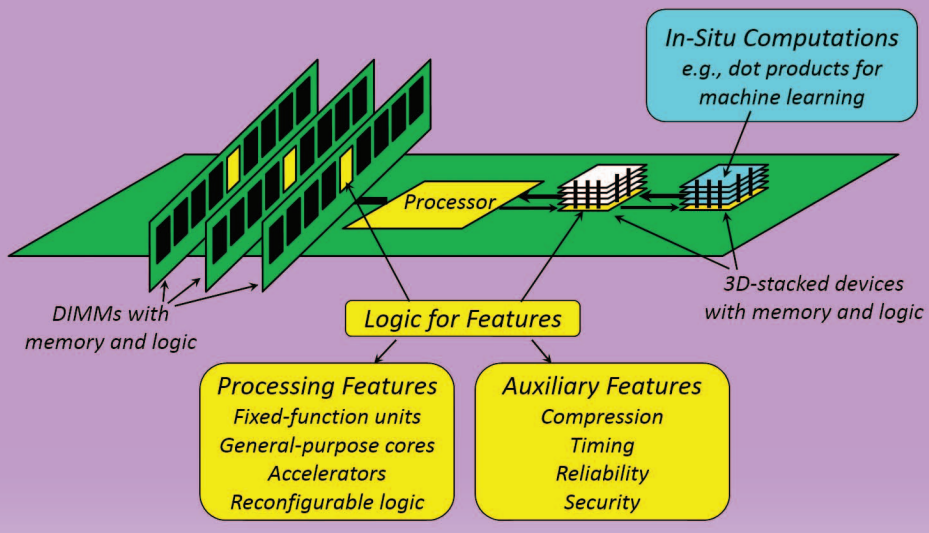

Making the Case for Feature-Rich Memory Systems

Rajeev Balasubramonian

Digital Object Identifier 10.1109/MSSC.2016.2546198
Date of publication: 21 June 2016

Many emerging workloads are constrained by the high cost of data access. Innovation in the memory system may soon be the primary driver of the computing economy. The result will be a memory system that is specialized, not commoditized. This article discusses the features that can be meaningfully added to memory devices. Not only do these features execute parts of an application, they may also take care of auxiliary operations that maintain high efficiency, reliability, and security.

Introduction

Memory products have long been commoditized and standardized, while the processor has remained a hot bed of innovation for many decades. However, in the coming decade, we can expect a reversal in roles.

Innovations to processor cores have started to taper out, and the microarchitectures of throughput-optimized

1943-0582/16©2016IEEE

and latency-optimized general-purpose cores are fairly well understood [16]. There will be a steady trickle of architecture innovations for general-purpose processors, but these are unlikely to disrupt the relatively flat average improvement curve for large benchmark suites. Without significant annual improvements, computer systems end up as commodities sold at low margins.

What then drives the computing industry forward? What is the motivation for hardware/architecture innovation?

A shift toward *specialization* is inevitable. There will likely be a significant low-margin market for general-purpose commodity systems and a second significant high-margin market for specialized systems. This is how the automobile industry has operated for decades. To some extent, this is already a reality today in the computing market. A desktop computer can be built for around US$500; this is how we build a cluster to do many architecture simulations in parallel. But a single graphics processing unit (GPU) card can cost ten times that amount, and this is what we use to run our machine learning algorithms.

Two phenomena will serve as the drivers of the computing industry in the coming decade. The first is the growing focus on *accelerators*. The second is a shift toward *feature-rich memory systems*. Both of these paths are relatively less traveled, i.e., they have the potential to uncover large benefits. Combined, these two phenomena will form the basis for specialized systems that can significantly outperform previous-generation systems and command a higher price tag.

Accelerators are being actively studied in architecture research circles. Accelerators have already been designed for popular data-intensive algorithms, e.g., data partitioning [43], database queries [44], sort [26], and machine learning [6].

As computational throughput on a processor increases, with help from accelerators, there is a corresponding demand for higher memory capacity and bandwidth. Enterprise-class workloads, e.g., SAP HANA [30] and SAS in-memory analytics [31], are well known for demanding low-latency access for massive data sets. This is an increasingly prevalent phenomenon as several industries grapple with analytics that can convert big data into big money.

For a few decades now, researchers have considered off-loading parts of an application to a processor embedded in the memory system.

In this era of big data processing, a large fraction of overall time and energy is expended in data access and data movement. Following Amdahl's law, the memory/storage system is clearly where system innovations can have the largest impact. This is especially true because the memory system has not been a target of architecture innovations for the past three decades. We are long overdue for specialized memory systems that are not constrained by standards or by an unwavering focus on cost per bit.

Memory system innovations can help a vendor distinguish its products from the competition. The new currency for a memory product will therefore be *features*. Cost per bit is a fine metric for the commodity general-purpose space, but it will be a secondary metric for specialized systems.

So what features can one place within the memory system? These features may include, for example, simple processing units, accelerators, logic for reliability, security, and compression. Such features are compatible with most memory technologies, while some features (e.g., those dealing with wearout) are an especially good fit for emerging nonvolatile memory (NVM) technologies. While minimal amounts of logic may be placed within memory dies, the more significant features will likely be placed in separate logic chips. These logic chips can be coupled with memory dies either with through silicon vias (TSVs) in a 3D-stacked package or with on-board traces in a dual in-line memory module (DIMM) form factor. This article discusses the features that can be meaningfully added to memory devices and the impact they can have on server architectures.

