Challenges of Programming a System with Heterogeneous Memories and Heterogeneous Processors

– A Programmer's View

Shuai Che, Arkaprava Basu, and Jonathan Gallmeier
{Shuai.Che, Arkaprava.Basu, Jonathan.Gallmeier}@amd.com
Advanced Micro Devices, Inc.

ABSTRACT

Recently there has been significant development and innovation in both frontiers of *Heterogeneous Memory* and *Heterogeneous Compute* domains. This paper summarizes the challenges, surveys related work, and proposes possible research directions to exploit both heterogeneous memory and compute resources in a computer system. We focus our discussion on the memory system and also touch issues related to heterogeneous compute.

CCS Concepts

•Hardware → Semiconductor memory; •Software and its engineering → Memory management; •Theory of computation → Parallel algorithms; •Computing methodologies → Parallel programming languages;

Keywords

Heterogeneous processor; heterogeneous memory; GPGPU; external memory algorithm

1. SYSTEM ARCHITECTURE

Heterogeneous Memory Hierarchy: Several new memory technologies such as die-stacked DRAM [15] and non-volatile memories (NVMs) [38] provide high bandwidth, new read/write latency characteristics, non-volatility and other new memory features. In the near future, programmers may face the challenge of programming a rather complex system consisting of diverse heterogeneous memory components, each of which presents a unique organization and characteristics. The concept of heterogeneous memory architectures has been a hot research topic [24]. The system architecture will influence how the software stack evolves and how applications are written.

Figure 1 a) shows a simple view on how such a system may look like from a hardware point of view. Figure 1 b) shows the components that may be involved in the software stack. In addition, conceptually we can deem such a system in an abstract view shown in Figure 1 c). It is easy to realize that the memory hierarchy design presents many different design alternatives. For example, one level of memory can be a backing store for another level, used to cache data for another level, or be in parallel with another providing NUMA accesses. Furthermore, each type of processor (e.g., CPU and GPU) may have their own memory subsystem (also processors in memory (PIM) and active storage). At a high level, the memory system has become increasingly complicated (e.g., a tree structure [13]). Importantly, different memory subtrees may present heterogeneity and different characteristics. No wonder that managing memory mapping and data movement in such a system is not trivial. A simple system abstraction and programming model is critically needed for programmers to program such systems.

Hardware vs. Software-managed memory: As a first requirement, a clear definition of the roles of hardware vs. software for memory management is necessary. For software management, it can be performed by the operating system, compilers, user-level libraries, application programs or their combinations. Given n levels of memories, a typical organization is there exists one or more "transitioning" levels as a separation point i, such that the $0 \rightarrow i$ levels are managed by hardware while the $(i+1) \rightarrow n-1$ levels are managed by software. In other words, this defines the main division point for *in-core* or *out-of-core* memory management. Which memory levels are treated as transitioning points has important implication on how programmers view the whole system, which in turn affects the development of software stack and what hardware features are exposed to software. The recent development in GPGPU and embedded systems has already reflected a trend that more memories are designed to be software-managed. In fact, at the same memory level, both software and hardware-managed memories can be provided (e.g., cache vs. scratchpad in GPU's L1). The goal is to provide programmers more flexible control and achieve high performance. Given increasing number of memory levels, we envision that programmers will take more responsibility to deal with explicit data movement in those levels managed by software.

There are different proposals treating new memories in different ways. For example, stacked DRAM can be utilized as a high-bandwidth, hardware-managed last-level cache, bridging the gap between SRAM-based on-chip cache and slower DRAM [22]. Alternatively, it can a part of the physical memory space in parallel to DRAM and managed by the OS virtual memory system. Advanced hot page detection and page migration can be integrated to leverage the stacked DRAM more efficiently [24]. This is also true for NVM [38]. In addition, the way how the memory is exposed can be different. A design can treat NVM as part of physical address space (i.e., accessible through a load/store interface) [9,36] or as fast storage [1]. In addition, a system can be dynamically configured to different designs to better meet diverse application needs. Of course, a solution depends on different requirements and the compatibility to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MEMSYS '16, October 03-06, 2016, Alexandria, VA, USA

© 2016 ACM. ISBN 978-1-4503-4305-3/16/10…$15.00

DOI: http://dx.doi.org/10.1145/2989081.2989097

legacy software, however there is no general guideline for system configurations for different scenarios. This makes application development a challenge because it is highly likely that applications assuming one model are not portable with another model. Though works have been done to standardize the model [3] for specific areas – a holistic design approach is needed by considering the whole system architecture.

