second, TUPIM is aware of the location of any registers during the execution and therefore, requests for remote register values if necessary.

2.3.2 Misprediction of Branch Instructions. The misprediction of an OoO processor often leads to a squashing request to any in-flight younger instructions and a reload to L1-cache. Likewise, in TUPIM, all undispatched PFIs are invalidated when encountering a misprediction. In the Tomasulo algorithm, the misprediction results are calculated and broadcasted through the CDB. This feature enables TUPIM to get immediately aware of branch misprediction. Therefore, it triggers a flush to the PFI Buffer and OPS to invalidate undispatched PFIs.

2.4 Coherence Management

In TUPIM, PFIs are interoperable with existing cache protocols. Since conventional fine-grained cache coherence protocols like MESI [18] can handle well with on-chip data consistency, the challenge lays on the data coherence between the host CPUs and the in-memory cores. We propose a simple-yet-effective approach for data coherence in TUPIM. TUPIM triggers an invalidation (for each read address of PFIs) and a flush (for each write address of PFIs) to the requested cache blocks when a PFI offloading happens. This ensures that neither the on-chip caches nor the main memory has a stale copy of the data before PFI execution. Besides, the...
in-order PFI committing feature ensures the data consistency at the memory side, without requiring complex hardware supports for extending cache coherence protocols toward the main memory. The invalidation and flush operations happen infrequently because PFIs are offloaded only if the involved data are considered not in the on-chip caches.

### 3 Evaluation

#### 3.1 Simulation Configuration

We modified GEM5 [21] to evaluate TUPIM. We modified the Deriv03CPU model in GEM5, which implements a basic OoO execution core based on the Tomasulo algorithm [17] and added the features of TUPIM into it. Besides, we implemented detailed PIM instructions as PEI [11] and extended it to support PFI operations. Table 1 summarizes the configuration of the baseline system. We compare the following three configurations:

1. CPU-only. This is a conventional architecture that uses HMC as the main memory and does not offload any instructions to the memory.
2. PIM-Atomic. Similar to PEI [11] and GraphPIM [5], this configuration offloads atomic operations to the memory. During executions, the system uses a locality-aware offloading approach to decide where the PIM operations should be executed. Different from PEI, this configuration assumes that the system can precisely predict if data are present in the on-chip caches, which acts as an ideal case of PEI as it will not trigger any stale data write to the memory.
3. TUPIM. This is the proposed TUPIM architecture.

The target applications of TUPIM are programs with a large memory bandwidth consumption and big memory access regions. We focus on scale-out applications in our evaluation. Therefore, we choose several typical benchmarks of scale-out applications from GraphBIG [22] used in [1, 5, 9, 11, 12], and implement them in C++. The names and input data sizes of the benchmarks are listed in Table 2. We use real-world workloads to make the results more reliable.

#### 3.2 Performance

Fig. 6 compares the performance. TUPIM achieves 2.2× speedup on average (3.67× at most) over CPU-only. TUPIM improves the performance by 1.42× on average over PIM-Atomic. In PIM-Atomic, PIM operations are marked in the source code with atomic locks inserted, and thus, instructions suffer from atomic overhead in the caches and processors [5]. However, in TUPIM, the PFIs can benefit from speculative execution, which implicitly avoids the atomic overhead.

However, we also observe that some applications, e.g., BC and TC, gain few benefits from TUPIM. For BC, it contains a large number of C++ standard template library (STL) operations, such as push_back() and pop(), which contain about 5-15 LSCs in each operation. Such massive PFIs will cause P-Table flushes, which flush useful PFI records and make the R-Table failed to add useful PFIs. Although we use the two-table hierarchy scheme to partly prevent unnecessary PFI flushes, it can still cause the decline of PFI recognition when encountering massive STL operations. However, this problem can be easily solved by either replacing the STL operations with pointers or increasing the P-Table and/or R-Table sizes. For TC, it usually involves good locality. The well-served caches cause few PIM offloads due to the good locality prediction. The PFI offloading overhead cannot be hidden, causing worse performance than PIM-Atomic. Moreover, the inter-PFI RAW dependency causes the offloaded PFIs re-dispatching frequently. However, for TC, TUPIM still performs better than the CPU-only baseline.

