



In-Memory Data Rearrangement for Irregular, Data-Intensive Computing

Scott Lloyd and Maya Gokhale, Lawrence Livermore National Laboratory

The data rearrangement engine (DRE) performs in-memory data restructuring to accelerate irregular, data-intensive applications. An emulation on a field-programmable gate array shows how the DRE could improve speedup, memory bandwidth, and energy consumption on three representative benchmarks.

The memory wall is perhaps the most prominent threat to our ability to analyze expanding data volumes. Although CPU innovations deliver teraflops computing nodes, irregular memory access prevents many data-intensive workloads from achieving corresponding performance improvements. Memory bandwidth improvements mainly benefit regular, streaming access patterns, so the challenge to keep CPU cores tasked with useful work remains. If only one-eighth of every cache line fetched from memory is used—8 out of 64 bytes—memory bandwidth and power are wasted, and performance suffers.

We have developed our data rearrangement engine (DRE) to lay out data dynamically in memory, when needed, and in the form the application needs. Thus we create a cache-optimized layout so that the CPU uses every byte of every cache line in the rearranged data structure. Using near-memory hardware logic, implementable in a logic layer of emerging 3D-memory packages such as the

Hybrid Memory Cube (HMC; www.hybridmemorycube.org), we effectively exploit the memory's vast internal bandwidth to maximize the use of limited off-package bandwidth and reduce the total energy consumed.

IRREGULAR APPLICATIONS AND CACHE

As CPU clock frequency plateaus, multicore and many-core architectures with heterogeneous computational units have emerged as the norm, enabling continued improvement in peak flops (floating-point operations per second) through spatial parallelism. To mitigate the gap between greatly increased peak flops and more modest improvements in memory bandwidth, many-core CPUs incorporate deep cache hierarchies to increase the likelihood that applications' memory accesses will be satisfied in cache.

However, memory-intensive applications with little spatial or temporal access locality might not benefit from cache hierarchies. For these applications, the

computational units often sit idle waiting for data.

For example, sparse matrix operations with a vector form the core of the popular PageRank algorithm, which traverses a Web graph to locate frequently referenced webpages. As Figure 1 shows, the algorithm accesses random locations in the graph and PageRank vector. Ideally the CPU cache will hold only PageRank vector entries corresponding to a vertex's edge list, as shown on the left.

DMA (direct memory access) and gather/scatter hardware integrated with the CPU enable it to initiate data structure rearrangement, but the full data structure must still traverse the memory bus. In our memory-integrated approach, DREs are on the memory package, and only the restructured data traverses the memory bus.

MEMORY-INTEGRATED COMPUTING

Streamlining the integration of logic with memory has been an ongoing research focus.¹⁻⁵ Successfully integrating the two enables the memory system to perform computation that a CPU typically handles, resulting in improved performance and decreased energy use. This is because the processing occurs close to the data without having to move it across chip interconnects from memory to the processor.

Many designs have been proposed for computational elements in memory. However, those designs were predicated on integrating logic into the same fabrication process as DRAM (dynamic RAM) cells and were not found to be commercially viable. With the advent of 2.5D and 3D packaging, placing computational elements directly in DRAM is no longer necessary. Stacking allows logic to be placed

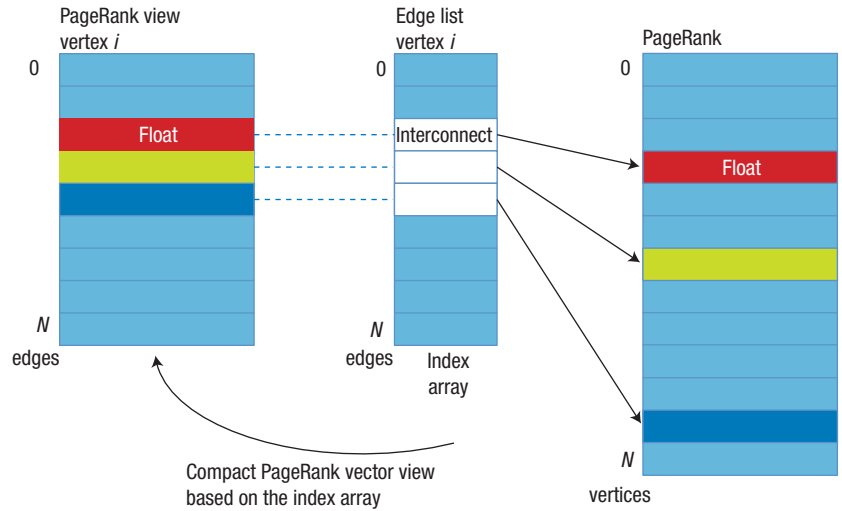


FIGURE 1. View assembly. The data rearrangement engine (DRE) assembles a PageRank view based on the adjacency list for vertex i , which is an index array into the full PageRank vector.

on a separate layer in a separate fabrication process, which makes memory-integrated computing structures more efficient than if they were in DRAM.

