This article has been accepted for publication in IEEE Computer Architecture Letters. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/LCA.2025.3600588

PIMsynth: A Unified Compiler Framework for Bit-Serial Processing-In-Memory Architectures

Deyuan Guo, *Member, IEEE*, Mohammadhosein Gholamrezaei, Matthew Hofmann, Ashish Venkat, *Member, IEEE*, Zhiru Zhang, *Fellow, IEEE*, and Kevin Skadron, *Fellow, IEEE*

***Abstract*—Bit-serial processing-in-memory (PIM) architectures have been extensively studied, yet a standardized tool for generating efficient bit-serial code is lacking, hindering fair comparisons. We present a fully automated compiler framework, PIMsynth, for bit-serial PIM architectures, targeting both digital and analog substrates. The compiler takes Verilog as input and generates optimized micro-operation code for programmable bit-serial PIM backends. Our flow integrates logic synthesis, optimization steps, instruction scheduling, and backend code generation into a unified toolchain. With the compiler, we provide a bit-serial compilation benchmark suite designed for efficient bit-serial code generation. To enable correctness and performance validation, we extend an existing PIM simulator to support compiler-generated micro-op-level workloads. Preliminary results demonstrate that the compiler generates competitive bit-serial code within $1.08\times$ and $1.54\times$ of hand-optimized digital and analog PIM baselines. Our code is publicly available at https://github.com/UVA-LavaLab/PIMsynth.**

***Index Terms*—processing in memory (PIM), bit-serial code generation**

I. Introduction

Processing-in-memory (PIM) architectures have been widely studied as a promising approach to address memory bottlenecks and leverage the high internal parallelism of DRAM. Bit-serial PIM architectures take advantage of the inherent massive parallelism of DRAM row operations, attracting considerable interest from both academia and industry. Analog bit-serial PIM designs, e.g., [1]–[5], often utilize triple-row activation (TRA) to perform majority (MAJ) logic operations directly within memory arrays with low area overhead, often referred to as processing-using-DRAM (PUD). Alternatively, digital PIM designs, e.g., [6]–[9], integrate logic gates near the sense amplifiers to perform bit-serial logic operations, possibly in combination with PUD.

Programming bit-serial PIM remains challenging due to the complexity of implementing bit-serial code variants for different instruction sets, data types, register configurations, and hardware constraints. Fig. 1 shows two bit-serial data dependency graphs of a 1-bit full adder as an example of how to program a digital bit-serial PIM [9] and an analog bit-serial PIM [4]. Without a compiler toolchain, significant manual effort is required to map such dependency graphs into the target PIM instruction set, considering instruction scheduling, register allocation, spilling, etc. This manual effort is often infeasible or unreliable when comparing bit-serial PIM architectures, due to the complexities of bit-serial algorithms and need for different architecture-specific optimizations.

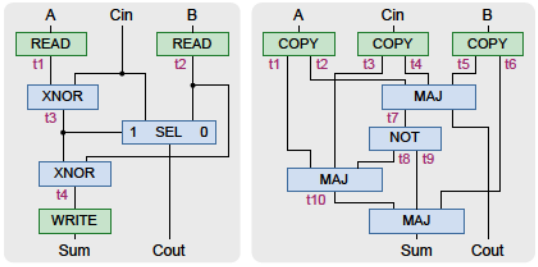

Fig. 1. Bit-serial data dependency graphs of 1-bit full adder: (left) digital bit-serial PIM; (right) analog bit-serial PIM. Analog PIM requires variable replication due to the input-destructive behavior of triple-row activation (TRA)-based majority (MAJ) operations, and it supports multiple outputs. Register allocation is further needed to utilize PIM hardware resources.

Previous PIM research has addressed the bit-serial compilation challenges specific to certain architectures. For example, SIMDRAM [4] introduced a majority-inverter graph (MIG)-oriented synthesis flow with row-to-operand allocation algorithms. CHOPPER [10] proposed a general compiler infrastructure designed for analog bit-serial PIM, incorporating the Usuba bit-slice compiler [11], LLVM [12] and several performance optimizations. However, to the best of our knowledge, these approaches have seen limited adoption, primarily due to being closed-source, limited generality, or requiring non-trivial manual effort.

