Intrinsics-HMC: An Automatic Trace Generator for Simulations of Processing-In-Memory Instructions

Aline Santana Cordeiro, Tiago Rodrigo Kepe, Diego Gomes Tomé, Eduardo Cunha de Almeida, Marco Antonio Zanata Alves

[1]Department of Informatics – Federal University of Paraná

{asc12, trkepe, dgtome, eduardo, mazalves}@inf.ufpr.br

Abstract. *Processor-in-Memory (PIM) architectures, such as the Hybrid Memory Cube (HMC), are emerging nowadays as a solution for processing large amount of data directly inside the memory. In this area, several researchers are proposing and evaluating new instructions and new PIM architectures. For such evaluations, trace-driven simulators, as the Simulator of Non-Uniform Cache Architectures (SiNUCA), are commonly used in order to model these new proposed systems. Such simulators provide fast prototyping of new architectures, while it requires the researcher to write simulation traces manually when evaluating new Instruction Set Architecture (ISA) proposals, which is an time consuming and error prone task. In this work, we propose a methodology for fast generation of simulation traces focused on HMC architecture, which consists on a high-level Intrinsics-HMC library and a modification inside the trace-generator tool from SiNUCA. Our proposal enables the researchers to write high level code in C/C++ languages using our library, which mimics the behavior of HMC instructions. These codes can be compiled and executed in traditional x86 architectures for verification. After ensure the code is correct and working, the user can use our modified version of SiNUCA-Tracer to translate HMC functions into HMC instructions know by the simulator, providing a convenient solution to generate traces and fast simulations of new PIM architectures. Results using the proposed technique applied on database application kernels show the correct translation and simulation of new HMC instructions using SiNUCA.*

1. Introduction

Processor-in-Memory (PIM) architectures [Patterson et al. 1997], [Elliott et al. 1999], [Balasubramonian et al. 2014], as the new Hybrid Memory Cube (HMC), are emerging in the last few years after the release of 3D-stacking technologies [Olmen et al. 2008]. The HMC presents high parallelism between the Dynamic Random Access Memory (DRAM) banks, ensuring low average latency during high pressure in memory (memory bursts). The HMC also supports PIM in order to mitigate data movement between memory and processor, where processing occurs in the same chip from the memory. Thus, many researchers are evaluating performance and energy consumption of existing HMC [Jeddeloh and Keeth 2012], [Khalifa et al. 2013], [Hadidi et al. 2017] or proposed new PIM architectures [Pugsley et al. 2014], [Alves et al. 2016].

In general, the processor designers and researchers relies on full-system or trace-driven simulators to evaluate performance of new Instruction Set Architecture (ISA) and architectural components. Full-system simulators require executable binaries compiled

for the specific ISA to be simulated, requiring thus a compiler ready for such new systems. On the other hand, trace-driven simulators are more flexible and deterministic, requiring only the simulation trace, which contains the instructions recognized by the simulator and the dynamic execution order of such instructions. The generation of traces is usually performed automatically using binary instrumentation tools for the existing ISA. However, for new PIM architectures, the instruction traces must be manually generated, using many times, the help of scripts to generate the dynamic traces. This manual task is error-prone and can demand a considerable amount time, depending on the complexity of the program to be simulated.

In this context, the main objective of this paper is to propose a method that allows automatic generation of simulation traces that uses HMC instructions directly from a high-level compiled program. We developed a library called Intrinsics-HMC that provides a series of functions that emulate the HMC behavior, based on the HMC specification version 2.1 [HMC Consortium 2017]. This library was written in C/C++ and uses x86 instructions only, so it can be normally linked, compiled and executed just to assure its correct operation. After the developer validates the code, the trace generator can be used to identify the HMC functions in the Intrinsics-HMC library and convert them to HMC instructions recognizable by the simulator. In order to demonstrate our translations of HMC functions and simulate the traces generated automatically, we used the Simulator of Non-Uniform Cache Architectures (SiNUCA) [Alves et al. 2015]. We also used the SiNUCA-Tracer and the dynamic binary instrumentation tool *Pin* from Intel to extract the simulation traces from the full execution of two database kernels developed in C++ language making use of our Intrinsics-HMC library.

The rest of this paper is organized, as follows: Section 2 details the basic concepts about HMC and SiNUCA. Section 3 explains how execution traces are generated, introducing the Intrinsics-HMC and the SiNUCA-Tracer. Section 4 analyzes the simulation results and Section 5 presents some related work. Finally, Section 6 concludes this paper and proposes future directions.

2. General Concepts

In this section, we present basic concepts about Hybrid Memory Cube (HMC) and an overview about the Simulator of Non-Uniform Cache Architectures (SiNUCA).

