This paper appears in the 2017 IGSC Invited Papers on Emerging Topics in Sustainable Memories

Power and Energy Implications of Misunderstanding DRAM

Erik Brunvand
University of Utah
elb@cs.utah.edu

Daniel Kopta
University of Utah
dkopta@cs.utah.edu

***Abstract*—When optimizing for energy in a modern computing system, it is critical to understand the primary source of energy usage: the memory system. Performing effective optimization in a traditional memory system requires knowing something about the complex and subtle behavior of dynamic random access memory (DRAM). This includes understanding DRAM chip organization and functionality, the organization of chips and data on a dual in-line memory module (DIMM), the structure of modern packaging options, and the behavior of the memory controller.**

In this position paper we describe some background of DRAM chip and system organization with some specific examples of how this knowledge can be used to enhance system behavior. We then give some examples of how understanding accurate DRAM behavior can influence energy and latency, and describe a detailed DRAM simulator (USIMM) that can be used to add high-fidelity DRAM models to system simulations. We use graphics hardware as a motivating example of a system that is both heavily reliant on the memory system, and that also has interesting latitude in terms of how the application accesses memory.

I. Introduction

For memory-bound applications where data are often not cache-resident, optimizing for main-memory DRAM access is critical for improving both speed and power performance. However, the details of how DRAM memory works, and the quirks associated with the low-level details of the DRAM chips, are often not well understood by application programmers. In this position paper we will expose some of the secrets of DRAM functionality by describing the internal structure of modern DRAM and how programmer knowledge of that structure can have a dramatic impact on application performance both in improved speed, and in decreased power consumption.

The dramatic difference in speed and power of a DRAM access vs. an on-chip memory access (e.g. cache) means that even small changes in main-memory traffic and behavior can have a large impact on system performance. For example, many studies attribute 25-40% of total power consumed in a datacenter to the DRAM system (e.g. [1], [2]). To give just one example, a proposed hardware architecture for ray tracing with cache hit rate percentages in the 90% range still shows almost 60% of the power consumed in the DRAM [3]. However, optimizing for main memory performance requires knowing something about the complex and subtle behavior of DRAM memory. This includes understanding DRAM chip organization and functionality, the organization of chips and data on a DIMM, the structure of cutting-edge advanced packaging options, and the behavior of the memory controller.

In this paper we describe some background of DRAM chip and system organization. We then motivate, with some examples, how much difference DRAM optimization can make, and describe a detailed DRAM simulator (USIMM) that can be used to add high-fidelity DRAM models to system simulations [4], [5]. We conclude with some thoughts on the importance of including high-fidelity DRAM simulation in any architectural exploration.

II. Background

A typical memory system for a modern general purpose computer consists of a variety of storage devices that give the illusion of uniform very large, very fast memory to the programmer or compiler. This consists of a hierarchy of different types of memory that have more capacity, but that become slower as the circuits become further away from the functional units in the CPU. This type of general memory hierarchy is well-known to computer architects (e.g. [6], [7]) but we give a brief overview for completeness.

Directly connected to the CPU functional units are the registers. From the point of view of the assembly code, the registers are typically seen as loaded directly from the main memory. In practice, there is a rich hierarchy of different memory types between the registers and the long term storage of the computing system. Memory accesses involve automatic fetching of required data from one level of the hierarchy to another on demand by the memory system hardware.

Closest to the registers is typically a cache memory. The cache is implemented using static random access memory (SRAM) circuits that are denser than the register file flip flops, but still constructed of active feedback circuits so that as long as the power is applied, the data remains in the memory. Caches exploit both temporal and spatial coherence in data access by caching copies of data from higher levels of memory into their smaller, but faster circuits. Register data items are served from the cache. If the data are not found in the cache they will be retrieved from the next level of the hierarchy automatically by the memory system, possibly displacing existing data in the cache. In a modern system there may be multiple levels of caching, each level consisting of larger capacity, but slower access times. Caches are always

978-1-5386-3470-7/17/$31.00

implemented using relatively fast SRAM and are typically integrated onto the same die as the processor.

Caches are backed up by "main memory." This memory is usually off-chip from the CPU and implemented using dynamic random access memory (DRAM) chips. DRAM is used only because of its high capacity and low cost. As will be described in the following section, DRAM is dynamic in two senses: it loses the data stored in a short amount of time (which requires periodic refreshing of the data) and even reads to DRAM are destructive to the contents of the memory (which requires write-back on every read). DRAMs have evolved to be optimized for refilling cache lines. This means that the access protocols are optimized for burst reads of contiguous data that matches a typical cache line refill request (e.g. 64B for many X86-based CPUs).