To address the challenge, we present a fully automated, end-to-end bit-serial compilation framework, PIMsynth, targeting both digital and analog bit-serial PIM architectures, to allow a deep and fair study of diverse PIM architectures. This framework integrates open-source logic synthesizers Yosys [13] and ABC [14] to convert bit-parallel computation into bit-serial, LLVM [12] for instruction scheduling and register allocation, PIMeval [9] simulator for verification and performance energy modeling, with multiple PIM-oriented transformation and optimization steps and code generation, as a unified compilation toolchain.

With this compiler framework, we provide a set of carefully designed Verilog combinational circuit modules as a bit-serial compilation benchmark suite. These modules represent conventional bit-parallel operations, e.g., arithmetic/Boolean

D. Guo, M. Gholamrezaei, A. Venkat, K. Skadron are with University of Virginia, Charlottesville, VA 22903 USA. (e-mail: {dg7vp, uab9qt, venkat, skadron}@virginia.edu).

M. Hofmann, Z. Zhang are with Cornell University, Ithaca, NY 14853 USA. (e-mail: {mrh259, zhiruz}@cornell.edu).

This work was supported in part by PRISM and ACE, two of seven centers in JUMP 2.0, an SRC program sponsored by DARPA; the NSF under grant PPoSS-2217071 and award #2403135; and Booz Allen Hamilton under contract FA-8075-18-D-0004.

This article has been accepted for publication in IEEE Computer Architecture Letters. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/LCA.2025.3600588

operations on 8/16/32/64-bit operands, but their circuit structure can strongly influence the performance of the generated bit-serial code. For example, using an arithmetic '+' operator often produces a wider data dependency graph and higher register pressure than sequential designs such as a ripple-carry adder. We therefore construct the suite to minimize dependencies and register usage, enabling high-performance bit-serial code generation and consistent evaluation.

The key contributions of this work are as follows.

1) A fully automated, end-to-end compilation flow for digital and analog PIM bit-serial architectures, supporting multiple instruction sets and register configurations;
2) A transformation and optimization flow to convert a digital circuit into an analog bit-serial PIM compatible intermediate representation (IR);
3) A benchmark suite of Verilog modules implementing bit-parallel operations;
4) An extension in PIMeval simulator to enable digital and analog bit-serial code execution for automated verification and performance energy modeling.

We evaluate the compiler across two representative digital and analog PIM architectures to demonstrate that it generates efficient bit-serial code comparable to handwritten baselines. Furthermore, this bit-serial compiler framework serves as a foundation for supporting a range of bit-serial PIM instruction sets, register configurations, and operations, enabling comprehensive analysis and comparison across diverse architectures.

II. Related Work

SIMDRAM [4] is an end-to-end framework for analog bit-serial PIM. In addition to its architectural and system-level contributions, it introduces a compilation flow that translates logic from AND-OR-Invert Graph (AOIG) into Majority-Inverter Graph (MIG), applies MIG-level optimizations, and performs row-to-operand allocation. Despite its comprehensive design, the framework has a few limitations: 1) It is closely tied to the SIMDRAM hardware architecture, limiting generality; 2) AOIG generation is not integrated into the compilation toolchain; and 3) the register allocation strategy lacks generality and provides limited support for spilling. MIMDRAM [5] is built on top of SIMDRAM, focusing on mapping high-level programs to bit-parallel operations in multi-instruction multi-data (MIMD) fashion, while the underlying bit-serial compilation is solved in the same way as SIMDRAM.

CHOPPER [10] is a compiler infrastructure proposed for analog bit-serial PIM architectures. Although it shares a similar high-level objective and investigates both register spilling and analog PIM specific optimizations, it has some limitations: 1) It is built on the Usuba bit-slice compiler [11], inheriting limited logic synthesis capabilities and lacking full support for converting general bit-parallel logic such as arithmetic operations into optimized bit-serial sequences. 2) It offers limited instruction set customization and relies on separate post-processing to support different architectures. 3) To the best of our knowledge, no open-source implementation of CHOPPER is publicly available, which limits reproducibility and hinders comparative evaluation.

