Saving Memory Movements Through Vector Processing in the DRAM

Marco A. Z. Alves, Paulo C. Santos, Francis B. Moreira, Matthias Diener, Luigi Carro
Informatics Institute – Federal University of Rio Grande do Sul – Porto Alegre, Brazil
{mazalves, pcssjunior, fbmoreira, mdiener, carro}@inf.ufrgs.br

ABSTRACT

Despite the ability of modern processors to execute a variety of algorithms efficiently through instructions based on registers with ever-increasing widths, some applications present poor performance due to the limited interconnection bandwidth between main memory and processing units. Near-data processing has started to gain acceptance as an accelerator device due to the technology constraints and high costs associated with data transfer. However, previous approaches to near-data computing do not provide general-purpose processing, or require large amounts of logic and do not fully use the potential of the DRAM devices. These issues limited its wide adoption. In this paper, we present the Memory Vector Extensions (MVX), which implement vector instructions directly inside the DRAM devices, therefore avoiding data movement between memory and processing units, while requiring a lower amount of logic than previous approaches. MVX is able to obtain up to 211× increase in performance for application kernels with a high spatial locality and a low temporal locality. Comparing to an embedded processor with 8 cores and 2 memory channels that supports AVX-512 instructions, MVX performs 24× faster on average for three well known algorithms.

Categories and Subject Descriptors

B.3.2 [**Hardware**]: Memory Structures—*Design Styles*

Keywords

Near-data computing, data movement, vector instructions

1. INTRODUCTION

For decades, the slow advancements of main memory technology and manufacturing processes have been overshadowed by Moore's law on processor manufacturing technology. As smaller transistors paved the way for ever-faster processing units, the same could not be done for memory devices, which have different trade-offs and design points [24, 22, 25]. This led to a performance gap between processing units and memory devices due to the high latency to fetch data from memory devices and the interconnection bottleneck between them.

Current processors require access to large amounts of data due to the increasing number of cores and the trend towards vector instructions (NEON, SSE, and AVX, among others). To deliver data and enable efficient use of processor cores, the industry started to provide multiple data channels and memory controllers, introducing parallelism between memory modules and therefore higher bandwidths. For embedded systems, the increase on the number of memory channels must be carefully considered since the energy consumed by multiple data buses and memory controllers can conflict with energy constraints.

A different approach to solve this issue is to move computation closer to the data, reducing the latency and energy to transfer data between memory and processor. This approach has a higher potential for applications with a low temporal locality. Algorithms that present low data reuse reduce the effectiveness of the cache memory, as they stream large data sets and evict data from caches with no reuse. For algorithms that also present high spatial locality, near-data computation has even higher potential by exploring the inherent spatial locality of the DRAM devices' row buffers.

Some of the previous work on near-data computing only proposed application-specific logic embedded inside or near the memory [28, 23, 27]. Others proposed to include full multi-core processors inside the memory module [19, 20], requiring a large amount of logic. Furthermore, only a few of the previous proposals use the full potential bandwidth from the memory arrays [9, 27].

In this paper, we present *Memory Vector Extensions (MVX)*, a new mechanism that implements a set of vector instructions inside the DRAM to perform near-data computing. The MVX implements a set of Functional Units (FUs) inside the DRAM devices and uses data provided directly by the rows of each memory bank. Through this approach, our mechanism is able to access large amounts of data simultaneously, and execute vector instructions on this data with a large parallelism to provide a better use of the data width available in memory devices.

Using MVX enables the processor to delegate operations to the memory over large data sets with high performance. MVX is designed to outperform normal processor vector instructions for algorithms that present high spatial locality

We acknowledge the support of CNPq and CAPES.

.

and low temporal locality, such as stream applications. This is because such applications cannot benefit from cache memories inside the processors to hide memory access latency. Our mechanism requires small modifications to the processor and the memory sub-system, but it does not degrade the normal processor operation. MVX has the following main features:

High performance: We identify and evaluate the gains that can be achieved when bypassing the interconnection bottleneck and performing vector operations directly inside the DRAM. By accessing the full row buffer width, our mechanism can achieve substantial improvements of streaming application kernels' performance.

Low logic overhead: Our mechanism requires vector units and a register bank inside the DRAM devices, additions which represent less hardware than previous proposals. We describe the required system modifications to implement our vector instructions extension inside the DRAM devices.

General-purpose design: MVX seamlessly integrates into current general-purpose systems, requiring few binary modifications, which can be easily generated by the compiler. With a small vector ISA extension provided by the processor, our mechanism enables execution of special instructions decoded by the processor inside the memory devices. For application kernels that do not use our mechanism, the system behavior is unchanged.

Flexible implementation: Our mechanism can be implemented in different ways depending on hardware constraints, such as latency, area and energy consumption. For example, by varying the number of FUs that operate in parallel, a trade off between area and latency can be achieved.

In our evaluation, we show how our mechanism behaves for three application kernels with different behaviors. We compare our mechanism to the use of multi-core vector instructions, extrapolating the number of cores and memory channels. Variations of the mechanism's parameters are presented in order to evaluate compiler improvements and also the different operation latency trade-offs.

MVX is up to 211× faster than the normal embedded single-core processor that supports Streaming SIMD Extensions (SSE) instructions with a single memory channel for application kernels with high spatial locality and low temporal locality. For the worst-case evaluated, which contains a high amount of data reuse, our mechanism is up to 20× faster. Comparing our single-threaded approach to an 8-core embedded architecture that supports Advanced Vector Extensions (AVX) instructions with 2 memory channels, our mechanism is 60× and 3× faster for the best and worst-case application kernels evaluated, respectively. Compared to previous work that performs near-data computing, MVX is on average 10× faster with a comparable overhead.

This paper is organized as follows: The next Section discusses previous work on near-data computation and compares it to our proposal. Section 3 details current DRAM devices and analyzes their technological constraints. Section 4 presents and discusses our proposed mechanism. Section 5 shows performance results for our mechanism compared to an embedded multi-core processor, as well as presenting baseline extrapolations to fully evaluate our mechanism. We also compare performance improvements and overhead to a previously proposed technique. Section 6 summarizes our conclusions and mentions ideas for future work.

2. RELATED WORK

Several studies have addressed near-data computation, generally aiming to reduce the costs related to data transfer between the processing units and DRAM. Since off-chip data movement is a major bottleneck for computer systems [25], the main goals are usually increasing performance and reducing energy consumption. Table 1 presents an overview of the characteristics of the related work and our proposal (MVX) presented in this section. MVX provides general purpose processing capabilities while requiring a reasonable amount of embedded logic to operate on a high data bandwidth.

Table 1: Summary of related work characteristics.