DRAM can be seen both as holding working-set data for the application, and also as a page-cache for operating system data pages stored on more permanent, even higher-capacity devices such as disks and solid-state memory. In the same way that DRAM access is optimized for cache-line-sized streams, disks and disk-equivalents are optimized for streaming page-sized chunks of data (typically 4kB on a standard linux OS).

While the memory system is typically designed to be transparent to the programmer (or compiler), it is well-known that considering data layouts and algorithms to match the properties of the memory system is a powerful optimization technique [8]–[10]. It is exactly these sorts of high-performance optimizations that we consider here, and argue that if a full system simulation is being used to explore a new architecture, that without high-fidelity simulation of the DRAM portion of the system, the accuracy of the simulation results is highly questionable.

III. DRAM Organization and Behavior

DRAM is the memory of choice for most main memory systems. This is primarily because of the high capacity of modern DRAM chips, and the low cost per bit. These are really the only redeeming features of DRAM. Although commodity DRAM chips have evolved to meet well-understood standards [11], they are subtle and complex to use, requiring sophisticated memory controller circuits to deal with them both at electrical and logical interfaces [6], [12]. They are also designed primarily to refill cache lines - a feature that makes them harder to use as general-purpose memory, but that also allows certain features of that behavior be exploited for better performance.

A. Circuit Organization

The fundamental bit-holding circuit of a DRAM is a storage capacitor. Charge is deposited or drained from the capacitor to indicate a 0 or 1 bit. Access to the capacitor is through a write transistor. These capacitors and associated transistors are typically implemented in a rectangular array on the chip (Figure 1). To increase bit density, these capacitors are sized as small as possible on the chip. This means that the stored charge leaks into the substrate and all data must be refreshed periodically to avoid losing information (typically every 32-64ms for recent DRAM chips [11]).

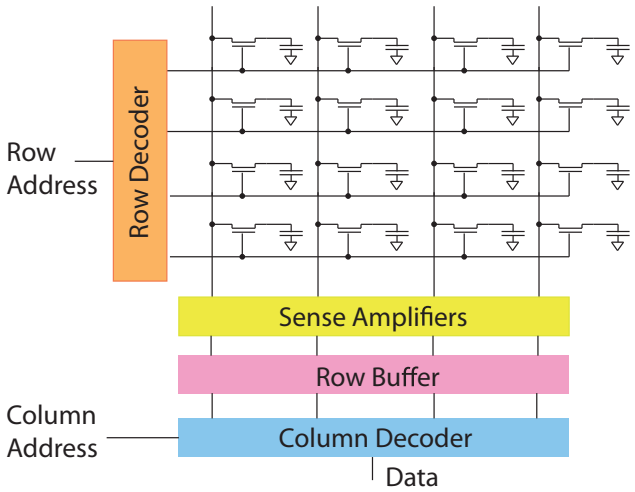

Fig. 1. Simplified schematic/block diagram of a DRAM on-chip memory array.

To amortize the decoding circuitry and the pins on the DRAM package, access is always done on entire rows at a time. For a read, the row is selected and the bit values are determined by the sense amplifiers. These amplifiers are needed because the value of the bit is read by coupling the storage capacitor to the vertical bit column resulting in a very small change in voltage on that column.

The sensed bits are stored in an SRAM *row buffer* so that the desired bits in the row can be selected from that buffer using the column address. The row buffer is also critical because the process of coupling the bit capacitors to the bit columns destroys the contents of that bit cell. The entire row must be written back after each read to restore the data in that row. This is also how refreshes are typically done - by making sure that every row in the DRAM array has been read at least once during the refresh interval. Writes to DRAM involve reading an entire row, followed by updating desired bits in the row buffer before writing the entire row back to the array.

B. Chip Organization

On a DRAM chip there may be many DRAM arrays, defined as *banks* of memory on the chip. This is required because of physical limits to the size of the DRAM bit arrays and the sensitivity of the amplifiers used to determine bit values. As an example, Figure 2 is taken from the data sheet of a DRAM chip sold by Micron [12]. This chip is not high-capacity by modern standards, but makes it easier to see the on-chip features in the block diagram. This example chip uses the double data rate 3 (DDR3) JEDEC standard [11] and contains 1Gb of total storage. That 1Gb is organized as 128M$\times$8b - the external data interface to the chip is eight bits wide.