Researchers have explored computing with 3D stacked memory, using the HMC's logic layer to hold processor arrays.⁶ The fundamental HMC architecture consists of a stack of DRAM layers connected to a base logic layer with through-silicon vias. The logic layer contains an integrated memory controller, which receives and transmits packetized data over a custom link protocol to an external (off-package) unit such as a CPU. To incorporate computation into this HMC processor design, some have suggested that throughput-oriented processors should be integrated into the logic layer, operating on streaming data directly from DRAM banks. In contrast, we perform data reduction—instead of computational offloading—in memory, similar to the data reorganization proposed by Pedro Diniz and Joonseok Park for field-programmable gate arrays (FPGAs).⁷

DATA REARRANGEMENT ENGINES

Unlike previous proposals that placed full-functionality processor arrays in memory, we propose lightweight DREs

that assist the main CPU by creating cache-friendly data blocks on demand in an in-memory buffer. In our design, load and store requests from the CPU traverse the normal cache hierarchy. Applications explicitly invoke DREs to rearrange data in the memory. When a DRE command finishes, the application accesses the rearranged data buffer, which then traverses the cache hierarchy instead of the original data blocks. For example, on request, a DRE creates a “gathered” subset of the PageRank vector in memory containing entries for a vertex's edge list. The application then accesses that reduced vector. Because the application needs every byte transferred, cache lines are fully utilized. As an additional benefit, computation on the restructured data can be vectorized, which is not possible when the data is scattered in memory.

OUR ARCHITECTURE DESIGN

Our approach partitions an application between restructuring data in memory and computing with it in the CPU. During execution, a DRE library routine assembles the desired view of a data structure fragment in a memory-resident SRAM (static RAM) buffer. The main application in the CPU then reads and updates the buffer, and the DRE

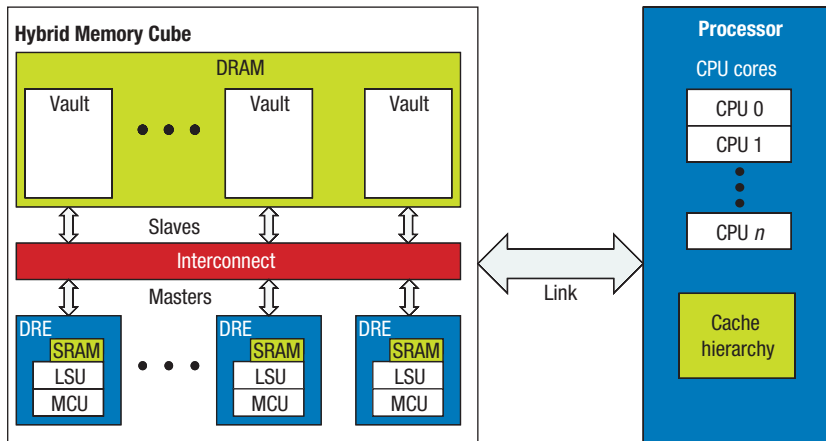


FIGURE 2. DREs connect to the internal interconnect. Each DRE holds a programmable reorder engine and an SRAM (static RAM) buffer. DRAM: dynamic RAM; LSU: load/store unit; MCU: microcontroller unit.

stores from the buffer back to DRAM. The DRE’s application-specific restructuring creates or stores a compact data-structure view; the CPU program loads, computes on, and stores that view. The DRE’s microarchitecture and associated API support this workflow.

The microarchitecture

The DRE is compatible with the HMC memory organization. On the HMC, DRAM banks are laid out in vertically structured vaults. A vault atomically accesses a 32-byte unit and interconnects with other vaults via a link controller that serializes and deserializes the data to and from the memory package. The HMC link protocol can handle multiple data packets from 128 bytes down to 16 bytes. Multiple high-speed links connect the HMC to the CPU and, potentially, to other HMC devices for greater capacity. In the latter configuration, inbound memory requests are either routed to an internal vault or forwarded to the destination on a pass-through link. The current HMC packet protocol only supports requests from an external host. However, a possible extension to the protocol and interconnect would enable a DRE to initiate memory requests from within a device and access data structures transparently across a large distributed memory.

