Active Memory Cube: A processing-in-memory architecture for exascale systems

R. Nair
S. F. Antao
C. Bertolli
P. Bose
J. R. Brunheroto
T. Chen
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
Z. Sura

Many studies point to the difficulty of scaling existing computer architectures to meet the needs of an exascale system (i.e., capable of executing 10^{18} floating-point operations per second), consuming no more than 20 MW in power, by around the year 2020. This paper outlines a new architecture, the Active Memory Cube, which reduces the energy of computation significantly by performing computation in the memory module, rather than moving data through large memory hierarchies to the processor core. The architecture leverages a commercially demonstrated 3D memory stack called the Hybrid Memory Cube, placing sophisticated computational elements on the logic layer below its stack of dynamic random-access memory (DRAM) dies. The paper also describes an Active Memory Cube tuned to the requirements of a scientific exascale system. The computational elements have a vector architecture and are capable of performing a comprehensive set of floating-point and integer instructions, predicated operations, and gather-scatter accesses across memory in the Cube. The paper outlines the software infrastructure used to develop applications and to evaluate the architecture, and describes results of experiments on application kernels, along with performance and power projections.

Introduction

Large petascale supercomputers are already being used to solve important problems in a variety of areas like material science, energy science, chemistry, earth systems, astrophysics and astronomy, biology/life systems, health sciences, nuclear and high energy physics, and fluid dynamics [1]. Each of these areas has specific problems and a set of algorithms customized to solve those problems, but there is overlap in the use of algorithms across these domains because of similarity in the basic nature of problems across domains. Several features are common to applications of interest to the supercomputing community. Chief among these is the pervasive use of the MPI (Message Passing Interface) programming model [2] with the partitioning of the application memory space into MPI domains, each typically no more than 2 to 4 GB in size. Many of these applications exhibit massive data parallelism, performing the same set of operations on billions of data elements. Moreover, most of the data processed in these systems exhibit fairly short tenancy in the processor, streaming into and out of the processor at high rates.

The use of even larger supercomputers is expected to both speed up solutions for these problems and enable the solution of problems with larger datasets or with finer resolution.

Digital Object Identifier: 10.1147/JRD.2015.2409732

Beyond this, however, there is an expectation that the ability to compute 10^{18} operations per second will also bring a qualitative change to problem solving, allowing exascale systems to be used in new types of problems such as uncertainty quantification [3] and in new areas, such as modeling of the human brain [4].

There are several challenges associated with designing exascale systems [5], the most important of them being the power consumed by these machines. At the rate of power consumption of current petascale systems, an exascale system could consume as much as 500 MW, far higher than the 20 MW targeted for these installations. The principal engine for the advances made by supercomputers over the past decades has been Dennard scaling [6], through which the electronics industry has successfully exploited the combination of increased density, better performance, and reduced energy per operation with steady improvements in silicon technology. With Dennard scaling, the target of 20 MW could have been reached in four generations of technology. However, with oxide thickness down to a few atoms, voltages can no longer be scaled proportionally without serious compromise to the reliability and predictability of device operation, and it is no longer reasonable to expect the same improvement in clock frequency for constant power density that was practical in the past.

The other challenge in the design of dense computational nodes is the limited bandwidth available to access memory external to the processor chip through the limited number of pins available at chip boundaries. The bandwidth of signaling through these pins has been increasing at a far lower rate than that of the computational capability within the processor chips. In contrast to the 1 byte per second (byte/s) bandwidth to memory for every floating point operation per second (flop/s) typical in the early generation of supercomputers, today's supercomputers provide barely 0.2 bytes/s per flop/s [7]. Memory hierarchies on the processor chip bridge the gap for some applications, but do so at the cost of power in static random-access memory (SRAM) caches, and only for applications that can exploit temporal or spatial locality of data.

One of the technologies that is maturing and that could help in both these respects is 3D stacking of silicon dies. 3D stacking not only helps in providing a compact footprint, but it also reduces the latency of communication between circuits on different layers. With judicious use of through-silicon vias (TSVs), 3D technology could also provide greater bandwidth between sections of the processor design without having to resort to large, expensive dies. The Hybrid Memory Cube (HMC) [8] is a novel memory platform in which the capacity of memory is dramatically increased by 3D stacking of several (four to eight) layers of DRAM dies, and connecting them to an additional base logic layer using TSVs. The base layer contains the memory controller, the built-in-self-test (BIST) logic, and an interface to a host processor, and uses CMOS technology optimized for such logic.

The logic layer in an HMC structure provides a convenient point where data transformations can be performed on the contents of the memory stack. Such transformations can filter data before the data are provided to a host processor for further processing, thereby reducing bandwidth requirements and latency at the interface between the host processor and memory. In other cases, such transformations can remove the need for the host processor to read or write any part of the data. In both cases, the reduction of data movement between chips results both in improved performance and in power savings. This paper describes the architecture of one such personalization of the logic layer in an HMC, called the Active Memory Cube (AMC).

The AMC is an example of the concept of processing in memory, which has been experimented with for several decades with limited success. The restriction of commodity DRAM architectures to conform to slowly-evolving standards, and the limitations placed on digital logic by the technologies used in the processing of DRAM have been difficult problems to overcome. The commercialization of HMC makes it possible to mix DRAM technology with logic technology in a 3D stack. By retaining the memory interface of the HMC, an AMC can appear to a host processor as ordinary memory accessible through standard memory read and write commands, while providing added capability of independently transforming the contents of memory. Special commands to memory have been provided on systems in the past, for example in atomic read-modify-write operations; the AMC extends the concept significantly further allowing commands to specify arbitrary programs to perform the transformations.

This versatility comes at some cost. The program to be executed in the AMC, pointers to data on which the program operates, and translation tables to map virtual addresses in user programs to real addresses in the AMC must be communicated from the host to the AMC. Much of the advantage of processing in memory could be lost if too much communication is needed between the host and processing elements within the memory. There is a subtle balance between keeping the processing elements on the AMC simple in order to maximize the power-efficiency of these elements, and making them versatile in order to reduce the need to communicate with the host processor. This has led to the design of the AMC processing elements as *lanes* that have characteristics of a general-purpose core, but tuned to be power and area-efficient for the streaming vector functions common in scientific workloads. This tradeoff yields an unconventional mix of features including the absence of caches and hardware scheduling of instructions, but the incorporation of vector registers, a vector instruction set, predicated execution, virtual addressing, and gather-scatter accesses directly to memory. The memory in the AMC is consistently and coherently accessible both from the lanes and from the host processor, thus avoiding the need to copy data back and forth between the memory associated with these units, as is needed in typical accelerators attached to processors.

With the hardware made efficient in area and power utilization, much of the burden of resource allocation and instruction scheduling falls on the compiler. The AMC compiler takes a standard program annotated to run on a parallel system using OpenMP** 4.0 [9] directives and demarcates the parts that should be run on the AMC. It exploits all of the vector and exposed pipeline features of the lanes in the AMC to derive an effective schedule of instructions for each lane. Runtime additions to the operating system provide the services needed by the compiler to allocate lanes and to direct the placement of data within the AMC and across multiple AMCs connected to the host.

The idea of using processing capability in the base logic layer of a stack such as the HMC has recently been exploited in other proposals such as NDC [10] and TOP-PIM [11]. These proposals generally incorporate cores of existing processor architectures in the base layer and hence cannot approach the energy-efficiency of the AMC, whose lane architecture is specifically designed with the energy-efficiency goal of scientific exascale supercomputers in mind.