Memory Space Abstraction: Software development in shared virtual memory is the easiest for programmers. Traditionally, programmers benefit from the virtual memory abstraction provided by OS. If we use the existing model for the new memories, it is unclear if a single, flat address space is a good abstraction for certain applications, because it does not represent the underlying memory topology and individual memory properties. Ideally the underlying software or hardware can intelligently detect and determine desirable mappings of data to different physical memories. However, it is unclear if an effective solution exists. The best approach to abstract memory spaces remains an open question. It may be useful that separate address spaces or virtual memory regions – along with hints for memory mapping (e.g., fast/slow, memory level or persistency [9])– are provided to programmers in order to control memory allocation and data placement (e.g., *pmalloc* [36]). Memory copy operations are also necessary to support data movement across different spaces or regions. This also requires innovation in API and design of programming languages (e.g., types).

Heterogeneous Compute: GPUs, FPGAs and other coprocessors offer alternatives to traditional CPUs and better compute efficiency for a variety of applications. However, to fully utilize them requires significant programming efforts especially in the presence of heterogeneous memories. In addition to the challenge of expressing concurrency, a critical issue is that different processors may not "see" the memory system in a uniform way. For example, a GPU may only be able to access its own device memory region (or can be part of the physical space) in certain systems. There has been some efforts to improve this issue, however additional work remains to be done. For example, recent development of HSA [14] allows the CPU and the GPU to share the same virtual address space and supports the "a-pointer-is-a-pointer" semantic. In addition, some initial work has also been proposed to allow GPU threads to directly access files and I/Os [31]. Future programming environments should incorporate these or similar features for better programmability.

2. APPLICATION AND ALGORITHM DEVELOPMENT

It is difficult to write applications for a system where the underlying system still has many undefined variables. Consequently, application development has been slow for such heterogeneous systems. This adversely limits the pace of architectural studies since there are no well-developed benchmarks to drive the design. For example, simulations usually use existing benchmarks with simple modifications or generated memory traces. Applications which directly take advantage of emerging memories are difficult to find. Existing CPU and GPU application development tends to be mature but they may need to be entirely rewritten to exploit the memory hierarchy and new memory features (e.g., non-volatility). Though prior works [8, 22, 24] leverage new memories without application changes, algorithmic innovation is needed to fully exploit the system.

External Memory Algorithms for New Memories: To allow applications to exploit the complex memory hierarchy and also the parallelism of heterogeneous processing elements, one option is to start by revisiting the development of *external memory* (or out-of-core) algorithms and adapt them to the new architecture with necessary innovation in algorithms, data structures, and other optimizations. For example, external memory algorithms are traditionally used to deal with large dataset problems which cannot fit into the main memory. The dataset is broken into chunks (e.g., in storage such as SSD and disk) and the algorithm defines the most efficient way to divide and move chunks across the main memory and the disk to compute a problem. In the past, the cache vs. memory relationship is different enough than the memory vs. disk relationship (e.g., bandwidth ratio), thus some external memory algorithms may not exploit cache efficiently [33]. However, today new memory technologies help bridge the gaps in latency and bandwidth, which may make these algorithms more promising. Characterizing these algorithms will be useful in understanding and designing better automatic mechanisms to improve the performance of existing applications without any code change. However, these algorithms need redesign and extensions to exploit a heterogeneous, more complex memory hierarchy. Since they originally assume a parallel disk model (PDM) [34, 35], research needs to take the asymmetric relationship of different memories (e.g., capacity, latency) into account. Explicit out-of-core data management can be very efficient. For example, recent work [20] on irregular graph algorithms with SSD shows that a workstation can approach the performance of a small machine cluster. We envision that future multi-node systems may integrate the advancement of both external and distributed memory algorithms [17]. Additionally, inside each chunk, data parallelism can be optimized by the heterogeneous processing elements. Also, there are designs which use NVMs in each node of a cluster to increase the memory capacity for compute purposes, besides use them for burst buffers and checkpointing [18].