Figure 2 shows a diagram of an HMC-like package in which DREs are also attached to the interconnect. A DRE consists of a programmable DMA unit (a load/store unit or LSU) along with a microcontroller unit (MCU) that executes a simple set of application-directed commands. The MCU program orchestrates LSU actions by generating command messages consisting of addresses and lengths for the LSU to fetch or store either in a fixed stride pattern or as specified in an associated stream of indexes. The LSU uses an SRAM buffer as a scratchpad. This buffer can also be directly addressed by the main CPU and serves as a shared-view buffer for communication between the CPU and DRE.

The API

Upon request, an application process acquires a DRE. The application specifies an MCU program; the OS loads the program into the MCU instruction memory, which is also in the logic layer. The application and MCU communicate with small messages: the application issues commands and receives completion notification by writing and reading a memory-mapped address range. Commands include the following:

- › Setup—loads parameters, such as the base addresses and either

the DMA size and stride for DMA operations or the index vector size and base address for gather and scatter operations.

- › Fill—copies from DRAM to the SRAM buffer according to the access pattern established during setup.
- › Drain—copies from the SRAM buffer into DRAM according to the access pattern established during setup.

CPU interaction

The CPU and DRE exchange control messages: the CPU issues commands and awaits completion; the DRE waits for commands, executes them, and notifies the CPU of completion. A range of reserved memory addresses is used to communicate parameters and completion flags. Polling is used to check for completion.

The CPU and DRE components cooperate to maintain cache and DRAM consistency by issuing cache flush and invalidate operations at well-defined synchronization points such as preceding and following fill operations. Cache consistency operations are explicitly invoked by application code and target the cache in either the CPU or the DRE’s MCU. A flush or invalidate operation is done to the entire cache or an address range, depending on the updated region’s size. Our implementations select the most efficient option. Any evaluation of the DREs’ potential benefits must take into account the overhead of maintaining consistency.

The CPU’s memory management unit (MMU) translates process virtual addresses to physical-memory addresses. Memory requests from the DRE must also be translated, so the DRE must have its own address

translation table. Graphics processors, high-performance networking such as InfiniBand, and some processing-in-memory proposals⁶ employ a general mechanism of mirroring the MMU.

We propose a simpler, albeit more restrictive approach; the application must allocate data to be accessed by the DRE in contiguous physical pages. This can be accomplished by using a custom allocator to gather a large contiguous physical range, as is being developed in transparent-large-page support in the OS. In addition, the pages must be pinned (commonly done by network interfaces in high-performance computing systems) to prevent subsequent relocation. The setup command gives the base physical page address; the DRE adds the base to each address being loaded or stored.

The advantages of this approach are that it requires minimal additional hardware, adds no performance overhead because the address can be assembled as it is loaded onto the request queue, and adds virtually no energy overhead because it involves a simple concatenation of bit fields. Addresses outside that range will trigger a return of control to the application to assemble a new contiguous range for the DRE to access.

THE EMULATOR

To quantitatively evaluate the DRE's performance and energy use, we developed an FPGA emulator modeling a CPU and DRE. We implemented it on a Xilinx Zynq-7000 system on chip (SoC).

Dataflow

Figure 3a shows the Zynq block diagram and emulation framework. The Zynq SoC has two main components:

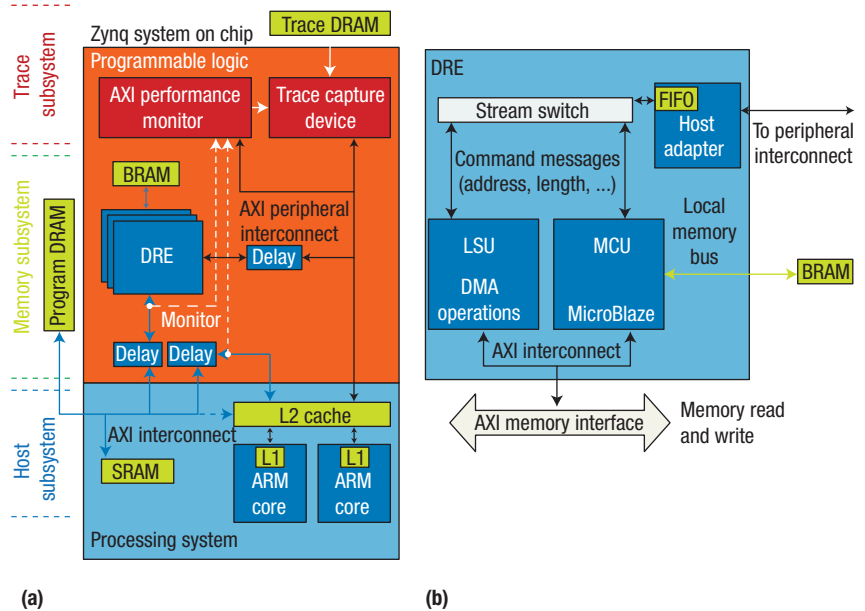


FIGURE 3. Emulation architecture. (a) The Zynq system on chip with the emulation framework. (b) DRE detail. AXI: advanced extensible interface; BRAM: block RAM; DMA: direct memory access; FIFO: first in, first out.

the processing system (PS) and programmable logic (PL). In the emulation, the ARM Cortex-A9 cores in the PS run the application and use dedicated memory: 1-Gbyte DRAM (program DRAM) holding instructions and data. The DREs (see Figure 3b) are implemented in FPGA logic in the PL, which also holds the emulation infrastructure to nonintrusively capture memory traffic.

During the emulation, memory requests from the ARMs that are not satisfied from cache are routed through the PL, even though the Zynq has a direct path from the ARMs to program DRAM. This lets the trace subsystem monitor the ARMs' memory accesses to program DRAM. Similarly, DRE memory requests to program DRAM are also captured. The AXI (advanced extensible interface) performance monitor filters the memory requests and forwards them to the trace capture device for storage in the 1-Gbyte trace DRAM. The PS-side SRAM emulates the DRE's SRAM scratchpad by looping SRAM accesses through the PL as well. Because the AXI performance monitor passively

reads system bus transactions, it does not perturb memory request timing.

Clock frequencies

Many configurable clocks spanning a range of frequencies are supported on the Zynq platform. The A9 cores on the Zynq-7000 can run from under 1 MHz to 800 MHz. The PL clock frequency depends on the specific design placed on the FPGA but is typically about 200 MHz. The double-data-rate program memory runs at 1,066 megatransfers per second.

Because the DRE runs in slower PL, we slow the CPU to run at a frequency with a ratio comparable to the target system. For example, if the DRE runs at 100 MHz and the CPU runs at 200 MHz, program runtimes when scaled by a factor of 10 represent components running at 1 and 2 GHz, respectively. The DRAM clocks are not slowed, so memory requests are routed through a set of programmable delay units (delay in Figure 3a) to emulate memory latencies consistent with the CPU and DRE frequencies. These delay units also allow emulation of a range of memory latencies encompassing

current and future technologies. CPU and DRE clock frequencies along with delay parameters are set with values that maintain the consistent ratios needed to emulate various CPU and active-memory configurations.

Our use of an SoC to emulate a system offers efficiency and challenges. Using fixed-logic intellectual property modules such as the development board's ARM cores, on-chip scratchpad, and memory architecture saves FPGA logic and development time. However, these fixed components also limit CPU design-space exploration and require coordinating multiple clocks to accurately model the desired system.

EXPERIMENTS

To evaluate the DRE's potential benefits, we ran irregular, data-intensive benchmarks over a range of emulated CPU-memory latencies.

Emulation parameters

The emulation targeted a standard CPU core and an HMC-like memory. Datapath widths and memory bandwidths conformed to standard configurations. The processor was a hypothetical 32-bit ARM A9 core running at 2.57 GHz with a 5-GBps memory bandwidth. Because the applications were memory bound, using a 32-bit rather than 64-bit processor did not affect runtime. The 1.25-GHz LSU had a 64-bit internal datapath and 10-GBps bandwidth. The 1.25-GHz MCU had a 32-bit datapath and 5-GBps bandwidth. The control path of sending commands to the DRE had a 340-ns round-trip latency. Measuring performance using a single CPU core and a single DRE enabled precise measurement of the application's memory access characteristics.

We derived the memory parameters from measurements on an Arira Design

Gen 2 HMC evaluation board (www.ariradesign.com/hmc-board), which has instrumentation to capture latency, bandwidth, and power consumption. On the basis of these measurements, we set the latency to access the DRAM array to 45 ns, reflecting the effects of random access in applications lacking long sequential data bursts. On the HMC, additional latency might occur because of congestion in vault request-and-response queues in the memory package.