2.1. Hybrid Memory Cube – HMC

HMC is a memory device formed by up to 8 stacked layers of Dynamic Random Access Memory (DRAM) and the base is a logic layer, as illustrated in Figure 1. The HMC is logically partitioned in 32 vaults. Each vault has a dedicated memory controller located in the logic layer and up to 16 DRAM banks (distributed among the layers of DRAM) connected together using Through-Silicon Vias (TSVs) [Olmen et al. 2008].

The HMC presents high parallelism for accessing data [HMC Consortium 2017], [Jeddeloh and Keeth 2012], [Pawlowski 2011]. Theoretically, the HMC can fetch data from the 32 different vaults at the same cycle, reaching a maximum theoretical bandwidth of 320 GB/s [Jeddeloh and Keeth 2012], which is 25% higher than High Bandwidth Memory (HBM) version 2 (256 GB/s) [Kim and Kim 2014]. Different from Double Data Rate (DDR) 3 memories, which transmit 64 bits per channel, the HMC uses

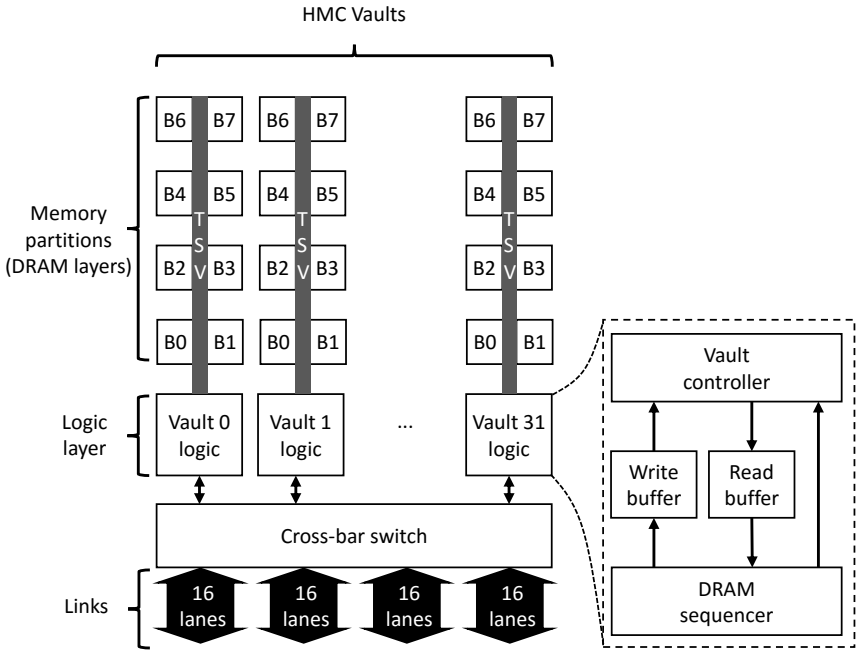

Figure 1. HMC block diagram with 32 vaults each one with 8 banks.

4 serial links formed by 16 full-duplex lanes each. These serial links can reach high frequencies with low interference during data transmissions [HMC Consortium 2017], [Thanh-Hoang et al. 2014]. Internally, there is a crossbar switch that allows these links to transfer data to/from any vault.

HMC supports read and write requests from 16 bytes up to 256 bytes. It also supports arithmetic and binary instructions that operates over 8 or 16 bytes. During the fetch and execution of HMC instructions, the processor treats these instructions the same way as the ordinary memory read or write requests. Thus, the processor fetches, decodes, computes the address and sends the instruction to the HMC. The instructions may also have an immediate operand to be used in the operation. This immediate is sent together to the HMC. When an instruction arrives in the HMC, it is forwarded to the vault responsible for that indicated memory address. The logic layer then interprets each instruction, fetching data and operating over it. Depending on the HMC instruction, the result is updated in the same memory address or sent back to the processor.

HMC can save up to 70% of energy compared to DDR3-1333 memory, presenting a theoretical speedup of 15× [HMC Consortium 2017, Jeddeloh and Keeth 2012, Pawlowski 2011]. However, is not clear if all kind of applications can benefit by current HMC instructions. In this way, studies about Instruction Set Architecture (ISA) extensions are important to evaluate new architectural components.

2.2. Simulator of Non-Uniform Cache Architectures – SiNUCA

During the evaluation of new processor architectures, only simulation represents a viable solution for designers, as the system to be evaluated is too complex to be handled by analytical models, and highly expensive to be prototyped [Jain 1990]. Thus, most computer architects use simulation tools. In contrast to full-system simulation, the trace-driven simulators do not require to actually executing the application instructions during

Table 1. Format of the SiNUCA input traces.