Mechanism name	**Small logic**	**High bandwidth**	**General purpose**	**No source code changes**
IRAM [19]			●	●
C-RAM [9]		●	●	●
NMP [23]			●	
LiM [27]	●	●		
DRAMA [10]	●		●	
NDCores [20]			●	
MVX	●	●	●	●

The Intelligent RAM (IRAM) [19] approach aims to increase the accessible data width by implementing more memory ports and data buses. To efficiently use this large bandwidth, the authors propose to implement a vector processor inside the DRAM module, where this processor is able to access the vector operands directly from RAM through the extra ports. IRAM extrapolates the system with up to 16 ports of 1024 bits, claiming that it is possible to accelerate the execution and simultaneously reduce energy consumption. However, [6] shows that not only does it require a large amount of logic, the approach taken in IRAM could quickly become obsolete for faster processors since it depends on extra buses and memory ports.

Elliot et al. [9] present the Computational-RAM (C-RAM), a proposal that resembles ours, with some important differences. In their proposal, multiple functional units are inserted together with the sense amplifiers, computing at the bit level. Considering that a DRAM device activates only a few memory sub-arrays per request, to control and power on all data rows and processing elements instantaneously represents a substantial constraint. Moreover, their technique requires a large number of functional units, coupled to every sense amplifier in the memory sub-arrays [15] (in our proposal we have a set of vector units per device, shared among all the row buffers). Finally, the C-RAM mechanism requires a very specific memory mapping of the application's data by the OS.

In [23], the authors present the Near Memory Processor (NMP), implementing an in-order 2-issue wide co-processor between processor/cache and main memory. Although this approach is limited by memory controller bandwidth, NMP presented has its own local large width scratchpad memory which enables data accesses with a high bandwidth. Despite the higher performance, the data width managed between main memory and NMP is limited to the original memory bus, maintaining compatibility with current architectures. The performance is also limited by the fact that the proposed architecture must fill the scratchpad memory before processing, an operation which depends on the memory and

Table 2: Comparison of previous near-data computation proposals.

Mechanism name	**Type of processing elements**	**Number of elements**	**Internal memory**	**Logic frequency**	**Data width**	**Data frequency**	**Integration technology**
IRAM [19]	In-Order + Vector proc.	1 per module	Reg. + Caches	DRAM	8x Bus	DRAM	DRAM module
C-RAM [9]	Functional Units	1 per sense amp.	3 Registers	DRAM	Arrays × Row buffers	DRAM	DRAM device
NMP [23]	In-Order BMT	1 per mem. ctrl.	Reg. + Scratchpad	Core	Channel	Bus	Processor die
LiM [27]	ASIC	1 per device	I/O FIFO	Bus	Bank column	DRAM	DRAM device
DRAMA [10]	CGRA	1 per device	I/O FIFO	Bus	Bank column	DRAM	DRAM device
NDCores [20]	In-Order	4 per device	Reg. + Caches	Core	Device	Bus	DRAM module
MVX	Vector FUs	1 set per device	Register bank	DRAM	Row buffer	DRAM	DRAM device

interconnection performance, which are well-known bottlenecks. Moreover, the programmer must carefully control the content of the scratchpad memory. The architecture presented in [8] also integrates processing elements between the processor/cache and the DRAM memory. Such mechanisms outside the processor also require specific address translation hardware to perform operations over the correct data inside the DRAM.

Taking advantage of die-stack technology, Zhu et al. [27] present a 3D-DRAM customized logic layer capable of accelerating application-specific data intensive computation. FFT and SpGEMM Application Specific Integrated Circuits (ASICs) were implemented as a 3D-DRAM layer with communication using Through-Silicon Via (TSV). The authors claim bandwidths of up to 668.4 GB/s when 16 banks and 1024 TSVs per bank are used to connect the proposed ASIC to the row buffers. The proposed mechanism does not support general purpose computations, which is the focus of our work.

In [10], the authors present a solution where Coarse-Grain Reconfigurable Arrays (CGRAs) are implemented on top of the memory devices using TSV technology. This approach accesses the data from global I/O (that is, the bus after the Double Data Rate (DDR) I/O gating) using TSV, being able to execute instructions over these data. The benefits emerge from the efficient data transmission system from memory devices to the proposed buffers and CGRAs. However, this approach does not use the full data bandwidth inside the DRAM devices.

In the work presented in [20], processor cores (NDCores) are inserted in the DRAM memory modules. In this approach, the authors inserted one low-power quad-core processor for each DRAM device in a memory module, with the logic outside the DDR device. Similar to the work presented in [10], each processor can access the data width presented at columns of each DRAM device, which represents less than 1% of the row buffer size. This small bandwidth limits the amount of data that can processed in parallel. This work provides a comparison with multi-core processors, presenting significant performance gains and energy savings.

Some proposals [5, 21] consider using the Hybrid Memory Cubes (HMCs) due to their great advantages in performance when compared to regular DRAM modules. In [21], the NDCores mechanism is extended to embedding processor cores inside the HMC, enabling low latency data access. The authors assume that each NDCore operates exclusively on their respective private memory space (256 MB in this case). The NDCores achieve performance mostly because they are not limited by the HMC to processor link bandwidth, and being able to nearly saturate the bandwidth inside the HMC device for the shown applications. These proposals require specific features from HMC while our technique can work with commodity DRAM devices.

Table 2 shows a comparison between the main near-data computation proposals. The first four columns present the processing element details considered in each proposed implementation. It can be seen that most approaches require full multi-core processors close to the DRAM devices [20, 23], showing that most previous work relies on a large amount of logic. Moreover, most of previous proposals use the same data bandwidth present outside the memory devices or memory modules (that is, smaller than the row buffer width). The fifth and sixth column present the data bandwidth related characteristics of each proposal, showing that only few previous proposals consider to access the full row buffer [9, 27]. The seventh column shows how close to the memory the proposals are implemented. In general, as the proposed processing elements are more complex, the farther from the memory device they are placed. This implies a bandwidth reduction and more data movement.

3. TECHNOLOGICAL CONSTRAINTS OF MEMORY DESIGNS

The main memory of current systems is a known performance bottleneck. Furthermore, it is a large contributor to a system's power consumption, especially for embedded systems, which have stringent restrictions. In this Section, we present the architecture of the DRAM devices and explore the sources of inefficiency of the main memory, motivating our approach to integrate functional units directly inside the DRAM devices. The DRAM system is presented at a level of abstraction that is sufficient to understand the terminology and key concepts of this paper. For a detailed description, we refer the reader to [7, 14].