Memory System Features

Figure 1 summarizes the overall approach of a feature-rich memory system. I will classify memory system features into two main groups: *processing features* and *auxiliary features*. The first group provides logic to execute parts of the application, and this logic can take the form of a general-purpose processor or an application-specific accelerator. The second group provides logic to perform auxiliary operations that are independent of the application but critical for overall system efficiency. Such operations may include wear leveling, encryption, compression, and coding.

Processing Features

For a few decades now, researchers have considered off-loading parts of an application to a processor embedded in the memory system. The area of *processing-in-memory* (PIM) was heavily researched in the 1990s but remained dormant for a decade after that for a variety of reasons, most notably, the economics of integrating logic and DRAM on a single die.

The area has now reemerged [4], thanks to improvements in technology [e.g., three-dimensional (3D) stacking], the demands of emerging workloads (e.g., big data workloads that benefit from high memory bandwidth), and, as

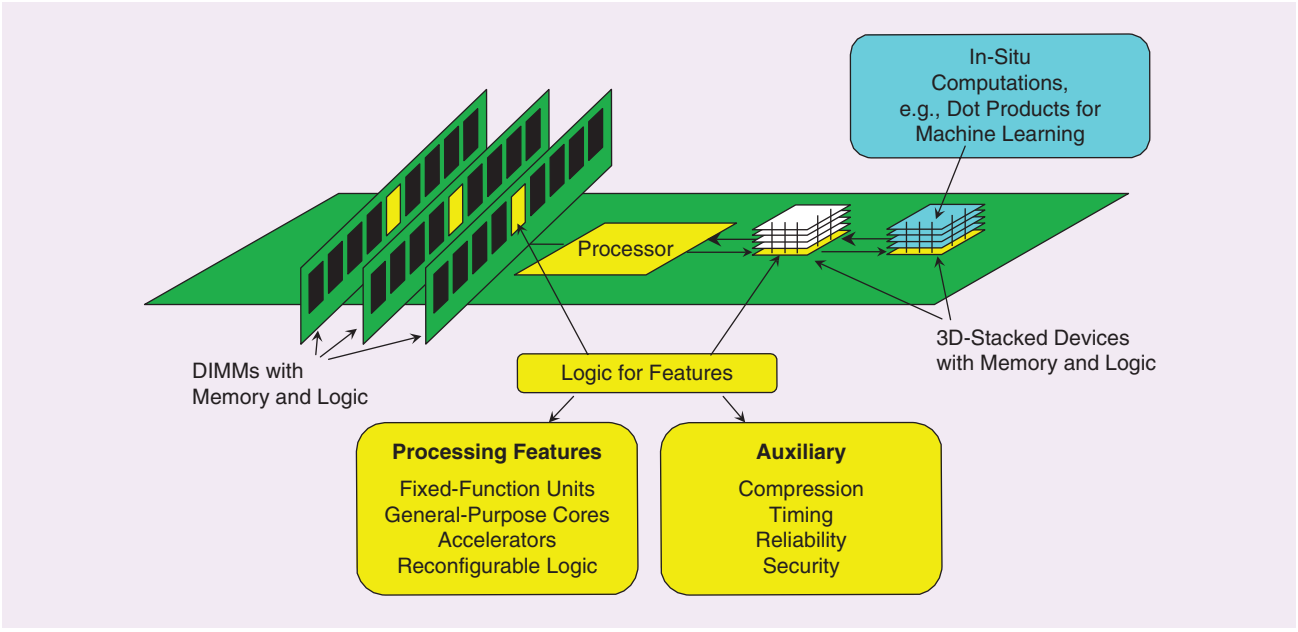

FIGURE 1: An overview of feature-rich memory architectures.

detailed in the introduction, the need for value additions in the memory system. The area has also broadened its scope—processors may be placed in a 3D-stacked package [28], [21], on a DIMM [27], and on solid-state drive (SSD) devices [7]. Accordingly, the term *near data processing* (NDP) is more descriptive and accurate than PIM. NDP research has been in the spotlight recently, with multiple papers at top-tier venues, a very successful series of workshops (WoNDP) at the International Symposium on Microarchitecture, and a special issue in *IEEE Micro*. Indeed, research in this area has closely tracked the famous Gartner hype curve [11], with a steep rise, a steep fall, and, hopefully now, a period of robust enlightenment and productivity.

So why is NDP useful? Consider an application that is searching for a particular record in a data set. Conventional architectures would move the entire data set to the processor, consuming all the memory bandwidth, expending large amounts of data movement energy and polluting the processor caches, only to isolate a few records of interest. This is the classic killer app for NDP. Such a "filtering" operation can be performed by a simple processor on the memory device. The near-data processor enjoys both lower latency and lower energy for memory access and higher memory bandwidth. Meanwhile, the processor's caches and memory bandwidth can be better used for other relevant data sets that exhibit higher locality.

Clearly, several research issues need to be addressed to realize the potential of NDP. Below, each of these issues are listed, along with example attempts to address them. While most of this discussion uses near-DRAM processing as a driving example, note that similar approaches can also be used for near-NVM and near-flash processing.

What Workloads Can Benefit from NDP?