Static Trace

```
#main
@1
MOV 1 0x95727 4 1 14 1 34 14 0 1 0 0 0 0 0
#main
@2
MOV 8 0x95717 3 1 14 1 65 14 0 1 0 0 0 0 0
ADD 1 0x95720 3 2 14 65 1 34 14 0 1 0 1 0 0 0
ADD 1 0x95723 4 1 14 1 34 14 0 1 0 1 0 0 0
CMP 1 0x95727 4 1 14 1 34 14 0 1 0 0 0 0 0
JBE 7 0x95731 2 2 35 34 1 35 0 0 0 0 0 1 0 0
```

Dynamic Trace

```
1
2
2
```

Memory Trace

```
R 4 0x1701448 1
#
R 4 0x1701448 2
R 4 0x1701452 2
W 4 0x1701452 2
R 4 0x1701448 2
W 4 0x1701448 2
R 4 0x1701448 2
#
R 4 0x1701448 2
R 4 0x1701452 2
W 4 0x1701452 2
R 4 0x1701448 2
W 4 0x1701448 2
R 4 0x1701448 2
```

the simulation. In fact, they just need to consider the behavioral details (algorithmic) and microarchitectural latencies for the given traced application. These simulators use execution traces of real applications. These traces are formed by one or multiple files that contains the flow of instructions observed during the program execution. The traces can be generated manually by researchers or automatically by binary instrumentation tools.

In this work, we use SiNUCA, which is a validated cycle-accurate, trace-driven simulator [Alves et al. 2015]. SiNUCA is based on x86 architecture and simulates the execution of mono and multi-threaded applications. It provides an adjustable number of out-of-order processor cores, modeling technologies such as Non-Uniform Cache Architecture (NUCA), Non-Uniform Memory Access (NUMA), Network-on-Chip (NoC) and DDR. SiNUCA also simulates components used in current state-of-the-art architectures, such as data prefetchers, non-blocking cache memories, detailed DRAM memory controller and branch predictors.

SiNUCA splits the simulation traces in three types: static, dynamic and memory, as illustrated in Table 1. The static trace consists of instructions formed by asm code, SiNUCA opcode number, size, read and write registers and other flags. The instructions are grouped in basic blocks, indicated by "@". The dynamic traces contain the sequence calls of basic blocks (from the static trace) performed by the application during normal execution. The memory traces contain the memory address and the instruction size for each memory access performed by the application.

3. The Trace Generator

Emulators and binary instrumentation tools are commonly used to generate trace for trace-driven simulators. Binary instrumentation tools such as Pin, has fast execution and low overhead as advantage. However, these tools depend on complete execution of an application in a real machine for generating an execution trace. Therefore, it becomes very difficult to generate traces of nonexistent instructions in current architectures. Emulators could also be used to generate such traces, but it would require modifications in it to recognize the new ISA as well a new compiler for such architecture. It is clear then, that a new methodology for fast and accurate trace generation is required. In this section, we describe the Intrinsics-HMC library and the trace generation with the SiNUCA-Tracer and Pin tools.

3.1. Proposal Overview

Figure 2 presents the steps required to generate simulation traces referred to HMC ISA. The Intrinsics-HMC library is composed of functions that mimics the behavior of HMC instructions [HMC Consortium 2017]. These functions are called in C/C++ programs and can be successfully compiled to the binary files using the x86 ISA. This binary is later executed with SiNUCA-Tracer (part of our proposal) and the *Pin* instrumentation to generate execution traces. During this trace generation phase, all occurrences of Intrinsics-HMC functions are converted in simulation HMC instructions.

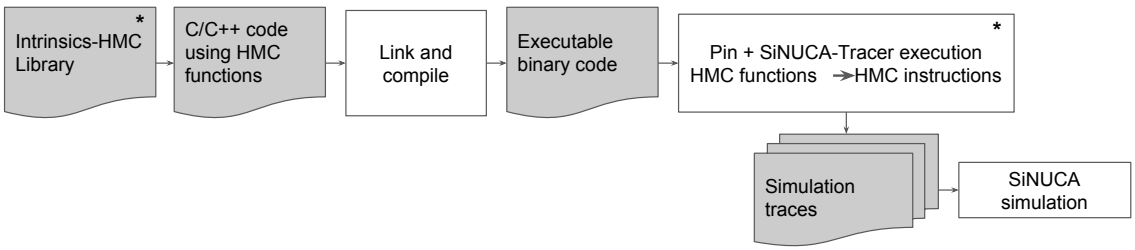

Figure 2. Sequence of steps to generate the simulator input traces (* indicates our main contributions).