As seen in the block diagram, there are eight banks of DRAM on the chip, each with 128Mb capacity. These banks are implemented as 64 copies of a 16k$\times$128 basic memory

array (called a "matrix" or "mat"). This means that for each row access to a bank, 128×64 = 8kb of data are sensed and transferred to that bank's row buffer. From that bank's row buffer, 64b are transferred to the chip's output FIFO for delivery to the external pins of the chip 8 bits at a time. A typical burst read will consist of eight transfers per burst, transferring all 64b from the output FIFO to the chip pins in four clock cycles (double data rate chips transfer data on both rising and falling edges of the clock). Even this description hides a great deal of the complexity of DRAM chip access. A modern DRAM chip has around 50 different timing parameters that must be considered when accessing the chip [12].

From the point of view of memory optimization, the row buffer can be a critical feature. Reading from the banks into the row buffer is the slowest operation on the DRAM, requiring a precharge of the bit columns, applying the row address, sensing the data, capturing the data into the row buffer, applying the column address, and transferring the selected bits from the row buffer to the chip outputs (with the writeback to the bank row happening concurrently). If the next data access is to data already in the row buffer (a so-called open-row access), then latency and energy are dramatically reduced for that access. It is also worth noting that this example has eight banks on the chip each with an 8kb row buffer, so there are eight separate row buffers to be managed for possible open-row access optimization.

C. DIMM Organization

DRAM chips in most computer memory systems are not used alone - they are assembled onto small circuit boards in a package known as a dual inline memory module (DIMM) to increase capacity (Figure 3). Using our previous chip example in Figure 2, if eight of these chips are used on a DIMM, and the memory access is spread across all eight chips, then the DIMM delivers 8×8b = 64b of data on each access. Since the chips are designed for burst access, a burst of eight accesses will result in a cache-line-sized chunk of 512b of data being transferred from DRAM to the cache.

From the point of view of a DIMM memory access, the bank of memory delivering the data is spread across all the chips on the DIMM. Using these example DRAM chips there would be eight banks implemented on the DIMM. The memory controller keeps track of the status of all the banks so that it can attempt to optimize access to those banks (using read and write coelescing through data queues residing in the memory controller, for example). The primary optimization is for open-row access, although there are other more subtle optimizations also possible.

Note that although Figure 3 shows all eight chips being used in each logical bank on the DIMM, it is also possible to use DRAM chips that, for example, have wider data interfaces and use fewer chips for each bank access. In this case, the additional chips could constitute another partitioning of memory, called a rank. Using chips with a 16b interface, for example, and eight chips on a DIMM, there could be two ranks on the DIMM, only one of which would be active on each memory cycle. Using multiple ranks increases the number of banks available on the DIMM, thereby increasing the number of row buffers for a given amount of memory.

IV. High-Fidelity DRAM Simulation

The important message from this discussion of DRAM memory systems is that accessing these chips is very complex, and that depending on the nature of an individual access the latency and power profile can be hugely different. The actual difference depends on such a number of conditions that broad statements are dangerous, but a system with optimized open-row access can show upwards of 10x more efficiency in both memory latency and energy than a system with more random accesses to the DRAM.

Keeping track of the state of all the DRAM banks in the memory, especially given the number of timing constraints, is a difficult task. This falls to the memory controller. As a programmer, one cannot usually modify the behavior of the memory controller itself, but one can optimize data layouts and algorithms to make life easier for the memory controller. In the same way that a programmer might take cache layouts into account when designing data structures, he/she might also take DRAM idiosyncrasies into account. The row buffers are a primary target that can be easily exploited by a programmer or compiler.

The result of this complexity is that it is extremely difficult to predict the "average" behavior (latency and power) of the memory system for a given workload. The behavior is so complex, and the interaction of different workloads has such an impact, that any system simulation that does not carefully model the detailed behavior of the DRAM is likely to not capture remotely realistic performance characteristics. In addition to the row-buffer behavior, this should include a model of a memory controller that keeps track of the detailed timing state of all the ranks and banks on the DIMMs, and includes structures such as write buffers and read-request buffers that are used to coalesce accesses.