To answer this, we first identify the types of computations that can be meaningfully off-loaded to the memory device. As a strawman, consider the generic NDP architecture in Figure 2, which includes a network of processor sockets and memory devices, each with multiple processing cores. This is essentially a distributed

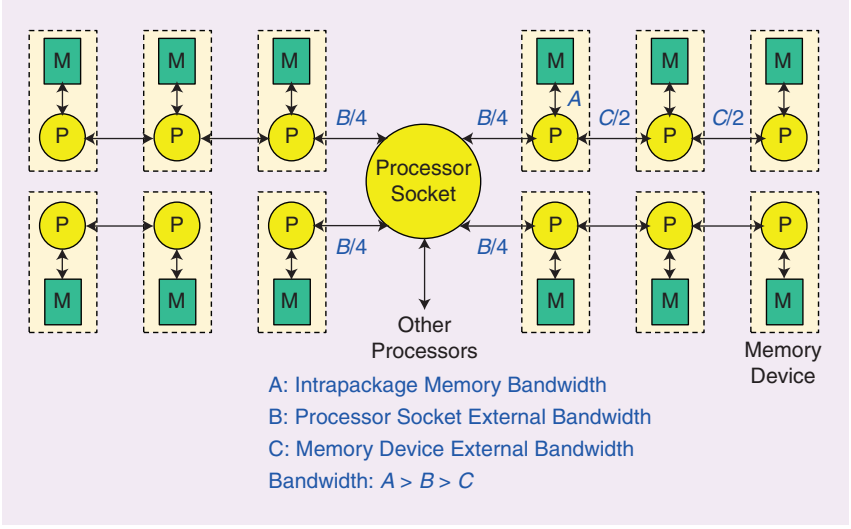

FIGURE 2: A generic bandwidth model for NDP.

computation model, with cores having asymmetric views of memory. A core on a memory device has a high-bandwidth link (*A*) to its local memory and a low bandwidth link (*C*) to nonlocal memory. A core on the processor socket has a collection of low-bandwidth links (*B*) to an aggregation of memory devices. In terms of bandwidth, $A > B > C$. If a computation can localize its memory accesses to data in a single memory device, the computation is best placed on that memory device so it can exploit the high bandwidth of *A*. If a computation has limited locality, it must determine if it is better to off-load to the memory device and exploit a combination of *A* and *C* or remain on the processor socket and exploit *B*.

This is clearly a simplistic view because it does not consider detailed network topologies nor the quality of the cores/caches on the processor/memory devices. But at a high level, it conveys the key point that a computation with localized memory accesses is an ideal candidate for near-data execution.

A second key component in this analysis is parallelism. Each additional memory device adds more memory and compute resources. If a computation can be parallelized across the many cores in memory devices, it is an even better candidate for near-data execution.

There are a few other considerations in the off-load decision.

- Is one of the cores or one of the cache hierarchies especially beneficial for the computation at hand?
- What is the cost of spawning a task (passing code and arguments to the memory device)?
- What is the cost of terminating a task (returning results to the processor socket)?
- What is the length of the off-loaded function?

It is nontrivial to factor in these issues to develop an automatic hardware/software off-load policy. It is, therefore, an active area of research. We briefly describe two examples here that represent opposite ends of the spectrum.

The work of Ahn et al. [2] attempts fine-granularity off-loads with *PIM-enabled instructions* (PEIs). Individual instructions can be executed either on the host processor or on the memory device (with a locality monitor that helps in this decision making). Consider the example where a single scalar value is being added to some word that is not currently cache resident. Without PEIs, an entire 64-B cache line is brought to the processor, an update is performed, and the entire 64-B cache line is sent back to memory. With PEIs, the 8-B scalar value is sent to the memory device (with appropriate control bits) so the update can be performed directly in the memory device.

In this particular example, PEIs yield a 16× decrease in bandwidth requirement, the length of the off-loaded function is one, and the cost of task spawning and termination is actually *lower* with PEIs than without PEIs. Ahn et al. also extend the instruction set architecture (ISA) so that nontrivial functionalities can be off-loaded to memory devices.

Meanwhile, work from our group [28] focuses on in-memory MapReduce applications that exhibit a very high degree of locality and task-level parallelism. Each map and reduce task is executed on a memory device that contains that data partition, dubbed *near data computing* (NDC). Task setup and teardown are nontrivial efforts, especially if data shuffling is required. But that overhead is palatable because each task executes for many thousands of cycles. The task latency itself is highly sensitive to memory bandwidth. The primary source of speedup is the high bandwidth within a collection of memory devices, which is far greater than the bandwidth into the processor socket.

The PEI approach can yield a nearly 1.5× average speedup for a range of memory-intensive workloads that do not exhibit cache line reuse, while the NDC approach can yield up to 15× speedup for a specific class of memory-intensive workloads that have high coarse-grained parallelism. This also provides insight on the data access patterns of workloads that benefit from NDP. In-memory MapReduce (e.g., in SPARK [46]) is a killer app that exhibits localized memory access and embarrassing levels of coarse-grained parallelism [28]. In-memory MapReduce frameworks have been shown to be useful for a wide range of applications: database operations, analytics, machine learning, and graph algorithms [46].

The PEI work shows benefits for a number of graph workloads where data traversal is random enough that caches are ineffective, and small computations within each graph vertex can be off-loaded to memory. They also extend the ISA to perform hash table probing, histogram bin indexing, and dot products over cache lines to accelerate data mining and machine learning applications. Other papers have also shown NDP benefits for other applications, e.g., graph processing [1], scientific kernels that map to coarse-grained reconfigurable accelerators (CGRAs) [10], scientific workloads [21], signal processing [15], and join algorithms [20].

How Should Data Be Organized Across Memory Devices?

A natural next issue is the interleaving and addressing of data across several memory devices. Unlike conventional double-data rate (DDR) memory that stripes a single cache line across multiple DRAM packages, an entire cache line or even an entire page in NDP must now be localized to a single memory package. This allows the core/accelerator on the memory package to perform fine- and coarse-grained computation without engaging in complex bit-level manipulations and without aggregating inputs from many sources. This already appears to be the default data mapping in emerging memory devices like the hybrid memory cube (HMC) [18]. However, when the same data is accessed by a host processor socket, it may lead to longer transfer times.