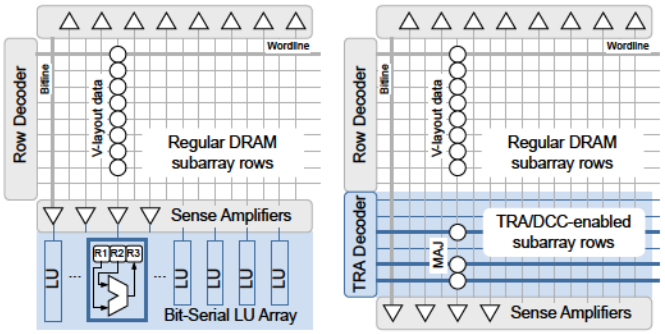

Fig. 2. Bit-serial PIM programming models: (left) digital; (right) analog. Both models assume a vertical data layout within DRAM subarrays. Digital PIM attaches a bit-serial logic unit (LU) to each sense amplifier, while analog PIM introduces triple-row activation (TRA)-based majority (MAJ) capability, with dual-contact cell (DCC) rows to perform NOT operations.

PIMLC [15] is a recent bit-serial compiler solution targeting SRAM- and ReRAM-PIM solutions, integrating a workload-resource aware scheduling (WRAS) algorithm to minimize latency. However, it does not support DRAM-based PIM architectures, which have distinct computing models, input-destructive analog operations, and digital registers.

III. Compiler Design

A. Bit-Serial PIM Programming Models

To support a range of PIM architectures, we define two bit-serial programming models for digital PIM and analog PIM respectively, as illustrated in Figure 2. Both models assume a vertical data layout and enable bit-serial SIMD computation within DRAM subarrays, but they differ in how computation is implemented and how operations are executed. Digital bit-serial PIM, e.g. [7], [9], introduces lightweight digital bit-serial logic units (LUs) attached to sense amplifiers, which can perform bit-serial operations on a small set of single-bit registers. Computation proceeds by reading memory rows into sense amplifiers, possibly copying values to registers, and executing bit-serial operations on the registers. A bit-serial logic operation is $10-20\times$ faster than a memory row read or write operation on DRAM, as described in Section IV-A. Analog bit-serial PIM, such as [4], performs computation directly in memory based on the triple-row activation (TRA) mechanism, which allows in-place majority logic. AND and OR operations are implemented using MAJ with constant zero and one inputs. We currently assume that all TRA-enabled rows are also dual-contact cell (DCC) rows for NOT operations. Computation proceeds by reading data from regular memory rows to a small group of TRA/DCC-enabled register rows, executing bit-serial operations, and writing results back to regular memory rows.

B. Bit-Serial Compiler Main Flow

The main flow of the bit-serial compiler is shown in Fig. 3. The compiler takes three inputs: 1) Verilog description of bit-parallel computation, 2) bit-serial PIM instruction set (ISA) defined as a standard cell library, and 3) the number of registers in the target PIM architecture. The compiler then performs logic synthesis, optimization, scheduling, code generation,

This article has been accepted for publication in IEEE Computer Architecture Letters. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/LCA.2025.3600588

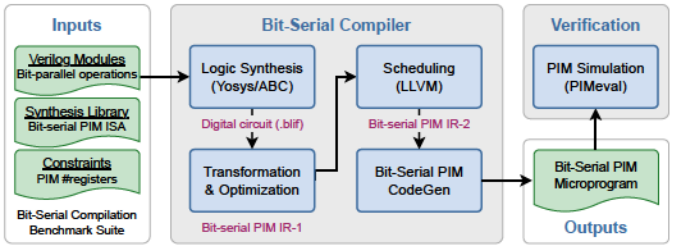

Fig. 3. PIMsynth bit-serial compiler main flow: A unified compilation framework that supports diverse digital and analog bit-serial PIM architectures

and simulation phases in a unified framework. If a new bit-serial PIM architecture follows either the digital or analog programming model and differs only in its ISA or register configurations, the compiler can already support it without modification, given the proper inputs. There are other PIM architectures beyond these two, e.g., with different analog instruction sets [8] or mixed analog-digital solutions [6]. Extending support to a broader range of PIM architectures, such as [2], [3], [6], [8], [16], [17], is left for future work.