There are a number of cycle-accurate DRAM simulators that have been reported in the literature including DRAMsim [13] and DRAMsim2 [14] that are primarily trace-driven, and Ramulator [15] that can operate on traces or as an integrated simulator with Gem5 [16]. For our example we will focus on the the USIMM [4] simulator that can operate on traces and also as a stand-alone event-driven DRAM simulator that can be easily integrated into any cycle-accurate simulator using a simple API.

V. Motivating Example: Ray Tracing and DRAM Behavior

In this section we will examine a motivating example in which a cycle-accurate simulator is used to better understand and optimize the hardware and software architecture of a memory-bound application such as graphics rendering.

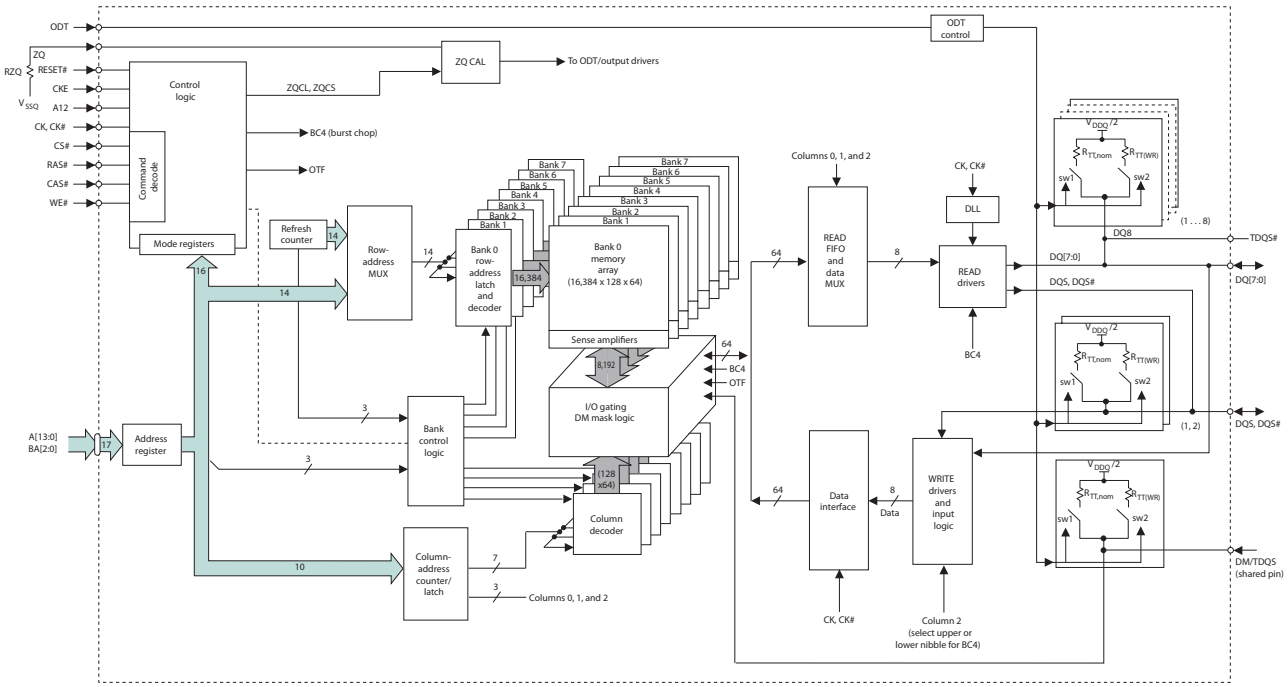

Fig. 2. Block diagram of a commercial DRAM chip from Micron [12]. This is an example of a 1Gb DDR3 DRAM configured as 128M x 8b. The memory arrays are configured as eight banks with each bank having an 8kb row buffer. On a read, 64b from the selected row buffer are transferred to the output FIFO where they are streamed out eight bits at a time.

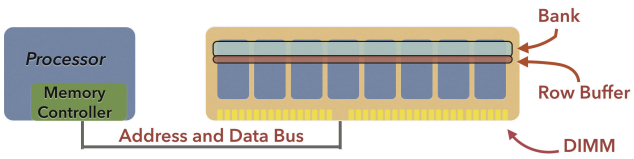

Fig. 3. Block diagram of a DIMM.

A. Ray Tracing Background

Ray tracing is a technique for rendering 3D computer graphics [17]. It is known for its ability to naturally create photo-realistic, high quality images, but also for its high computational cost, and the heavy load it puts on the memory system. Due to this high quality, high cost trade off, it is typically used for rendering movies and still images, rather than for real-time purposes such as video games.