The sections in the rest of this paper provide a description of the structure of the AMC, details about the organization of the lanes, a description of the software development infrastructure, and results of some experiments on a simulated version of the AMC.

Structure of the active memory cube

As mentioned before, the AMC is built upon the fundamental structure of the HMC, sharing with the HMC the architecture of the DRAM stack. The memory dies in an HMC stack are partitioned into blocks, and all blocks in the same physical location on all dies are grouped together into an entity called a *vault*. A corresponding *vault controller* individually manages each vault. All vault controllers are interconnected to one another and, through *link controllers*, to external I/O links via an internal network.

The vault controllers, interconnect network, and link controllers are located on a logic layer separate from the DRAM dies. As indicated, the AMC adds several processing elements, referred to as *lanes*, in the base logic layer, connected to an enhanced interconnect network as shown in **Figure 1**. By using the same interconnect network and the same transaction protocols between a lane or a link and the vault controllers, a lane can address memory locations within its AMC just as an external processor can. The link controller can send and receive read and write commands using special memory-mapped input/output (MMIO) addresses to and from any lane. This allows an external host processor to communicate directly with a lane, and for a lane to indicate some event to the host without involving the vault controllers.

Memory structure

The prototype design for the AMC described in this paper, assumed to be implemented in 14-nm technology, has an 8 GB stack comprising eight layers of DRAM, each having 32 partitions of 32 MB. Each partition has two *banks* each composed of 16 K pages with each page holding 8 128-byte cache lines. Aligned partitions from each of the 8 layers are grouped into a vertical *vault* as in the HMC. The 16 banks in a vault are connected to its vault controller in the base logic layer through a stack of TSVs. Address and command information are delivered to the DRAM layers through a TSV broadcast bus and decoded by each vault layer, but captured only by the layer to which the address decodes.

Each request to a bank of memory delivers a packet of 32 bytes plus 32 bits of metadata. The 32 bits of metadata are divided into 30 bits for ECC, one bit for coherence control and one bit for snoop filtering support required by typical host multiprocessor implementations, e.g., those of the IBM POWER* architecture. Each 32-byte packet is delivered to the vault controller in four lane-clock ticks. At a 1.25 GHz clock rate, this results in a peak bandwidth of 10 GB/s per vault, or 320 GB/s over the full cube.

Host processor interface

The host processor is assumed to be an implementation of the POWER architecture directly connected to eight such AMCs, communicating with each AMC over a bidirectional link with a bandwidth of 32 GB/s in each direction. Each of the eight AMCs is daisy-chained to another AMC through another bidirectional link. This provides a total memory capacity of 128 GB and a bandwidth of 256 GB/s to the memory in each direction. In comparison, the total internal bandwidth to the vaults in the 16 AMCs is 5 TB/s. The bandwidth is visible to 512 lanes, 32 per AMC, each running at a frequency of 1.25 GHz.

Lanes within an AMC can address only the memory located within that AMC. Communication and synchronization between AMCs must be coordinated by the host processor. This is not an unreasonable restriction because the amount of memory needed for each process, e.g., an MPI task, in large-scale scientific machines is typically less than 8 GB.

The vault controller

The vault controller on the base logic die receives all requests to the DRAM layers either from the host processor or from individual lanes within the AMC. Requests from the host are typically for entire 128 B cache lines. However, since no

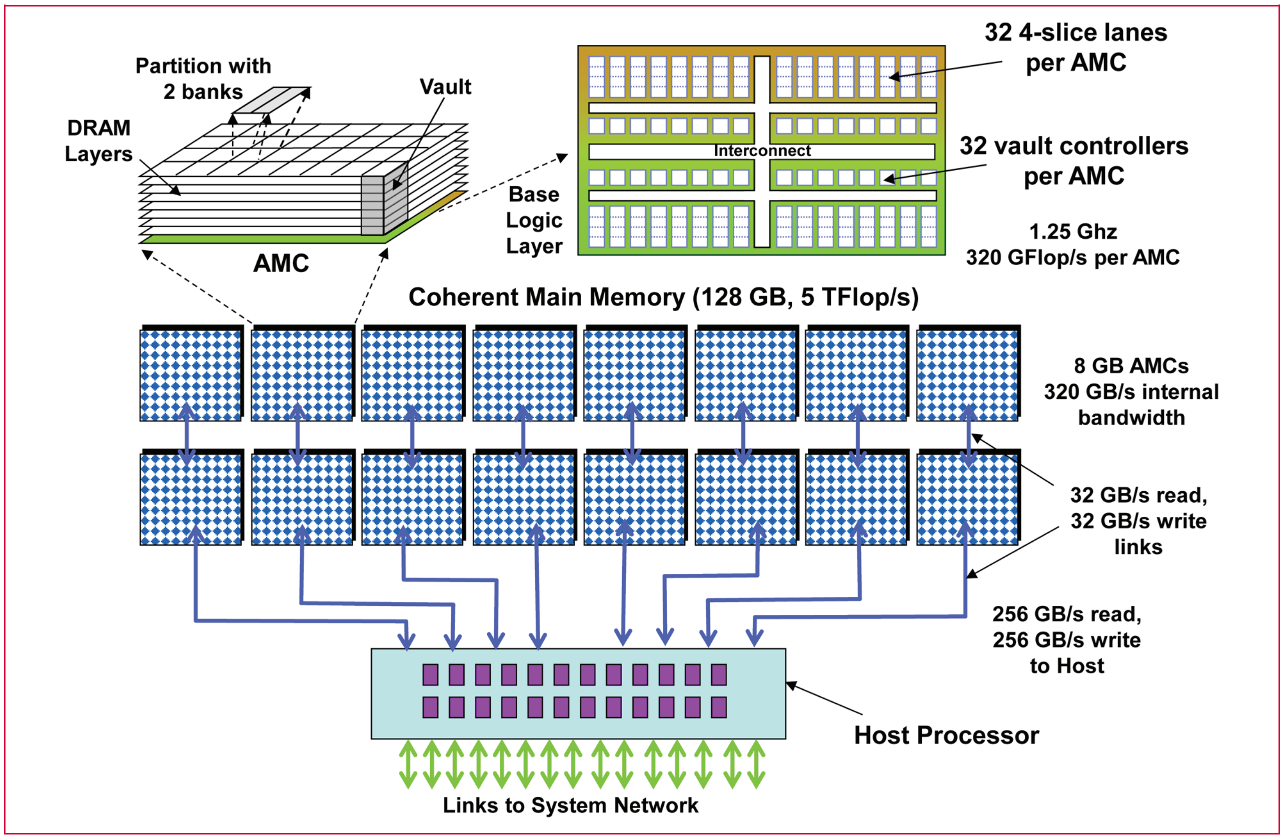

Figure 1

System node incorporating Active Memory Cube.

caches are incorporated within the AMC, smaller widths can be requested by lanes. In order to save power and reduce the latency of access for requests smaller than 32 bytes, the vault controller delivers the needed 8- or 16-byte part requested but saves the entire 32-byte packet coming down the TSVs in a *vault buffer* capable of holding 256 such packets. A hit in the vault buffer avoids the need to send the request up to the DRAM layers.

