In-Memory Processing: A look into "A Logic-In-Memory Computer", by Harold S. Stone, IEEE 1970

Peppas Athanasios, Iliadis Thrasyvoulos-Fivos

***Abstract*—In this work we discuss the use that Processing-in-Memory designs can have in modern systems. Starting from the very early steps towards PIM, we take a critical look to "A Logic-In-Memory Computer" and try to evaluate the foundations that were laid on processing in-memory architectures. We pinpoint the persistent ideas from the 70s up to today in the work as well as touch on some inaccuracies and overlooked problems. After that, we show ways that accelerate and ease the embedding of PIM units in contemporary systems, such as 3D-stacked Memories and DRAM usages.**

I. Introduction

A. Current Computing Trends and Problems

Both on the personal and on the industrial scale, computers systems virally adopt the processor-centric design. The main model is separating the computation capability from the memory/storage capability of any given system, connecting the two components through long, costly and as empirically witnessed bottleneck-prone networks. This is because all of the computing done on the system happens only on the CPU and nowhere else, hence the need to move the data for any calculation (no matter how small or insignificant) all the way to the processor. Not only is this a field for potential performance loss due to latencies and errors in the data stream, but empiricall evidence on the separation of processing and memory capabilities have been said to use *"more tha 62% of the energy consumed by four major commonly-used mobile consumer workloads "*(including software like Chrome browser among others)[1].

The major problem of this design is therefore the performance loss and energy waste attributed to data movement. In general it can be summed that five major factors are behind this phenomenon][1]. Briefly:

- Low bandwidth/High latency due to the narrowness of the path between the memory controller and the main memory.
- Complex extra mechanisms to tolerate the data access from main memory, which when not needed/utilized properly by the system cause an unecessary overhead in energy demand. A great example of the vicious cycle of trying to balance the extra data movement is the following:
  1) Data movement is causing (already) a lot of latency and energy waste.
  2) We employ complex mechanisms to deal with this.
  3) The mechanisms produce additional complexity and latency.
- Inefficient use of the many employed caches.
- Modern applications produce random memory access patterns, therefore incapacitating usual memory-efficiency strategies.
- The interconnections between memory and CPU themselves impose latency problems.

B. Processing-in-Memory

From the time of the article's writing to today Processing-In-Memory (or In-Memory Processing, PIM, Near-Data-Processing, NDP) is a developing field. Generally in computer science, Processing-In-Memory is a group of technologies for the processing of data stored in an in-memory database. Modern PIM architectures rely on the use of RAM because of its faster data accessing times and the emerging demands in Business Intelligence, but older systems implemented such systems based on disk storage and other slower hardware.

The problem that PIM technologies are trying to address is reducing the movement of data for computation. Modern architectures are designed with

little consideration on data movement optimization. The CPU is treated as the only place that computations can take place, and data can be processed only after they are moved into proper registers (similarly for accelerators, e.g. GPUs), therefore stressing the bandwidth between CPU and memory. Apart from the bandwidth power consumption levels are also increasing. These drawbacks are particularly apparent in mobile devices and server setups, with a recent work showing that *"more than 62% of the entire system energy of a mobile device is spent on data movement between the processor and the memory hierarchy for widely-used mobile workloads"*[2], [1].

The result of this is data access being a key bottleneck, especially with emergin data-heavy applications ,as well as energy consumption becoming a limiter. With data movement being very costly in terms of bandwidth, these design trends are forcing the field towards other more aware in terms of data movement architectures, one of which is the In-Memory-Processing model [1].

In Section II we show the main points of "A Logic-In-Memory Computer" and give an overview of the key ideas that emerge from it. Section III is where we evaluate the strengths and weaknesses of the original work, while in Section IV we discuss problems that were overlooked and give our recomendations on work to be done.

II. Paper Overview

The article we will be discussing in this section is one written in 1970 by Harold S. Stone, titled "A Logic-In-Memory Computer" and published in the IEEE IEEE Transactions on Computers Journal of that year. It defends the point that according to observations made at that time[3], the cost of microelectronic components is likely to drop significantly therefore enabling system architects to embed microelectronic memories (mainly, but also other microelectronic components) to computers efficiently. The paper then introduces a case study of the IBM 360/85 [4] and it's use of a microelectronic memory as a cache, subsequently showing that it is both feasible and lucrative to use microelectronic components even at the time of writing. The author then starts suggesting a different kind of microelectronic memory, one infused with some computing capabilities, and provides a few example functions of said memory. Apart from that he also shows ways of utilizing the new functionalities through giving explicit control of some aspects of the cache to the programmer.

The paper is widely assesed to be one of the earlier and most pioneering works on the Logic in Memory design model. It spawned a variety of research by later works and aspects of it are still being used and cited to this day.

A. Cache Memory Benefits

As described, the IBM 360/85 was a computer system that used a high-speed buffer memory as a cache in between the CPU and the main memory. The purpose of this architecture is to make the main memory seem as if it had the performance of the microelectronic cache using clever memory management algorithms and utilizing the superior speed and performance of the middle hardware.

The empirical knowledge concluded at the time (correctly to this day) that memory "memory accesses tend to be highly correlated". Using that we can design memory management algorithms so that the most active areas of memory tend to reside in the cache, making future accesses on the same data a lot faster.