Ray tracing is a simulation of light transport, so the core of the ray tracing algorithm is determining which nearest geometric scene primitive (usually a triangle) a ray of light intersects with. To speed up this process, almost every ray tracer uses an *acceleration structure*, a data structure that organizes the scene primitives in a way that facilitates fast pruning of the set of geometry primitives a ray intersects. The most popular type of acceleration structure is the bounding volume hierarchy (BVH) [18], [19]. A BVH, like most other acceleration structures, is a tree, and determining which geometry a ray intersects involves traversing the tree in a non-deterministic order. This traversal creates unpredictable memory access patterns that prove difficult for caches to filter, causing not only more accesses to DRAM, but also more incoherent access patterns. This problem is exacerbated when many threads are operating concurrently, as is typically the case in "embarrassingly parallel" graphics applications such as ray tracing.

Steady research over the past two decades has continued to advance the cutting edge of ray tracing performance, through both algorithmic and architectural advances. Much of this research has focused on addressing the incoherent nature of control flow and data access patterns caused by BVH traversal in order to improve SIMD utilization or cache hit rates [20]. These techniques usually involve processing multiple rays together in a group called a "packet", with each ray in the packet being spatially coherent (having a similar origin and direction). Ideally, a coherent packet of rays will take a very similar path through the BVH, since traversal order is determined by the origin and direction of the ray. This amortizes the cost of loading a BVH node from memory over multiple rays.

More directly, another technique groups together rays that are known to traverse the same subset of nodes in the BVH, called a *treelet*, where treelets are designed to fit within the cache hierarchy [21], [22]. During traversal, rays are accumulated at treelet root nodes; when a ray crosses a treelet boundary, it is enqueued in a buffer associated with the new treelet. Once a critical mass of rays collects in a treelet, one of the processors or thread-blocks is assigned to process those rays through that treelet. Thus a processor will operate for a

prolonged period of time on only the cache-resident data for a single treelet, greatly increasing cache hit rates.

These ray coherence techniques are designed to improve cache hit rates, but do not necessarily address DRAM accesses. While it is certainly true that reducing the number of DRAM accesses by filtering them with a cache can improve DRAM performance, carefully controlling the *pattern* of accesses can have an even more dramatic effect.

B. GPU Simulation

As a case study, we examine a custom ray tracing system called STRaTA with hardware support for treelet traversal [23]. The goal of STRaTA is to reduce energy consumption by reducing data movement both on and off-chip, and is evaluated with a cycle accurate GPU simulator. This simulator initially used a naïve DRAM model, assuming a fixed average latency and energy consumption for each access. Not surprisingly, increased cache hit rates achieved through using treelets translates to a corresponding decrease in DRAM energy consumption. However, the goal of the STRaTA system is to target data movement, so ignoring the complexities of DRAM is likely a considerable mistake. We augmented the GPU simulator used in STRaTA with a high fidelity DRAM simulator called USIMM (Section VI). The results revealed that although STRaTA was successful at reducing off-chip memory accesses, DRAM energy consumption was not reduced substantially. To truly address data movement energy consumption, we must target DRAM access patterns as well as the caches.

We note that the nature of treelet ray tracing introduces an interesting phenomenon that data is loaded from DRAM in bursts. When a thread-block switches to a new treelet, initially the treelet data will not be cache-resident, and all threads will simultaneously request loads that miss in the last level cache. After this initial burst of cache miss accesses are fulfilled, the thread-block will cease generating DRAM requests while it operates on the cached treelet.

This bursty nature of accessing DRAM presents an interesting opportunity: if we can map all of those accesses to one, or a small number of DRAM rows, then those rows can be opened once and streamed fully to the processor before closing. This type of operation is ideal for retrieving data out of a DRAM system. We achieve this by rearranging the BVH tree so that all nodes in a treelet occupy a contiguous address range that is a small multiple of the DRAM row buffer size. Figure 4 illustrates this, where each treelet maps to exactly two separate rows.