3.1 DRAM Implementation Details

Traditional main memory modules are formed by multiple devices that act in a coordinated way [14]. The highest level memory structure is the module, which consists of a set of devices. A module may have multiple ranks, each rank consisting of multiple devices, which will operate in synchrony. The devices are composed of a set of banks, and all the devices in a given rank react to an operation signal, always operating in the same bank for a given signal. These banks are composed of sub-arrays, formed by rows that are accessed per column. The DRAM protocol manages these arrays using these 5 basic, simplified operations: *precharge* (prepares the arrays and sense amplifiers to read a new row), *row access strobe* (reads a specific row using the sense amplifiers into a SRAM row buffer, with 1 buffer for each bank), *column access strobe* (bursts data of a specific column of the

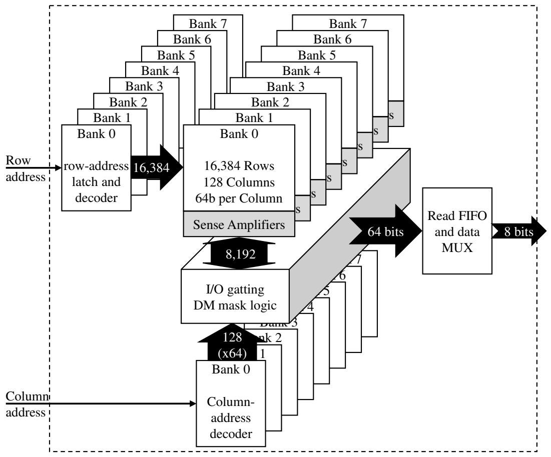


Figure 1: DDR 3 x8 functional block diagram of a single device [17].

row buffer from the DRAM devices to the bus), *column write* (receives data from the bus and overwrites the addressed column of an open row) and *refresh* (refresh capacitor charges a row, usually done automatically by each device).

Since processors have been increasing their throughput demand, the DRAM evolved to meet the requirements of modern processors. DDR memories have emerged as a major technological breakthrough, providing the ability of transmitting data at both clock edges. Its evolutions as DDR 2, DDR 3, DDR 4 and so forth generally increased the I/O frequency by increasing the data burst capability and bus operating frequency. However, the organization of a DDR DRAM device in all versions suffered few architectural modifications. Figure 1 shows a schematic of the basic DDR 3 x8 device. Despite these advancements in memory technology, the operating frequency of the basic devices to a certain data width is limited, providing a lower throughput than what is required by modern processors. Thus, besides the burst technique, sets of devices are deployed in a module to increase parallelism and increase data throughput.

A widely used technique for high performance systems is the integration of multiple channels and multiple memory controllers. This technique allows memory modules to be used in parallel, although on a limited bus width per module. However, even for systems with a high memory bandwidth, if we consider streaming applications that present a high spatial locality and low temporal locality, the cache hierarchy will represent a waste of resources in terms of performance and energy consumption. This happens because the processor will not use it, as the data will be brought into the cache and removed as soon as possible to make room for new data.

3.2 Evaluating Memory Constraints

To evaluate the sources of inefficiency in a system, we present in Figure 2 the number of execution cycles of three application kernels: *vector sum*, *5-point stencil computation* and *matrix multiplication*. These applications represent three different scenarios with a zero, medium and high amount of data reuse, respectively. We evaluated the behavior of these three applications with different memory parameters and a different number of threads. Each bar shows the number of execution cycles from 1 to 16 threads/cores using an Atom-inspired processor as baseline, with 1 channel and 1 memory module attached to it. The first two bars evaluate the impact of vector instructions set SSE 4.2 and AVX-512. The third bar shows the impact of larger L2 cache memories, which are shared between 2 cores. The fourth bar shows how performance scales given a zero latency for DRAM operations (representing ideal DRAM devices and module latencies), but keeping the bus width. The last bar shows how performance scales given a higher bandwidth (representing close-to-ideal bandwidth), by using 64 full-width parallel memory channels.

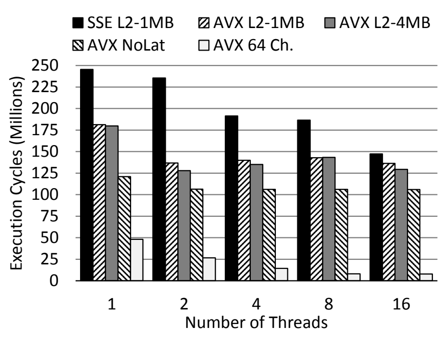
Observing the experimental results, we can see that bandwidth is the real constraint to performance. However, scaling the number of channels (in order to increase the bandwidth) demands resources and increases power consumption. Therefore it is not a scalable solution nor a valid alternative for embedded systems. The intuitive solution for this is to place the processing logic closer to data, alleviating or even avoiding the costs related to data movement, as it has been reasoned that there are varied advantages that can be achieved [6, 27, 28]. There are two logical approaches: put more memory closer to the processors through usage of embedded DRAM [1, 16], or enable processing in the DRAM modules [27].

In the detail of Figure 1 it is possible to observe that a large data width is always available when a row is read inside the DRAM devices, although at a low operating frequency. These rows of data are the focus of this work. Even if the transfer rate is reduced due to the low operating frequency of the DRAM devices (that is, 200 MHz for device and 1600 MHz for I/O), if we were to compute inside the DRAM devices using row buffers (that is, operation in 1 KB width data vectors), we could theoretically achieve a throughput of up to 204.8 GB/s, considering the parallelism between 8 banks. This means that accessing all DRAM devices in this way, assuming 8 devices at 200 MHz, it is possible perform at rates of 1.23TB/s calculation throughput (considering the sum of all 1 KB row buffers from 8 devices). This peak throughput is obtained considering that only the Row Precharge (RP) and Row Address Strobe (RAS) signals are required to obtain a full row buffer, as no Row Cycle (RC) and Column Address Strobe (CAS) delays are necessary.

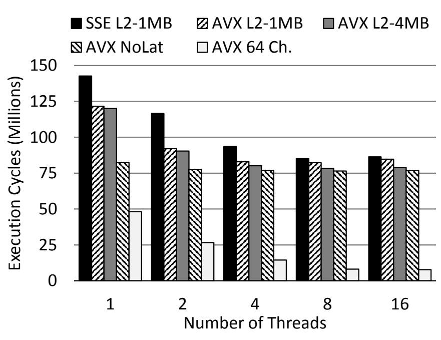
In this paper, we propose to insert Single Instruction Multiple Data (SIMD) units inside the DDR memory devices. The core idea is to perform vector instructions using entire open row buffers using vector functional units. In this way, we eliminate the bandwidth constraint and enable parallelism between vector operations in the DRAM and the processing cores, while we require to embed a logic into the DDR devices that is much smaller than a processor. Considering that a common DDR 3 device provides between 1 and 2 KB row buffer sizes, and in most of the cases memory modules have between 8 and 4 devices, this mean that our mechanism can operate over registers holding up to 8 KB.

4. THE MVX MECHANISM

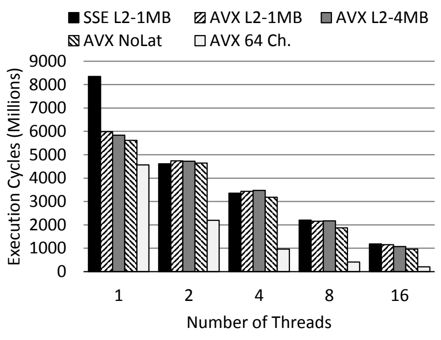
The main focus of this work is to reduce the data movement between processor and main memory for embedded systems, exploring the current implementation of modern DRAM devices. In this section, we introduce our mechanism, the Memory Vector Extensions (MVX), detailing the required processor and memory system modifications as well as the operations of the mechanism.



