Instruction Set Design Methodology for In-Memory Computing through QEMU-based System Emulator

Kévin Mambu, Henri-Pierre Charles, Julie Dumas, Maha Kooli

▶ **To cite this version:**

Kévin Mambu, Henri-Pierre Charles, Julie Dumas, Maha Kooli. Instruction Set Design Methodology for In-Memory Computing through QEMU-based System Emulator. 32rd International Workshop on Rapid System Prototyping, Oct 2021, Rennes (Virtual Conference), France. hal-03449840

HAL Id: hal-03449840
https://hal.science/hal-03449840

Submitted on 2 Dec 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Instruction Set Design Methodology for In-Memory Computing through QEMU-based System Emulator

Kévin Mambu, Henri-Pierre Charles, Julie Dumas, Maha Kooli
Univ. Grenoble Alpes, CEA, LIST
F-38000 Grenoble, France
firstname.name@cea.fr

***Abstract*—In-Memory Computing (IMC) is a promising paradigm to mitigate the von Neumann bottleneck. However its evaluation on complete applications in the context of full-scale systems is limited by the complexity of simulation frameworks as well is the disjunction between hardware exploration and compiler support. This paper proposes a global exploration flow in the scale of Instruction Set Architectures (ISA) to perform both modeling and the generation of compiler support to perform ISA-level exploration. Our emulation methodology is based on QEMU, implements a performance model based on hardware characterizations from the State-of-the-Art, and allows the modeling of cache hierarchies, while our compiler support is automatically generated and based on a specialized compiler. We evaluate three applications in the domains of image processing and linear algebra on a reference IMC architecture, and analyze the obtained results to validate our methodology.**

***Index Terms*—QEMU, IMC, Instruction Set Design, Cache Modeling, Power Modeling, Compiler, System Emulation**

I. Introduction

Von Neumann architectures are inherently limited by performance bottlenecks, in the form of the *Memory Wall* and the *Energy Wall*. Previous works such as [1] show that the discrepancy in energy cost between a CPU and L1 cache can vary between $\times 10$ for 8 KB and $\times 100$ for 1 MB memories. In-Memory Computing (IMC) is a solution to implement non-von Neumann architectures and mitigate the Memory Wall bottleneck by moving computation directly to memory instead of exchanging data between it and the CPU, in order to achieve a wider level of parallelism while reducing bandwidth usage, and thus, the latency cost and power consumption of computer architectures. The integration of IMC to conventional von Neumann architectures opens the path to new challenges, as the performance of an IMC architecture depends on the operations supported by its instruction set but also its integration in a full-scale system with a complex memory hierarchy and access to I/O and peripherals, *e.g.* network cards. However, this need for IMC to be evaluated in a real-life setup does not scale to the level of abstraction of most simulation frameworks, and the complexity required to model such systems in order to perform the evaluation of IMC in a system integrating the previously mentioned features to the level of Instruction Set Architectures (ISAs).

Moreover, the exploration of emerging architectures such as IMC is often disjointed between hardware and software exploration, in terms of compiler & language support for the latter. This discontinuity in terms of research methodologies makes the bridge between emerging architectures and compilers difficult to sustain, as low-level modeling and micro-benchmarking first need to be performed before considering compiler support even though performing ISA-level exploration allows for the characterization of results interesting for HW-SW exploration.

In this paper, we propose a complete environment to perform exploration of IMC architectures at the level of ISAs, with complete applications able to be run on full-scale systems instead of micro-benchmarks while also generating the compiler support for these architectures. The goal of this ISA-level design methodology is to quickly and effortlessly perform exploration beneficial for software and hardware exploration later on.

The rest of the paper is organized as follows. Section II presents a State-of-the-Art regarding simulation frameworks and general-purpose and IMC-dedicated software compiler solutions. Section III presents our reference IMC architecture, *e.g.* its instruction set, its programming model and its system integration. Sections IV and V focus respectively on the modeling and software stack of our environment, while section VI is an overview of the global exploration flow. Finally, section VII describes the evaluation of a reference IMC architecture on three applications – Frame differencing, Sobel filter & Matrix multiplication – as well as a commentary of the results, and section VIII exposes our conclusion and perspectives.

