PROGRAMMING MULTIPROCESSORS WITH EXPLICITLY MANAGED MEMORY HIERARCHIES

Scott Schneider and Jae-Seung Yeom, *Virginia Tech*
Dimitrios S. Nikolopoulos, *FORTH-ICS and University of Crete*

A study of two applications programmed using three models of varying complexity reveals that implicit management of locality can produce code with performance comparable to code generated from explicit management of locality.

Multicore processors with explicitly managed memory (EMM) hierarchies originated in the domain of games and graphics[1,2] and are now emerging as general-purpose high-end computing platforms. More recently, processor vendors for mainstream computing markets such as Intel and AMD have introduced similar designs.[3] All of these processors have data-parallel components as accelerators. This acceleration is achieved through multiple scalar or single-instruction, multiple-data (SIMD) cores, high on-chip bandwidth, and explicit data transfers between fast local memories and external dynamic RAM (DRAM). Explicit data transfers enable programmers to use optimal caching policies and multiple streaming data buffers that allow overlapping computation with data transfer latency.[4]

Managing data locality in EMM multicore processors requires tradeoffs in performance, code complexity, and optimization effort. Multicore processors based on coherent hardware-managed caches provide the abstraction of a single shared address space. This abstraction frees programmers from having to explicitly manage data as it moves through the memory hierarchy and between cores. The hardware automatically synchronizes data in main memory and across the caches in each core so that all cores have a consistent view of shared memory. In contrast, software-managed local memories introduce disjoint address spaces that programmers are responsible for keeping consistent. Because programmers must explicitly manage data locality, they can decide when to place data in local memories, what data to replace, and what the data layout is in local memories, which can differ from the layout of data in off-chip DRAM.[5]

We used the Cell Broadband Engine processor as an experimental testbed to analyze support for expressing parallelism and locality in programming models for multiprocessors with EMM hierarchies. We implemented two applications using three programming models of varying complexity to explore abstractions for specifying the working sets of parallel tasks, controlling task granularity, and scheduling data transfers to and from local memories. The two scientific parallel applications, PBPI[6] and Fixedgrid,[7] stress both computational power and memory bandwidth.

 Published by the IEEE Computer Society

In comparing implicit versus explicit programming models for managing locality, we found that programming models with implicit locality management via compiler and runtime support can increase programmer productivity: Programmers write less code, maintain a high-level view of locality, and rely more on the compiler and the runtime environment for parallelism and locality management. However, explicit management of locality is often necessary for performance optimization, and the explicit control of private address spaces enforced in some programming models helps in this direction.

CELL PROGRAMMING CHALLENGES

Figure 1 shows the Cell processor's architecture. The power processing element (PPE) is a traditional PowerPC core with hardware-managed caches and vector processing extensions. The eight synergistic processing elements (SPEs) are 128-bit vector cores with software-managed caches. The SPEs can communicate with one another over high-bandwidth buses, and the PPE and SPEs use the same buses to communicate with the memory interface controller. Compared to programming for homogeneous multicore processors and shared-memory symmetric multiprocessors (SMPs), programming for the Cell presents several unique challenges.

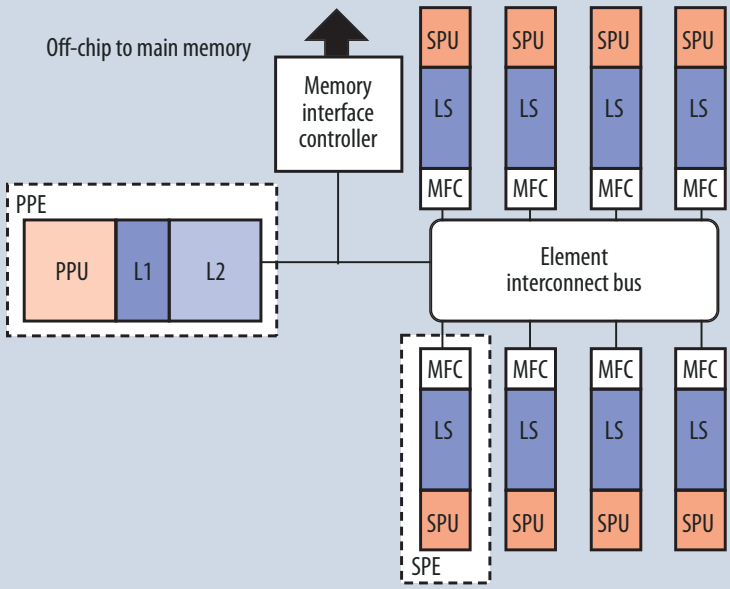

Figure 1. Cell processor architecture. The power processing element (PPE) is a traditional PowerPC core with hardware-managed caches and vector processing extensions. The synergistic processing elements (SPEs) have a programmer-controlled local store (LS) that communicates with the memory interface controller through the memory flow controller (MFC).