This is what the IBM 360/85 tries to implement. The particular cache that can hold 16 secors of memory each of 1024 bytes has an operating time at around 80ns per access. Comparing that to the massive 750 - 8000ns that the main memory operates at we can hint at the advantages of the design. We benefit from the performance when the data needed resides on the cache and if that is not the case we bring the data from the main memory. The task of trying to maximize the amount of accesses that hit on the cache is the job of the memory management algorithms.

In this case the algorithm dictates that on the event of a sector needing to come to cache but there being insufficient space, the least recently used sector will be replaced. Although this is as we will see in a moment rare, the time needed to bring a sector to cache is roughly equivalent to 10 main memory cycles.

The paper then claims that through simulation - as there had not yet been actual results from a real IBM 360/85 reported - it can be estimated that "95% of the CPU memory requests were directed to data

sectors that were already resident in the cache" and the total performance of the system was equivalent to 80% of the speed of an identical machine that used a microelectronic main memory. The logical conclusion of this sector is therefore that with a small microelectronic buffer used in such way we can benefit from performance levels suit for much more expensive systems.[5].

B. Logic-In-Memory Array as Cache

So the above section shows that with the use of fairly simple and small microelectronic we can make the whole main memory perform at levels similar to the cache. The idea behind this next chapter is that of enhancing such small, cheap microelectronic hardware units not only with memory managing capabilities but also with accordingly simple processing capabilities. If we manage this, then the cache will make the main memory appear not only as if it has the memory management capabilities of the cache but also it's processing power. This is of course made feasible because of the earlier observation that most data needed by programs already reside in the cache, so hypothetical instructions running on the cache will almost always find the needed data instantly.

With this in mind, the author suggests some types of instructions that he thinks could be useful to programmers and to the system's performance alike. Keep in mind that these are not implementation guides or hard plans for instructions, and are more of route suggestions based solely on the empirical knowledge of the author. These include:

- Search on Masked Equality: An instruction with 3 operands, a pattern, a mask and an address. The instruction is loosely described as searching one memory sector and matching the pattern with the first word that matches it.
- Search on Masked Threshold: Similar to the above instruction but can return not only patterns that are equal to the first operand but also greater. Other search commands can be likewise formulated.
- Copy Tag Bit, Tag Bit AND: With according operands for bit positions, these types of instructions can allow for in-memory handling of word bits.
- Sector ADD: Taking two sectors as arguments, the corresponding words of the two sectors are added together. Similarly we can have multiplication and sector copy.
- Sector Scale: A sector and an operand, we can for example multiply the sector with the given operand.

Two things noted on behalf of these proposed instructions by the article are about their utility and their cost of implementation. As for the utility as already noted the author also points out the instructions are meant to be illustrative of the uses a PIM cache can have.

The cost of implementation of most of these functions is no more cumbersome than adding a few registers on the hardware, with whatever that means for financial cost and hardware complexity. The sector commands can be a little trickier. The paper provides a clever implementation using only one bank of adders and a bus system used for each word of the sector, thus making this implementation more feasible. In any case if the cost to utility ratio of such instructions is proved to be non-profitable designers can simply choose to avoid such insctructions altogether. Fig. 2 shows the above mentioned mechanism.

The main difficulty of the design stage around these processing capabilities is according to the author the identification of the performance improvement in contrast to the implementation cost of any proposed instruction, those aforementioned included.

C. Cache Control

The final section of the paper examines the idea of giving explicit control of the cache to the programmer. In most systems from the time of writing since today, the cache is invisible to the programmer and only handled by the operating system. This reduces unecessary complexity of the code and most of the time leads to increased performance, as the OS is more likely to handle the cache memory efficiently. But replace the standard CPU centric computer system with a processing-in-memory one and Harold S Stone says you may have notable advantages from giving cache control to the programmer.

The types of instructions the user could be able to give to the cache ultimately fall into these 3 categories:

- "Hold" a Sector in cache.
- "Release" a Sector from the cache.

- Load a memory area or a group of data in a particular way that benefits my program.

We can easily understand the use of the 2 first scenarios,e.g. where a programmer may know better about the use of particular data fields and may wish to always keep them in cache. For the last point however, more explicit examples can be given, and the paper provides a use case.

The way matrices are loaded in memory are often designed so that either rows or columns can be fetched to the cache. This is as we can imagine beneficial to the program time. But by giving control to the programmer, we could design a technique so as to make both rows and columns of the array interlaced in the memory modules of memory and have a row's data on say S, S+1, S+2 addresses while a columns data on the S, S+K, S+2K etc, where K is an appropriate number dependent on the number of memory modules used.

Lastly the paper references other techniques that could gain from such a design, mostly on the field of parallel processing, and the author states once again that the mechanisms provided in this chapter are illustrative and in no way the only things that could be done by controlled cache buffers.

III. Evaluation

Critical evaluation of the ideas in the article

IV. Further Problems

Problems with bandwidth described in literature

References

[1] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "A modern primer on processing in memory," 2021.

[2] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu, "Lazypim: An efficient cache coherence mechanism for processing-in-memory," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 46–50, 2017.

[3] P. Christie and D. Stroobandt, "The interpretation and application of rent's rule," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 6, pp. 639–648, 2000.

[4] 2021.

[5] H. S. Stone, "A logic-in-memory computer," *IEEE Transactions on Computers*, vol. C-19, no. 1, pp. 73–78, 1970.