Maintaining an entire host processor cache line of 128 bytes along with its metadata bits in a single DRAM page avoids the bandwidth and power overhead of splitting a line among multiple pages. However, in order to support small granularity requests from lanes to the same cache lines, the default policy for DRAM access was changed from the closed-page mode of the HMC [12], where the page is closed immediately after one access, to an open-page mode, where the page is closed only if subsequent access is to a different page on the same layer of the vault.

The vault controller is an ideal place to perform atomic operations, which are variants of read-modify-write operations typically supported by memory controllers. Atomic operations usually read data from a memory location, manipulate the data by performing logical or arithmetic operations with other data, and then write the results back to memory. If return data is warranted, the old data, the modified data, or an indication of success/failure is returned back to the requestor depending on the atomic operation type. The AMC supports a notably rich set of atomic operations including fetch-and-op, compare-and-swap, and even an atomic floating-point add operation. Combining atomic operations over a 32-byte block results in significant reduction of interconnect traffic for several vector applications.

Interconnect structure

The vaults and lanes are organized in four quadrants, each having eight vaults and eight lanes. Two sets of networks connect the links, lanes, and vaults—one for the request traffic going from links and lanes to the vaults, and another for response traffic from the vaults to the links and lanes. Each quadrant implements an internal local request and response crossbar. The quadrants themselves are fully interconnected with 20 GB/s point-to-point links. Requests from the links/lanes can be directed to vaults in the given local quadrant or to vaults in other quadrants. Local quadrant

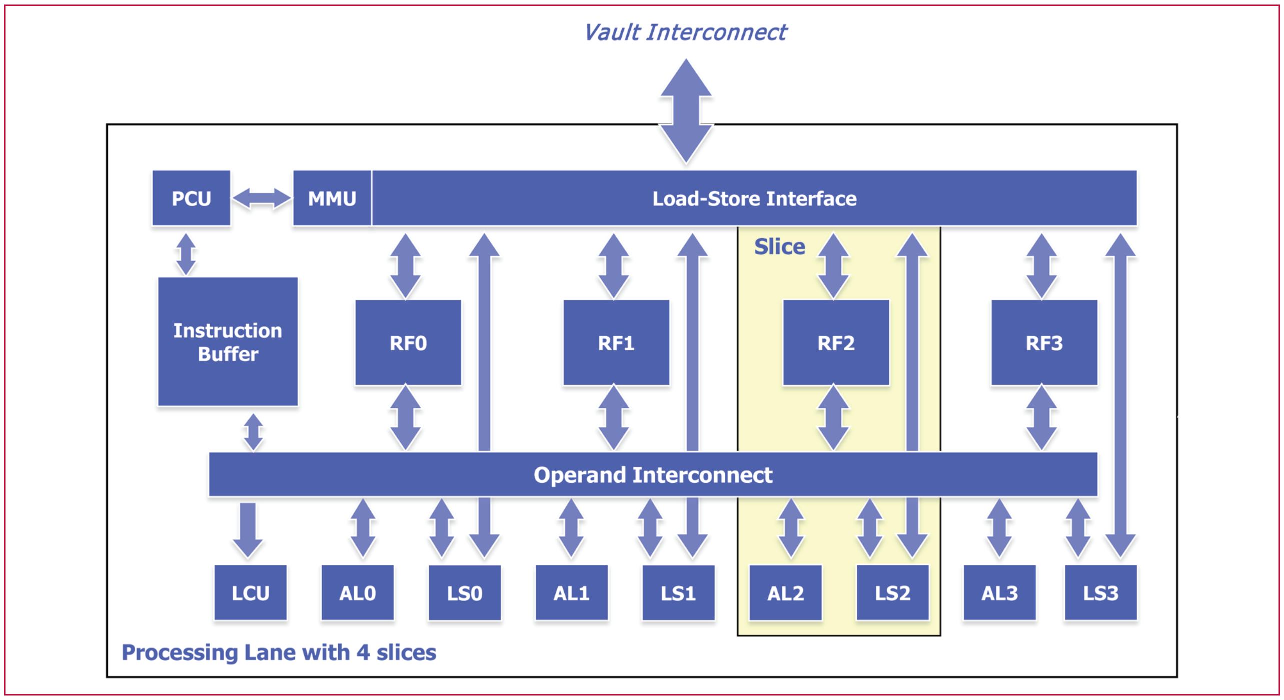

Figure 2

High-level organization of the AMC lane (AL*n*: arithmetic and logical unit *n*; LS*n*: load-store unit *n*; RF*n*: register file *n*; LCU: lane control unit; MMU: memory management unit; PCU: processor communications unit).

vault accesses need to hop across just one crossbar. For remote accesses, the requests have to be sent through the first request crossbar, which then connects it to the appropriate quadrant, from where the request packet has to make a second hop across the request crossbar before reaching the destination vault in the remote quadrant. Similarly, the response packet has to travel through two response crossbars to get back to the source link/lane.

The latency of access to a location in memory depends not only on whether the access is to a local quadrant versus a remote quadrant, but also on whether the request hits in the vault buffer, whether the page being accessed is already open, and finally on the existence of conflicts in the fabric. The minimum latency of access is designed to be 24 cycles when the access is a hit to the vault buffer in a local quadrant.

Coherence management

The host can access the same memory locations on an AMC as the lanes in that AMC. The vault controller has coherence logic that maintains coherence of access to the memory locations shared between the lanes and the host. The granularity of coherence tracking is 128 bytes, identical to the granularity of tracking among caches on the host. The AMC tracks the caching of each line by storing this information in the metadata region for each 128 B memory line. The state bit used for tracking is replicated across each 32-byte sector of the line and is used to track the following states:

- *INT_RW*—Agents within the AMC, like a lane, have ownership of this line and can read or write this line.
- *EXT_RW*—The host has ownership of this line and can read, cache, and write this line.

An access to a line by the host results in the state being set to EXT_RW if it is not already in that state. An access to a sector in the EXT_RW state by a lane will incur a delay to allow the host to invalidate or flush cached copies of the corresponding line. The state is then changed to INT_RW and the access is completed. The delay incurred is similar to that which would be incurred in keeping caches coherent on the host.

Only sectors of lines in the INT_RW state are buffered in the vault buffer, and even these sectors are written through to the DRAM on write requests from lanes. Thus, there are no coherence issues within an AMC because the lanes themselves have no caches.

Lane architecture

Figure 2 shows a high-level diagram of the organization within a lane of the AMC. Each lane has its own register files and two principal subsystems: a load/store subsystem to perform read and write accesses to memory contained in the vaults, and a computational subsystem that performs transformations of data fetched and prepares data to be written back to the memory vaults. The load/store subsystem of each lane has a memory-address translation unit that is cognizant of the mapping from effective addresses used in a program to real addresses recognized by the vault controllers. In addition to these two principal subsystems, each lane has a *Lane Control Unit* (LCU) that directs the flow of instructions into the other subsystems, and a *Processor Command Unit* (PCU) that perform all operations related to communicating with the host processor, synchronization among lanes, and loading of configuration state before lane execution.

Vector instruction set architecture

The computation part of each processing lane in the AMC is composed of four computation *slices*. Each slice includes a memory-access pipeline (also called the load/store pipeline) and an arithmetic pipeline as depicted in Figure 2. The arithmetic pipeline performs arithmetic and logic operations on floating-point and integer data types at a 64-bit granularity. The instruction set architecture (ISA) includes vector instructions that repeat a given operation on vectors of operands, one element per cycle. This is the classical temporal single-instruction-multiple-data (SIMD) paradigm pioneered in the early Cray computers [13]. At the same time, for data types that are smaller than 64 bits, the ISA includes instructions that perform the same operation on multiple elements packed into 64-bit operands in the same cycle, similar to the spatial SIMD paradigm of more recent multimedia architectures. These narrower vector sub-word elements can be 32-bits, 16-bits, or 8-bits wide.