II. Related Works

A. Modeling & emulation frameworks

Frame -work	Accu -racy[1]	System type[2]	Develop -ment effort	Simulation speed	Fidelity
LLVM	I	CM	+	+	+
gem5	C	S	+++	+++	+++
SystemC	C	M	++	++	+++
QEMU	I	ISS / S	+/ +++	+/ +	++/ +++

+++: High, ++: Moderate, +: Low
[1]*I: Instruction-accurate, C: Cycle-accurate*
[2]*CM: Cost Model, S: System, M: Micro-architecture, ISS: Instruction Set Simulator*

TABLE I
Comparison of various simulation flows

Table I presents the comparison of different simulation frameworks found in the literature, the retained criteria are

the type of architecture modeled and their accuracy, but also their fidelity to a physical implementation and the development effort required to perform ISA exploration. [2] implements an instruction-level cost model based on LLVM to emulate its energy cost by instrumenting LLVM bytecode. This is a very fast method of prototyping in terms of development effort and simulation speed, but it shows low fidelity to a physical implementation. Other solutions such as [3] [4] [5] use elaborate frameworks such as SystemC or gem5 to perform cycle-accurate system modeling. These frameworks show very high fidelity to physical implementations but low simulation speed due to their complexity. Moreover, the development effort required to model ISAs does not always scale to ISA-level exploration, that is to the level of instructions and user-visible resources. We retain QEMU [6] as a suitable tool due to its capabilities of full-system emulation, based on instruction-level translation, but also its potential to perform rapid prototyping and exploration. Previous works such as [7] and [8] base their simulation environments on QEMU to perform respectively micro-architectural simulation and HW-SW co-simulation with SystemC. Our present work shows that QEMU is also able to perform ISA-level exploration for the quick evaluation of IMC architectures at system-level.

B. Software solutions for In-Memory Computing

General-purpose code compilers such as `gcc` and `clang` can be used to program IMC architectures through macros, like [9] for example. The problem is that this solution shows limited expressiveness and code portability, as such a dedicated solution for IMC would be preferable. In the literature, there exist few compiler solutions specialized for IMC architectures and flexible enough to perform ISA exploration. [10] proposes a compiler technique and a decision model to efficiently offload instructions to IMC architectures but does not describe a code compiler per se. [11] presents the Duality Cache architecture, which implements a subset of Nvidia's PTX assembly instructions, and a compilation flow based on the Nvidia CUDA compiler `nvcc`. This solution is effective for compatible source code, but it is tied to a specific ISA and lacks the flexibility desired to compile code to target various IMC instruction sets. Moreover, the use of CUDA induces significant overhead due to data and instruction scheduling which might be difficult to control and evaluate in the scope of ISA exploration. [12] is the most flexible solution we found in terms of IMC support, an LLVM-based compiler tool-chain able to auto-vectorize scalar C code and targeting IMC architectures. However, it is implemented as part of the Intel x86 back-end of LLVM, which limits the degree of exploration achievable to evaluate the ISA at system-level. Moreover, the use of automatic vectorization makes evaluation difficult to control, and a more explicit solution is preferable. We decide to base our software stack on the Hybrogen environment [13], a compiler environment targeting heterogeneous architectures. Its programming language and dedicated compiler, HybroLang, is able to compile a domain-specific language with dedicated data types to lower-level languages such as C, which makes it easy to use and integrate in an existing compilation flow. It also integrates a database manager to organize various ISAs. We adapt this compilation environment for the support of IMC architectures modeled through our simulation stack.

III. In-Memory Computing Architecture

A. Computational SRAM

The Computational SRAM (C-SRAM) architecture emulated in this paper, is an SRAM-based IMC architecture able to perform computation directly inside the memory array, thus reducing bandwidth utilization by substituting multiple data transfers with fewer C-SRAM instructions. In the base specification, computation is performed between rows — *i.e.* on physically aligned data in the memory array — by using an ALU in its periphery. ALU operations can be parametered to perform parallel computation on 8-bit to 32-bit operations. Previous works [2], [14], [15] provided details regarding the specification, the design and the characterization of this architecture, and additional works [9], [16] investigated C-SRAM integration in elaborate scenarios. In this paper, our experimental C-SRAM architecture is a 128-bit single unit supporting 16×8 up to 32×4 vector operations.

B. Programming Model

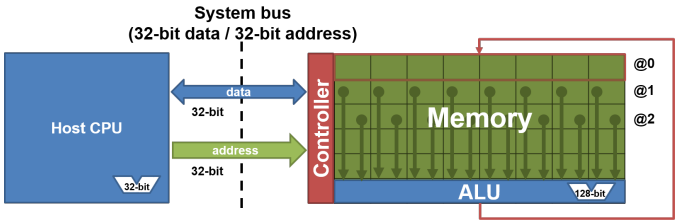

Fig. 1. The architectural integration of C-SRAM allows IMC instructions to be sent by host CPU through the address and data buses.

Fig. 1 presents the programming model of the C-SRAM from the point-of-view of the developer. The host CPU programs C-SRAM using a dedicated instruction set through its address and data bus to make host-agnostic integration possible.