3.2. Intrinsics-HMC Library

The logic layer of HMC devices can execute arithmetic instructions and bitwise operations. Therefore, in this section we present functions for creating the Intrinsics-HMC library in order to reproduce the behavior of HMC statements. The types of implemented functions, together with the description of their functionalities are listed below, and in this case, the read and write instructions are not implemented in the library because their traces are generated from the traditional instructions (i.e., load or store requests) normally decoded and executed by the processor being simulated. We split the HMC instructions into four classes of operations, as follows:

Arithmetic: the *sum (add)* and the *increment (inc)* operations. The *add* operation is performed over the operand coming from the DRAM and the immediate, both informed in the HMC instruction. This *add* operation works either with $2\times$ 8-bytes or with $1\times$ 16-bytes operands. The *inc* operation uses only one operand to read a value from the DRAM, increment it by one, and then store it back.

Bitwise: the *bit write (bwr)* and the *swap* operations. The *bwr* operation sends inside the immediate two fields, the bit mask and the write data. For a given address, two operands of 8-bytes each will be loaded and using the bit mask, only specific bits will be updated. In the *swap* operation, the 16-bytes immediate is written to the memory, and the old value is returned to the processor.

Boolean: *and*, *nand*, *or*, *nor* and *xor* operands are used between memory operations and immediate addresses.

Comparison: There are four operations in this class, the *compare and swap if greater than (casgt)*, *compare and swap if less than (caslt)*, *compare and swap if zero (caszero)* and *equal to (eq)*. These operations perform over $1\times$ 8-bytes or $1\times$ 16-bytes operands.

Notice that each operation class supports different operand formats and sizes. There are also variations in the return type sent back to the processor by the HMC. For

further details regarding these operations please refer to the HMC specification version 2.1 [HMC Consortium 2017]. The Intrinsics-HMC library, written in C++ language, covers all the functions described in specification and uses a data type standard to reproduce the HMC operations, described in table 2.

Table 2. Intrinsics-HMC library data types standard.

Data type	Description
__h16l1	Data type equivalent to a short unsigned
__h64l1	Data type equivalent to a long unsigned
__h64l2	Data type equivalent to a vector of 2 log unsigned
__h128ll1	Data type equivalent to a long long unsigned

We define the new data types starting with *'__h'* to denote that this data type refers to the HMC ISA. The number in sequence indicates the data length in bits (16, 64 or 128), together with the letter *'l'* used to refer to *long unsigned*, or *'ll'* to *long long unsigned*. Finally, the last number indicates how much variables that data type allocates (one or two) for each instruction.

```
1  #include "../hmc.hpp"
2
3  int main(int argc, char *argv[]){
4    uint128_t mem_ret;
5    mem_ret =  _hmc128_nor_s
6                (&mem_op1, imm_op2);
7  }
```

Code 1. Intrinsics-HMC function call example.

```
1  __h128ll1 _hmc128_nor_s
2   (__h128ll1 *mem_op, __h128ll1 imm_op){
3      __h128ll1 r = *mem_op;
4      *mem_op = ~(*mem_op | imm_op);
5      return r;
6  }
```

Code 2. Intrinsics-HMC source code for *nor* HMC operation.

Codes 1 and 2 present a *nor* operation between a memory operand and an immediate performed by the function call to *_hmc128_nor_s()*. Notice that, during the function call, the programmer is free to use the usual data types defined by the compiler instead of the ones defined in our library because they are equivalent. However, these new data types are declared to maintain the data type standard of the HMC. Once the program is compiled, the programmer can perform tests and code debug to ensure correctness before the trace generation step.

3.3. Modified SiNUCA-Tracer

With the binary code in hand, the Intel Pin instruments it with the Pin tool called SiNUCA-Tracer. Pin is developed by Intel and integrated with SiNUCA to instrument and analyze code, allowing program developing with routines provided by its own called Pin tools. These Pin tools can be integrated to a program to determine which code parts will be analyzed and inspected and in sequence, what kind of analysis/operation should be done in these parts (memory, execution, cost and performance).

The Pin tools should be applied in binary code when source code does not need more changes. The analysis tools should be developed in C or C++ and analysis occurs at the same time as execution, this is why Pin is known as a just-in-time compiler. In our work, we are using Pin in version 3.2 (revision number 81205).

SiNUCA-Tracer starts opening the program binary image and traverses it. For each routine identified it records into the static trace output file all the instructions present in that routine. The instructions are split into Basic Blocks (BBL). These BBLs contains

all the instructions starting from the instruction after some branch/jump (target instruction) and ends at the first branch/jump found in the execution flow. This way, the static trace may contain the same instruction present in more than one BBL.