1) Logic Synthesis Phase: The bit-serial compiler framework integrates Yosys and ABC synthesizers to convert bit-parallel computation Verilog into digital circuit netlists in BLIF. Verilog provides flexible ways of describing the structures of bit-parallel computation and serves as standard input to logic synthesizers. Digital circuits can be naturally interpreted as bit-serial data dependency graphs for the next phase of compilation. Using Verilog helps with low-level mapping, but adding support for high-level programming languages is a natural direction for future work. Another compiler input is the bit-serial PIM ISA definition. The PIM ISA is converted to a standard cell library in GenLib format, used by Yosys and ABC to perform technology mapping from a general digital circuit into a circuit using specific logic gates or operations supported by the target architecture.

2) Transformation & Optimization for Digital/Analog PIM: During this phase, the compiler converts the output of logic synthesis, i.e., digital circuits, into bit-serial PIM IR-1 (bit-serial data dependency graph) for scheduling considering key characteristics of digital and analog bit-serial PIM. The purpose of these transformation steps is to convert digital circuits into PIM operations, particularly for analog PIM.

Based on the digital bit-serial PIM programming model, digital bit-serial logical operations are performed on a small set of single-bit registers within each bit-serial LU, which can be directly mapped to logic gates in the digital circuits generated by the logic synthesis phase. However, analog bit-serial PIM is more complicated due to its unique characteristics and requirements: 1) Because of the input-destructive behavior of TRA, input variables will be updated after performing an analog MAJ operation, and this further creates the requirements of protecting global input variables and preventing using the same variable to drive more than one input-destructive operations. 2) Analog PIM has DCC capability, which embeds NOT operations as part of TRA. 3) The analog PIM AAP primitive supports more than one output variable for efficient one-to-multi copying.

To meet the above requirements and fully exploit the computational potential of analog bit-serial PIM, we design a multi-step flow that transforms and optimizes a digital circuit with NOT/MAJ/AND/OR gates into an analog PIM compatible bit-serial IR, more comprehensive than prior work [4], [10], as shown in Fig. 4. Step ❶ is for replicating global inputs to avoid impact from analog TRA operations. Step ❷ is for mapping AND and OR gates into MAJ with additional zero and one inputs. Step ❸ is to eliminate inverters in the circuit by negating the inputs of MAJ and taking advantage of DCC. Step ❹, ❺, ❻ are three alternative approaches to resolve the input-destructive requirements of analog PIM, by leveraging the inout inputs and multi-outputs of TRA, or insert a copy as a final measure.

3) Scheduling Phase: We integrate LLVM [12] to perform instruction scheduling, register allocation, and spilling. This phase lowers bit-serial PIM IR-1 (before register allocation and spill insertion) to IR-2 with register allocation and spill insertion. For both digital and analog bit-serial PIM, a spilled register causes an additional pair of memory row read and write, which can impact bit-serial execution time substantially. We implement bit-serial-friendly Verilog inputs and perform priority-aware topological sort before scheduling to relieve register pressure, and we leave the exploration of optimal scheduling algorithms for future work.

4) Bit-Serial Code Generation Phase: The compiler translates bit-serial PIM IR-2 into executable bit-serial code. To facilitate result verification, we extend the PIMeval simulator to support bit-serial PIM primitives. Code generation is based on a set of PIMeval APIs for memory row read and write, digital PIM logical operations, analog PIM AP/AAP operations, and DCC row access. It is extensible to support other formats, including bitwise C code for bit-serial PIM IR verification.

5) Simulation & Verification Phase: Given the complexities of generated bit-serial code for digital and analog PIM, verification is essential. After compilation, the framework generates test functions and test inputs for the generated bit-serial code, and perform micro-op-level simulation using PIMeval.

IV. Evaluation

A. Bit-Serial PIM Timing Parameters

Timing parameters for performance evaluation are derived from a DDR4_8Gb_x16_3200 model listed in Table I.