It is also important to resolve how a memory device may potentially access data in a different memory

device. One popular option is to never allow this, requiring the application on the host processor socket to marshall any necessary data before spawning an NDP task (as was done by Pugsley et al. [28]). Another option is to simply treat every core as being part of a full-fledged shared-memory multiprocessor system, i.e., every core can issue loads and stores to any globally visible address regardless of whether the core resides on the host processor or on the memory device. This entails more software/hardware complexity because it requires the memory device to maintain a coherent translation look-aside buffer (TLB) and serve as an originator of memory requests.

This brings us to yet another key and somewhat unsolved issue—how is virtual memory handled? One solution, as suggested in the PEI work [2], is to leave virtual memory management entirely up to the host processor. When a task is spawned on the memory device, it is provided the necessary arguments as physical addresses; the task is not allowed to touch data beyond the cache lines (or pages) that were provided as arguments. Another solution, as suggested by Pugsley et al. [28], is to organize the data on a memory device into a few large pages. This is a good fit for many big data applications, and it reduces the overheads associated with page faults, large TLBs, etc.

What Microarchitecture Is Best for Near Data Processing?

In most prior work, the cores on the memory device have been designed to be "wimpy." While some proposals incorporate full-fledged general-purpose wimpy cores [28], such as the 80 mW ARM Cortex A5 cores in NDC [28] that can execute entire general-purpose map or reduce tasks, others only implement custom functional units or accelerators [2], [3], [15]. In the work of Ahn et al. [2], the functional unit is only capable of executing a single PIM-enabled instruction, and the most complex functional unit handles dot-product computations for the words in a cache line. In another example, Akin et al. [3] design a 178-mW functional unit that can permute data.

In addition to these fixed-function units, we have also seen examples of reconfigurable accelerators, such as the use of CGRAs by Farmahini-Farahani et al. [10] and predefined accelerator primitives that can be chained together to perform more complex operations [15]. Also, there are proposals to combine general-purpose wimpy cores and accelerators, e.g., for in-memory MapReduce workloads, Pugsley et al. [26] execute map and reduce phases on Arm cores, while the sort phase between map and reduce is handled by a fixed-function accelerator. These are all compelling design points on the classic generality versus efficiency spectrum.

So why is it best to pursue a "wimpy" core instead of a low-latency out-of-order core? An argument that is frequently cited is the reluctance to embed a high-power core in a 3D-stacked package for fear that it may lead to thermal issues. But this is often a red herring. For example, adding a few watts to a 13-W HMC device [18] is unlikely to pose a hazard, especially if some of the external bandwidth can be eliminated [28]. A more detailed study by Eckert et al. [9] makes exactly that argument.

The more credible argument in favor of wimpy cores is that it actually leads to higher performance as it enables the creation of a throughput-optimized compute substrate that can leverage the high bandwidth afforded by NDP. As mentioned earlier, one of the main benefits of NDP is that plugging in more memory modules leads to more cores and a large-scale parallel system. This is most useful for tasks with high degrees of parallelism. For such a highly parallel task, the path to high performance at a fixed power budget is to use many low-power cores, not a few high-power cores [28]. To be more precise, for a highly parallel task, we can optimize throughput at a fixed power budget by optimizing energy per instruction [28]. Therefore, it is best to use cores or accelerators that are optimized for low energy and not low latency. This also enables the use of many cores or accelerators per memory device, an important requirement if we want to saturate the available bandwidth.

Where Can Processors/Accelerators Be Placed?

About two decades ago, there was a strong push to place computation on the memory die itself. With a potentially lower focus on cost per bit in the future, that approach may yet have merit. But so far, few have chosen to revisit that direction. A few works by Seshadri et al. [33]–[35] have proposed small changes to DRAM arrays to support bit manipulations and efficient data movement.

The vast majority of NDP studies in the last few years have focused on 3D-stacked memory devices. This approach leaves the DRAM dies relatively untouched, while leveraging TSVs to support very high intrapackage bandwidth. By localizing the cores/accelerators to a separate die, they can be implemented in a superior logic process. This approach is often touted as the solution that offers the benefits of NDP at relatively low cost, and that is compatible with the natural evolution of DRAMs (3D stacking). However, early indications are that 3D-stacked DRAMs, especially those that include a logic die, will not be cheap. In certain segments, the cost increase will be well worth the higher performance.

Given the high cost of 3D-stacked DRAM, it is worth exploring if some (most?) of the benefits of NDP can also be provided with conventional non-3D-stacked DRAM? This is an area that is relatively under studied and more research needs to be done. One example proposal by Pugsley et al. [27], *NDC-Module*, re-designs a DIMM by placing many simple processor chips on the DIMM and connecting them to their adjacent commodity DRAM chips. The key here is that in a conventional DIMM and server, the on-DIMM buses can offer very high bandwidth