Because our benchmarks ran in isolation, we modeled the effects of interference from other jobs in a workload by emulating congestion delay at three rates: 0 ns (no delay) for a light load, 20 ns for a medium load, and 40 ns for a heavy load. This latency affected both CPU and DRE memory accesses. We set SRAM latency for the DRE's scratchpad to 10 ns. SRAM latencies vary widely; this value represented the average of latencies reported in the literature. We set the round-trip link latency to transfer packets between the memory package and CPU to 24 ns, again derived from measurements. We modeled the DRAM energy at 19.4 picojoules per bit (pJ/bit), the SRAM energy at 1 pJ/bit, and the link energy at 10.3 pJ/bit. We obtained the DRAM and link energy estimates from measurements on the HMC board. We estimated the SRAM energy on the basis of results reported in the literature.⁸

Irregular applications often access random 8-byte data values. To study how hardware support affects this behavior, we evaluated the energy impact of a potential "narrow vault" architecture modeled on the HMC in which the DRAM can be accessed internally in 8-byte units. On the standard HMC, the DRAM is accessed in 32-byte units, and even a 16-byte request will touch 32 bytes in a vault.

Benchmarks

For these configurations, we conducted a detailed evaluation on three representative benchmark programs: RandomAccess, PageRank, and ImageDiff. Each benchmark had two versions. The CPU-only version was the benchmark in its original form. In the DRE-assisted version, the CPU program communicated with the DRE to load and store an SRAM view buffer, but the CPU performed all computation.

In each benchmark, restructured data in the view buffer was in a compact form that let the compiler vectorize CPU computations. However, this was not possible in the CPU-only version when the data was scattered in memory. Both benchmark versions were serial; in the DRE-assisted version, the CPU core waited for DRE command completion to perform computation and did not double-buffer to hide the DRE latency. The benchmarks ran standalone with a one-to-one virtual-to-physical mapping to enable accurate measurement of each DRE operation phase.

RandomAccess(<http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess>) used the DRE gather and scatter hardware. It is the best example of extreme irregular applications; it measures memory performance in the presence of a completely random access pattern in combination with minimal computation. The benchmark reads, modifies, and writes back random elements of a table that occupies up to half the total memory. In our benchmark, the table was 0.5 Gbytes. The benchmark iteratively performs the computation

```
T[ran[j] & (TableSize-1)] ^= ran[j];
```