(a) Vector sum.



(b) Stencil.



(c) Matrix multiplication.

Figure 2: Number of execution cycles for multiple threads performing a vector sum, stencil computation and matrix multiplication.

As explained in Section 3, MVX makes use of the fact that during normal DDRx operation, while the memory controller opens a new row and requests a column of data, a wider range of addresses is available inside the row buffers. Our approach is based on direct access to data available in the sense amplifiers of the open rows. To take full advantage of the data available inside the memory devices, we implement vector functional units along an additional register bank inside the DRAM. These new functional units will answer to specific processor instructions which we introduce.

The main objective of MVX is to perform large-scale computation inside the memory, avoiding expensive data transfers between the memory and processor. Our mechanism behaves similarly to vector instructions that are available in current processors, such as Neon, MMX, SSE and AVX. However, with our mechanism, the processor uses special MVX instructions that are sent to be operated inside the DRAM. The next sections will detail the architectural modifications required by our mechanism, discuss implementation possibilities and evaluate the integration overhead.

4.1 DRAM Modifications

To perform vector instructions inside the DRAM, we require two main logic additions to the DDR device, a register bank and the vector functional units. Figure 3 illustrates the MVX inside a DRAM device. This figure presents a DDR 3 x8 device, even though our mechanism can be easily adapted for different DDRx device layouts (for example, different row bank size, data bursts, etc.). The additional register bank available inside the DRAM device, this component is decoupled from the row buffers in such way that it can be used to store full row buffers from any bank inside the device. Each register is capable of handling an entire row buffer. Thus, open row signals can be issued to different banks to achieve higher performance. Such a register bank could also be used to accelerate non-MVX code by caching row buffers [26], however, this is not evaluated in this paper. In order to support instructions that need to transfer data between memory devices (such as shift or shuffle instructions), a possible implementation would require a bus interconnecting the MVX register banks from different devices. We use this basic implementation in our evaluation.

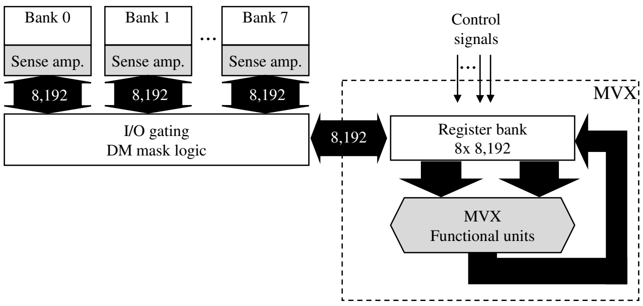


Figure 3: Single DDR 3 x8 device with the modifications required by our mechanism.

Although the MVX operates in a sequential way, the functional units act as a restricted data-flow processor. A given operation may start as soon as the registers are available. To support that data-flow, we must have a flag associated with each register that indicates if the operand is ready. Each MVX instruction must erase this flag for its destination register, and re-enable it whenever the instruction becomes ready. This system enables the DRAM to open rows from different banks in parallel, and also ensures that once a MVX instruction requires operands that are not ready, execution will stall.

Upon registers being ready, the functional units operate in several steps to process the entire row buffer. The number of steps depends on the number of available functional units. We further explore the trade-offs of the number of functional units in Section 5. All functional units operate at the DRAM device frequency. After completion, every MVX instruction sends an acknowledgment signal to the processor, such that our instructions behave similar to a normal memory request. These acknowledgment signals provide important information for the processor regarding the status of each operation, such as overflow, division-by-zero and other exceptions. For the Intel AVX instructions, a set of 17 bits is enough to provide the information regarding the operation status [12]. During our mechanism evaluation, we considered an acknowledgment of 64 bits in order to correctly simulate the impact of this transmission in the final performance.

4.2 Memory Controller Modifications

In our mechanism, the memory controller is responsible for handling the MVX instructions on its internal buffers and sending the instruction to the DRAM in-order. This reduces the amount of logic required inside the DRAM. Although MVX instructions must be treated in-order, the normal read and write operations can still be scheduled using the standard memory controller policy.

When the memory controller receives the MVX lock operation, the memory controller must manage to lock the MVX mechanism inside the DRAM to operate only to that particular thread that requested the lock. This is important to guarantee the consistency of the registers when two or more threads are performing MVX operations. In case the memory is already locked, the MVX instructions from the requester waits until the DRAM gets unlocked. After a lock is granted, the MVX instructions are able to perform their operations, avoiding that one thread modifies the content of registers that are being used by a different thread. Considering that such a lock mechanism is only necessary to maintain the MVX registers consistent, other non-MVX read and write operations can still be issued and serviced normally even when the MVX is locked. A simple mechanism could make use of this locking system to power gate all MVX resources after a certain threshold period of time, reducing any energy overhead during idle periods.

4.3 Processor Modifications

Few modifications inside the processor are required for our mechanism. We require an ISA extension to create our MVX instructions. We design our instructions such that their execution is similar to a subset of AVX instructions. These MVX instructions have to be written in the application's source code, just like normal vector instructions, with the difference that these operations will be applied over a wider data register inside the memory devices.

The MVX instructions use a new register bank inside the DRAM to perform operations. To avoid resource conflicts, a MVX code snippet needs to be wrapped by MVX's lock and unlock instructions. These instructions will perform a lock in the MVX structures for a specific thread, unlocking it whenever an unlock is executed. The application's code with the MVX instructions will pass through the pipeline just as a memory load operation would. MVX instructions that do not require memory addresses, such as MVX lock and unlock, will bypass the address generation unit and wait to be transmitted inside the Memory Order Buffer (MOB).

All the MVX instructions are sent to the MOB to be delivered to the memory subsystem. These instructions wait inside the MOB for an answer from the memory system, which acknowledges the operation as successful or raises exceptions. The processor uses these instructions' acknowledgments to control execution flags such as overflow, not-a-number, among others. The MVX instructions that perform load or store work with virtual addresses and must be translated by the Translation Look-aside Buffer (TLB) and checked for correct permissions to access the given address range. After passing through the TLB, the requests follow the cache memory hierarchy, bypassing the memory caches. The directory must be changed, as once a request arrives into the directory, we ensure that the directory performs a write-back and invalidates all the data in the range at which the specific MVX instruction will operate.