Access Patterns: An important goal of application development is to minimize data movement and improve effective memory bandwidth utilization for better performance and energy efficiency. This requires a better understanding of memory access patterns especially in a larger granularity (e.g., in chunk or page). Many algorithms do not access data in a sequential, streaming fashion. They read and write multiple chunks which are scattered throughout different memory levels or locations in the same level. Simple "demand paging" like solution is not sufficient. Without an explicit memory control interface or smart paging mechanisms, they may not be able to take full advantage of the memory hierarchy. Previous work has studied various basic algorithms to improve I/O efficiency. These problems present diverse access patterns, ranging from: 1) the simplest scenario where all reads/writes happen locally within a chunk, through 2) a more complicated scenario where data communications happen across a subset of chunks (e.g., sorting, grid computation, linear algebra algorithms, and fft [21, 33]), to 3) the most challenging scenario where memory accesses are random [7,20,29]. Because some problem is data dependent [5,20,21], the algorithm needs to be carefully designed to determine what data to load to lower-level memories (closer to cores) in order to reduce the overall communication cost. As an example, this may be achieved through prediction (e.g., sampling in sorting [21]) or novel data structures and scheduling mechanisms (*memory-shard* and *sliding window* for GraphChi [20]). Furthermore, one may prematurely think an algorithm designed for distributed memory systems will naturally map well to multi-level memory system. However this is not true, because certain algorithms that are efficient on distributed memory parallel systems have no efficient multi-level memory schedules [33].

Library: To improve programming productivity, library development for useful software building blocks and basic algorithms is

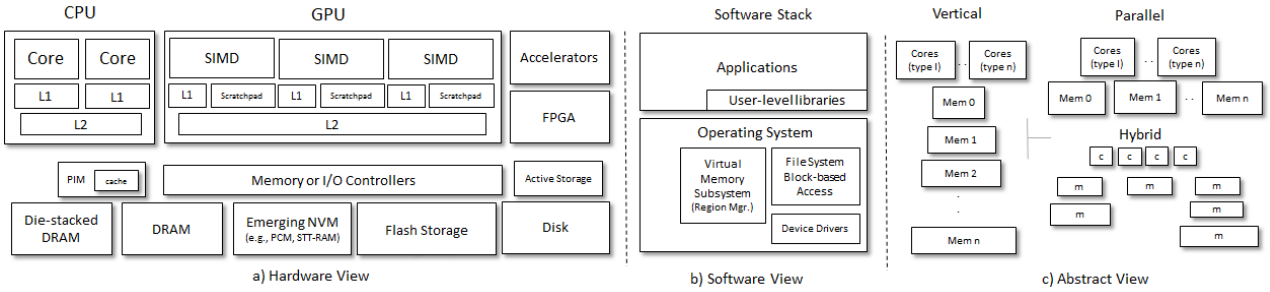

Figure 1: An architecture consisting of heterogeneous memories and processors

needed. Due to previously mentioned challenges, libraries which can transparently leverage both heterogeneous memories and heterogeneous processors can alleviate programmers from dealing with architectural details. For example, important structures such as stacks, queues, priority queues, maps and their associated operations (e.g., those in C++ STL) can be developed and extended for a system similar to the one in Figure 1. In the meantime, important parallel primitives (e.g., reduction, sort, scan) processing very big data sets can be accelerated with GPUs and other accelerators, by taking advantage of the new memories. For example, data can be divided to small chunks , each mapped to a GPU workgroup/thread block.

3. RUNTIME, DATA LAYOUTS, AND OPERATIONS

Scheduling and Load Balancing: An important issue is how to schedule and place chunks in the memory hierarchy in an order to maximize locality and reuse, and limit redundant data movement. For example, to compute the result of a chunk in some applications (e.g., grid, FFT), data from several chunks are needed as input, therefore these chunks are ideally scheduled to be loaded adjacently to a memory level; similarly, each chunk is further decomposed into smaller chunks and loaded into another level closer to cores. Apparently data chunks present interdependencies among each other and the scheduling needs to follow the natural data flow. In addition, from the compute perspective, there is another dimension of task dependency [30]. Both control and data dependencies should be considered together to decide an optimized scheduling order by analyzing both the task and data flow graphs. In addition, a desirable schedule will allow an asynchronous way of execution by overlapping computation and data movement. Because both cores and memories present heterogeneity (e.g., compute throughputs, NUMA). Dynamic load balancing mechanisms are necessary and can be integrated to the runtime framework for problems where application behavior is difficult to predict (e.g., by static scheduling).

Data Layout: For the same application, execution on different architectures may favor different memory layouts and access patterns in a heterogeneous environment. SIMD organizations generally perform best when each thread or lane of a SIMD operation accesses adjacent data (e.g., memory coalescing), while scalar organizations (CPU) perform best when a single thread accesses adjacent data. Therefore, some application may prefer a row-major layout on the CPU while a col-major layout is preferred on the GPU [6]. For sparse matrix problems, the choice of data layouts (e.g., CSR, ELL) not only depends on architectures but also program inputs [4]. Furthermore, some applications may switch access patterns in different execution phases, which further complicates the issue [23]. One can imagine when data migrates across memory levels, chunks can be transformed and stored in different formats. In fact, for applications with sufficient data reuse, maintaining diverse layouts is beneficial [6, 32]. However, this requires the runtime system to reduce the transformation overhead (e.g., latency and shadow memory capacity) with optimized buffering and data reorganization operations [2]. It may also be useful that layout control is exposed to the programming API.