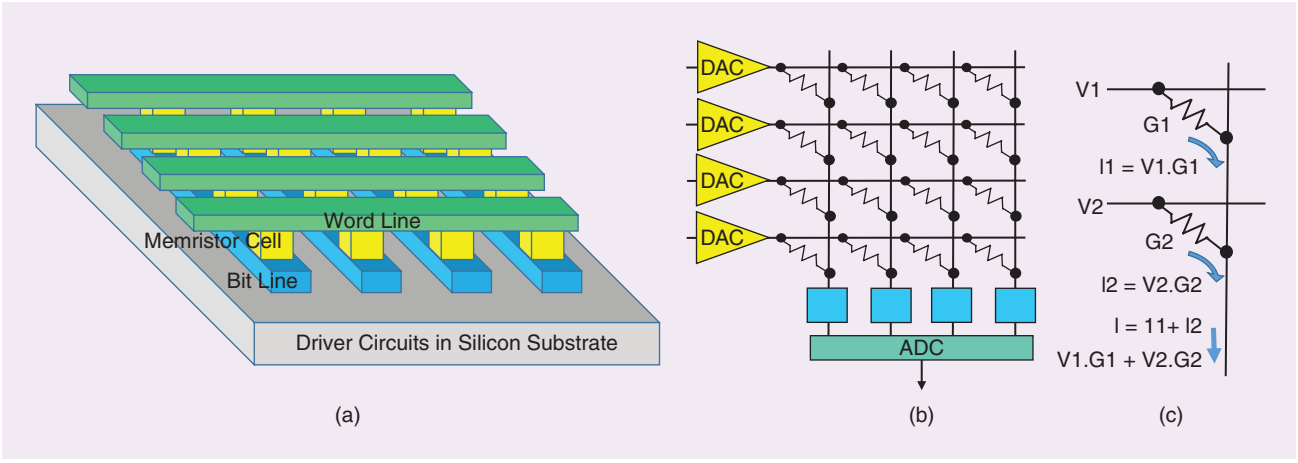

FIGURE 3: A memristive crossbar unit for analog dot product operations.

levels; but these buses (on multiple DIMMs) are eventually multiplexed on to a single shared DRAM channel that carries data into the processor socket. Thus, the high aggregate intra-DIMM bandwidth isn't entirely available to a conventional processor. Moving computational logic to the DIMM allows the system to leverage the high intra-DIMM bandwidth and linearly scale available bandwidth as more DIMMs are added to the system. In other words, 3D-stacked TSVs are not the only source of high bandwidth; on-DIMM buses are a great (and cheaper) alternative source. The NDC-module design of Pugsley et al. [27] is optimized for high-bandwidth memory access for a parallel workload, such as in-memory MapReduce. It is not a good fit for applications that cannot be easily parallelized.

Computation can be placed near DRAM memory [27], near NVM (PCM and memristors [36]), or near flash-based solid-state drives [7]. Note that the benefits of NDP are determined more by the ratio of intra- and interpackage data bandwidths, which can be high in most memory technologies.

What Programming Models Are Required?

This remains a key challenge in the widespread adoption of NDP. Not only does NDP require the programmer to grapple with the usual challenges of parallel/distributed programming (e.g., how is data partitioned across the compute units, what are the semantics/synchronization for dealing with shared data or serialized critical sections), but the programmer has to also identify the best location (i.e., host processor or near-data processor or near-data accelerator) for any computation. Consider the following example approaches to this challenge. The NDC proposal of Pugsley et al. [28] does not introduce a new programming model—it tries to leverage an existing infrastructure and developer base. That architecture is targeted at a class of workloads that can be handled by the intuitive MapReduce programming model.

The PEI proposal of Ahn et al. [2] requires the application developer or compiler to identify individual instructions that can be mapped to functional units on the memory device. The burden on the developer or compiler is low because the PEI hardware automatically decides where that instruction is best executed. The larger community realizes that we are moving in a direction where specific computations will be off-loaded to accelerators, whether they are near memory or not. The evolution of these programming models and standards will likely play out over the next five years. NDP introduces a significant wrinkle to this evolution because of how caches/memory are asymmetrically exposed to the various processors.

In-Situ Computing

I will end this section with a discussion of a unique in-memory accelerator, a design that leverages emerging memristive technology and a memory array organization that not only stores data but also performs operations on that data.

Figure 3(a) shows a memristive crossbar array, where cells are implemented as metal-oxide materials between overlapping word lines and bit lines on different metal layers. The array has no access transistors, and it can be represented logically as a grid of resistances, as shown in Figure 3(b). Each cell can be individually programmed by applying the appropriate combination of voltage pulses at the word lines and bit lines [45]. During a read operation, as depicted in Figure 3(c), a set of voltages V_1–V_N are applied to the N word lines. If the cell conductances of the first column are G_1–G_N, the current emerging from the first bit line can be represented as $V_1 \times G_1 + V_2 \times G_2 + \cdots + V_N \times G_N$, based on Kirchoff's law. In other words, the current in the first bit line is the dot product of the vector of input voltages and the vector of cell conductances in the first column. In parallel, each of the bit lines is now performing a dot product of

the same input voltage vector and its vector of cell conductances. The memristive crossbar array is therefore a powerful analog vector-matrix multiplier that leverages Kirchoff's law to perform a large number of parallel multiply-accumulate operations. Digital-to-analog conversion is required when providing input voltages, and similarly, analog-to-digital conversion (ADC) is required before the outputs can be buffered.

The previously given analog dot-product unit can be very useful in accelerating applications that involve dot products on large data sets. Machine-learning applications fit this bill. While noise and precision are important concerns in analog units, machine-learning applications are known to be tolerant to noise. An upcoming paper [36] shows how a mixed analog-digital architecture, ISAAC, can execute entire deep-learning algorithms at very high efficiency. In-situ computing targets the biggest bottlenecks in these algorithms—storage, access, and compute for many millions of parameters—with a compact and parallel crossbar unit. In spite of ADC overheads (which are significant), ISAAC is able to achieve efficiency gains of nearly 15× over a state-of-the-art digital accelerator for deep learning [6].