4.4 Logic Overhead

MVX can be implemented with different widths of vector FUs, creating a trade-off between performance and area. Table 3 presents the estimated number of transistors required for each different implementation of MVX and compares it to total size of each DRAM device. For our baseline configuration with 256 FUs, MVX requires about 47 million transistors, which represents less than 1.5% of the DRAM size. Since the number of transistors required for the registers and muxers is low compared to the number of transistors used by the FUs, the total number of transistors scales almost linearly with the number of FUs. Although we do not use floating-point instructions in our evaluation, we include the FP FUs in our calculation to consider a general version of MVX. The NDCores mechanism [20] discussed in Section 2 has an estimated area overhead of 26 million transistors, based on an ARM Cortex-A5 architecture [4], which is slightly higher than our configuration with 128 FUs per device. In our experimental evaluation, we will compare NDCores to our MVX proposal.

Table 3: Estimated logic overhead per DRAM device.

Component	**# of transistors for each config.**			
	256 FUs	**128 FUs**	**64 FUs**	**32 FUs**
#FUs × 32 bit - Int. ALUs	0.5 M	0.3 M	0.1 M	0.1 M
#FUs × 32 bit - Int. Shift.	2.6 M	1.3 M	0.6 M	0.3 M
#FUs × 32 bit - Int. Mul.	5.4 M	2.7 M	1.3 M	0.7 M
#FUs × 32 bit - Int. Div.	4.9 M	2.4 M	1.2 M	0.6 M
#FUs × 32 bit - FP Units	32.7 M	16.3 M	8.2 M	4.1 M
8× 8192 bit - Registers	800 K	800 K	800 K	800 K
8 Banks - Muxers	230 K	230 K	230 K	230 K
Total of transistors	**47.2 M**	**24.1 M**	**12.6 M**	**6.5 M**
% of a 512 MB device	**1.1%**	**0.59%**	**0.31%**	**0.16%**

4.5 Technological Integration Alternatives

Previous near-data computing proposals integrate their mechanisms through three main approaches: DRAM integration inside the memory device, separate logic inside the memory module and 3D stacked integration. For our evaluation, we considered the DRAM integration inside the memory device alternative. However, other integration possibilities could be adopted, each with different trade-offs.

Integrating the mechanism inside the DRAM memory device is a limiting factor for the insertion of large amounts of logic inside the memory, primarily because the technology used in DRAMs limits the speed that can be achieved for logic circuits while requires larger area than other integration cells [14]. MVX does not aggressively modify the current DRAM architecture, placing the control logic inside the memory controller. We consider that MVX functional units and register bank could be integrated directly next to I/O gating logic, where a designer might even be able to take advantage of shared resources.

Techniques such as 3D stacking using TSV [18] emerged as solutions for integration of different technologies directly in DRAM devices. This solution allows the usage of more efficient integration technology to implement the MVX logic, enabling a significant reduction of energy and power consumption. However, this approach could reduce the amount of data that MVX can access in parallel if using an interconnection narrower than the row buffer.

Another approach would be to implement MVX logic outside the devices, inside the memory modules. This implementation would also reduce data transfers outside the DRAM module while allowing different operating frequencies. However, the high cost to extract a large bandwidth from the devices must be considered.

```
void vecsum(int a[], int b[], int c[], int N){
    for(int n=0; n<N; n++)
        c[n] = a[n] + b[n];
}
```

(a) C code.

```
vecsum:
    add     $0x1,%rbp
    movdqu  (%r11,%r9,1),%xmm0  ; ld a[n]
    movdqu  (%rbx,%r9,1),%xmm1  ; ld b[n]
    paddd   %xmm1,%xmm0  ; add a[n], b[n]
    movdqu  %xmm0,(%r10,%r9,1) ; st c[n]
    add     $0x10,%r9
    cmp     %rbp,%r13
    ja      <vecsum>
```

(b) SSE assembly code.

```
vecsum_MVX:
    add     $0x1,%rbp
    MVXlock ; lock MVX
    MVXmovdqu (%r11,%r9,1),%mvx0  ; ld a[n]
    MVXmovdqu (%rbx,%r9,1),%mvx1  ; ld b[n]
    MVXpaddd  %mvx1,%mvx0  ; add a[n], b[n]
    MVXmovdqu %mvx0,(%r10,%r9,1) ; st c[n]
    MVXunlock ; unlock MVX
    add     $0x2000,%r9
    cmp     %rbp,%r13
    ja      <vecsum_MVX>
```

(c) MVX assembly code.

```
vecsum_MVX_unrolled:
    add     $0x1,%rbp
    MVXlock
; ld a[n], a[n+1], a[n+2], a[n+3]
    MVXmovdqu $0x0000(%r11,%r9,1),%mvx0
    MVXmovdqu $0x2000(%r11,%r9,1),%mvx1
    MVXmovdqu $0x4000(%r11,%r9,1),%mvx2
    MVXmovdqu $0x6000(%r11,%r9,1),%mvx3
; ld b[n], b[n+1], b[n+2], b[n+3]
    MVXmovdqu $0x0000(%rbx,%r9,1),%mvx4
    MVXmovdqu $0x2000(%rbx,%r9,1),%mvx5
    MVXmovdqu $0x4000(%rbx,%r9,1),%mvx6
    MVXmovdqu $0x6000(%rbx,%r9,1),%mvx7
; add a[n],b[n]; a[n+1],b[n+1]; ...
    MVXpaddd  %mvx4,%mvx0
    MVXpaddd  %mvx5,%mvx1
    MVXpaddd  %mvx6,%mvx2
    MVXpaddd  %mvx7,%mvx3
; st c[n], c[n+1], c[n+2], c[n+3]
    MVXmovdqu %mvx0,$0x0000(%r10,%r9,1)
    MVXmovdqu %mvx1,$0x2000(%r10,%r9,1)
    MVXmovdqu %mvx2,$0x4000(%r10,%r9,1)
    MVXmovdqu %mvx3,$0x6000(%r10,%r9,1)
    MVXunlock
    add     $0x8000,%r9
    cmp     %rbp,%r13
    ja      <vecsum_MVX_unrolled>
```

(d) MVX assembly code with the main loop unrolled 4 times.

Figure 4: Four code versions of the vector sum benchmark. The C code shown in Figure (a) generates the other 3 versions.

4.6 Binary Generation and Optimizations

In order to use MVX instructions, we require no changes to source codes. However, the code needs to be recompiled in order to make use of the MVX instructions. In our proposal in this paper, we consider that the compiler aligns the data for MVX at the row buffer size (8 KB in our evaluation). Inside the DRAM, MVX performs operations sequentially. However, we adopted a loop unrolling technique in order to better organize the loads and operations in such a way that using the register bank to store the data, the memory controller can issue the row access signals as early as possible, exposing better the bank parallelism inside the DRAM.

To perform a loop unrolling, we set the compiler objective of the target code to perform memory loads as early as possible, which is a technique that is already commonly performed by compilers [11]. Figure 4 presents three different assembly codes for the vector sum kernel, the x86 version using SSE instructions with XMM registers and two versions using the MVX ISA. For the MVX version, we present the kernel with the normal loop and the unrolled loop. Preliminary results have shown an average of 68% improvements when unrolling loops. We use the loop unrolling technique for all the results of our experiments.