During the identification of instructions and BBLs, all the instructions that can perform load or store requests are instrumented in order that, whenever it is executed, the memory address accessed will be written into the memory trace file. Notice that we are dealing with x86 ISA, which may contain a single instruction that perform up to 3 memory accesses (two loads and one store). The head of each BBL identified will receive an identification and will be instrumented in such way that, whenever such BBL is executed, it will write into the dynamic trace file its identifier. This way, a loop over a BBL present in the execution will only store the BBL identification in the dynamic trace to denote each repetition.

After understanding the SiNUCA-Tracer mechanism, we modified it first to identify all the Intrinsics-HMC functions. Each function call must be translated to the correct HMC operation. In this case, the goal is to simulate traces as they have executed by a computer that has an architecture with support to HMC ISA, therefore, the SiNUCA-Tracer suppresses temporarily the generation of x86 traces.

Notice that compiler may have created real dependencies between the registers from outside and inside each Intrinsics-HMC function. After the translation of these functions to HMC instructions, such dependencies must be kept. During the binary instrumentation, we analyze all the input and output dependencies. We call input dependency, all registers that are read inside the function, but the register was written before the function call. The output dependency corresponds to all registers written inside the function that can be read after the function return. Therefore, for each HMC instruction, it contains as read registers a list of all input dependencies, and for the write registers a list of all the output dependencies.

Figure 3 illustrates the analysis performed to keep the right dependencies during each HMC function translation. In this example, we can observe the static trace code for the *add* function (lines 1∼27). The lines 28∼30 contains the translated HMC instruction to perform the *add* operation. This figure brings read (bold with shade), write (bold), base and index (underlined) registers for each instruction. We can see that registers 5 and 6 (line 3) are first read before any previous write, it means that such registers are real dependencies, and thus must be present on the final HMC instruction. The same does not occur for read register 10 (line 10), it was previous written by instruction on line 9, so it does not represent a real dependency to the HMC instruction. This way, when performing the trace generation for HMC, our mechanism analyzes all the input and output registers (inside the circles in the figure) from our function and we use these as read and write registers respectively in the simulable HMC instruction.

In order to replace the functions call by simulable HMC instructions, we added new basic blocks in the static trace, each containing just one HMC instruction according to the function provide by the library. When generating the dynamic trace, we replaced each call to HMC function by the call to one of these basic block containing the respective HMC instruction. Nevertheless, the memory trace will have the specific memory address used by the HMC function. Such address is extracted from routine parameters by Pin tool.