TABLE I
Bit-Serial PIM Timing Parameters

Timing Parameter	Formula	Latency (ns)
DRAM clock period	tCK	0.63
DRAM row active time	tRAS = 52 * tCK	32.76
DRAM row precharge time	tRP = 22 * tCK	13.86
DRAM column-to-column delay	tCCD = 4 * tCK	2.52
Digital PIM row read or write	tRAS + tRP	46.62
Digital PIM logic operation	tCCD	2.52
Analog PIM AP/AAP operation	tRAS + tRP	46.62

B. Bit-Serial Compilation Benchmark Suite

The PIMsynth benchmark suite consists of Verilog implementations of 8/16/32/64-bit integer arithmetic, relational and logical operations, min/max, shift, and population count, for

This article has been accepted for publication in IEEE Computer Architecture Letters. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/LCA.2025.3600588

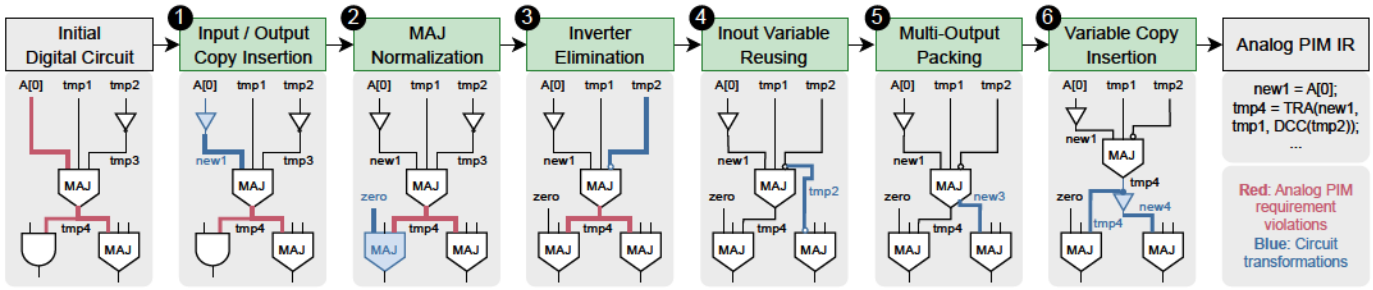

Fig. 4. Transformation and optimization steps to convert a digital circuit into analog PIM IR, by leveraging the input-destructive behavior of triple-row activation (TRA), dual-contact cell (DCC)-based NOT, and multi-output activate-activate-precharge (AAP) operations

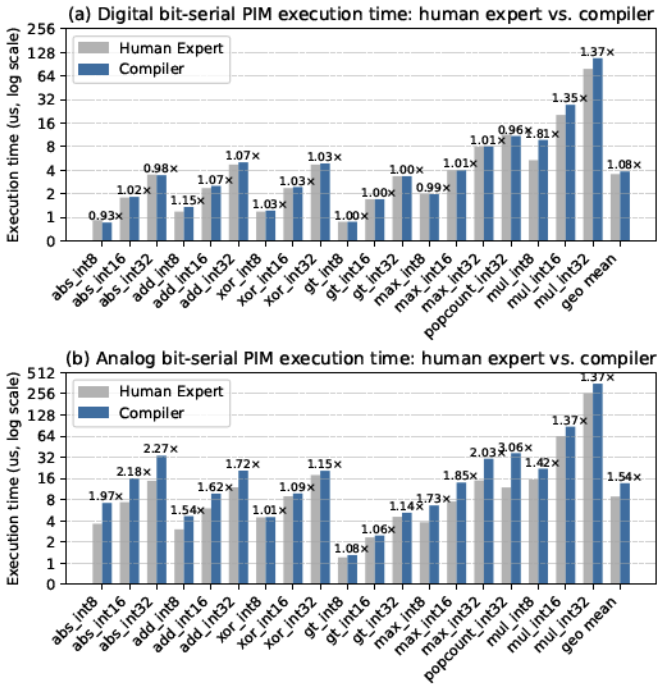

Fig. 5. Comparison of digital and analog bit-serial PIM execution time between human expert-written baseline and compiler generated code

evaluating compiler efficiency in mapping bit-parallel operations to bit-serial ISAs. The compiler and suite are designed to be readily extensible, e.g. to more complex operations such as floating-point arithmetic and look-up table (LUT) based logic.