5. EXPERIMENTAL EVALUATION

This section presents the simulation details, the application kernels and the evaluation results comparing our mechanism to the baseline embedded system and previous work.

5.1 Configuration Parameters and Baseline

To evaluate our MVX mechanism, we used an in-house cycle-accurate simulator [2, 3]. The simulation parameters are inspired by Intel's Atom processor with the Silvermont Out-of-Order (OoO) micro-architecture [13]. Table 4 shows the simulation parameters used for our tests. The *SSE* baseline system supports the SSE 4.2 vector instructions using 128 bits registers.

The Silvermont micro-architecture only supports SSE instructions on up to 8 cores and 2 memory channels. In order to build a possible future scenario for comparison, we also extrapolate the baseline configuration and build a second system named *AVX*. This second system supports the AVX vector instructions using 512 bits registers (the same used on Xeon-Phi processors). We then evaluate our proposal against configurations with up to 16 cores, and up to 64 memory channels.

As the MVX hardware is a set of vectorial functional units and a register bank, we performed the experiments with the

Table 4: Baseline and MVX system configuration.

OoO Execution Cores - 2 GHz; 16 cores; Front-end 2-wide;
1 branch per fetch, 8 parallel in-flight branches;
14 stages (3-fetch, 3-decode, 3-rename, 2-dispatch, 3-commit);
24-entry fetch buffer, 32-entry decode buffer, 32-entry ROB;
2-alu, 1-mul. and 1-div. INT units (1-3-20 cycle);
1-alu, 1-mul. and 1-div. FP units (5-5-20 cycle);
1-load and 1-store functional units (1-1 cycle);
MOB entries: 10-read and 10-write;

Branch Predictor - 4 K-entry 4-way set-associative BTB;
Two-Level PAs predictor; 16 K-entry BHT, 2-bits prediction;

L1 Data + Inst. Cache - 32 KB, 8-way, 2-cycle; 64 bytes line;
LRU policy; MSHR entries: 8-request, 8-write-back, 1-prefetch;
Stride Prefetcher: 1-degree, 16-strides table;

L2 Cache - 1 MB shared for every 2 cores;
16-way, 4-cycle; 64 bytes line; LRU policy;
MSHR entries: 16-request, 8-write-back, 2-prefetch;
Inclusive LLC; MOESI coherence protocol;
Stream Prefetcher: 2-degree, 16 prefetch distance, 32-streams;

DRAM Controller and Interconnection - Bi-directional ring;
On-chip DRAM controller, Open-row first policy, 1-channel;
DDR3-1333, 8 burst length at 3:1 frequency ratio;
8 DRAM banks, 8 KB row buffer per bank (1 KB per device);
CAS, RP, RCD, RAS and CWD latency (9-9-9-24-7 cycles);

MVX Processing Logics - Operation frequency: 166 MHz;
Latency (cycles): 1-alu, 3-mul. and 20-div. integer units;
Latency (cycles): 5-alu, 5-mul. and 20-div. floating-point units;
Up to 256 sets of functional units (INT + FP) per device;
1 register bank per device, with 8 registers of 8,192 bits each;
Interconnection between registers: 3 cycles latency;

MVX working at DRAM device operating frequency, where all the vectorial operations could be performed in parallel. However, experiments with a reduced number of functional units are also presented.

5.2 Application Kernels

For our evaluations we used three different applications kernels: the *vector sum*, the 5-points *stencil* and the 2-dimensional *matrix multiplication*. The vector sum application calculates a sum of two vectors of integers, storing the result in a third vector. During normal execution, the processor performs sums with vector instructions over an array of contiguous elements. The vector sizes evaluated were 1, 2, 4, 8, 16, 32 and 64 MB.

The stencil application performs a single step application of a 5-point stencil over a matrix of integer elements, adding up the 5 neighbors elements, multiplying the result by two and then storing every result in an output matrix. During the computation, the algorithm uses two aligned loops, using vector instructions over multiple contiguous elements, in such a way that after each inner loop iteration, the calculation over multiple points are ready. The matrices are square and the sizes evaluated were 1, 2, 4, 9, 16, 36 and 64 MB.

The matrix multiplication application uses three integer square matrices, $C = A \times B$. The algorithm multiplies a single element from matrix A with multiple contiguous elements from matrix B, accumulating the results into multiple contiguous elements of the matrix C. The matrices evaluated have sizes of 1, 2, 4, 9 MB.

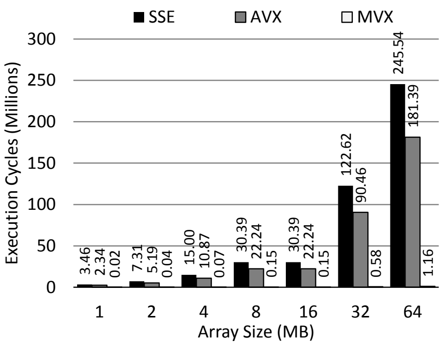
The vector sum application represents the most favorable case for our mechanism since it does not reuse data and only performs a stream over contiguous elements. The stencil presents some data reuse, making use of the cache memory. The matrix multiplication application has a high amount of data reuse, thus benefiting greatly from the cache memories. The assembly code of the kernel of these three applications was obtained from the gcc compiler, and it is similar to the code generated by Intel compiler, using the SIMD extensions present in the simulated processors.

5.3 Results Varying the Input Size

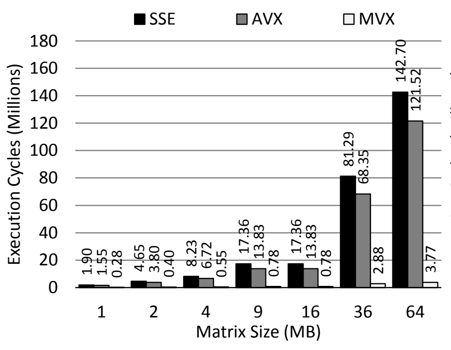
Figure 5 presents the number of execution cycles for the three evaluated application kernels using only one thread and varying the input size of each application. We can then observe how each system scales when the data size increases.

Comparing our mechanism to the baseline system that supports SSE, we can observe that our mechanism performed on average 204×, 18× and 10× faster for the vector sum, stencil and matrix multiplication applications, respectively. When compared to the AVX, the MVX performed 147×, 14× and 6× faster than the baseline for the same applications.

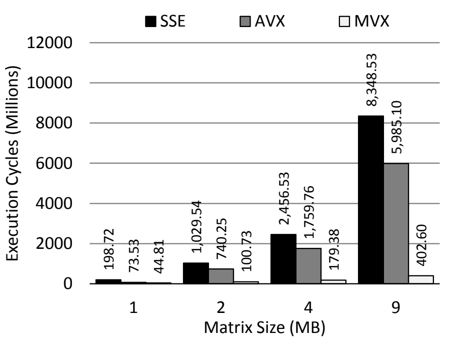
This first experiment also shows the benefits of the wider vector units inside the processor, where the AVX performed on average 1.41× faster than the SSE. Thus, for the next results we keep only the AVX model for the rest of the comparisons.