C. Results

The combination of restructuring the algorithm to operate on blocks of data (treelets), and explicitly organizing those blocks for efficient row buffer access has a drastic effect on DRAM performance, both in terms of latency and energy consumption. We analyze performance for twelve ray tracing benchmarks using a baseline ray tracer and our DRAM-centric modifications to STRaTA. Read latency is reduced by up to

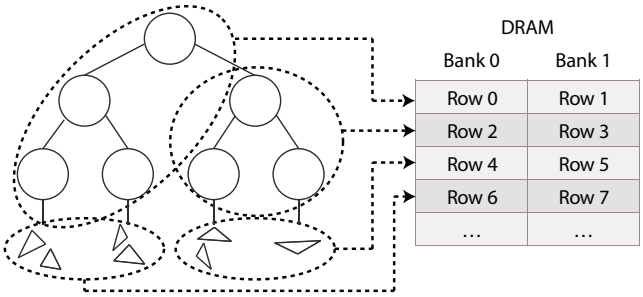

Fig. 4. Treelet row mapping.

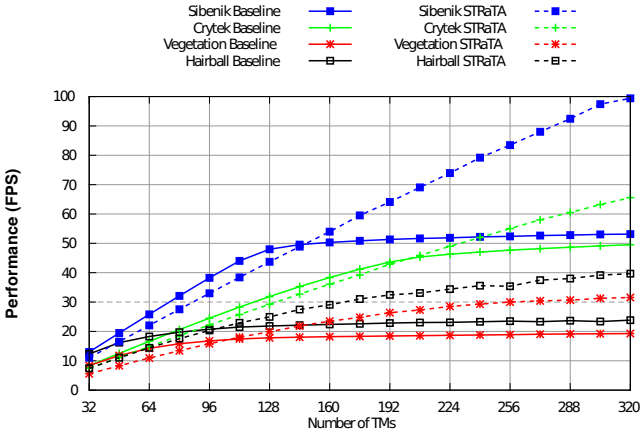

Fig. 5. Parallel scalability with and without carefully controlling DRAM access patterns.

85% in the best case, and 71% on average, and DRAM energy consumption is reduced by 24% on average, even though some benchmarks show increased data transferred. This is all due to an average increase in row buffer hits of more than 50% [3].

Optimizing for row buffer hits is effective not only for reducing latency and energy, but also for getting high throughput data out of DRAM. Memory system performance is often reported in terms of peak bandwidth, but achieving that data rate is all but impossible without carefully controlling access patterns. The memory system can often be the bottleneck, especially for applications with many threads demanding data. By increasing row buffer hits, our improved STRaTA system effectively increases DRAM throughput, allowing for scaling to many more threads, as illustrated in Figure 5 [3]. Each thread multiprocessor (TM) in the STRaTA system represents a block of 32 threads. Figure 5 shows that the baseline performance (solid lines) quickly plateaus due to memory starvation as the number of threads increases, while STRaTA (dashed lines) sees continued performance gains.

Identifying and rectifying these DRAM performance characteristics would be impossible without an accurate DRAM simulator. Augmenting a cycle accurate processor simulator with a DRAM simulator provides a much more complete picture of a full system, revealing performance bottlenecks and opportunities in both hardware and software design.

VI. USIMM DRAM Memory Simulator

The Utah Simulated Memory Module (USIMM) is a highly accurate DRAM and memory controller simulator [4]. It tracks the full state of each bank in the memory system based on well known precise timings for the various phases of each memory access [12]. The memory controller maintains read/write queues, and enacts a scheduling policy to determine which accesses to issue on each cycle. Based on the access phases and duration that each DRAM operation must go through, USIMM tracks the power consumption of each DIMM [24]. For example, an open-row access does not activate the same phases as a row miss.

By default, USIMM uses a realistic scheduling policy and timing parameters, but these are customizable. This makes USIMM a powerful tool for research and exploration of "what if" scenarios [25]. Users can program their own scheduling policy, e.g. to try to increase open-row accesses for a particular application domain. Even without customization, USIMM is an invaluable tool for augmenting existing processor simulators with accurate DRAM modeling.

USIMM is driven either by memory traces or by on-demand access events generated by another simulator. The simple API allows for easily inserting read/write accesses on a given cycle, triggering the memory clock, and receiving access completion events. An existing processor simulator can simply send last level cache misses to USIMM, and then act appropriately when notified of the operation completion.

VII. Conclusions

The main memory system of a modern computer is highly complex. Architectural simulations regularly use detailed simulation models of registers (with possible renaming) and caches because the behavior of those portions of the memory system are very dependent on the way the application uses memory and data. What is not as common is to also use a detailed simulation of the DRAM main memory. We argue that this is a mistake. Even in systems with high cache hit rates, the impact of using main memory more carefully can be large. This is especially true for applications with large memory requirements that cannot fit into caches.