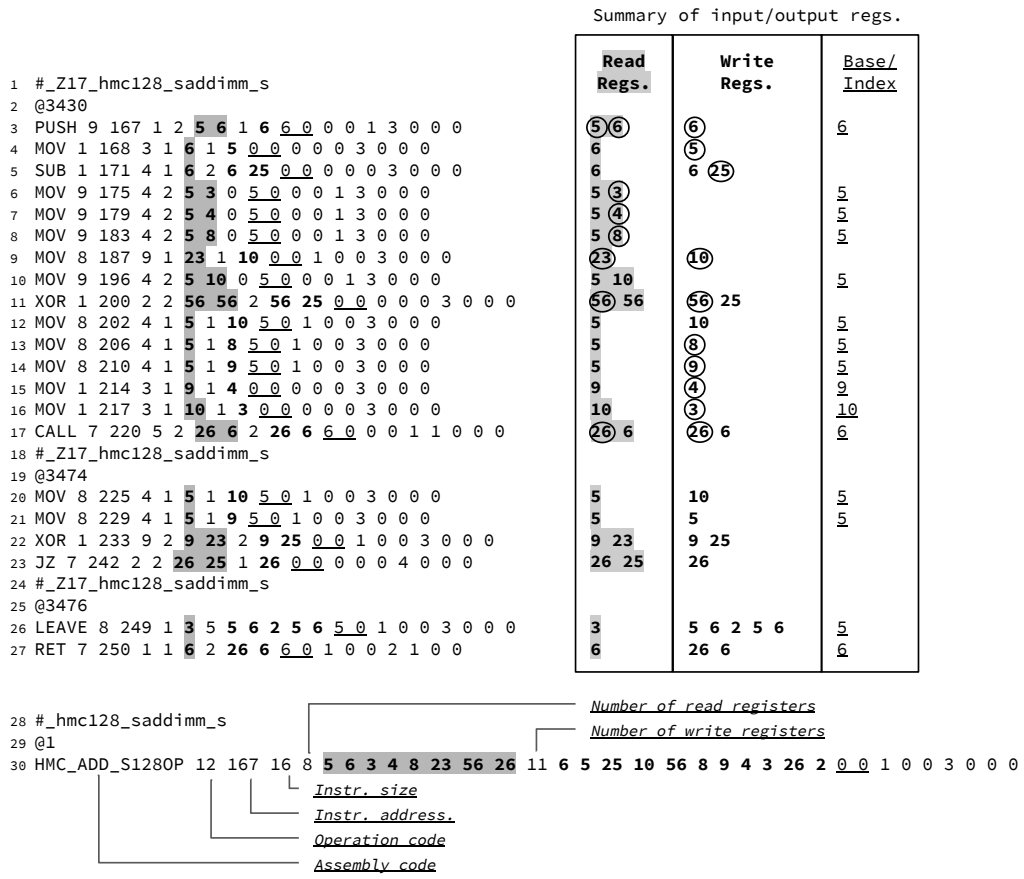

Figure 3. Example illustrating the x86 function translated to HMC instruction, also presenting the read and write register dependencies.

At the end of the execution, the generated traces can be used to feed the SiNUCA in order to simulate the program execution considering an ISA containing HMC instructions.

Currently, only the instructions provided in the HMC specification, are available in the Intrinsics-HMC library. However, such library is easy to extend, thus enabling the creation of new functions that still do not exist. For such extensions, few modifications are required inside the Intrinsics-HMC library and the SiNUCA-Tracer, so that the correct translation of the function is made to a new instruction supported by the simulator.

4. Methodology and Experimental Results

In this section we present the features of the micro benchmark used describing the process to prepare the traces. We also present the results generated by the simulations describing the benefits of using Intrinsics-HMC.

4.1. Benchmark applications

In order to validate our approach, we evaluated the simulation with a micro benchmark composed of a database *join* algorithm and *select scan* from a real query from TPC-H[1]. Those programs were chosen due to their behavior of data streaming which is suitable to exploit in-memory processing coupled with HMC data processing capabilities.

[1] A standard benchmark for decision support in database systems: `http://www.tpc.org/tpch`.

```
#include "../hmc.hpp"

void nljoin(vector<__h64l1> &outer, vector<__h64l1> &inner,
vector<__h64l1> &join_index) {
  for(size_t i=0; i < outer.size(); ++i) {
    for(size_t j=0; j < inner.size(); ++j) {
      if( _hmc64_equalto_s(outer[i], inner[j]) == 1 ) {
        join_index[i] = j;
        break;
      }
    }
  }
}
```

Code 3. Nested Loop Join using Intrinsics-HMC.

The Join algorithms are the kernel of the join operator in Database Management System (DBMS) which combines two relations (tables) by comparing the join attributes and generating a set of tuples (records or rows in a table) that matching these attributes. Commonly, the join attributes are primary and foreign keys. One of the forerunner join algorithm is the nested loop join (NLJoin), depicted in code 3. That algorithm traverse two vectors, each one representing a relation, the outer loop interact in the largest relation and the inner in the smallest one. Inside inner loop is performed a comparison between the join attributes, in which the HMC intrinsic function *_hmc64_equalto_s* is used, case these attributes match a join index[2] is performed.

From TPC-H was selected the query 6 because it scans some columns and their values are accessed just once, i.e., the values, after used, are not touched anymore in the query plan. The query 6 is presented in code 4, it performs a select scan by applying predicates in three columns (WHERE clause) and projecting a sum of two columns just for the tuples that passed in the predicates evaluation.

Code 5 is an implementation of query 6, using Intrinsics-HMC. It encompasses a loop to traverse four columns stored in arrays, the columns involved in the WHERE clause are evaluated by the intrinsic functions: *_hmc64_cmpswapgt_s* (compare and swap if greater than) and *_hmc64_cmpswaplt_s* (compare and swap if less than). Just the tuples that passed by the predicates evaluation are added to the resulting variable.

```
SELECT
  sum(l_price * l_disc) as revenue
FROM
  lineitem
WHERE
  l_date >= date '1994-01-01'
  AND l_date < date '1994-01-01' +
      interval '1' year
  AND l_disc between 0.05 AND 0.07
  AND l_quant < 24;
```

Code 4. Query 6 from TPC-H in SQL code.

```
void query6() {
  for(size_t i=0; i < l_date.size(); ++i) {
    _hmc64_cmpswapgt_s(&l_date[i], 19940101);
    _hmc64_cmpswaplt_s(&l_date[i], 19950101);
    _hmc64_cmpswapgt_s(&l_disct[i], 5);
    _hmc64_cmpswaplt_s(&l_disc[i], 7);
    _hmc64_cmpswaplt_s(&l_quant[i], 24);
    if( l_date[i] != 19940101 &&
      l_date[i] != 19950101 &&
      l_disc[i] != 5 &&
      l_disc[i] != 7 &&
      l_quant[i] != 24)
    {
      res += l_price[i] * l_disc[i];
    }
  }
}
```

Code 5. Query 6 using Intrinsics-HMC in C code.