C. Instruction Set Architecture

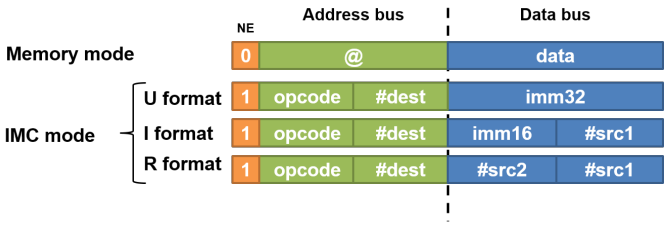

Fig. 2. C-SRAM Instruction Formats.

As seen on Fig. 2, the operating mode of the C-SRAM when receiving requests can be toggled using the Most Significant

Bit (MSB) of the address bus, which we call NE. $NE = 0$ sets the C-SRAM in Memory Mode, with data bus and address bus containing resp. the address and data of the transfer request issued by the host CPU. $NE = 1$ sets the C-SRAM in Computing Mode, the address bus contains the 32 left-most bits of the instruction while the data bus contains its 32 right-most bits, allowing us to implement 64-bit instructions. The C-SRAM instruction set defines three instruction formats: the R-format for 2-row operations, the I-format for 1-row 1-immediate operations and the U-format for 1-immediate operations.

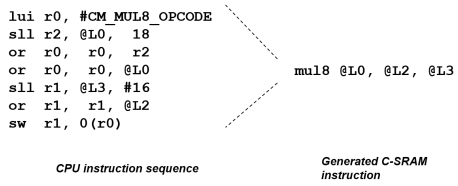

Fig. 3. Generation of C-SRAM instructions by the host CPU.

The host CPU generates C-SRAM instructions on-the-fly by encoding them into store instructions, as previously mentioned. Fig. 3 shows the generation of an 8-bit vector multiplication C-SRAM instruction. This example code shows a data dependency between the two consecutive C-SRAM instructions. This dependency issue is addressed in the implementation of the C-SRAM architecture through various micro-optimizations, in order to execute both instructions in a single clock cycle. The operand addresses `Lx` does not correspond to the virtual addresses of the vectors but their physical addresses, *i.e.* their row index. Determining the row index from the virtual address can be done by performing simple masking and shifting operations. Each field is initialized in accordance to the C-SRAM ISA, and the `sw` instruction issues the instruction to the C-SRAM unit, with the address register encoding the left-most bits and the data register the right-most bits of the instruction — resp. `r0` and `r1` on Fig. 3. The resulting execution model of a C-SRAM architecture can be seen on Fig. 4. The interleaving of general-purpose CPU instructions and C-SRAM instructions enables mixed parallelism without altering the host ISA.

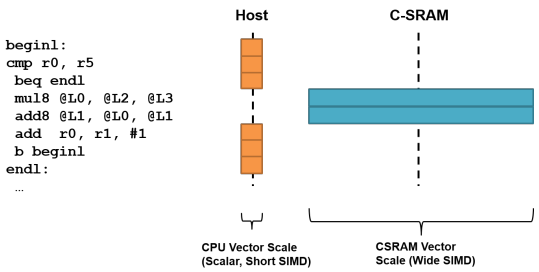

Fig. 4. Interleaved instruction flow between host CPU and C-SRAM.

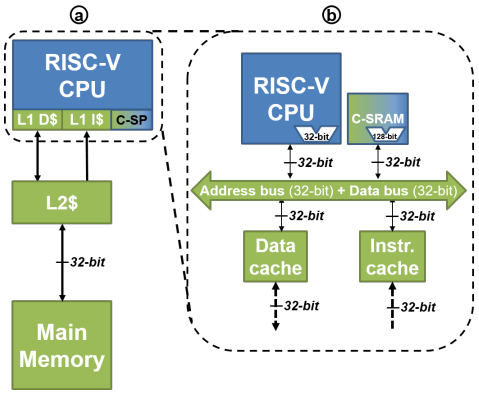

Fig. 5. Integration of a C-SRAM unit in a computer architecture with a 2-level cache hierarchy.

D. System integration of the Computational SRAM

Due to its architecture-independent programming model, the C-SRAM can be integrated in various computer architecture layouts, from low-power embedded computing architectures with a Micro-Controller Unit (MCU) and a long-latency memory, to High-Performance Computing (HPC) architecture.

The architecture layout used for this paper is presented on Figure 5: a 2-level cache hierarchy with L1 data and instruction cache units, and a L2 unified cache connected between the L1 cache level and the main memory. It can be integrated in a Multi-Processor System-on-Chip (MPSoC) architecture.