Useful Operations: Several operations may help to enable efficient data movement. These include memory-to-memory and DMA operations which bypass original round trips to CPU cores. For example, special *memcpy* can be developed to enable data transfers between different memory modules or levels. This can be accomplished with optimized network transmission and routing for fast data delivery. Memory operations which reorganize data can use the help from PIM [16]. Finally, coarser-grained prefetching is useful for many applications in such systems. For example, the active data chunk can reside in lower-level memories. When it predicts a next chunk is going to be used soon (a simple example is stack and queue operations), prefetching is more efficient to execute at the data chunk and page granularity.

4. PROGRAMMING ENVIRONMENT

Extension to Existing Programming Models: The ideas of previous works [9–12, 25–28, 36] can be used to facilitate the programming environment development for such systems. The capability to explicitly manage parallelism (e.g., fine-grained parallelism in OpenCL™/CUDA) and data movement (for out-of-core memories) is needed in an integrated programming environment. Alternatively it may be possible a high-level language and API (e.g., OpenMP) are extended with hints so that the compiler can emit code with calls to low-level libraries and transform the program to exploit the memory hierarchy with explicit data transfers. For example, Colvin and Cormen [10] design an API which allows programmers to define shapes in traditional disks as *outofcore*. And the compiler translates the code into both in-core and out-of-core codes respectively. As another example, new directives are proposed to map data structures to high-bandwidth memories [19]. In either case, a new programming model may require the feature to identify both the underlying memory topology and properties of individual memories (e.g., volatility, size). Recently, we have seen some trend moving towards this direction. HSA [14] includes the concept of memory scopes in the memory model. Memory scopes are used to limit the visibility of a memory operation to specific hardware processing elements, and support various memory-ordering options (e.g., sequential, acquire-release, and relaxed consistencies). This model can be extended to use in our system. Also, the capability to allow programmers to leverage the knowledge of numerical algo-

rithms to direct data placement and movement will be useful [37]. Previous works also propose programming styles [9, 36] for NVM and include the support of basic primitives, data structures, and pointer management to ensure persistency with transactions. Research needs to be done to incorporate these concepts into a unified programming model.

Programming Abstraction and Recursive Execution: Given a tree-like memory hierarchy (Figure 1 c), new abstractions to program memories are needed. However, most existing models describe the distribution and horizontal communication of data among nodes (symmetric) of a parallel machine. We propose that a possible solution could be to use a similar concept to the Sequoia-type model [13] and design it to support a heterogeneous memory hierarchy and heterogeneous processors. For example, programmers can specify the working sets (e.g. dimensions and sizes of data chunks with an API) mapped to different memory levels (e.g., SSD, NVM, stacked DRAM, cache, etc.), respectively. The data can be *recursively* decomposed, scheduled and moved towards lower-level memories across the memory hierarchy. The computation tasks are mapped to leaf chunks (the smallest decomposition) of the recursion tree. The task can be written and distributed to different processor types. To achieve this, the topology structure of the memory hierarchy can be maintained, in order for the scheduler to distribute tasks to each subtree in an efficient way. This can be in the form of memory tree with each node representing a memory node (with its characteristics) and each tree level representing a memory level. This model is also helpful for enabling load balancing, since each tree node can keep track of the data (e.g., capacity usage) and ongoing tasks belonging to its owned subtree. The information can be used for enabling dynamic load balancing.

5. CONCLUSION

This paper summarizes some issues which are important to make it easy to program a system with heterogeneous memories and heterogeneous processors. Our discussion covers several aspects including the system architecture, roles of hardware vs. software, application and algorithm development, and programming model and runtime. This is by no means a complete list and some points are even debatable. However, it is important that the issues are clearly defined and then resolved to advance the research in improving heterogeneous systems.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

6. REFERENCES

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *Proceedings of USENIX 2008 Annual Technical Conference*, June 2008.

[2] B. Akin, F. Franchetti, and J. C. Hoe. Data reorganization in memory using 3D-stacked DRAM. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, June 2015.

[3] Storage Networking Industry Association. NVM programming model. Technical report, 2013.

[4] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec 2008.