Though Tesseract [1] achieves much higher speedup (10×), Tesseract and TUPIM are not comparable due to the opposite design goals. Tesseract is an application-specific architecture but TUPIM is universal. Tesseract heavily needs code re-writing and re-compiling but TUPIM does not. We try to make TUPIM universal and compatible to existing architectures while minimizing the introduced overhead but Tesseract is dedicated only for certain applications.

#### 3.3 Energy Consumption

Fig. 7 shows the energy consumption of the un-core components, normalized to that of CPU-only. Since a data movement consumes 3 orders of magnitude higher energy than an arithmetic operation [23], we only estimate the energy consumption of the un-core components. The controller energy is estimated based on [24]. The HMC link energy is evaluated...
based on [7], assuming 3pJ per bit for packets. We find that TUPIM reduces the energy consumption by 15.7% on average over the CPU-only baseline. The main improvement comes from the offloaded PFIs. We also observe that TUPIM consumes more energy than PIM-Atomic for several workloads, especially for BC, DC and TC. These workloads often exhibit many branch mispredictions, which in turn frequently trigger data re-sending from the host CPUs to the memory. From an average point of view, although TUPIM consumes almost the same energy as PIM-Atomic, our scheme can execute unmodified binaries and thus, significantly reduce the deployment overhead for PIM applications.

3.4 Hardware Overhead Estimation

The hardware overhead of TUPIM mainly includes four aspects: 1) the PFI detection unit; 2) the PFI History Tables; 3) the PFI Locality Prediction Unit and 4) the PFI execution units (including the dependency check units and OPS).

We can provide an approximate estimation on the hardware overhead. However, we are not able to provide a detailed evaluation because of the following two reasons.

First, TUPIM has to interact with host CPUs. For example, locality prediction utilizes the adders of the arithmetic logical units (ALUs) for indirect addressing calculation. It is impossible to conduct a detailed hardware overhead evaluation unless we implement a detailed CPU. Nevertheless, to get an approximate estimation, we conservatively assume that the extra logic in each core for performing the operations of TUPIM has similar overhead as an ALU. Since ALUs only consume a few percentages (e.g., 3% [25]) of the total CPU power, the logic overhead of TUPIM is also a few percentages.

Second, the hardware overhead of TUPIM depends on the ISA. For example, when TUPIM is running on x86, OPS needs 1.16KB, the PFI buffer needs 0.75KB, and the P-Table and the R-Table need 2KB and 1KB, respectively. Things change for different ISAs, because many hardware parameters, like the register number, depend on the specific ISA. Nevertheless, the storage overhead of TUPIM is just a few KBs which can be ignored.

4 CONCLUSIONS

In this paper, we propose TUPIM, a transparent and universal PIM architecture that could execute unmodified binaries and at the same time take the advantages of PIM. Our work is the first effort that makes PIM transparent to programmers. TUPIM is a significant advance over the state-of-the-art because it transparently expands the scope of PIM to deploy all applications without any code or compiler modifications. We design detailed mechanisms to enable the transparency and universality of the architecture and also ensure data coherence and correct executions. Our experiments show that TUPIM achieves 2.2× speedup on average (up to 3.67×) and 15.7% energy reduction, compared with CPU-only executions.

ACKNOWLEDGEMENTS

This work was supported by National Key R&D Program of China (No. 2018YFA070150), by Key Research Program of Frontier Sciences, CAS (No. ZDBS-LY-JSC012), by National Natural Science Foundation of China (Nos. 61804155 & 61834006), by Youth Innovation Promotion Association CAS, by Young Elite Scientists Sponsorship Program by CAST (No. 2018QNRC001), and by Beijing Academy of Artificial Intelligence (BAAI).

REFERENCES