C. Comparison Against Manually Optimized Baseline

We evaluate a subset of benchmarks selected based on baseline availability and types of operations. Baselines are highly compact bit-serial code optimized by human experts over months of effort, while the compiler generates results in seconds or minutes. Digital bit-serial PIM baseline results are obtained from [9], using an ISA of NOT/AND/XNOR/SEL, with 4 single-bit registers. Analog bit-serial PIM baseline results are collected from [4], [5], which models TRA/DCC operations using AP/AAP primitives in 6 register rows.

Figure 5 shows the comparison between bit-serial compiler-generated code and manually optimized baseline. For digital PIM, the results are strong, with performance comparable to or better than the manually optimized baseline, and a geometric mean of 1.08×. For analog PIM, the compiler-generated code remains competitive, with a geometric mean within 1.54× of the baseline. We observe compilation inefficiencies in analog operations such as add_int8 and max_int8, due to difficulties in exploiting the analog MAJ operation as efficiently as human experts. Despite these inefficiencies, the overall results demonstrate the effectiveness of the bit-serial compiler, providing significant reduction in manual effort.

V. Conclusions and Future Work

This paper presents a bit-serial compiler framework for digital and analog bit-serial PIM code generation. This achieves performance within 1.08× and 1.54× of hand-optimized baselines, while eliminating manual programming effort. Future work will explore improved synthesis and scheduling algorithms and extend support to a wider range of bit-serial targets, and compare to related compiler approaches.

References

[1] V. Seshadri *et al.*, "Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology," in *MICRO*, 2017.
[2] F. Gao *et al.*, "ComputeDRAM: In-memory compute using off-the-shelf DRAMs," in *MICRO*, 2019.
[3] X. Xin *et al.*, "ELP2IM: Efficient and low power bitwise operation processing in DRAM," in *HPCA*, 2020.
[4] N. Hajinazar *et al.*, "SIMDRAM: A framework for bit-serial SIMD processing using DRAM," in *ASPLOS*, 2021.
[5] G. F. Oliveira *et al.*, "MIMDRAM: An end-to-end processing-using-DRAM system for high-throughput, energy-efficient and programmer-transparent multiple-instruction multiple-data computing," in *HPCA*, 2024.
[6] S. Li *et al.*, "DRISA: A DRAM-based reconfigurable in-situ accelerator," in *MICRO*, 2017.
[7] T. Finkbeiner *et al.*, "In-memory intelligence," *IEEE Micro*, vol. 37, no. 4, 2017.
[8] R. Zhou *et al.*, "FlexiDRAM: A flexible in-DRAM framework to enable parallel general-purpose computation," in *ISLPED*, 2022.
[9] F. A. Siddique *et al.*, "Architectural modeling and benchmarking for digital DRAM PIM," in *IISWC*, 2024.
[10] X. Peng *et al.*, "CHOPPER: A compiler infrastructure for programmable bit-serial SIMD processing using memory in DRAM," in *HPCA*, 2023.
[11] S. Belaïd *et al.*, "Tornado: Automatic generation of probing-secure masked bitsliced implementations," in *Proc. of the Int'l. Conf. on the Theory and Applications of Cryptographic Techniques*, 2020.
[12] C. Lattner *et al.*, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004.
[13] C. Wolf, "Yosys open synthesis suite," 2016.
[14] R. Brayton *et al.*, "ABC: An academic industrial-strength verification tool," in *CAV*, July 2010.
[15] C. Tang *et al.*, "PIMLC: Logic compiler for bit-serial based PIM," in *DATE*, 2024.
[16] S. R. S. Raman *et al.*, "Sachi: A stationarity-aware, all-digital, near-memory, Ising architecture," in *HPCA*, 2024.
[17] S. R. S. Raman *et al.*, "SPARK: Sparsity aware, low area, energy-efficient, near-memory architecture for accelerating linear programming problems," in *HPCA*, 2025.