An instruction, also referred to as the *long-instruction word* (LIW) in the ISA, specifies all operations that must be executed simultaneously in the nine pipelines of the lane. The ISA also provides the capability to specify a vector length to be applied to all vector instructions in the LIW. The vector length may be directly specified in the instruction or may be obtained from a special *iteration count register*. Thus, a single LIW in the AMC can express three different levels of parallelism: parallelism across functional pipelines, parallelism due to spatial SIMD in sub-word instructions, and parallelism due to temporal SIMD over vectors.

Each slice of a processing lane contains three register files: a *vector computation register* (VCR) having 16 entries of 32 vector elements, each 64 bits wide, a *scalar computation register* (SCR) having 32 entries, each 64 bits wide, and a *vector mask register* (VMR) having four entries of 32 vector elements, each 8 bits wide. In general, the operations in a slice read from and write into registers belonging to that slice. However, a slice can also read values from the VCR of another slice.

The arithmetic pipeline in the ALU (arithmetic logic unit) includes support for a large set of floating-point and integer operations needed in different types of workloads. The data types supported include double-precision and single-precision floating point, and byte, 2-byte halfword, 4-byte word, and 8-byte doubleword integer operations. The ISA includes multiply-add instructions for all data types, special instructions in support of single-precision complex arithmetic, and long-latency instructions such as divide, square-root and inverse square-root.

The load/store pipeline is capable of specifying a vector operand as a target. The separate addresses needed to load each of the vector elements are computed either by specifying a scalar stride to be added to a base operand, or by specifying a vector register as one of the address operands. The latter option facilitates the gathering and scattering of data between any part of the AMC memory and vector elements or sub-word elements.

The eight bits of the VMR allow an arithmetic or a load/store operation of a vector element or sub-word element to be predicated. Complex predicates can be computed through a rich set of logical operations supported on VMR operands.

Pipeline organization

Most floating-point instructions have a latency of 8 cycles; divide, square root, and inverse square root have a latency of 26 cycles, while integer instructions and data manipulation instructions have a latency of 4 cycles. In order to keep the hardware complexity as low as possible, the pipelines are exposed in the ISA, implying that it is the responsibility of the assembly language programmer or the compiler to take into account the instruction latencies in the pipeline when scheduling dependent instructions. This avoids the need for complex dependency-checking and instruction-stalling logic found in standard processors. The only exception to this is the interlock implemented for load operations, which have unpredictable latency. Target registers of load operations are associated with a *pending bit* that indicates a pending load. Use of a register that has this pending bit set results in a stall of all activities in the lane until the load data arrives from memory.

Performance and power consumption can be improved by specifying *bypassed* operands in instructions: results of operations from one slice may be fed directly to dependent operations on any slice two cycles earlier, compared with using the results of operations from their target register locations.

One double-precision floating-point multiply-add execution can complete in each slice during each 1.25 GHz cycle, resulting in a peak floating point capability of 10 billion floating-point operations per second (Gflop/s) per lane or 320 Gflop/s per cube. The resulting ratio of 1 byte/s memory bandwidth per flop/s is significantly better than the 0.2 ratio in an IBM Blue Gene*/Q node.

Load-store operations

The bandwidth available for load/store requests from a lane to the interconnect network is 8 bytes per cycle. This is adequate for most applications of interest. A deep, 192-entry *load-store queue* (LSQ) is provided to buffer requests to memory and to enable *combining* of loads or stores when possible. Combining allows multiple 8-byte requests to adjacent portions of a 32-byte sector to be satisfied through a single transaction on the interconnect fabric and at the vaults.

User-level POWER effective addresses are computed in the load-store pipeline and translated into AMC real addresses using an 8-entry effective-to-real address translation unit (ERAT). Misses in the ERAT force the lane execution to stall and are handled by the PCU which sends a command back to the host processor through the link controller. This command generates an interrupt handled by the operating system, which obtains a translation for the address causing the miss and sends it back to the PCU to be installed in the ERAT. The PCU then wakes up the lane pipelines from the stall and retries the memory operation that caused the miss.

Instruction buffer

The instruction fetch mechanism of typical processors, with its instruction cache, instruction prefetch and branch prediction mechanisms, is costly both in area and in power consumed. The iterative nature of scientific kernels, their relatively small size, the predictability of branches, and the ability to hide branch latencies through vector instructions all call into question the need for complex instruction caches and hardware schemes for predicting and prefetching instruction streams. With its temporal and spatial SIMD features, the AMC can save both in instruction footprint and in power consumed by amortizing the cost of fetching instructions over many more operations compared to typical processors. Further reduction of complexity and power in a processing lane was realized in the AMC by eschewing an instruction cache in favor of a *lane instruction buffer* (LIB) having 512 entries, each holding a single LIW. The size of the LIB is sufficient for most functions because the LIW format and the temporal and spatial SIMD features of the ISA allow very dense code. The LIB is managed in software either by being loaded directly by the host or through LoadLIB instructions in the AMC ISA.

Conditional and unconditional branches in the instruction stream are restricted to targets within the LIB. Conditional branches examine the value in a condition register that is typically stored as a side effect of computing a vector mask using a vector compare instruction. The ISA features a decrement-count-and-branch-if-zero instruction to keep track of iterations in a loop. The ISA also features an indirect branch that derives its target from the value in a scalar register. This, along with a branch-and-link instruction, greatly facilitates the placement and invocation of small subroutines in the LIB.

Simulation and software infrastructure

Several tools are needed in order to evaluate the design of the AMC and to make it convenient for use by programmers. This section first describes a simulator that was built to evaluate the performance of workload targeted for the AMC. It then outlines the modifications made to the operating system to provide a suitable configuration environment and a runtime to exploit the AMCs in a full system. The section finally describes the programming model and compiler to aid a programmer in developing code for a system containing AMCs.

Simulator

The AMC simulator is an execution-driven simulator based on the Mambo simulator [14], which has been used in several IBM processor development projects. The simulator creates a model for an entire node consisting of a host POWER processor and 16 AMCs. The simulator models all features of the AMC faithfully, including the timing details of the AMC pipelines, the interconnect network, the vault controllers, and the DRAM layers in the vaults. The simulator can be operated in various modes to facilitate understanding of the architecture and the performance of applications. For example, one mode of the simulator allows the user to specify a fixed latency to memory. This mode helps the user in breaking down the memory access delay into delay that is inherent in the scheduling of code and delay due to congestion at the network and at the vaults. The simulator also has options to display statistics on the usage of various resources in the AMC.

Because the focus of investigation was the performance of the AMCs themselves, the host processor was modeled functionally, rather than in a timing-accurate manner. This allowed simulation of the full system, including simulation of the operating system and host regions of the application program. Timing-accurate simulation of these sections would have been prohibitively slow.

To improve the performance of the simulator further, a parallel version of the simulator was developed, which simulated AMC lanes as parallel threads of execution in a shared memory environment with the host processor itself as a separate thread of execution. Simulation of the host processor was further improved significantly using caching and optimization techniques commonly employed in just-in-time compilers.