The main culprit is DRAM. This type of memory has such a large latency and power footprint that small changes in the use of that memory can have large impacts on overall system performance. Also, the underlying DRAM memory is so internally complex that it is difficult, or impossible, to predict in advance how the memory system will react to a particular application. Beyond the accuracy argument, it's not possible for a programmer to think about optimizing an application for the memory system if the simulation does not properly model the memory system.

We believe that it is critical for system simulations to use high-fidelity DRAM models, especially if they are being designed with the goal of reducing energy. There are a variety of high-fidelity DRAM simulators available. What is essential is that researchers use one when exploring and reporting on new system architectures.

References

[1] L. Barroso and U. Holzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2009.

[2] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: Eliminating server idle power," in *ASPLOS XIV*. ACM, 2009.

[3] D. Kopta, K. Shkurko, J. Spjut, E. Brunvand, and A. Davis, "Memory considerations for low energy ray tracing," *Computer Graphics Forum*, vol. 34, no. 1, pp. 47–59, 2015.

[4] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley *et al.*, "USIMM: the Utah SImulated Memory Module," University of Utah, Tech. Rep. UUCS-12-02, 2012.

[5] E. Brunvand, D. Kopta, and N. Chatterjee, "Why graphics programmers need to know about DRAM," in *SIGGRAPH '14: ACM SIGGRAPH 2014 Courses*. ACM, 2014.

[6] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. Elsevier Science, 2010.

[7] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[8] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3rd ed. Pearson, 2015.

[9] F. Franěk, *Memory as a Programming Concept in C and C++*. Cambridge University Press, 2004. [Online]. Available: https://books.google.com/books?id=gGYK1brqPeEC

[10] NVIDIA CUDA Documentation, http://developer.nvidia.com/object/cuda.html.

[11] JEDEC, "Global Standards for the Microelectronics Industry," https://www.jedec.org/.

[12] Micron Technology, Inc, "1Gb: x4, x8, x16 DDR3 SDRAM," https://www.micron.com/~/media/documents/products/data-sheet/dram/ddr3/1gb_ddr3_sdram.pdf.

[13] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, "DRAMsim: a memory system simulator," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 100–107, Nov. 2005. [Online]. Available: http://doi.acm.org/10.1145/1105734.1105748

[14] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: a cycle accurate memory system simulator," *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, pp. 16–19, Jan. 2011. [Online]. Available: http://dx.doi.org/10.1109/L-CA.2011.4

[15] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[16] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt *et al.*, "The Gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[17] T. Whitted, "An improved illumination model for shaded display," *Communications of the ACM*, vol. 23, no. 6, pp. 343–349, 1980.

[18] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "OptiX: a general purpose ray tracing engine," in *ACM SIGGRAPH 2010 papers*, ser. SIGGRAPH '10. New York, NY, USA: ACM, 2010, pp. 66:1–66:13.

[19] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst, "Embree - a kernel framework for efficient CPU ray tracing," in *ACM SIGGRAPH 2014 papers*, ser. SIGGRAPH '14. New York, NY, USA: ACM, 2014.

[20] I. Wald, P. Slusallek, C. Benthin, and M. Wagner, "Interactive rendering with coherent ray tracing," *Computer Graphics Forum (EUROGRAPHICS '01)*, vol. 20, no. 3, pp. 153–164, 2001.

[21] P. Navrátil, D. Fussell, C. Lin, and W. Mark, "Dynamic ray scheduling to improve ray coherence and bandwidth utilization," in *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on*, Sept. 2007, pp. 95 –104.

[22] T. Aila and T. Karras, "Architecture considerations for tracing incoherent rays," in *Proc. High Performance Graphics*, 2010, pp. 113–122.

[23] D. Kopta, K. Shkurko, J. Spjut, E. Brunvand, and A. Davis, "An energy and bandwidth efficient ray tracing architecture," in *Proc. High-Performance Graphics*. ACM, 2013, pp. 121–128.

[24] *Calculating Memory System Power for DDR3 - Technical Note TN-41-01*, Micron Technology Inc., 2007.

[25] MSC, "2012 memory scheduling championship," 2012, http://www.cs.utah.edu/~rajeev/jwac12/.