where ran is a sequence of random 64-bit numbers. Like the Graph 500

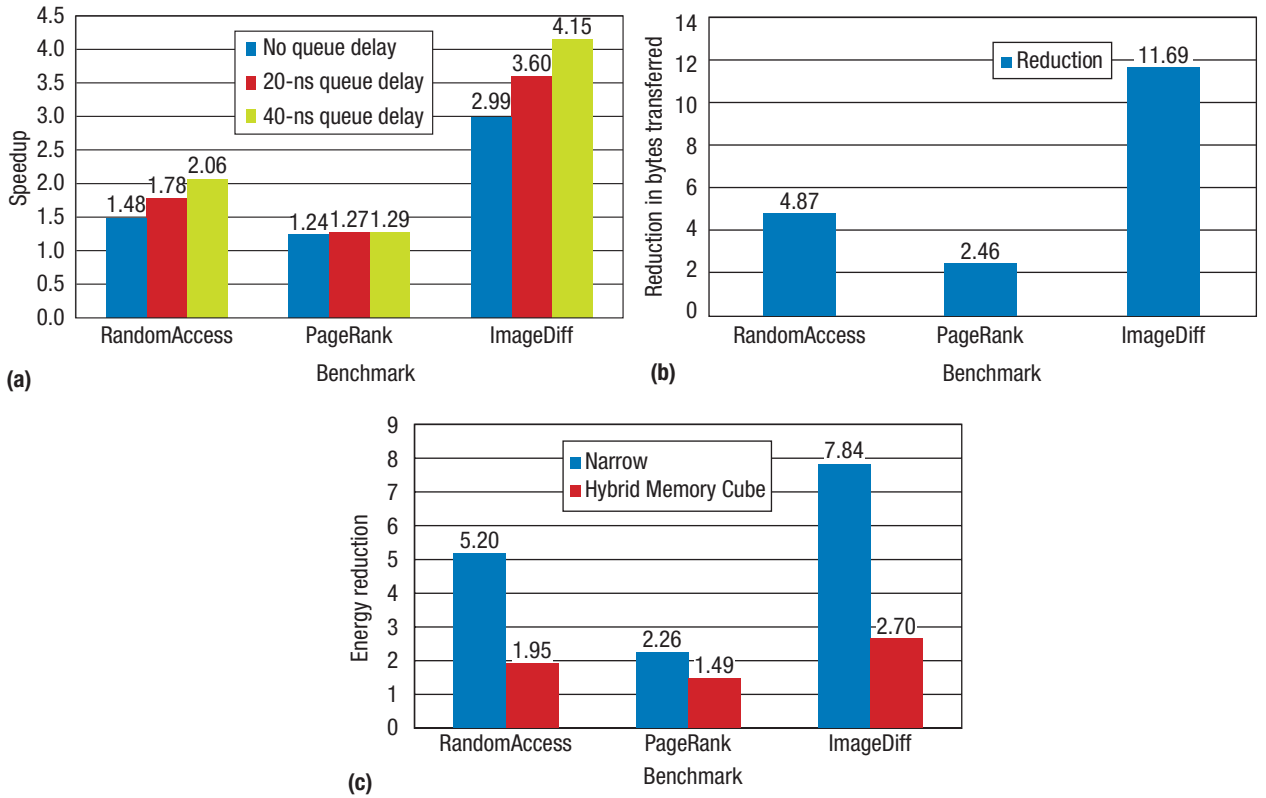


FIGURE 4. Benchmark application performance. (a) Speedup. (b) The reduction in bytes transferred for DRE-assisted versions of the benchmarks. (c) The reduction in energy consumption for a hypothetical narrow-access memory and for the Hybrid Memory Cube. For these benchmarks, using a DRE always provided an advantage.

breadth-first search benchmark, RandomAccess encapsulates the core of one class of irregular, data-intensive applications.

PageRank used the DRE gather hardware. It is a popular data-intensive irregular benchmark characterized by floating-point computation on a sparse matrix (graph) and vector (PageRank). We used a synthetic scale-free input graph with 2^{22} vertices and an adjacency list representation of the graph. The algorithm iterated through the list of vertices and updated each vertex's rank. It replaced indirect, scattered accesses to the PageRank vector with direct, contiguous accesses into the edge list's PageRank view.

ImageDiff performed image differencing of reduced-resolution imagery, using strided DMA. We included this benchmark to demonstrate that even a regular streaming access can appear irregular from the viewpoint of cache reuse, enabling effective DRE assistance when the stride exceeds

the cache line length. The benchmark loads two high-resolution 2D images into memory and, given a decimation factor, subtracts the corresponding pixels in reduced-resolution views in both the x and y dimensions. We used a decimation factor of 16. In the CPU-only version, the CPU performed the image differencing and stored the differenced image in memory. In the DRE-assisted version, the DRE loaded two view buffers with corresponding blocks of the decimated images.

All three benchmarks exercised the synchronization and cache-consistency management methods we described earlier. The application's CPU part issued setup, fill, and drain commands by sending messages to the DRE through special memory addresses. The DRE executed the commands and returned completion messages to the CPU. To maintain memory consistency, the CPU issued a cache invalidate operation to update the cache with the DRE's fill of the SRAM

view buffer. It also issued a cache flush to write its updated view buffer contents to the memory so that the DRE could drain the buffer. The DRE issued the corresponding cache flush and invalidate operations to update its cache if needed. Our performance evaluation included the time to perform these operations.

Evaluation

Figure 4 summarizes the benchmarks' performance results. For these benchmarks, using a DRE always improved speedup, memory bandwidth, and energy consumption.

The speedup results (see Figure 4a) highlight the effects of intrapackage queue delays. The DRE improved speedup in the contrived case of exclusive access to memory in serial execution. However, even more speedup occurred in the more typical case of interfering memory requests, as shown by the green bars, which indicate a 40-ns queue delay.