An assembler was developed to generate binaries from assembly language kernels. Several such kernels were built for architectural verification of the simulator. Assembly-language kernels were also written for several common library routines and for several scientific applications.

OS and runtime support

Extending the range of applications beyond simple mathematical kernels requires an interface that simplifies exploiting AMCs in a system. Primary support for utilizing AMC resources is provided by the operating system, which is the lightweight Compute Node Kernel (CNK) used in the Blue Gene/Q [15]. System calls are used for allocating and configuring resources, but once allocated, those resources are managed by a user-level library, and presented to the application programmer in an application programming interface (API) implemented through a combination of CNK services and a low-level runtime library.

The AMC API allows for logical naming of resources, giving the operating system and runtime freedom to allocate physical resources as needed by multiple processes. At the same time the API exposes all hardware aspects that might be of interest to programmers such as the quadrant and vault structure of individual AMCs.

Different data structures of an application may need to be placed in memory in different layouts. Data that is used only by the host processor would normally be striped across all AMCs in order to maximize the bandwidth usage. However, data structures that are manipulated by lanes within an AMC must be physically allocated to the memory in that AMC. The AMC API supports two types of memory-placement calls. It supports the relocation of an existing region of the client's address space to a particular AMC, a quadrant within the AMC, or even a vault within a quadrant. In addition, it supports the allocation of new regions with such specific location characteristics.

Finally, the AMC API provides for the allocation of regions of memory for passing parameters, for use as a stack by code in a lane, and for returning values from the AMC lanes back to the host.

Compiler

The AMC architecture implements a balanced mix of multiple forms of parallelism (heterogeneous processors, multithreading, instruction-level parallelism, vector, and spatial SIMD) to provide excellent power-performance for real-world high-performance computing applications. Further, AMC processing lanes have radically different memory access properties (latency, bandwidth, coherence) than typical processor or accelerator cores. The unique power-aware architecture, and especially the exposed pipeline, requires strong compiler support in order to productively and effectively use a system incorporating AMCs.

The AMC compiler supports C, C++, and FORTRAN language front-ends. It relies on the user to annotate source code with OpenMP 4.0 accelerator directives. These directives allow the user to identify code sections that will execute on AMC lanes, and to identify regions of data accessed during execution on an AMC lane. The compiler generates AMC binary code for the identified code sections, and code to manage the transfer of execution between the AMC and the host.

The compiler independently analyzes loop nests within the code targeted for the AMC to exploit the vector facilities provided in the AMC architecture. Compiler transformations like loop blocking, loop versioning, and loop unrolling are used to uncover the potential for exploitation of all forms of parallelism available on an AMC lane. These transformations are applied in an integrated manner, rather than separately, in order to minimize code size while retaining good code quality.

At the compiler backend, a mapping phase first intelligently partitions instructions onto one of four slices using a multilevel graph partitioning heuristic [16]. Software pipelining, a technique in which successive iterations of a loop are overlapped and scheduled for parallel execution, is then employed to improve utilization of the functional units on an AMC lane. A polynomial-time heuristic based on the Swing Modulo Scheduling algorithm [17] is used as the scheduling algorithm. The algorithm honors the exposed-pipeline nature of the architecture, taking into account instruction latencies and register port restrictions in arriving at a valid and effective schedule. The algorithm also attempts to control the rate of issue of loads and stores to memory scheduling them early enough to hide the latency while avoiding filling up the LSQ.

Software instruction cache

The techniques used by the compiler target optimal utilization of resources and hence good performance, but they come at a cost in LIB space. Function calls within AMC code sections can help in reducing the number of entries needed, especially when the calls are made to carefully optimized library routines. However, a more general mechanism, a software instruction cache similar to the one described in [18], was developed to overcome the limitation of 512 entries in the LIB. When using the LIB as a software instruction cache, it is partitioned into multiple equal-sized slots. The compiler arranges the AMC code in memory in blocks of size smaller than or equal to the size of the LIB slots, and uses the LoadLIB instruction to dynamically load instruction blocks into LIB slots as required by the program control-flow. Various optimizations are employed to limit the overhead of branching and of loading new sections of code.

Experimental results

Several experiments have been performed using the AMC simulator, principally to characterize the performance and power consumption of the AMC, and to examine the fundamental ability of the AMC to execute vectorizable regions of various types of applications. A few representative experiments are described below.

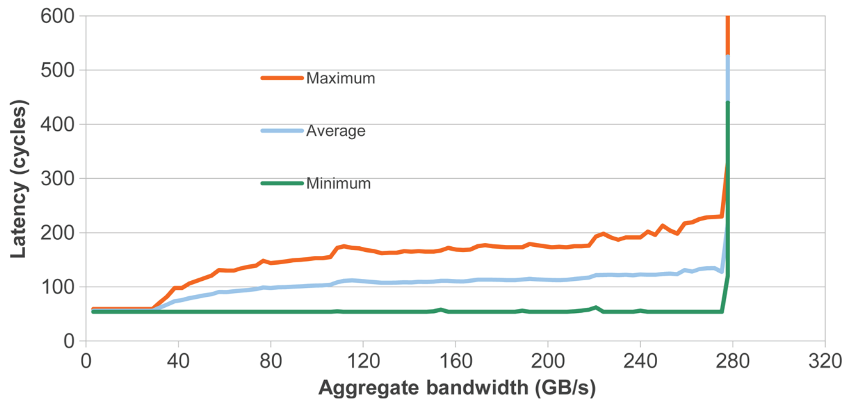

Figure 3

Memory latency and bandwidth characteristics of a sequential access pattern using vault-local addressing in closed-page mode.

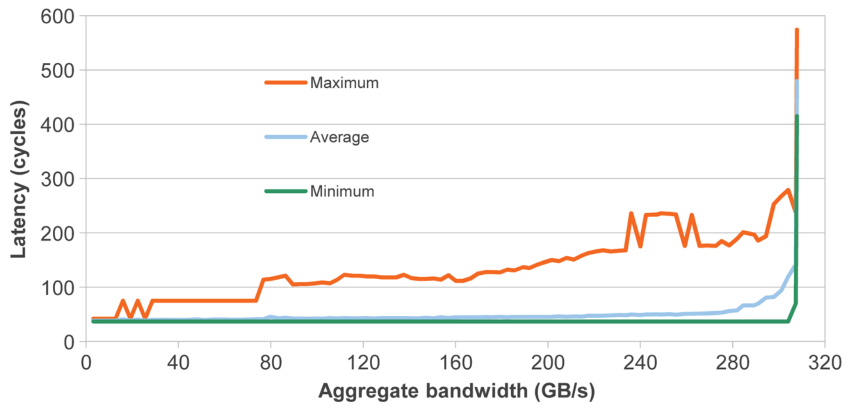

Figure 4

Memory latency and bandwidth characteristics of a sequential access pattern using vault-local addressing in open-page mode.

Bandwidth and latency characteristics of memory and fabric

Figure 3 shows the latency for a sequential access pattern from a lane using an addressing scheme that places adjacent cache lines in the same vault with the vault controller operating in a closed-page policy where the DRAM page is closed after each access. All requests processed by the vault controller are 32 B reads, exploiting the request-combining capability of the lane for sequential accesses. Three graphs are shown corresponding to the minimum latency observed for each value of bandwidth in a random set of access sequences, the maximum latency, and the average latency. The achievable aggregate bandwidth stays below 280 GB/s, 87% of the DRAM interface bandwidth, because each DRAM page is cycled four times for each 128 B of consecutive data. The closed-page policy hurts performance for this access pattern. **Figure 4** shows the results for the same access pattern, but using an open-page policy where the DRAM page is left open after each request, the default case for AMC. The graph shows that up to 310 GB/s of aggregate bandwidth, almost 97% of the peak, can now be exploited. This matches the theoretical limit, accounting for the penalty incurred for switching the driver on the TSVs when a different die of the DRAM stack is selected.