In general, any resistive-RAM cell is a good candidate for use in such an analog dot-product engine. HfOx-based memristors are an especially good fit because of their high on/off ratio. Since memristor cells exhibit a nonlinear I-V curve, they can also seamlessly apply sigmoid-like activation functions to the dot-product operation, as is done in machine learning algorithms. Another example of in-situ computing is the DRAM-based Automata Processor that can accelerate algorithms that rely on pattern-matching [42].

Auxiliary Features

The previous section discusses a variety of computational capabilities that can be moved to the memory device; these explicitly accelerate portions of the application itself. We now turn our attention to other useful functionalities that can be executed on the memory device; such features can improve overall system metrics in an application-agnostic manner. Many of the proposals in this section target phenomena (such as endurance, iterative writes, and new error types) that are more applicable to emerging NVMs than to DRAM.

Compression

Memory compression was introduced in commercial products over a decade ago, e.g., IBM MXT [41]. However, that idea didn't catch on for a variety of reasons.

- The latency of compression and decompression added a nontrivial penalty to memory access times.
- There is significant software/hardware complexity in managing compressed data: 1) it takes nontrivial logic to estimate the location of a compressed block, and 2) when a block is modified and the new compressed version turns out to be larger, several other blocks may have to be moved to make room for the larger block.
- Invasive changes are required to all parts of the operating system (OS) that deal with memory management.

The last few years have seen significant advancements that make compressed memory a very attractive choice today. First, new compression algorithms, such as Base Delta Immediate (BDI) compression [25], are able to achieve sufficiently large compression ratios for a sufficiently large set of applications, while consuming fewer than a handful of nanoseconds for compression and decompression.

Second, new compression architectures [24] have been developed that, in the common case, make it easier to locate a compressed block, and avoid shuffling data around when compressed blocks grow in size.

Third, recent papers [37], [32] have made the argument that it is the greed for higher memory capacity that increases the OS complexity from compression. Therefore, these papers have made the case that compression should be used for a variety of reasons but not to tightly pack more blocks into a limited space. So a 64-B storage area in memory is reserved for a single 64-B data block, even though the block may only consume a fraction of that storage after compression. This leaves the OS page management mechanisms unchanged. There are many other benefits of reading/writing smaller blocks [37]: lower energy, the ability to do more reads/writes in parallel, higher endurance for NVMs, the ability to add more metadata for error correction, etc. Sathish et al. [32] exploit some of these benefits for a GPU and GDDR5 memory, while Shafiee et al. [37] exploit them in the context of CPUs and DDR3. Their approaches place only one burden on the OS: managing compression metadata in a separate region of memory. Even this obstacle can be scaled; a recent paper [22] shows how error correction code (ECC) metadata can incorporate compression metadata.

With these new technologies in place, compression is a feature that no longer needs heavy involvement from the processor or the OS. It could be a feature that is entirely managed by logic on memory devices, either in 3D-stacked devices or on DIMMs, and completely oblivious to the rest of the system.

Memory Timing

Just as the management of compression can be off-loaded to the memory devices, other low-level memory management features can also be off-loaded to memory. For example, as DRAMs scale or as new NVMs come to market, new problems might emerge. Because of process variation, some regions of a die may be faster than others, and some regions may be more error prone than others. Similarly, operations that consume large amounts of current (e.g., writes in NVMs) will introduce static or dynamic IR-drop and pose severe worst-case constraints on DRAM timing parameters [38].

Rather than expose these so-called "blemishes" or conservative worst-case

parameters to the system, memory devices may choose to perform low-level timing management on their own [23]. The processor issues read or write commands to the memory device, and the device figures out the fastest sequence of commands that can service that data. If the manufacturing process is highly problematic, it leads to more complex management circuits and higher performance than a competitor that chooses to not provide smart timing management.

Reliability

In many domains, error-correction support within memory is a strong requirement. Modern error-correction solutions, especially those for chipkill [8], incur significant overheads for storage, energy, and in some cases performance. As DRAM technologies scale, errors may become more common. Instead of relying on the processor socket to implement a reliable solution over an unreliable DRAM platform, there is value in implementing a reliable DRAM platform in the first place so that the processor doesn't have to worry about error correction. The advantage in implementing an error correction solution in DRAM (either in a 3D-stacked package or on a logic chip on a DIMM or on the DRAM die itself) is as follows: 1) the processor need not be exposed to the bandwidth overhead of fetching ECC bits, 2) there is reduced data movement energy, and 3) error correction solutions can be designed that are tailored to the specific vulnerabilities of the vendor's manufacturing process.

Reliability will be an even bigger concern in NVMs because of wearout, drift, the iterative nature of writes, and other interference effects [5]. With internal mechanisms to handle these problems, memory vendors can make trade-off choices that would not otherwise have been palatable. For example, one would not design a weaker write process if it resulted in a high raw bit-error rate being exposed to the outside world. But with an in-built error correction process, a not-so-perfect write mechanism may yield an overall lower power consumption, while also yielding a low bit-error rate. Note that codes can also be constructed to reduce data transmission energy or reduce write energy/wearout in NVMs, as is done with data bus inversion in DDR4 or by a few other recent papers [39], [19].