IV. QEMU-based System Emulator

A. QEMU Overview

QEMU is a fast machine emulator using a mechanism known as Dynamic Binary Translation (DBT). An internal dynamic translator, the Tiny Code Generator (TCG) scans guest binary as a tree of blocks and translate them into semantically equivalent host assembly code to execute on the host architecture. the TCG also implements an IR to perform code manipulation between guest and host binary.

a) QEMU operating modes: Linux user mode uses a CPU model for the TCG to translate guest code to the host, and system calls are handled by the host OS. This operating mode is similar to *semi-hosting mode* in ARM CPU architectures. *System mode* describes an entire Virtual Machine (VM) with one or multiple cores, access buses, memory units and peripherals to run applications, in most cases a guest OS. This mode can be used to model complete architecture with a high level of fidelity, the modeled platforms being described in C through an internal API. We decide to implement our architecture models using *Linux user mode* for the benefits of modeling semi-hosted systems, which delegate system calls to the host OS. With this solution, run-time instrumentation can be focused solely on the binary code of the application to evaluate instead of also instrumenting a guest OS, while still making accesses to I/O and peripherals available through the host OS.

b) TCG Plugin System: The QEMU TCG Plugin System allows for end-users to implement out-of-tree features using dynamically linked libraries. The original API is primarily designed for code instrumentation. While the Plugin system does not natively expose internal functionalities and structures, it is possible to use it to implement call-backs to specific run-time events out-of-tree, without modifying the main source code of the QEMU project.

B. Simulation Methodology

Component	Events
CPU	Executed arithmetic/branch instruction, Executed load instruction, Executed store instruction
C-SRAM	Load request, Store request, Executed C-SRAM instruction
Cache memory	Load request, Store request, Cache look-up, Read miss, Write miss
Main memory	Load request, Store request

TABLE II
LIST OF THE EVENTS GENERATED AND COUNTED AT RUN-TIME BY OUR SIMULATION METHODOLOGY, PER MODELED COMPONENT.

At translation time, every executed guest instruction triggers callbacks available through TCG plugins for users to set their own instrumentation helpers. Though *Linux user mode* QEMU does not model architectural details such as the memory hierarchy, we are able to functionally model our IMC architecture and cache memories by writing TCG helpers performing event counting & generation at run-time. Table II shows the list of components we are able to model, and the associated list of events generated and counted for each.

a) Emulation of IMC instructions: The programming model of C-SRAM, as explained in the previous section, describes the issuing of instructions by the host CPU through the address and data buses to the C-SRAM controller to decode and execute operations on memory. This non-intrusive integration allows IMC instructions to be modeled without altering the host ISA, by scanning CPU load/store instructions whose address corresponds to accesses to C-SRAM in memory mode. From that, the TCG helper calls for a "decode tree" function to extract the arguments of the C-SRAM instruction from the parameters of the memory access and execute the appropriate function.

b) Cache modeling: QEMU does not perform native cache modeling in *Linux user mode*, but we wanted the possibility of performing ISA exploration of IMC depending on its providing instruction set but also its integration in a multi-level cache hierarchy. In order to model complete memory hierarchies, we describe generic call-backs and data structures in order to emulate cache controllers depending on their write-policy. For the work presented in this paper, we implement cache memories with write-through or write-back policies, and with pseudo-random replacement policy [17]. Cache read/write call-backs can be chained to model a multi-level memory hierarchy from CPU to main memory.

1) Performance and Energy Model: Let I a given instruction of the binary run on the architecture described through our methodology:

$$E_{total} = \sum_{I=First\ instruction}^{Up\ to\ completion} f_{Instruction}(I, ArchState) + f_{MemAccess}(I, ArchState) + f_{C\text{-}SRAM}(I, ArchState) \quad (1)$$

$$L_{total} = \sum_{I=First\ instruction}^{Up\ to\ completion} g_{Instruction}(I, ArchState) + g_{MemAccess}(I, ArchState) + g_{C\text{-}SRAM}(I, ArchState) \quad (2)$$

Equations 1 & 2 describe respectively the energy and latency model of our simulation methodology. The total energy and latency cost E_{total} and L_{total} are the sum of the latency and energy cost evaluated at every executed instruction, depending on on their type – Non-memory access instruction, memory access instruction or C-SRAM instruction – and the architectural state at said instruction.

V. The Hybrogen compiler environment

A. HybroLang Programming Language

```
#[
int 32 1 ImageDiff(sint[] 8 16 a, sint[] 8 16 b, sint[] 8 16
res, int 31 1 len)
{
        int 32 1 i;
        for(i = 0; i < len; i = i + 1)
        {
                res[i] = a[i] - b[i];
        }
        return 0;
}
]#
```

Data width — Vector width — Specialized data types

Memory accesses According to data type

Fig. 6. Language features of HybroLang in the sample code of Frame differencing : support of specialized data types and parameterization of variables <data_width vector_width>

The HybroLang language is heavily inspired by the C language for its data type operands (`int`, `float`, pointers, *etc*), but also implements more specialized data types variants such as `sint`, to enable the support saturated arithmetic [13]. It also supports the explicit parameterization of variables with a pair <`data_width vector_width`>, as seen on Figure 6. These parameters make possible to explicitly perform vector computation on heterogeneous architectures without the need for intrinsic functions, using the operator symbols provided by the grammar of HybroLang. Finally, HybroLang is designed to be integrated in a host source code of different language (*e.g.* C, Javascript, Python) and trans-compiled source-to-source to generate the described code at run-time. Our instruction set design methodology uses a C back-end for debug & evaluation purposes.