[5] M. Bender, G. Brodal, R. Fagerberg, R. Jacob, and E. Vicari. Optimal sparse matrix dense vector multiplication in the i/o-model. *Theory of Computing Systems*, 47(4):934–962, 2010.

[6] S. Che, J. W. Sheaffer, and K. Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2011.

[7] Y-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, Jan 1995.

[8] C. Chou, A. Jaleel, and M. K. Qureshi. Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2014.

[9] J. Coburn, A. Caulfield, L. Grupp A. Akel, R. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar 2011.

[10] T. H. Cormen and A. Colvin. ViC*: A preprocessor for virtual-memory C*, 1994. http://www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail\&id=oai%3Ancstrlh%3Adartmouthcs%3Ancstrl.dartmouthcs%2F%2FPCS-TR94-243.

[11] T. H. Cormen and E. R. Davidson. Fg: A framework generator for hiding latency in parallel programs running on clusters. In *Proceedings of the 17th International Conference on Parallel and Distributed Computing Systems*, Jan 2004.

[12] J. Dean and Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of ACM*, 51(1):107–113, 2008.

[13] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, Nov 2006.

[14] Heterogeneous System Architecture (HSA). Web resource. http://hsafoundation.com/.

[15] High Bandwidth Memory (HBM). Web resource. http://www.amd.com/en-us/innovations/software-technologies/hbm.

[16] N. Jayasena, D. Zhang, A. F. Farahani, and M. Ignatowski. Realizing the full potential of heterogeneity through processing in memory. In *the 3rd Workshop on Near-Data Processing*, Dec 2015.

[17] M. Jung, E. H. Wilson, W. Choi, J. Shalf, H. M. Aktulga, C. Yang, E. Saule, U. V. Catalyurek, and M. Kandemir. Exploring the future of out-of-core computing with compute-local non-volatile memory. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov 2013.

[18] S. Kannan, A. Gavrilovska, and K. Schwan. Optimizing checkpoints using nvm as virtual memory. In *Proceedings of*

International Parallel and Distributed Processing Symposium, May 2013.

[19] L. Kaplan. Managing the memory hierarchy. In *the Twentieth Anniversary Meeting of the SOS Workshop*, Mar 2016.

[20] A. Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, Oct 2012.

[21] X. Li, P. Lu, J. Schaeffer, J. Shillington, and P. S. Wong. On the versatility of parallel sorting by regular sampling. *Journal Parallel Computing archive*, 19(10):1079–1103, 1993.

[22] G. H. Loh and M. D. Hill. Supporting very large caches with conventional block sizes. In *Proceedings of International Symposium on Microarchitecture*, Dec 2011.

[23] Z. Majo and Thomas R. Gross. Matching memory access patterns and data placement for numa systems. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, April 2012.

[24] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh. Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In *Proceedings of International Symposium on High Performance Computer Architecture*, Feb 2015.

[25] MPI I/O. Web resource. http://beige.ucs.indiana.edu/I590/node86.html.

[26] NVIDIA CUDA. Web resource. http://www.nvidia.com/object/cuda_home_new.html.

[27] OpenCL. Web resource. http://www.khronos.org/opencl/.

[28] OpenMP. Web resource. www.openmp.org.

[29] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov 2010.

[30] S. Puthoor, A. Aji, S. Che, M. Daga, W. Wu, B. M. Beckmann, and G. Rodgers. Implementing directed acyclic graphs with the heterogeneous system architecture. In *GPGPU-9*, Mar 2016.

[31] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. Gpufs: Integrating a file system with gpus. *ACM Trans. Comput. Syst.*, 32(1), 2014.

[32] I-J Sung, J. A. Stratton, and W-M W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *PACT*, Sept 2010.

[33] Sivan Toledo. External memory algorithms. chapter A Survey of Out-of-core Algorithms in Numerical Linear Algebra. 1999.

[34] J. S. Vitter and E. A. M Shriver. Algorithms for parallel memory ii: Hierarchical multilevel memories. Technical report, Durham, NC, USA, 1993.

[35] J. S. Vitter and E. A. Shriver. Algorithms for parallel memory i: Two-level memories. Technical report, Providence, RI, USA, 1992.

[36] H. Volos, A. J. Tack, and M.l M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar 2011.

[37] P. W, D. L, Z. Chen, J. S. Vetter, and S. Mittal. Algorithm-directed data placement in explicitly managed non-volatile memory. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, May 2016.

[38] Y. Xie. Modeling, architecture, and applications for emerging memory technologies. *IEEE Design Test of Computers*, 28(1):44–51, 2011.