Security

It is clear that security is a metric that will grow in prominence. A first step in protecting sensitive information is to encrypt data. There are good arguments for performing this encryption (and corresponding decryption) on the processor or on the memory. If one is paranoid about an attacker having physical access to the hardware and, in particular, the memory bus, then plain text information should never emerge out of the processor socket, and the encryption is best performed on the processor. By giving the processor control over encryption, the application can also lower overheads by only encrypting the most sensitive information. On the other hand, data encryption on the memory device makes sense if the threat model is that someone may steal a DIMM. In such cases, a more lazy encryption policy can be implemented. One can also implement an aggressive prefetch-and-decrypt policy on the memory device to reduce read latency; doing this on the processor would incur a heavy bandwidth penalty.

There exist other threat models that require more drastic security measures. If an attacker has physical access to hardware (e.g., a malicious cloud operator), we have already seen that data encryption is an important first step. Even with data encryption, the attacker can still engage in replay attacks [29] (returning the old encrypted value of data to disrupt the application) or gather sensitive information [13], [17] by snooping on the address trace (note that addresses are still sent out in plain text). To address these vulnerabilities, researchers have constructed solutions, Merkle trees [12] and oblivious RAM [40], respectively, that organize memory blocks into a logical tree structure and require fetching all the blocks in a given subtree. Both solutions incur a very high penalty in terms of bandwidth. Moving some of these solutions to the memory device, as was done by Gundu et al. [14], can lower the overheads by leveraging the high intra-package or intra-DIMM bandwidth.

Closing Thoughts

In summary, this article has made the argument that memory devices can be much more than "dumb" units that store rows of bits. They can be augmented to perform parts of the application or other auxiliary system-level features. This can directly impact overall system performance and energy. While this approach will no doubt increase the cost per bit for memory devices, it offers an opportunity for value addition in an era where customers will likely pay more for specialized systems.

Acknowledgments

This work was supported in part by IBM Research, Hewlett Packard Labs, and by National Science Foundation grants 1302663 and 1423583. This article draws heavily on the work of students in the Utah Arch lab (Seth Pugsley, Ali Shafiee, Akhila Gundu, Manjunath Shevgoor, Anirban Nag, Niladrish Chatterjee, Meysam Taassori, and Arjun Deb) and collaborators (Alper Buyuktosunoglu, Al Davis, Feifei Li, Naveen Muralimanohar, Vivek Srikumar, and Viji Srinivasan).

References

[1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proc. ACM/IEEE 42nd Annual Int. Symp. ISCA*, Portland, OR, 2015, pp. 105–117.

[2] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *Proc. ACM/IEEE 42nd Annual Int. Symp. ISCA*, Portland, OR, 2015, pp. 336–348.

[3] B. Akin, F. Franchetti, and J. Hoe, "Data reorganization in memory using 3D-stacked DRAM," in *Proc. ACM/IEEE 42nd Annual Int. Symp. ISCA*, Portland, OR, 2015, pp. 131–143.

[4] R. Balasubramonian, J. Chang, T. Manning, J. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-data processing: Insight