Local memory spaces

Effective use of the Cell requires offloading as much computation as possible to the SPEs. The difficulty of this requirement is that the SPEs are divorced from the normal memory hierarchy. Each SPE has a local static RAM (SRAM) store of 256 Kbytes, and this is the only memory it can directly address. Programmers must transfer data from main memory explicitly through direct memory accesses (DMAs). Consequently, they must know the memory access patterns in their application and move data in and out as needed.

Small local store

The 256-Kbyte local store associated with each SPE contains all of the code and all of the data the SPE uses. Consequently, there is a tradeoff: The more code that is loaded into an SPE, the less data it can operate on. Because this space also contains stack frames, recursive functions are more limited in their depth of recursion than in an architecture with a conventional hardware-managed memory hierarchy.

The limited amount of data that can fit into the local store requires streaming to achieve good performance. Programmers must be able to anticipate what data they will need for future computations and initiate DMAs for this data while the other data is in use for computation. Overlapping computation and communication can hide the latency associated with DMAs, a necessity for sustained performance.

Strided access

A single DMA only transfers contiguous data. The Cell has no architectural support for strided access to main memory, which is required for accessing, say, the columns in a matrix that is stored in row-major format. To best transfer noncontiguous data to and from main memory, programmers must construct DMA lists. Each entry in a DMA list specifies a separate DMA, and programmers must ensure that the memory address for each subsequent entry in the list adheres to the stride they want.

Data alignment

All DMAs of less than 16 bytes must be naturally aligned in both main memory and in the SPE's local store—that is, transfers of 1, 2, 4, and 8 bytes must be aligned on a 1-, 2-, 4-, or 8-byte boundary, respectively. Transfers larger than or equal to 16 bytes must be aligned on a 16-byte boundary, but for best performance data should be aligned on a cache line (128 bytes).

```
#pragma cell reduction(+: double l = lnL)
        private(double* freq = model->daStateFreqs,
                int N4 = N / 4)
        shared(double* sroot = tree->root->siteLike,
               int* weight = g_ds.compressedWeight.v)
{
  int i;
  for (i = 0; i < N4; i++) {
    int j = i * 4;
    double temp;
    temp = sroot[j]*freq[0] + sroot[j+1]*freq[1] +
           sroot[j+2]*freq[2] + sroot[j+3]*freq[3];
    temp = log(temp);
    l +=  weight[i] * temp;
  }
}
```

(a)

```
void task<leaf> Sum::Leaf(in double A[L],
                          inout double B[L])
{
  B[0] += A[0];
}

void task<inner> Likelihood::Inner(in double sroot[N],
 in double freq[M], in int weight[P], out double lnL[L])
{
  tunable T;
  mapreduce(unsigned int i = 0 : (N+T-1)/T) {
    Likelihood(sroot[i*T:T], freq[0:3],
               weight[i*T/4:T/4], reducearg<lnL,Sum>);
  }
}

void task<leaf> Likelihood::Leaf(in double sroot[N],
                  in double freq[M], in int weight[P],
                  inout double lnL[L])
{
  unsigned int i,j;
  double temp;

  for (i = 0; i < P; i++) {
    j = i * 4;
    temp = sroot[j] * freq[0] + sroot[j+1] * freq[1] +
           sroot[j+2] * freq[2] + sroot[j+3] * freq[3];
    temp = log(temp);
    lnL[0] += weight[i] * temp;
  }
}
```

(b)

Figure 2. Two examples of likelihood calculation code in PBPI: (a) Cellgen and (b) Sequoia.

PROGRAMMING MODELS

Our study implemented two applications using two high-level programming models, Cellgen and Sequoia.[8] We contrasted these models with programming for the Cell directly using the IBM Software Development Toolkit (SDK) 3.0.