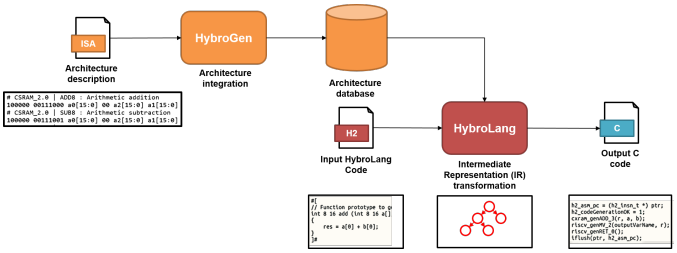

Fig. 7. Code generation flow of HybroGen & HybroLang

B. HybroGen Compilation Tool-Chain

The Hybrogen environment includes two programs: `HybroGen`, which manages the database of supported ISAs and `HybroLang`, the code compiler implementing the language. Figure 7 represents the code generation flow the HybroGen tool-chain. `HybroGen` takes as main input a .isa file, describing the various operations supported by a given architecture, the encoding of its instructions as well as their formats and stores this information in a database. `HybroLang` compiler transforms .hl input code into an Intermediate Representation to perform various manipulations before writing it back into host language — in our case, into C code interleaving CPU and C-SRAM instructions. This output code can then be compiled with any C compiler (*e.g.* `gcc`, `clang`), making the Hybrogen environment easy to integrate in existing project builds.

VI. Global Emulation, Compilation & Evaluation Flow

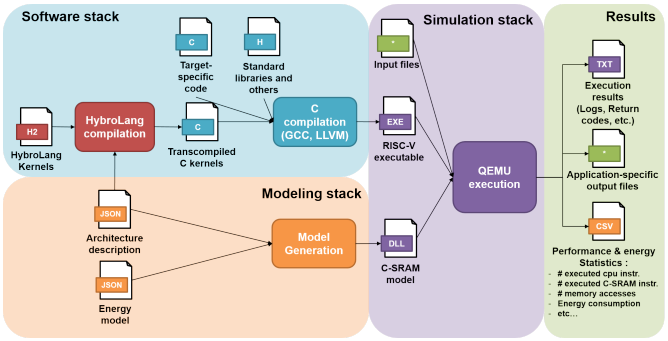

Fig. 8. Proposed compilation & simulation flow: software stack allowing the code generation using HybroGen compilation environment & HybroLang programming language presented in [13], modeling stack representing C-SRAM architecture, simulation stack representing the QEMU-based propose platform, and results giving statistics regarding energy and performance.

We propose in this article a unified emulation, compilation & evaluation flow to model, evaluate and compile code for IMC-based heterogeneous systems.

Fig. 8 presents the model generation steps. The inputs are an architecture describing the memory hierarchy, and an energy model with latency and energy costs for each component, taken from post-Place-and-Route evaluations, the literature and silicon measurements. Both inputs are written in JSON format, and passed to a Python tool-chain to generate the C sources of the plugin, which is compiled against the QEMU project as a dynamic library. The result plug-in can be loaded at launch-time to an unaltered QEMU binary to emulate the behavior of a computer architecture with a memory hierarchy integrating a C-SRAM unit. As of now, the cost of data transfers between the CPU, the C-SRAM and the memory hierarchy are modeled, while the cost of data transfers with I/O and peripherals are abstracted. This allows us to perform the evaluation of full-scale applications using I/O and peripherals at a minimal development cost.

VII. Experimental Results

A. Application Kernels

Application	Vector Element size	Complexity	Pattern type	Avg. data redundancy
Frame differencing	8-bit	$\mathcal{O}(n)$	Row-major	1
Sobel filter	16-bit	$\mathcal{O}(n)$	Complex	≈ 18
Matrix multiplication (squares)	32-bit	$\mathcal{O}(n^3)$	Row-major / Column-major	n^2

TABLE III
Characteristics of retained applications.