from a workshop at MICRO-46," *IEEE Micro.*, vol. 34, no. 4, pp. 36–42, Aug. 2014.
[5] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla, B. Rajendran, S. Raoux, and R. S. Shenoy. (2010). Phase change memory technology. [Online]. Available: http://arxiv.org/abs/1001.1164v1
[6] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and and O. Temam, "DaDianNao: A machine-learning supercomputer," in *Proc. MICRO-47*, Cambridge, U.K., 2014, pp. 609–622.
[7] A. De, M. Gokhale, R. Gupta, and S. Swanson, "Minerva: Accelerating data analysis in next-generation SSDs," in *Proc. IEEE 21st Annu. Int. Symp. Field-Programmable Custom Computing Machines*, Washington, DC, 2013, pp. 9–16.
[8] T. J. Dell, "A whitepaper on the benefits of chipkill-correct ECC for PC server main memory," IBM Microelectronics Division, Tech. Rep., Nov. 1997.
[9] Y. Eckert, N. Jayasena, and G. Loh, "Thermal feasibility of die–stacked processing in memory," in *Proc. 2nd Workshop Near Data Processing*, 2014.
[10] A. Farmahini-Farahani, J. Ahn, K. Morrow, and N. Kim, "NDA: Near–DRAM acceleration architecture leveraging commodity dram devices and standard memory modules," in *Proc. IEEE 21st Int. Symp. HPCA*, Burlingame, CA, 2015, pp. 283–295.
[11] Gartner. (2015). Gartner's 2015 hype cycle for emerging technologies. [Online]. Available: http://www.gartner.com/newsroom/id/3114217
[12] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas, "Caches and Merkle trees for efficient memory authentication," in *Proc. 9th International Symp. High Performance Computer Architecture*, Nov. 2003.
[13] O. Goldreich, "Towards a theory of software protection and simulation by oblivious RAMs," in *Proc. 19th Annu. ACM STOC*, New York, 1987, pp. 182–194.
[14] A. Gundu, A. Shafiee, M. Shevgoor, and R. Balasubramonian, "A case for near data security," in *Proc. 2nd Workshop Near Data Processing,* 2014.
[15] Q. Guo, T. Low, N. Alachiotis, B. Akin, L. Pileggi, J. Hoe, and F. Franchetti, "Enabling portable energy efficiency with memory accelerated library," in *Proc. MICRO-48*, New York, 2015, pp. 750–761.
[16] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. 5th ed. New York: Elsevier, 2011.
[17] M. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack, and mitigation," in *Proc. NDSS*, 2012.
[18] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance," in *Proc. Symp. VLSI Technology*, Honolulu, HI, 2012, pp. 87–88.
[19] R. Maddah, S. Seyedzadeh, and R. Melhem, "Cafo: Cost aware flip optimization for asymmetric memories," in *Proc. 21st Int. Symp. HPCA*, Burlingame, CA, 2015, pp. 320–330.
[20] N. Mirzadeh, O. Kocberber, B. Falsafi, and B. Grot, "Sort vs. Hash join revisited for near-memory execution," in *Proc. Workshop Architectures and Systems for Big Data,* 2015.
[21] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Che, C. -Y. Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O'Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S. Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam, and Z. Sura. "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM J. Res. Dev.*, vol. 59, no. 2/3, pp. 17:1–17:14, May 2014.
[22] D. Palframan, N. Kim, and M. Lipasti, "COP: To compress and protect main memory," in *Proc. ACM/IEEE 42nd Int. Symp. ISCA*, Portland, OR, 2015, pp. 682–693.
[23] T. Pawlowski, "The future of memory technology," in *Proc. Memory Forum*, 2014.
[24] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *Proc. MICRO-46*, New York, 2013, pp. 172–184.
[25] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *Proc. 21st Int. Conf. PACT*, New York, 2012, pp. 377–388.
[26] S. Pugsley, A. Deb, R. Balasubramonian, and F. Li, "Fixed-function hardware sorting accelerators for near data mapreduce execution," in *Proc. 33rd IEEE Int. Conf. ICCD*, New York, 2015, pp. 439–442.
[27] S. Pugsley, J. Jestes, R. Balasubramonian, V. Srinivasan, A. Buyukto-sunoglu, A. Davis, and F. Li, "Comparing different implementations of near data computing with in-memory mapreduce workloads," *IEEE Micro.*, vol. 34, no. 4, pp. 44–52, June 2014.
[28] S. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "NDC: Analyzing the impact of 3D-stacked memory+logic devices on mapreduce workloads," in *Proc. ISPASS*, 2014, pp. 190–200.
[29] B. Rogers, S. Chhabra, Y. Sohilin, and M. Prvulovic, "Using address independent seed encryption and bonsai merkle trees to make secure processors os and performance-friendly," in *Proc. 40th Annu. IEEE/ACM Int. Symp. MICRO*, Chicago, IL, 2007, pp. 183–196.
[30] SAP. (2015). In-memory computing: SAP HANA. [Online]. Available: http://scn.sap.com/community/hana-in-memory
[31] SAS. (2015). SAS in-memory analytics. [Online]. Available: http://www.sas.com/en_us/software/in-memory-analytics.html
[32] V. Sathish, M. J. Schulte, and N. S. Kim, "Lossless and lossy memory I/O link compression for improving performance of gpgpu work- loads," in *Proc. 21st Int. Conf. PACT*, New York, 2012, pp. 325–334.
[33] V. Seshadri, K. Hsieh, A. Boroumand, D. Lee, M. Kozuch, O. Mutlu, P. Gibbons, and T. Mowry, "Fast bulk bitwise AND and OR in DRAM," *IEEE Comput. Archit. Lett.*, vol. 14, no. 2, pp. 127–131, 2015.
[34] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. Gibbons, M. Kozuch, and T. Mowry, "Gather-scatter dram: In-DRAM address translation to improve the spatial locality of non-unit strided accesses," in *Proc. MICRO-48*, New York, 2015, pp. 260–280.
[35] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhi-menko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization," in *Proc. MICRO-46*, New York, 2013, pp. 185–197.
[36] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. Strachan, M. Hu, R. Williams, and V. Srikumar, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proc. ISCA*, 2016.
[37] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis, "MemZip: Exploiting unconventional benefits from memory compression," in *Proc. HPCA*, 2014, pp. 638–649.
[38] M. Shevgoor, J.-S. Kim, N. Chatterjee, R. Balasubramonian, A. Davis, and A. Udipi, "Quantifying the relationship between the power delivery network and architectural policies in a 3d-stacked memory device," in *Proc. MICRO-46*, New York, 2013, pp. 198–209.
[39] Y. Song and E. Ipek, "More is less: Improving the energy efficiency of data movement via opportunistic use of sparse codes," in *Proc. MICRO-48*, New York, 2015, pp. 242–254.
[40] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An extremely simple oblivious RAM protocol," in *Proc. CCS*, New York, 2013, pp. 299–310.
[41] R. Tremaine, P. Franaszek, J. Robinson, C. Schulz, T. Smith, M. Wazlowski, and P. Bland, "IBM memory expansion technology (MXT)," *IBM J. Res. Dev.*, vol. 45, no. 2, pp. 287–301, 2001.
[42] K. Wang, M. Stan, and K. Skadron, "Association rule mining with the micron automata processor," in *Proc. IEEE Int. IPDPS*, Hyderabad, India, 2015, pp. 689–699.
[43] L. Wu, R. Barker, M. Kim, and K. Ross, "Navigating big data with high-throughput energy-efficient data partitioning," in *Proc. 40th Annu. ISCA*, New York, 2013, pp. 249–260.
[44] L. Wu, A. Lottarini, T. Paine, M. Kim, and K. Ross, "Q100: The architecture and design of a database processing unit," in *Proc. 19th Int. Conf. ASPLOS*, New York, 2014, pp. 255–268.
[45] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *Proc. 21st Int. Symp. HPCA*, Burlingame, CA, 2015, pp. 476–488.
[46] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. NSDI*, Berkeley, CA, 2012. p. 2.

About the Author

Rajeev Balasubramonian is a professor in the School of Computing at the University of Utah. His research focuses on memory systems and interconnects. He has a B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Bombay, and M.S. and Ph.D. degrees, both in computer science, from the University of Rochester. He is a Member of the IEEE.