[2]More explanations about join index can be found in `http://cs-www.cs.yale.edu/homes/dna/papers/vldb.pdf`.

4.2. Generated Traces and Simulation Results

The results of the simulations with the generated trace from Intrinsics-HMC are illustrated in figure 4. The simulations illustrates the execution of the *select scan* and *join* operations from typical DBMS.

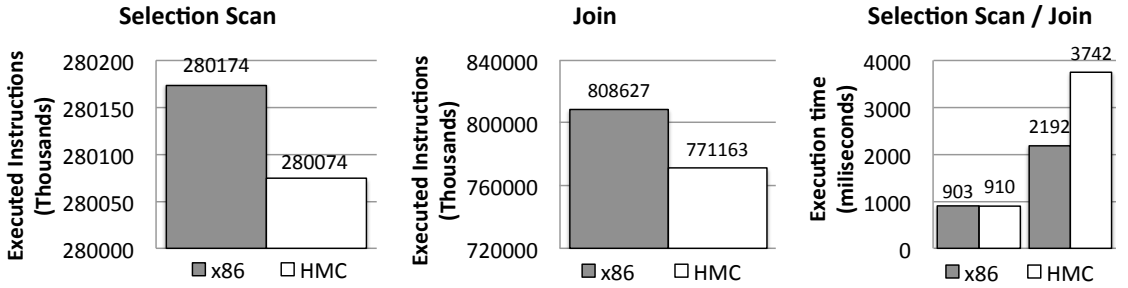

Figure 4. Simulation results for two database operations: select scan and join.

Figure 4 presents the total number of processed instructions and the execution time for each experiment, representing both selection and join operations with HMC instructions compared with x86 instructions only. When the operations are executed interleaving x86 and HMC instructions the total number of instructions reduces since some x86 instructions (from HMC-intrinsics) are replaced by HMC instructions. Comparing the results from the number of executed instructions, it is clear the complete translation of the code through our proposal, but the result also demonstrates an increase in the execution time when using HMC instructions such as the results presented in previous work [Alves et al. 2016], [Hadidi et al. 2017].

5. Related Work

The arrival of the HMC motivated several work over the past decade[Alves et al. 2016], [Hadidi et al. 2017], [Oliveira et al. 2017]. Together with HMC, the High Bandwidth memory (HBM) emerged with a 2.5D stacked memory architecture. In the case of HBM, the vendor can supply a logical die with a memory controller and a set of specific instructions. Thus, we could easily apply our technique to HBM. However, in this paper we choose HMC because it uses a simple and well documented ISA. Moreover, HMC offers a higher bandwidth of 320 GB/s compared to HBM, which achieves only 128 GB/s in the first release, and 256 GB/s on the second version [Kim and Kim 2014].

Most of Processor-in-Memory (PIM) researchers proposed to improve the architecture efficiency, using simulators to perform its analysis. On [Alves et al. 2016], the HMC receives a mechanism called HMC Instruction Vector Extensions (HIVE) to execute vectorized instructions in the logic layer of the HMC. However, the researchers had to generate simulation traces manually to evaluate the mechanism (HIVE). Due to lack of emulators and compilers capable of generating vector codes running in memory, this work refrained from using only simple workloads. On a different direction, the work proposed in [Hadidi et al. 2017] uses a FPGA that generates and sends customized requests to a coupled physical HMC. Although they make use of a physical hardware, any evaluation of new architectural components or different ISA should demand modifications to the compiler also requiring hardware prototypes, which is a costly task.

The work proposed in [Oliveira et al. 2017] presented the Precise Cycle Parallel PIM Simulator (CLAPPS) that allows the modeling of custom PIM architectures for simulation. Compared to gem5 simulator with SMC Simulation Environment (SMC-Sim) [Azarkhish et al. 2016], CLAPPS was developed to provide a more precise model for PIM architectures. In this way, the authors have created an architecture similar to the HMC, based on the specification 2.0 [HMC Consortium 2017]. However, on CLAPPS there is still a necessity of an efficient way to generate input workloads that uses the HMC instructions. Moreover, CLAPPS only simulates HMC memory and it depends on integration to some processor simulator in order to obtain realistic results. The CasHMC [Jeon and Chung 2017] is a cycle-accurate simulator for HMC. It was developed in C++ and covers most specific architecture details of HMC specification. Different from SiNUCA, CasHMC models the HMC as a simple memory, without any PIM capability. Our proposal in this paper improves the SiNUCA features by allowing the automatic generation of traces with HMC instructions from binary code, while also allow some customization in the operation size. Furthermore, our methodology is adaptable to other simulators, and it can be extended to support a different ISA.