Table III presents the three applications implemented on the C-SRAM architecture. The criteria we retain to evaluate the results of our experimentation are their algorithmical complexity and their vector element size, which have an impact on the arithmetical complexity of the applications and whether ot not the main generated access pattern is regular or complex. We also define for each application their *Average data redundancy factor*, *e.g.* the number of times an input element is accessed during the entire life cycle of the application:

- *Frame differencing*, used in Computer Vision to perform motion detection [18], is an application which performs saturated subtraction between two (or more) consecutive frames in a video stream to detect differences. It has linear complexity in both computing and memory.
- *Sobel filter* is an application which applies two 3×3 convolution kernels on an input image to generate its edge-highlighted output. It is a standard operator in Image Processing as well as Computer Vision to perform edge detection [**?**]. It has linear complexity in computing an memory, and shows constant data redundancy (2×9 reads per input pixel, on average).
- *Matrix-matrix multiplication* is used in various domains such as Signal Processing or physics modeling, and a standard of Linear Algebra as the `gemm` operator [19]. It has cubic ($\mathcal{O}(n^3)$) complexity in computing and memory, and shows quadratic ($\mathcal{O}(n^2)$) data redundancy.

The variety of this selection in terms of complexity and data reuse generates various distinct run-time behaviors to analyze. All three applications were retained for their importance and relevance to domains such as image processing and computer

vision. Most are defined as standard functions in Domain-Specific APIs such as OpenVX [20].

B. Results

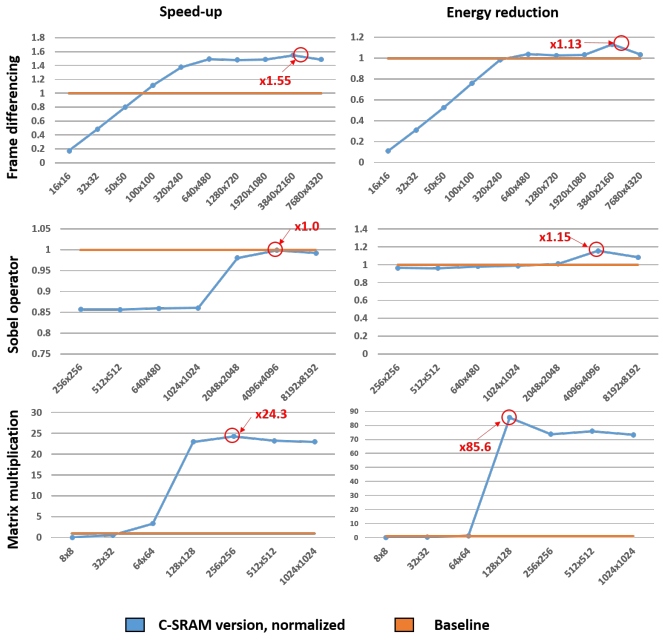

Fig. 9. Normalized speed-up and energy evaluated on the C-SRAM architecture for each application.

1) Speed-up and Energy Reduction: Figure 9 presents the speed-ups and energy reduction measured for each applications on both experimental architectures.

For Frame Differencing, the execution time of the C-SRAM implementation is inferior to the scalar implementation on 16×16, 32×32 and 50×50 resolutions. From resolution 100×100 onward, the C-SRAM implementation performs up to 55% faster than the scalar implementation. The energy reduction of the C-SRAM implementations follows the same trend with an improvement of 2 to 13% starting 320×240 resolution. Both statistics present a roofline starting between 100×100 and 320×240 resolutions. The cause of this limitation is the capacity of the L1 data cache and the bandwidth between it, the host CPU and the C-SRAM unit.

Sobel filter performs worst on the C-SRAM implementation than the scalar implementation for image resolutions 256×256 to 1024×1024. Resolution 2048×2048 shows a significant threshold where the performance reaches that of the scalar implementation. This is because the size of the input data set becomes large enough to compensate for the cost of data management. Moreover, the data management code of the C-SRAM implementation makes use of the data redundancy in sobel filter to limit the number of accesses towards the rest of the memory hierarchy. Its energy cost relative to the baseline presents a similar threshold between resolutions 1024×1024 and 4096×4096 before trending towards 15%.

For Matrix Multiplication, both the execution time and the energy consumption are higher on the C-SRAM implementation than the scalar implementation for trivial instances: 8×8 and 32×32 matrix dimensions. Starting matrix dimension 64×64, the size of the input data set is large enough to amortize the cost of data management, and matrix dimension 128×128 shows a significant theshold, due to the reuse of data stored in C-SRAM, compared to the scalar implementation. The overall performance trends towards a speed-up of $\times24$ and an energy reduction of $\times75$, compared to the scalar implementation.