Figure 4 also shows the lower average latency of access in the open-page mode. It further shows that the latency of access increases from about 50 cycles, when bandwidth utilization on the fabric is low, to as much as 180 cycles or more when bandwidth utilization, and hence congestion on the fabric, is high.

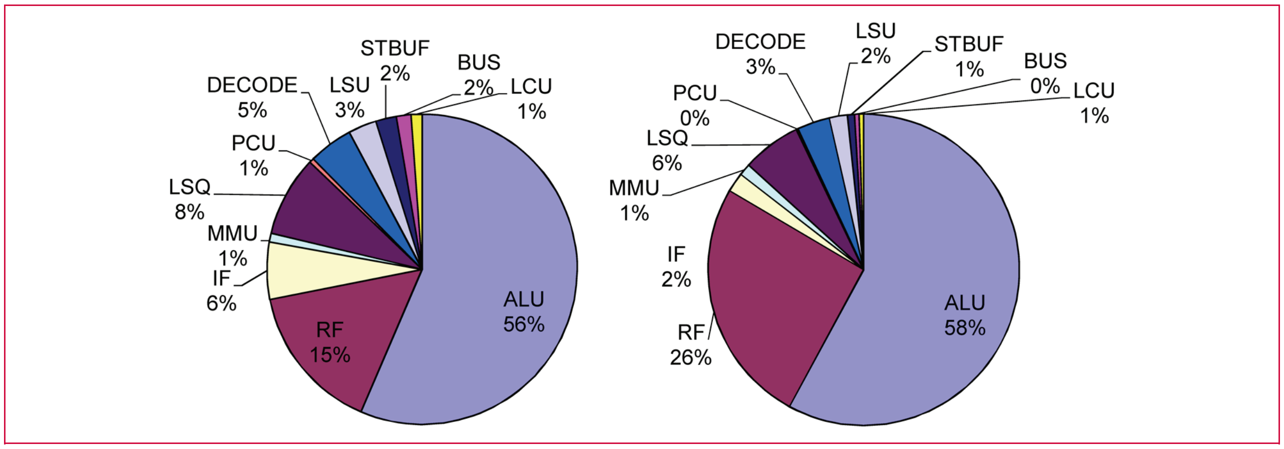

Figure 5

Relative Contribution of units in the AMC lane. *Left*: Area. *Right*: Power consumption while running DGEMM (ALU: arithmetic and logical unit; LCU: lane control unit; BUS: operand interconnect; STBUF: stall buffer; LSU: load-store unit; PCU: processor communications unit; LSQ: load-store queue; MMU: memory management unit; IF: instruction buffer and fetch unit; RF: register files).

DGEMM performance and power

A common kernel used in the rating of supercomputer installations is the double-precision matrix-matrix multiplication routine (DGEMM), a key part of the High Performance Linpack (HPL) benchmark [19], which evaluates the function $\mathbf{C} = \mathbf{C} + \mathbf{A} \times \mathbf{B}$, where $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are matrices. This routine was coded in assembly language and evaluated on an AMC. The metric of performance for this benchmark is the flop/s averaged over the course of execution. With appropriate blocking techniques and use of atomic floating point operations with combining, it was possible to achieve as much as 83% of peak performance on the AMC. Thus a system node with 16 AMCs can sustain as much as 4.3 teraflop/s per node. The power consumed per AMC is around 10 W for a design based on 14 nm technology, resulting in a total of 160 W and an efficiency of 26 Gflop/s/W. This efficiency will be reduced somewhat due to the additional power consumed by the host processor. Because there is hardly any work done by the host during the computation, aggressive power gating can limit the power consumed by parts of the system outside the AMCs. A conservative estimate for the final power/performance of an AMC-based system for this routine is 20 Gflop/s/W. In comparison, the Blue Gene/Q system (45 nm technology) delivers 2.1 Gflop/s/W, while the current best system in the Green500 rankings, the Tsubame system (28 nm technology) delivers 4.4 Gflop/s/W. Given the trends in technology scaling, it is unlikely that these systems will scale beyond 10 Gflop/s/W in the 14 nm technology that has been assumed in this paper.

The core work in a typical computation in any processor is performed within the ALU and the register files, which account for less than a third of the area and power in contemporary processors. Even in an efficient processor like that of the Blue Gene/Q, the area occupied by the vector floating point units and associated register file account for only 38% of the core area and 27% of the power. In contrast, **Figure 5**, which shows the fraction of the area and power going into various units within a lane of the AMC, indicates that the ALU and register files account for as much as 72% of the area and 85% of the power of an AMC lane, thus validating all the microarchitectural decisions described earlier.

DAXPY performance

DAXPY [20] evaluates the function $\mathbf{Y} = \mathbf{Y} + a \times \mathbf{X}$, where a is a constant value, and $\mathbf{X}$ and $\mathbf{Y}$ are vectors. This routine loads two elements from memory and stores one element back to memory for each floating-point multiply-add. Thus, this routine stresses the bandwidth to memory rather than floating-point utilization. When all 32 lanes were used to perform this computation, it was possible to retire 4.95 computations per cycle per AMC with the vectors striped across all vaults and as many as 7.66 computations per cycle per AMC with the vectors blocked within vaults. The latter rate equates to a read bandwidth of 153.2 GB/s and a write bandwidth of 76.6 GB/s, per AMC, or 2.4 TB/s (read) and 1.2 TB/s (write) across the node while executing DAXPY, nearly 10 times the bandwidth that would be achieved if the routine were executed on the host processor.

Characterizing compiler performance

To quantify the performance of the compiler, a compiled version of a double-precision volume computation that performs a determinant calculation on a single lane was compared with a hand-written assembly language version. The loop was annotated with the OpenMP 4.0 pragma `#pragma omp target map(x, y, z, vol)`, which directs the compiler to place the input and output arrays `x`, `y`, `z`, and `vol` on the AMC and to initialize the ERAT tables in each of the lanes of the AMC accordingly.

Programming in assembly language is tedious because the kernel naturally maps only to three out of the four slices in the lane. In contrast, the compiler is more easily able to exploit all four slices. Moreover, the scheduler is able to reduce the stalls due to delay in fetching from memory and due to a full load-store queue down to only 2.7% of the total execution time, leading to a floating-point utilization of 33%, which is 71% better than that of the hand-written version.

A similar exercise was carried out for the DGEMM benchmark mentioned earlier. On a single lane version of the code, the compiler has been able to achieve 75% floating-point efficiency, compared to the 85% that has been achieved by hand-written code on a single lane. The compiler has also produced code for a number of kernels of interest to the supercomputing community for single lanes, for multiple lanes, and for MPI applications using multiple AMCs with one MPI domain per AMC. Many of these kernels had large instruction footprints for which the compiler automatically generated code using the software instruction-cache mechanism to overcome the size limitations of the LIB. Further tuning and experiments are being carried on these and other benchmarks and the results will be presented in a future paper.