Cellgen

Cellgen implements a programming model similar to OpenMP.[9] Figure 2a shows an example of Cellgen code taken from PBPI.

Programmers identify data-parallel sections of their code, then annotate these sections to mark them for parallel execution and data handling. This model provides the abstraction of a shared-memory architecture and an indirect abstraction of data locality. While the parallel sections are annotated with their datasets, the code inside these regions is not; it is written in the same way it would be for a sequential program.

The programmer annotates the data as private or shared, using the same keywords as in OpenMP. Private variables follow OpenMP semantics: They are copied into local stores using DMAs, and each SPE gets a local copy of the variable. Shared variables are further classified using reference analysis as in, out, or I/O variables. This classification departs from OpenMP semantics and serves as the main vehicle for managing locality in the Cell. In data needs streaming into the SPE's local store, out data needs streaming out of local stores, and I/O data needs streaming both in and out of local stores. Classification is transparent to the programmer and internal to the compiler. The Cellgen compiler and runtime system manages locality by triggering and dynamically scheduling the associated data transfers. Note that the abstraction of shared memory is not implemented on top of a coherent software cache.[10] The compiler and runtime system transparently manage coherence and locality.

Streaming data is paramount for two reasons: The local stores are small, so they can only contain a fraction of parallel-task working sets, and the DMA latency is significant. Overlapping DMAs with computation is necessary to achieve high performance. Data classified by the compiler as in or out is streamed using double buffering, while I/O data is streamed using triple buffering. The number of states a variable can be in determines the depth of buffering. In data can be either simultaneously streaming in or computing; out data can be either simultaneously computing or streaming out; I/O data can be simultaneously streaming in, computing, or streaming out. The Cellgen compiler creates a buffer for each of these states. The goal is to maximize computation/DMA overlap by having data in two (in and out) or three (I/O) states simultaneously.

SPEs operate on independent loop iterations in parallel, as scheduled by Cellgen. It is the programmer's responsibility to ensure that loop iterations are in fact independent, which is also the case in OpenMP.

Sequoia

The second class of programming models that we considered expresses parallelism through explicit task and data subdivision. A representative of this model is Sequoia, in which programmers construct trees of dependent tasks, with the inner tasks calling tasks further down the tree. Final computation occurs in leaf tasks. At each level, Sequoia decomposes the data and copies it to the child tasks as specified, which enforces the model that each task has a private address space. Figure 2b repeats the code example of Figure 2a using Sequoia.

Sequoia strictly enforces locality because tasks can only reference local data. In this manner, there is a direct mapping of tasks to the Cell architecture in which the SPE local storage is divorced from the memory hierarchy. By providing a programming model in which tasks operate on local data, and providing abstractions to subdivide data and pass it on to subtasks, Sequoia can abstract away the underlying architecture. Programmers explicitly define data and computation subdivision through an architecture-independent notation. Using these definitions, the Sequoia compiler generates code for data subdivision and transfer for the specific architecture.

Comparing the two examples in Figure 2 shows that the same computation takes significantly less code to express using the Cellgen model. However, the current Cellgen model is designed for data-parallel computations. While programming with Sequoia takes more code, it provides a more expressive, general language. The tradeoff is that programming models with implicit locality management are more concise, but explicit models can solve a broader class of problems.

Cell SDK

Cell SDK exposes Cell architectural details to programmers. It provides libraries for low-level, Pthread-style thread-based parallelization, and sets of DMA commands based on a get/put interface for managing data transfers.

Programming in Cell SDK is analogous to, if not harder than, programming with the message passing interface (MPI) or Pthreads on a typical cluster or multiprocessor. Cell SDK programmers must parallelize their program explicitly with threads, implement application-specific scheduling loops in each thread, and manually schedule all data transfers. Hence, programmers need a deep understanding of both thread-level parallelization and the Cell hardware.

While programming models can transparently manage data transfers, Cell SDK requires programmers to explicitly identify and schedule all data transfers. Further, programmers are solely responsible for synchronizing threads running on different cores, maintaining data coherency, aligning data, and setting up and sizing buffers to achieve computation/communication overlap. However, hand-tuned parallelization also has well-known advantages: Programmers with insight into the parallel algorithm and the Cell architecture can maximize locality, eliminate unnecessary data transfers, and optimally schedule data and computation on cores.