IRREGULAR APPLICATIONS

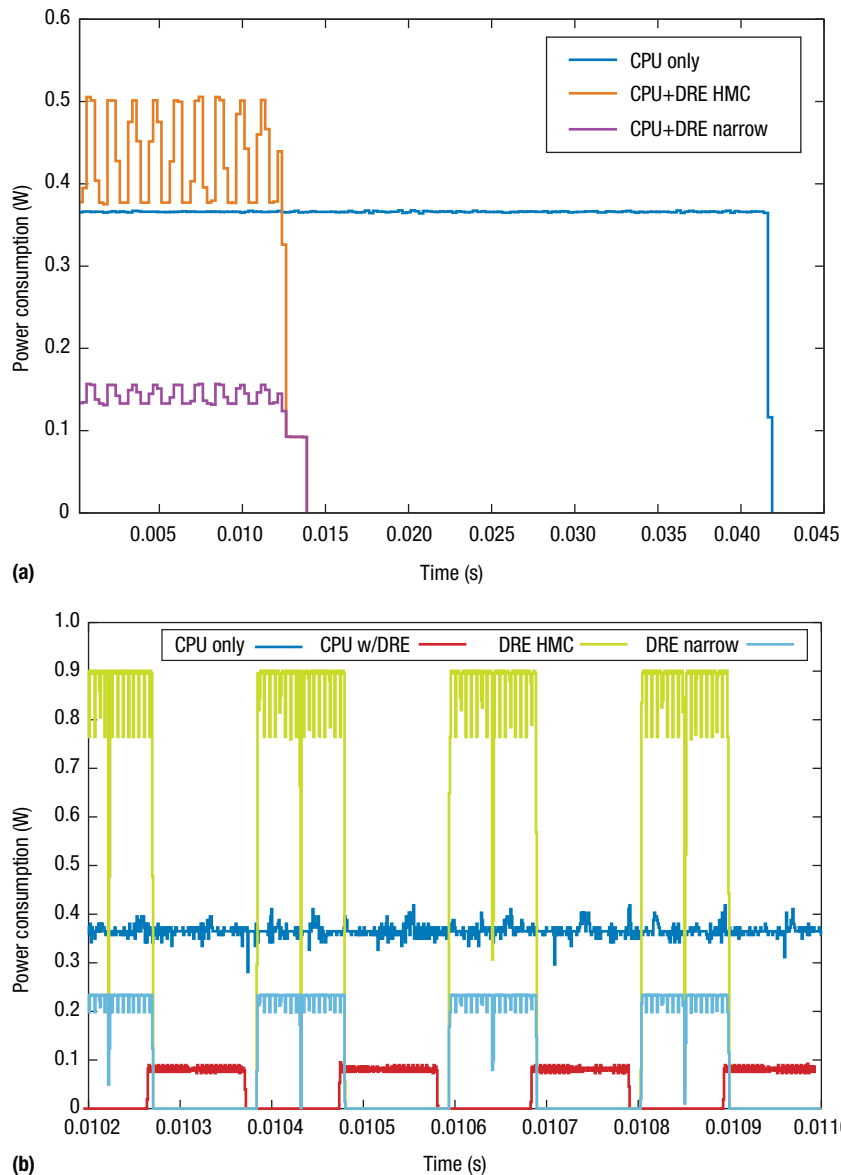


FIGURE 5. The ImageDiff power profile. (a) The entire run. (b) An enlarged segment of the run. The DRE-assisted version in Hybrid Memory Cube (HMC) mode used more power than the CPU-only version, but for a much shorter time. The DRAM energy was 19.4 picojoules per bit (pJ/bit), the SRAM energy was 1 pJ/bit, the off-chip energy was 10.3 pJ/bit, and the decimation factor was 16.

PageRank's speedup ranged from 24 to 29 percent. The more modest speedup compared with the other benchmarks was due to the large number of vertices with few edges, which is characteristic of a scale-free graph. Because of synchronization overhead between the CPU and DRE, it is only profitable to process long edge lists on the DRE. The number of "hub" vertices and the edge list lengths increase with

scale, so larger graphs should show greater speedup.

Memory bandwidth savings (see Figure 4b) ranged from 2.46 \times to 11.69 \times , showing the data reduction afforded by the DRE.

We modeled energy use (see Figure 4c) for two access modes: HMC and Narrow. The HMC bars reflect the HMC's access unit, which required 32-byte access even when only 8 bytes

were requested. The Narrow bars reflect the potential energy savings if we could have employed 8-byte access. In HMC mode, energy savings were at least 49 percent and were as high as 2.7 \times . If Narrow mode had been possible, the energy savings would have jumped to 7.80 \times .

Figure 5 shows a running power profile captured from ImageDiff memory traces (owing to space limitations, the other power profiles are not shown). The full trace (see Figure 5a) shows that the DRE-assisted version in HMC mode (32-byte access) used more power than the CPU-only version, but for a much shorter time. Narrow mode used significantly less power. Both modes showed overall energy savings (see Figure 4c). The enlarged segment (see Figure 5b) shows the hand-off between the CPU and DRE in the DRE-assisted version.

Our results show that rearrangement in active memory substantially improved speedup, memory bandwidth, and energy consumption in benchmarks whose access patterns were representative of irregular, data-intensive analytics workloads. The improvements were particularly significant considering that the DRE performs data reduction, not computational offloading. The improvements were enabled by transforming a cache-unfriendly data layout into a tightly packed, locality-enhancing format before the data traverses the memory bus.