6. Conclusions and Future Work

In this paper, we present a methodology for aiding the simulation of emergent HMC architectures and new instructions. Our Intrinsics-HMC library allows writing full codes in high-level languages by utilizing functions that emulate new instructions for in-memory HMC processing. This saves time and reduces errors when writing assembly / simulated code language, which is a major benefit for PIM architecture designers and researchers.

Our proposal also allows the generation of simulated traits through the SiNUCA-Tracer that translates the HMC behavioral functions to HMC simulated instructions. We focused our proposal on the SiNUCA simulator as use case, but other simulators can also benefit from the presented outcomes. Besides, the trace generator SiNUCA-Tracer can also be extended allowing to simulate new PIM architectures. Our Intrinsics-HMC library is freely available and can be accessed in the repository `https://github.com/AlineS/intrinsics-hmc`.

As future work, we consider to modify the SPEC-CPU 2006 [Henning 2006] benchmark suite in order to use the Intrinsics-HMC library. Moreover, we also consider to re-validate previous work that performed the evaluation of new PIM architectures.

References

Alves, M. A. Z., Diener, M., Moreira, F. B., et al. (2015). SiNUCA: a validated micro-architecture simulator. In *High Performance Computation Conf.*

Alves, M. A. Z., Diener, M., Santos, P. C., and Carro, L. (2016). Large vector extensions inside the HMC. In *Conf. on Design, Automation & Test in Europe*.

Azarkhish, E., Rossi, D., Loi, I., and Benini, L. (2016). A case for near memory computation inside the smart memory cube. In *Workshop on Emerging Memory Solutions*.

Balasubramonian, R., Chang, J., Manning, T., et al. (2014). Near-data processing: insights from a MICRO-46 workshop. *IEEE Micro*, 34(4).

Elliott, D. G., Stumm, M., Snelgrove, W. M., et al. (1999). Computational RAM: Implementing Processors in Memory. *Design and Test of Computers*, 16(1).

Hadidi, R., Asgari, B., Mudassar, B. A., Mukhopadhyay, S., Yalamanchili, S., and Kim, H. (2017). Demystifying the characteristics of 3d-stacked memories: a case study for hybrid memory cube. *arXiv preprint arXiv:1706.02725*.

Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4).

HMC Consortium (2017). *HMC specification 2.1*.

Jain, R. (1990). *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons.

Jeddeloh, J. and Keeth, B. (2012). Hybrid memory cube new DRAM architecture increases density and performance. In *Symp. on VLSI Technology*.

Jeon, D.-I. and Chung, K.-S. (2017). Cashmc: A cycle-accurate simulator for hybrid memory cube. *IEEE Computer Architecture Letters*, 16(1).

Khalifa, K., Fawzy, H., El-Ashry, S., and Salah, K. (2013). Memory controller architectures: A comparative study. In *Int. Design and Test Symp.*

Kim, J. and Kim, Y. (2014). Hbm: Memory solution for bandwidth-hungry processors. In *Hot Chips Symposium*.

Oliveira, G. F., Santos, P. C., Alves, M. A. Z., and Carro, L. (2017). A generic processing in memory cycle accurate simulator under hybrid memory cube architecture. In *Int. Conf. on Embedded Computer Systems: Architectures, MOdeling and Simulation*.

Olmen, J. V., Mercha, A., Katti, G., et al. (2008). 3D stacked IC demonstration using a through silicon via first approach. In *Int. Electron Devices Meeting*.

Patterson, D., Anderson, T., Cardwell, N., et al. (1997). A case for intelligent RAM. *IEEE Micro*, 17(2).

Pawlowski, J. (2011). Hybrid memory cube (hmc). *Hot Chips*, 23.

Pugsley, S., Jestes, J., Balasubramonian, R., et al. (2014). Comparing Implementations of Near-Data Computing with In-Memory MapReduce Workloads. *IEEE Micro*, 34(4).

Thanh-Hoang, T., Shambayati, A., Deutschbein, C., Hoffmann, H., and Chien, A. (2014). Performance and energy limits of a processor-integrated fft accelerator. In *High Performance Extreme Computing Conf.*