APPLICATIONS

For our analysis we used two scientific applications, Fixedgrid and PBPI.

Programming models with implicit locality management are more concise, but explicit models can solve a broader class of problems.

Fixedgrid

Fixedgrid is an atmospheric modeling application that describes chemical transport via third-order upwind-biased advection discretization and second-order diffusion discretization.

To calculate mass flux in a 2D domain, Fixedgrid must calculate a two-component wind vector, horizontal diffusion tensor, and concentrations for every species of interest. In a domain of $N \times M$, the calculation's complexity is $O(NM)$. To determine ozone concentrations in a 600 $\times$ 600 domain as in our experiments, Fixedgrid used approximately 3,240,000 double-precision values (24.7 Mbytes) to calculate 1,080,000 values (8.24 Mbytes) at each time step. These calculations access noncontiguous data, which requires special handling on the SPEs in the form of DMA lists. Reordering the data on the PPE so that the SPE can access contiguous data is possible but hinders performance significantly.

PBPI

PBPI is a parallel implementation of the Bayesian phylogenetic inference method, which constructs phylogenetic trees using a Markov-chain Monte Carlo sampling method. On the Cell, calculation of the likelihood values (as shown in Figure 2) for each generation is distributed among all SPEs. Where N is the size of the data used in all calculations, all offloaded algorithms have complexity $O(N)$.

Our experiments used a dataset of 107 taxa with 19,989 nucleotides for a tree. Three computational loops are called for a total of 324,071 times and account for most of the program's execution time. The first loop accounts for

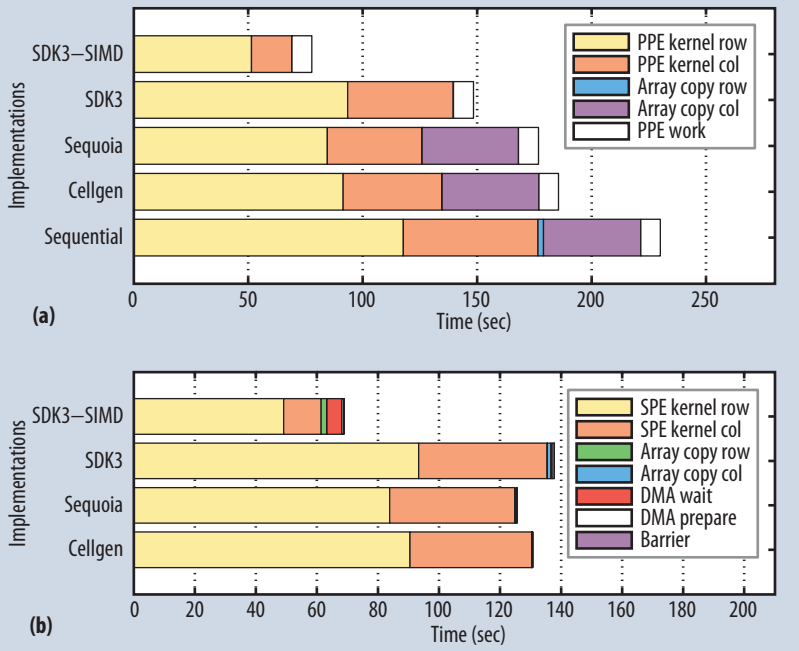

Figure 3. Fixedgrid implementation performance. (a) PPE timing profile. "PPE kernel" measures the time to complete each offloaded function. "Array copy" is the time spent on copying arrays from main memory for each discretization function. "PPE work" includes array initialization and file I/O time. (b) SPE timing profile. "SPE kernel" is the time spent on core computation excluding DMA and array copying overheads. "Array copy" is the time spent on copying arrays to the SPE local store. "DMA wait" is data transfer time not overlapped with computation in addition to the time for checking for DMA completion. "DMA prepare" is the time to prepare DMA addresses and lists along with the time to queue DMA commands.

88 percent of the calls and requires 1.2 Mbytes to compute a result of 0.6 Mbytes; the second loop accounts for 6 percent of the calls and requires 1.8 Mbytes to compute a result of 0.6 Mbytes; and the third loop also accounts for 6 percent of the calls and requires 0.6 Mbytes to compute a result of 8 bytes.