Conclusion

This paper described a new processing-in-memory architecture called the Active Memory Cube, which implements a set of processing units in the base layer under a stack of DRAM dies. The instruction-set architecture and the microarchitecture of the processing units were tuned to the vector requirements of common scientific applications and to the low-power requirements of future exascale systems. Complex features implemented in the hardware of traditional architectures were omitted in the interest of maximizing the utilization of the limited area available in the base logic layer and maximizing the power-performance for applications of interest. At the same time, the hardware microarchitecture was exposed to software through the instruction-set architecture to allow software to tune applications for performance.

Software infrastructure was designed around the AMC concept, including a cycle-accurate simulator, a set of operating system features, and an interface that facilitates configuration and management of the resources in the AMC. The programming paradigm at the node level chosen to demonstrate the features of the AMC is OpenMP 4.0. A compiler that takes programs annotated with OpenMP 4.0 directives and produces executable code for the AMC is nearing completion. Experiments conducted demonstrate the salient features of the AMC, namely its internal bandwidth that is an order of magnitude larger than that available from typical processors to memory, its capability to provide an order of magnitude larger number of operations per second compared with traditional processors, the ability to run significant portions of workload close to memory, and the ability to exploit the large degree of parallelism provided by the architecture. The combination of these features allows power consumption in an AMC to be less than half of what would be needed in traditional systems in the same technology running floating-point-intensive applications.

The AMC is a testament to the idea of processing in memory and the general notion of gaining computational efficiency by moving computation to the data instead of the traditional technique of bringing data to a central processing unit [21].

Acknowledgments

This work was supported in part by Department of Energy Contract B599858 under the FastForward initiative. We thank the members of the seven labs of the Department of Energy who gave feedback during the course of the project. The final architecture was influenced also by feedback received from members of the Jülich Supercomputing Center and the IBM Exascale Innovation Center in Germany. The project benefited from the advice of N. A. Eisley, L. Lastras, K. Rodbell, and F. Petrini and the application coding efforts of A. R. Choudhury, H. Imai, Y. Sabharwal, and V. Saxena. This paper has benefited significantly from the comments provided by the editor of the *IBM Journal of Research and Development* and the reviewers. Finally, we acknowledge the constant support, encouragement, and advice from IBM Fellow J. A. Kahle.

*Trademark, service mark, or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

**Trademark, service mark, or registered trademark of OpenMP Architecture Review Board in the United States, other countries, or both.

References

1. J. Dongarra et al., “The international exascale software roadmap,” *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 1, pp. 3–60, Feb. 2011.
2. A. Skjellum, E. Lusk, and W. Gropp, *Using MPI: Portable Parallel Programming With the Message Passing Interface*. Cambridge, MA, USA: MIT Press, 1999.
3. H. Simon, T. Zacharia, and R. Stevens, “Modeling and simulation at the exascale for energy and the environment,” Dept. Energy, Washington, DC, USA, Tech. Rep. [Online]. Available: http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Townhall.pdf

4. H. Markram, “The human brain project,” *Sci. Amer.*, vol. 306, no. 6, pp. 50–55, 2012.
5. A. Gara and R. Nair, “Exascale computing: What future architectures will mean for the user community,” in *Parallel Computing: From Multicores and GPUs to Petascale*, B. Chapman et al., Eds. Amsterdam, The Netherlands: IOS Press, 2010, pp. 3–15.
6. R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, “Design of ion-implanted MOSFETs with very small physical dimensions,” *IEEE J. Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct. 1974.
7. The IBM Blue Gene TeamThe Blue Gene/Q Compute Chip, Hot Chips 23, 2011. [Online]. Available: http://www.hotchips.org/wp-content/uploads/hc_archives/hc23/HC23.18.1-manycore/HC23.18.121.BlueGene-IBM_BQC_HC23_20110818.pdf
8. J. T. Pawlowski, “Hybrid memory cube (HMC): Breakthrough DRAM performance with a fundamentally re-architected DRAM subsystem,” in *Hot Chips 23*. Boise, ID, USA: Micron Technology, Inc., 2011. [Online]. Available: http://www.hotchips.org/wp-content/uploads/hc_archives/hc23/HC23.18.3-memory-FPGA/HC23.18.320-HybridCube-Pawlowski-Micron.pdf
9. OpenMP ARBOpenMP Application Program Interface version 4.0, 2013. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
10. S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, “ND: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads,” in *Proc. Int. Symp. Perform. Anal. Syst. Softw.*, 2014, pp. 190–200.
11. D. P. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, “TOP-PIM: Throughput-oriented programmable processing in memory,” in *Proc. 23rd Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2014, pp. 85–98.
12. J. Jeddeloh and B. Keeth, “Hybrid memory cube: New DRAM architecture increases density and performance,” in *Proc. Symp. VLSIT*, 2012, pp. 87–88.
13. R. M. Russell, “The Cray-1 computer system,” *Commun. ACM*, vol. 21, no. 1, pp. 63–72, Jan. 1978.
14. P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. Van Hensbergen, and L. Zhang, “Mambo: A full system simulator for the PowerPC architecture,” *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 4, pp. 8–12, Mar. 2004.
15. M. Giampapa, T. Gooding, T. Inglett, and R. W. Wisniewski, “Experiences with a lightweight supercomputer kernel: Lessons learned from Blue Gene’s CNK,” in *Proc. ACM/IEEE Int. Conf. High Perform. Comput., Netw., SC*, Nov. 2010, pp. 1–10.
16. G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Aug. 1998.
17. J. Llosa, A. González, E. Ayguadé, and M. Valero, “Swing modulo scheduling: A lifetime-sensitive approach,” in *Proc. Conf. Parallel Architectures Compilation Tech.*, 1996, pp. 80–86.
18. J. E. Miller and A. Agarwal, “Software-based instruction caching for embedded processors,” *ACM SIGOPS Oper. Syst. Rev.*, vol. 40, no. 5, pp. 293–302, Dec. 2006.
19. J. J. Dongarra, P. Luszczek, and A. Petitet, “The LINPACK benchmark: Past, present and future,” *Concurrency Comput. Practice Experience*, vol. 15, no. 9, pp. 803–820, Aug. 2003.
20. DAXPY subroutine, Linear Algebra Package, LAPACK 3.5.0. [Online]. Available: http://www.netlib.org/lapack/explore-html/d9/dcd/daxpy_8f.html
21. R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, “Near-data processing: Insights from a MICRO-46 workshop,” *IEEE Micro*, vol. 34, no. 4, pp. 36–42, Jul./Aug. 2014.

Received August 15, 2014; accepted for publication September 3, 2014

Ravi Nair *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (nair@us.ibm.com).* Dr. Nair is a Distinguished Research Staff Member in the Exploratory System Architecture Group at the IBM T. J. Watson Research Center. His current interests include large-scale computer design, processor microarchitecture, and approximate computing.

Samuel F. Antao *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (sfantao@us.ibm.com).* Dr. Antao is a Postdoctoral Researcher in the Computer Architecture department at the IBM T. J. Watson Research Center. His interests include compiler code generation and optimization.

Carlo Bertolli *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (cbertol@us.ibm.com).* Dr. Bertolli is a Research Staff Member in the Advanced Compiler Technology Group at the IBM T. J. Watson Research Center. His interests include high-performance architectures, compilers, and applications.

Pradip Bose *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (pbose@us.ibm.com).* Dr. Bose is a Distinguished Research Staff Member at the IBM T. J. Watson Research Center. His interests include power-efficient and reliable processor/system architectures.