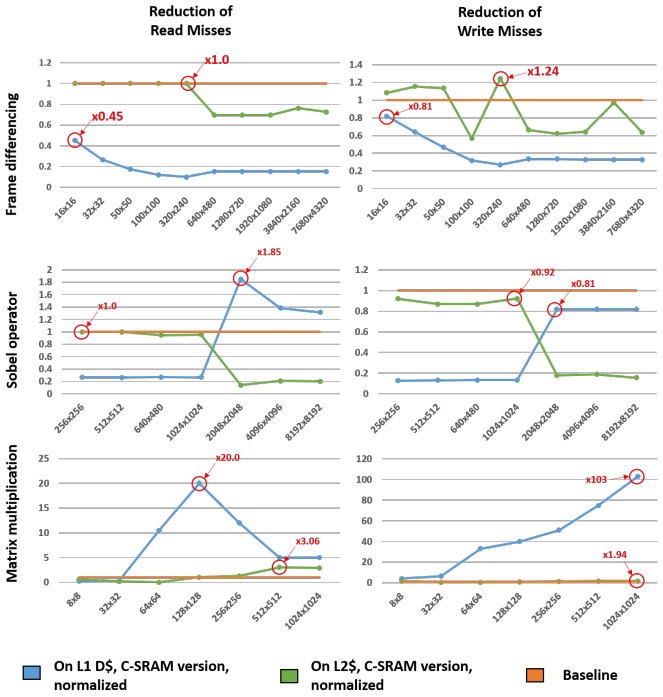

Fig. 10. Normalized reduction of read & write misses evlauated on the C-SRAM architecture for each application.

2) Impact of the memory hierarchy on the performance: Figure 10 presents the reduction of read and write misses induced by the addition of C-SRAM for each application. All three presented applications are inherently memory-bound, and the current integration of the C-SRAM at the bottom of the memory hierarchy without any dedicated transfer bus between it and the main memory makes performance gains inherently dependent on the memory hierarchy. A large number of cache misses happen due to the limited capacity of the cache memories but also because of the conflict misses between control variables (iterators, generated addresses, etc) and the input data destined for the C-SRAM unit. Moreover, the lack of dedicated transfer mechanisms to assist the CPU creates a consequential overhead in terms of CPU instructions. Finally, the limited bandwidth between each cache memory further aggravates the impact of cache misses on the performance.

Frame differencing and sobel filter are linear with limited opportunities of data reuse to compensate the eventual cache

misses. The reduction of cache misses on the L1 data cache vary between the two however, in the case of Frame Differencing the reduction of read & write misses is consistently increasing, due to the conflict misses between the input data and the limited capacity of the L1 cache.

For sobel filter, the reduction of L1 misses increases with the size of the input data set up to 2048×2048 images before stabilizing. This is due to the generated access patterns to input data being more regular on the C-SRAM version than the scalar version and also the data duplication being performed by accessing data already stored in the C-SRAM. However, the high complexity of the data management code and the limitation of the memory bandwidth still impede the overall performance, as can be seen on the reduction of read & write misses decreasing for the L2 cache.

The overall reduction of cache misses in the case of Matrix Multiplication is very high for the Matrix Multiplication due to the loop tiling. The highest tiling factor achieved with the dimensions of the experimental architecture was $8 \times 8 \times 4 = 256$, which greatly maximize the data reuse, compared to the scalar implementation of Matrix Multiplication. The data size which presents peak performance in terms of reduction of read misses as well as overall performance is dimension 128×128, with a read miss reduction factor of $\times 20$ on the L1 data cache, as well as a peak energy reduction of $\times 85$. On dimension 1024×1024, its peak reduction of L1 write misses is achieved at $\times 103$.

All previous observations are coherent with the criteria retained for each application on Table III, as memory-bound linear applications with fewer data redundancy offer less opportunities of performance gains than memory-bound cubic applications with higher data redundancy.

VIII. Conclusion

In this paper, we present a novel exploration flow to perform exploration of IMC architectures at ISA-level with complete applications running on full-scale systems instead of micro-benchmarking. This exploration flow allows for the effortless and quick exploration beneficial for software and hardware exploration later on, by generating both the simulation model and the compiler support for a given IMC ISA. We evaluate a reference architecture on three applications to validate our simulation methodology, and show that the results are coherent with the algorithmical complexity of our applications and with the dimensions of out reference architecture.

We plan to use this exploration flow for other paradigms such as Stochastic Computing, and Quantum Computing to evaluate the integration of quantum accelerators in traditional von Neumann systems.

References

[1] M. Horowitz, "1.1 Computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb. 2014, pp. 10–14, iSSN: 2376-8606.

[2] M. Kooli, H.-P. Charles, C. Touzet, B. Giraud, and J.-P. Noel, "Smart Instruction Codes for In-Memory Computing Architectures Compatible with Standard SRAM Interfaces," p. 6.

[3] Y. M. Qureshi, W. A. Simon, M. Zapater, D. Atienza, and K. Olcoz, "Gem5-X: A Gem5-Based System Level Simulation Framework to Optimize Many-Core Platforms," in *2019 Spring Simulation Conference (SpringSim)*, Apr. 2019, pp. 1–12.