(a) Vector sum.

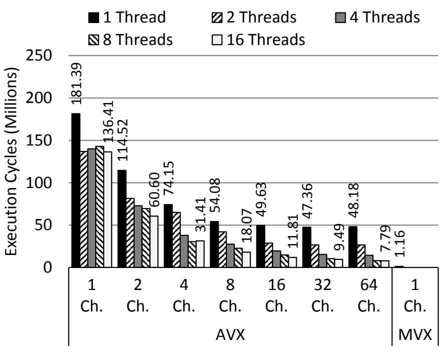


(b) Stencil.

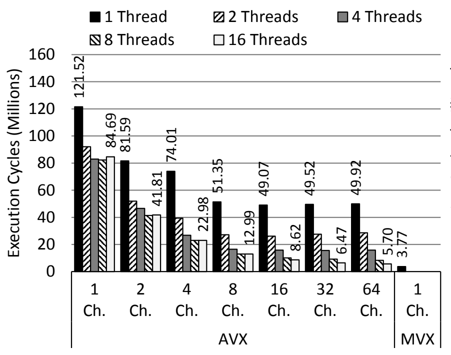


(c) Matrix multiplication.

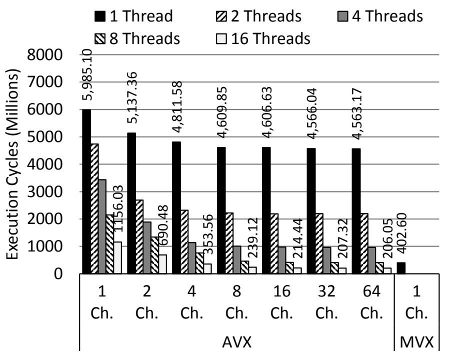
Figure 5: Applications' number of execution cycles varying the input size.



(a) Vector sum.



(b) Stencil.



(c) Matrix multiplication.

Figure 6: Applications' number of execution cycles varying the number of channels.

5.4 Results Varying the Number of Threads and Memory Channels

Figure 6 evaluates the baseline performance scalability when the number of threads and memory channels increases, for the three applications kernels. This experiment enables a comparison between our mechanism to a baseline exploring the parallelism of multiple cores while we also extrapolate the number of memory channels on the system that supports AVX. With this experiment we intend to show the break-even point for the implementation of our mechanism when compared to highly parallel architectures with larger number of cores and bandwidths for memory access. However, for the vector sum, we found that our mechanism performs 6× faster than a baseline with 64 channels and 16 cores.

When increasing the amount of data reuse, for the stencil application, our mechanism still performs 1.5× faster than the baseline that uses 64 channels and 16 cores. For the matrix multiplication, which has a high amount of data reuse, the break-even point occurs only with 16 cores and more than 4 memory channels.

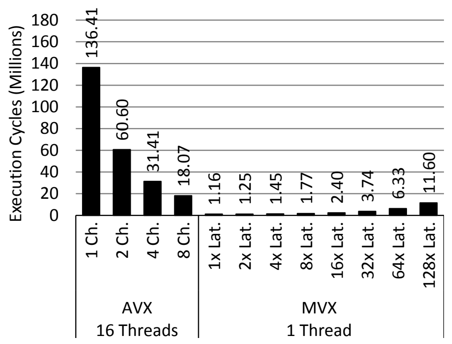
5.5 Results Varying the MVX Latency

Considering that different integration technologies could lead to different latencies for our mechanism operation, Figure 7 evaluates MVX with different latencies to perform its operations on the functional units. This experiment gives us insight regarding two project decisions: first, our mechanism could be implemented using fewer functional units, performing the operations over fewer operands in a multi-cycle way; second, MVX could be implemented in a lower frequency than the memory (that is, lower than 166 MHz). In both cases the latency to perform a single operation would be multiplied by a factor, depending on the number of functional units or the operation frequency.

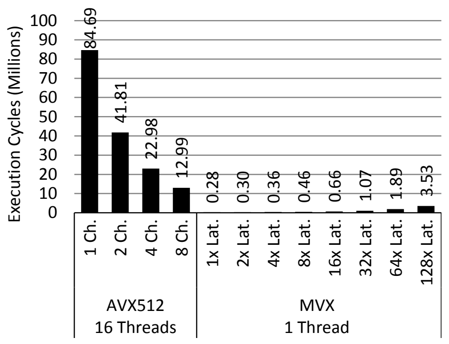
For vector sum and stencil, which present zero or low amount of data reuse, we can observe that even increasing the latency by 128×, our mechanism continues to deliver a higher performance than the baseline system with 8 channels and 16 cores. Executing the matrix multiplication application, our mechanism with an increased latency of 4× is still better than a system with 1 or 2 channels and 16 cores. We can see with these results that our mechanism presents a wide design space with sustained performance gains.

5.6 Comparison to Related Work

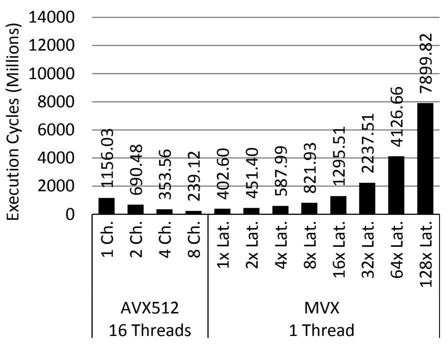
We compare our MVX proposal to the NDCores mechanism [20] discussed in Section 2. We have implemented the NDCores with 4 cores per DRAM device, and a total of 8 devices. We can see that MVX is on average 25×, 4× and 3× than NDCores for vector sum, stencil and matrix multiplication kernels. We can observe that for stencil and matrix multiplication applications, the NDCores makes a better use of its cache, achieving a better result. However, for vector sum, the MVX have a good advantage over NDCores. Comparing MVX with smaller logic overhead than NDCores (that is, MVX with 128 or less FUs), our proposal achieves substantially better results than the previous work for all the evaluated application kernels and input sets.



(a) Vector sum.

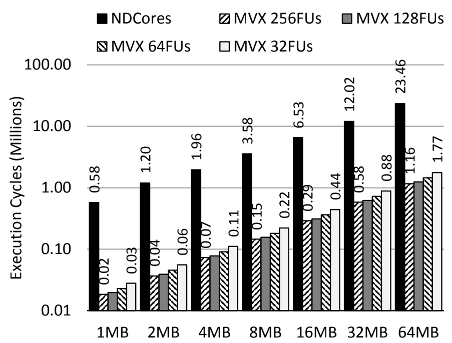


(b) Stencil.