Jose R. Brunheroto *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (brunhe@us.ibm.com).* Mr. Brunheroto is a Senior Software Engineer the Data Centric Systems department at the IBM T. J. Watson Research Center. His interests include processor architecture and simulation of full systems including supercomputers.

Tong Chen *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (chentong@us. ibm.com).* Dr. Chen is a Research Staff Member in the Advanced Compiler Technologies Group at the IBM T. J. Watson Research Center. His interests include programming languages, compilers and computer architecture.

Chen-Yong Cher *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (chenyong@us.ibm.com).* Dr. Cher is a Research Staff Member at the T. J. Watson Research Center, where he has worked on performance, thermal, process variation, power, and reliability for microprocessors.

Carlos H. A. Costa *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (chcost@us.ibm.com).* Dr. Costa is a Research Staff Member in the Data Centric Systems department at the IBM T. J. Watson Research Center. His interests include parallel and distributed systems, high-performance computing, and analytics.

Jun Doi *IBM Research Division, Tokyo, Japan (doichan@jp.ibm.com).* Mr. Doi is a Research Staff Member in the deep computing and analytics area at IBM Research - Tokyo. His research interests include supercomputing, computer simulation, computer graphics, and visualization.

Constantinos Evangelinos *IBM Research Division, Cambridge, MA 02142 USA (cevange@us.ibm.com).* Dr. Evangelinos is a Research Staff Member in the Computational Science Center at the IBM T. J. Watson Research Center. His interests include parallel high performance computing and computer architecture.

Bruce M. Fleischer *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (fleischr@us.ibm.com).* Dr. Fleischer is a Research Staff Member in the VLSI (very-large-scale integration) Design department at the IBM T. J. Watson Research Center. His interests include processor architecture and design with an emphasis on designer productivity.

Thomas W. Fox *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (foxy@us.ibm.com).* Mr. Fox is a Senior Engineer and the manager of the Design Systems department at the IBM T. J. Watson Research Center. His interests lie in the architecture and design of CPUs, floating-point arithmetic and instruction units, and 3D graphics systems.

Diego Sanchez Gallo *IBM Research Brazil, 04007-900 São Paulo, Brazil (dsgallo@br.ibm.com).* Mr. Gallo is a Researcher in the Systems of Engagement and Insight department at IBM Research—Brazil, and a Ph.D. candidate at University of Sao Paulo. His interests include high-performance computing, computer networks, peer-to-peer computing, and crowdsourcing.

Leopold Grinberg *IBM Research Division, Thomas J. Watson Research Center, Cambridge, MA 02142 USA (leopoldgrinberg@us.ibm.com).* Dr. Grinberg is a Research Staff Member in High Performance Computing Applications and Tools Group. His interests include massively parallel computing, high-order methods, and particle simulations.

John A. Gunnels *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (gunnels@us.ibm.com).* Dr. Gunnels is a Distinguished Research Staff Member in the Data-Centric Systems department at the IBM T. J. Watson Research Center. His interests include code verification, graph algorithms, and hardware accelerators.

Arpith C. Jacob *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (acjacob@us.ibm.com).* Dr. Jacob is a Research Staff Member in the Advanced Compiler Technologies Group at the IBM T. J. Watson Research Center. His interests include parallelizing compilers and special-purpose accelerators.

Philip Jacob *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (jacobp@us.ibm.com).* Dr. Jacob is a researcher in the Data Centric Systems department at the IBM T. J. Watson Research Center. His research interests include supercomputing and 3D chip architecture and design.

Hans M. Jacobson *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (hansj@us.ibm.com).* Dr. Jacobson is a Research Scientist in the Computer Architecture department at the IBM T. J. Watson Research Center. His interests include power efficient microarchitectures, power modeling, power management, logic design, and advanced clocking techniques.

Tejas Karkhanis *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, New York 10598 (tkarkha@us.ibm.com).* Dr. Karkhanis is a Research Staff Member at the IBM Thomas J Watson Research Center. His research interests include various aspects of enterprise class and high-performance class computing systems and software.

C. Kim *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (kimchang@us.ibm.com).* Dr. Kim is a Research Staff Member at the IBM T. J. Watson Research Center. His interests include computational science and application optimization.

Jaime H. Moreno *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (jhmoreno@us.ibm.com).* Dr. Moreno's interests include processor architectures, application-specific and domain-specific architectures, data centric architectures, interaction among architectures and emerging technologies.

J. Kevin O'Brien *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (caomhin@us.ibm.com).* Mr. O'Brien is the manager of the Advanced Compiler Technology department. He has worked on parallelization, vectorization and optimization transformations in IBM compilers for over 30 years.

Martin Ohmacht *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (mohmacht@us.ibm.com).* Dr. Ohmacht is a Research Staff Member at the IBM T. J. Watson Research Center. His research interests include computer architecture, design and verification of multiprocessor systems, and compiler optimizations.

Yoonho Park *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (yoonho@us.ibm.com).* Dr. Park is a Research Staff Member in the Data Centric Systems department at the IBM T. J. Watson Research Center. His interests include operating systems, networks, and computer architecture.

Daniel A. Prener *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (prener@us.ibm.com).* Dr. Prener is a Research Staff Member in the Computing as a Service department of the IBM T. J. Watson Research Center. He has worked in computer architecture, compilers, and approximate computing.

Bryan S. Rosenburg *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (rosnbrg@us.ibm.com).* Dr. Rosenburg is a Research Staff Member in the Exascale System Software Group at the IBM T. J. Watson Research Center. His interests include operating-system support for very large systems.

Kyung Dong Ryu *LG Electronics, Software Platform Lab, Seoul, Korea (kd.ryu@lge.com).* Dr. Ryu is a Vice President of Research and Development in the Software Platform Laboratory at LG, and a former Manager of the Exascale System Software group at the IBM T. J. Watson Research Center. His interests include operating systems, system software, software platforms, Internet of Things for smart consumer electronics, data-centric systems, and cloud computing.

Olivier Sallenave *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (ohsallen@us.ibm.com).* Dr. Sallenave is a Research Scientist in the Advanced Compiler Technologies department at the IBM T. J. Watson Research Center. His interests include compiler optimization and programming models.

Mauricio J. Serrano *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (mserrano@us.ibm.com).* Dr. Serrano is a Senior Software Engineer in the Computer Sciences department at the IBM T. J. Watson Research Center. His interests include computer architecture, performance modeling and compiler technology.

Patrick D. M. Siegl *Chair for Chip Design for Embedded Computing (C3E), Technische Universität Braunschweig, 38106 Braunschweig, Germany (psiegl@c3e.cs.tu-bs.de)*. Mr. Siegl is a PhD candidate in the Department of Chip Design for Embedded Computing at the Technische Universität Braunschweig (Germany). Simulation of processor architectures, low-power reconfigurable computing, as well as low-level algorithm optimizations targeting accelerators are his main fields of interest.

Krishnan Sugavanam *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (sugavana@us.ibm.com)*. Mr. Sugavanam is a Senior Engineer in the Data Centric Systems group at the IBM T. J. Watson Research Center. His primary interest is in the field of computer architecture with emphasis on cache design.

Zehra Sura *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (zsura@us.ibm.com)*. Dr. Sura is a Research Staff Member in the Advanced Compiler Technologies group at the IBM T. J. Watson Research Center. Her interests include memory access optimizations and the analysis and transformation of programs for parallel processing.