Overheads to using the DRE include coherence transactions to maintain memory consistency, and communication between the CPU and DRE. Despite these overheads, our approach using a shared scratchpad

ABOUT THE AUTHORS

SCOTT LLOYD is a computer scientist at Lawrence Livermore National Laboratory (LLNL). His research interests include high-performance computer architecture, near-memory processing, and reconfigurable computing. Lloyd received a PhD in computer science from Brigham Young University. He is a member of the IEEE Computer Society. Contact him at lloyd23@llnl.gov.

MAYA GOKHALE is a computer scientist at LLNL. Her research interests include data-intensive persistent-memory architectures and reconfigurable computing. Gokhale received a PhD in computer science from the University of Pennsylvania. She is a member of Phi Beta Kappa, a Distinguished Member of Technical Staff at LLNL, and a Fellow of IEEE. Contact her at gokhale2@llnl.gov.


demonstrated improvement in all the metrics we evaluated.

Another aspect to consider is the memory bank's access granularity. For irregular applications, 8-byte granularity would save substantial energy for DRE-assisted irregular applications. Although reducing access size might introduce unacceptable complexity into DRAM, researchers suggest that such an organization will be better suited to future persistent memories.⁹

For this evaluation, we wrote the DRE-assisted versions of the benchmarks manually to closely control the low-level hardware interfaces for effective hardware and API codesign. Our evaluation's promising results support the next step of building higher-level tools to minimize application effort. These tools would

- ▶ encapsulate the view buffer interactions in libraries similarly to communication libraries,
- ▶ hide DRE interaction in high-level language classes that use the libraries, and
- ▶ use compiler pragmas to indicate DRE interaction.

Our research focused on a single CPU core interacting with a single DRE. A more comprehensive use case would include multiple DREs managed by one or more cores. Each DRE would therefore have its own view buffer, and—as in the case with a single DRE and CPU—the application would have to coordinate synchronization and cache consistency if data structures are shared. Another important aspect is how the reduced bandwidth used by the irregular part of the workload can be exploited by

other, more regular applications. Although this is plausible, more research is necessary to evaluate the effect on full system throughput. 

ACKNOWLEDGMENTS

Lawrence Livermore National Laboratory performed this research under the auspices of the US Department of Energy under contract DE-AC52-07NA27344. We thank Roger Pearce for the original Page-Rank benchmark.

REFERENCES

1. M. Gokhale, W. Holmes, and K. Iobst, "Processing in Memory: The Terasys Massively Parallel PIM Array," *Computer*, vol. 28, no. 4, 1995, pp. 23–31.
2. J.B. Brockman et al., "A Low Cost, Multithreaded Processing-in-Memory System," *Proc. Workshop Memory Performance Issues (WMPI 04)*, 2004, pp. 16–22.
3. J. Draper et al., "A Prototype Processing-in-Memory (PIM) Chip for the Data-Intensive Architecture (DIVA) System," *J. VLSI Signal Processing*, vol. 40, no. 1, 2005, pp. 73–84.
4. J. Gebis et al., "VIRAM1: A Media-Oriented Vector Processor with Embedded DRAM," presented at the Student Design Contest, 41st Design Automation Conf. (DAC 04), 2004; <http://csl.stanford.edu/~christos/publications/2004.dac.iram.pdf>.
5. L. Zhang et al., "The Impulse Memory Controller," *IEEE Trans. Computers*, vol. 50, no. 11, 2001, pp. 1117–1132.
6. R. Nair, "Active Memory Cube," presented at the 2nd Workshop Near-Data Processing (WoNDP), in conjunction with the 47th IEEE/ACM Int'l Symp. Microarchitecture (MICRO-47), Dec. 2014; www.cs.utah.edu/wondp/Nair.pdf.
7. P.C. Diniz and J. Park, "Data Reorganization Engines for the Next Generation of System-on-a-Chip FPGAs," *Proc. 2002 ACM/SIGDA 10th Int'l Symp. Field-Programmable Gate Arrays (FPGA 02)*, 2002, pp. 237–244.
8. B. Rooseleer, S. Cosemans, and W. Dehaene, "A 65 nm, 850 MHz, 256 Kbit, 4.3 pJ/Access, Ultra Low Leakage Power Memory Using Dynamic Cell Stability and a Dual Swing Data Link," *Proc. 2011 ESS-CIRC*, 2011, pp. 519–522.
9. J. Meza, J. Li, and O. Mutlu, "A Case for Small Row Buffers in Non-volatile Main Memories," *Proc. IEEE 30th Int'l Conf. Computer Design (ICCD 12)*, 2012, pp. 484–485.



Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.