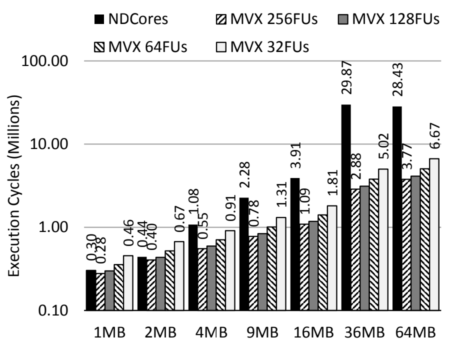


(c) Matrix multiplication.

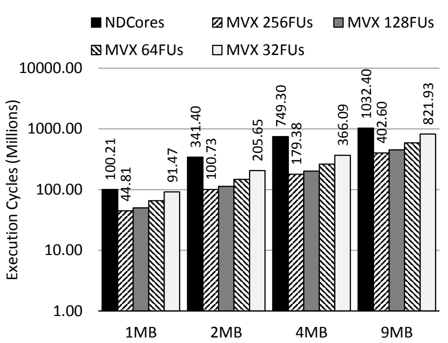
Figure 7: Applications' number of execution cycles varying the latency per MVX operation.



(a) Vector sum.



(b) Stencil.



(c) Matrix multiplication.

Figure 8: Applications' number of execution cycles comparing MVX to the NDCores mechanism [20]. The y-axis in all three figures is in logarithmic scale.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced Memory Vector Extensions (MVX), a new approach to perform near-data computing implemented directly in the DRAM devices. Similar to the related work in the area, we showed that reducing data transfer is an efficient approach to increase performance by performing calculations closer to the main memory, saving memory movements between memory and cores.

MVX is capable of achieving high performance gains by executing vector instructions over a large volume of data inside DRAM devices. Through our experiments, we proved that MVX is capable of executing up to 211× faster than a single-threaded core. Moreover, when we extrapolated the number of cores and memory channels of the baseline architecture, MVX was still a better choice for applications with zero or low amount of data reuse. Contrary to previous work, our design can be implemented in a multitude of ways using a relatively low amount of hardware, as we show that even a reduced number of functional units can greatly improve the performance by avoiding data transfer. This is important, as new technologies, such as TSV or CGRA, have not met adoption due to several manufacturing constraints.

In the future, we plan to evaluate our mechanism in a HMC environment, evaluate complex benchmarks that use floating point instructions, and also evaluate the impact and implementation of our mechanism in a system with multiple memory channels and memory controllers. In this way, we can evaluate how our mechanism behaves for larger, performance-oriented systems.

7. REFERENCES

[1] J. W. Adkisson, R. Divakaruni, J. P. Gambino, and J. A. Mandelman. Embedded dram on silicon-on-insulator substrate, 2002. US Patent 6,350,653.

[2] M. A. Z. Alves. *Increasing Energy Efficiency of Processor Caches via Line Usage Predictors.* PhD thesis, Universidade Federal do Rio Grande do Sul, 2014.

[3] M. A. Z. Alves, M. Diener, F. B. Moreira, C. Villavieja, and P. O. A. Navaux. Sinuca: A validated micro-architecture simulator. In *High Performance Computation Conf*, 2015.

[4] ARM. *Cortex-A5 Technical Reference Manual.* 2010.

[5] E. Azarkhish, D. Ross, I. Loi, and L. Benini. High performance axi-4.0 based interconnect for extensible smart memory cubes. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2015.

[6] N. Bowman, N. Cardwell, C. Kozyrakis, C. Romer, and H. Wang. Evaluation of existing architectures in iram systems. In *Workshop on Mixing Logic and DRAM*, 1997.

[7] B. T. Davis. *Modern dram architectures.* PhD thesis, University of Michigan, 2001.

[8] J. Draper, J. Chame, M. Hall, et al. The architecture of the diva processing-in-memory chip. In *Int. Conf. on Supercomputing (ICS)*, 2002.

[9] D. Elliott, M. Stumm, W. Snelgrove, C. Cojocaru, and R. McKenzie. Computational ram: Implementing processors in memory. *Design and Test of Computers, IEEE*, 1999.

[10] A. Farmahini-Farahani, J. Ahn, K. Compton, and N. Kim. Drama: An architecture for accelerated processing near memory. *Computer Architecture Letters*, 2014.

[11] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *ACM SIGPLAN Notices*, 1986.

[12] Intel. *Intel ® Xeon Phi TM Coprocessor Instruction Set Architecture Reference Manual.* 2012.

[13] Intel. Intel Atom Processor E3800 Product Family. Technical report, 2015.

[14] B. Jacob, S. Ng, and D. Wang. *Memory systems: cache, DRAM, disk.* Morgan Kaufmann, 2008.

[15] D. Lee, Y. Kim, V. Seshadri, et al. Tiered-latency dram: A low latency and low cost dram architecture. In *Int. Symp. on High Performance Computer Architecture (HPCA)*, 2013.

[16] G. H. Loh and M. D. Hill. Efficiently enabling conventional block sizes for very large die-stacked dram caches. In *Int. Symp. on Microarchitecture*, 2011.

[17] Micron. 1gb: x4, x8, x16 ddr3 sdram features, 2006. 1Gb_DDR3_SDRAM.pdf - Rev. N 11/14 EN.

[18] J. V. Olmen, A. Mercha, G. Katti, et al. 3d stacked ic demonstration using a through silicon via first approach. In *Int. Electronic Devices Meeting (IEDM)*, 2008.

[19] D. Patterson, T. Anderson, N. Cardwell, et al. A case for intelligent ram. *Micro, IEEE*, 1997.

[20] S. Pugsley, J. Jestes, R. Balasubramonian, et al. Comparing implementations of near-data computing with in-memory mapreduce workloads. *Micro, IEEE*, 2014.

[21] S. Pugsley, J. Jestes, H. Zhang, et al. Ndc: Analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads. In *Int. Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[22] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the memory wall: The case for processor/memory integration. In *Int. Symp. on Computer Architecture (ISCA)*, 1996.

[23] M. Wei, M. Snir, J. Torrellas, and R. B. Tremaine. A near-memory processor for vector, streaming and bit manipulation workloads. Technical Report UIUC DCS-R-2005-2557, University of Illinois at Urbana-Champaign, Dept. of Computer Science, 2005.

[24] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 1995.

[25] D. P. Zhang, N. Jayasena, A. Lyashevsky, et al. A new perspective on processing-in-memory architecture design. In *Workshop on Memory Systems Performance and Correctness (MSPC)*, 2013.

[26] Z. Zhang, Z. Zhu, and X. Zhang. Cached dram for ilp processor memory access latency reduction. *IEEE Micro*, 2001.

[27] Q. Zhu, B. Akin, H. E. Sumbul, et al. A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing. In *Int. 3D Systems Integration Conf. (3DIC)*, 2013.

[28] Q. Zhu, T. Graf, H. E. Sumbul, et al. Accelerating sparse matrix-matrix multiplication with 3d-stacked logic-in-memory hardware. In *High Performance Extreme Computing Conf. (HPEC)*, 2013.