[4] W. A. Simon, Y. M. Qureshi, M. Rios, A. Levisse, M. Zapater, and D. Atienza, "BLADE: An in-Cache Computing Architecture for Edge Devices," *IEEE Transactions on Computers*, vol. 69, no. 9, pp. 1349–1363, Sep. 2020, conference Name: IEEE Transactions on Computers.

[5] R. Gauchi, V. Egloff, M. Kooli, J.-P. Noel, B. Giraud, P. Vivet, S. Mitra, and H.-P. Charles, "Reconfigurable tiles of computing-in-memory SRAM architecture for scalable vectorization," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*. Boston Massachusetts: ACM, Aug. 2020, pp. 121–126. [Online]. Available: https://dl.acm.org/doi/10.1145/3370748.3406550

[6] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, Apr. 2005, p. 41.

[7] N. Derumigny, F. Gruber, T. Bastian, C. Guillon, L.-N. Pouchet, and F. Rastello, "From micro-OPs to abstract resources: constructing a simpler CPU performance model through microbenchmarking," *arXiv:2012.11473 [cs]*, Jan. 2021, arXiv: 2012.11473. [Online]. Available: http://arxiv.org/abs/2012.11473

[8] A. Charif, G. Busnot, R. Mameesh, T. Sassolas, and N. Ventroux, "Fast Virtual Prototyping for Embedded Computing Systems Design and Exploration," in *RAPIDO2019 - 11th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, Valence, Spain, Jan. 2019, pp. 1–8. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02023805

[9] R. Gauchi, M. Kooli, and P. Vivet, "Memory Sizing of a Scalable SRAM In-Memory Computing Tile Based Architecture," p. 6.

[10] R. Hadidi, L. Nai, H. Kim, and H. Kim, "CAIRO: A Compiler-Assisted Technique for Enabling Instruction-Level Offloading of Processing-In-Memory," *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 4, pp. 1–25, Dec. 2017.

[11] D. Fujiki, S. Mahlke, and R. Das, "Duality Cache for Data Parallel Acceleration," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2019, pp. 1–14, iSSN: 2575-713X.

[12] H. Ahmed, P. C. Santos, J. P. C. Lima, R. F. Moura, M. A. Z. Alves, A. C. S. Beck, and L. Carro, "A Compiler for Automatic Selection of Suitable Processing-in-Memory Instructions," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2019, pp. 564–569, iSSN: 1558-1101.

[13] J. Dumas, H.-P. Charles, K. Mambu, and M. Kooli, "Dynamic compilation for transprecision applications on heterogeneous platform," *Journal of Low Power Electronics and Applications JLPEA*, vol. 11, no. 3, 2021.

[14] J.-P. Noel, V. Egloff, M. Kooli, R. Gauchi, J.-M. Portal, H.-P. Charles, P. Vivet, and B. Giraud, "Computational SRAM Design Automation using Pushed-Rule Bitcells for Energy-Efficient Vector Processing," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, pp. 1187–1192.

[15] J.-P. Noel, M. Pezzin, R. Gauchi, J.-F. Christmann, M. Kooli, H.-P. Charles, L. Ciampolini, M. Diallo, F. Lepin, B. Blampey, P. Vivet, S. Mitra, and B. Giraud, "A 35.6 TOPS/W/mm^2 3-Stage Pipelined Computational SRAM With Adjustable Form Factor for Highly Data-Centric Applications," *IEEE Solid-State Circuits Letters*, vol. 3, pp. 286–289, 2020, conference Name: IEEE Solid-State Circuits Letters.

[16] V. Egloff, J.-P. Noel, M. Kooli, B. Giraud, L. Ciampolini, R. Gauchi, and C. Fuguet, "Storage Class Memory with Computing Row Buffer: A Design Space Exploration," p. 6.

[17] "ARM Cortex-A Series Programmer's Guide." [Online]. Available: https://developer.arm.com/documentation/den0013/d/Caches/Cache-policies/Replacement-policy

[18] G. Thapa, K. Sharma, and M. M.K.Ghose, "Moving Object Detection and Segmentation using Frame Differencing and Summing Technique," *International Journal of Computer Applications*, vol. 102, no. 7, pp. 20–25, Sep. 2014. [Online]. Available: http://research.ijcaonline.org/volume102/number7/pxc3898647.pdf

[19] "BLAS (Basic Linear Algebra Subprograms)." [Online]. Available: http://www.netlib.org/blas/

[20] "OpenVX - Portable, Power-efficient Vision Processing," Dec. 2011, section: API. [Online]. Available: https://www.khronos.org/openvx/