PERFORMANCE ANALYSIS

We compared the performance of each implementation of both applications, as shown in Figures 3 and 4. The experimental environment was a Sony PlayStation 3 running Linux with a 2.6.24 kernel and Cell SDK 3.0. On a PS3 running Linux, only six SPEs are available to user-land code. Each data point represents the best of 40 runs; we found this more reproducible and representative than the average.

The original SDK3 implementations have vectorized kernels. Because both Cellgen and Sequoia are primarily concerned with locality management, they do not produce SIMD code. However, autovectorization is not precluded by design or engineering considerations.[11] To control for vectorization, we compared the Cellgen and Sequoia implementations against SDK3 implementations with and without SIMD kernels.

Fixedgrid

Fixedgrid has two types of computational kernels: row and column discretization. The former requires row data from a contiguous region of memory, and the latter requires column data from a noncontiguous region of memory. For each time-step iteration, the application calls the former twice as much as the latter. The sequential implementation maintains a transposed copy of each matrix. The application copies the values of each transposed matrix as a whole from the original matrix before the column discretization kernel. After the computation, it copies the values back as a whole to the original matrix.

Fixedgrid's SDK3 implementations use DMA lists to transfer columns of data from matrices. DMA lists are the only mechanism that Cell provides to perform scatter/gather operations. Because DMA lists perform best when each list entry is at least 16 bytes, the SDK3 implementation transfers two columns at once. In the SIMD implementation, vector operations simultaneously work on the interleaved columns. Unlike the other Fixedgrid implementations, the SDK3 implementation does not require column data to be reordered on the PPE or SPE.

The Cellgen and Sequoia versions do not support array-column accesses. Those implementations instead rearrange noncontiguous data by performing array transpositions on the PPE. Transposes introduce copying overhead on the PPE, as Figure 3a shows.

Overall, we found that the lack of support for automatic generation of DMA scatter/gather operations is the key reason for the performance gap between the high-level programming models and the hand-tuned implementation of Fixedgrid.

PBPI

Applications with a fine granularity of parallelism are sensitive to the size and frequency of DMAs between the SPE and main memory. Because PBPI is such an application, we experimented with different buffer sizes, as Figure 4a shows.

The major factors that influence performance in all three cases are the computational kernel, which is either manually written or generated for the SPE; the overhead of DMA-related operations; SPE overheads imposed by the programming model runtime; and the overhead of signaling between PPE and SPE, as Figure 4b shows.

Vectorizing the computational kernel in the SDK3 implementation of PBPI improves performance by 20 percent. The computational kernel is the only significant difference between the SDK3 and Cellgen implementations; the communication overheads are similar. Cellgen's computational kernel in PBPI is faster than Sequoia's because Sequoia relies on a compiler-generated data structure for generalized array access and incurs dereferencing overhead on each access.

Overlapping DMA latency and computation is important for performance with applications like PBPI. The best overlap is achieved with different buffer sizes in the three implementations. The DMA wait overhead is minimal when the buffer size is 2 Kbytes for the SDK3 implementation, while the overhead is minimal at 4 Kbytes for Cellgen and 8 Kbytes for Sequoia. This difference occurs because the programming models provide different abstractions for accessing arrays with their own associated overheads. Hence, the costs of their computational kernels and data transfers are different. Optimal performance is achieved when the sum of the computation costs and all related data transfer overheads is minimal.

In the SDK3 version of PBPI, the epilogue—which includes the computation and communication for the final iterations that cannot be unrolled—is inefficient: One DMA is issued for each iteration. In Cellgen and Sequoia, one DMA is issued for the entire remainder of the data. This inefficiency explains why the SDK3 implementation performs worst with the largest buffer size.

Sequoia has additional overheads on the SPE including barriers, reductions, and extra copies of scalar variables, which are artifacts of the Sequoia compilation process. Such overheads become noticeable when there is a large number of offloaded function calls. There are 324,071 offloaded function calls in a PBPI run, while there are only 2,592 in a Fixedgrid run.

At the end of a leaf task, Sequoia sometimes requires the SPEs to synchronize at a barrier. In contrast, Cellgen does not require such a barrier among SPEs. Instead, each SPE waits until all outstanding DMAs have completed and then sets a status value in its local store to indicate completion. The PPE polls these values from each SPE directly, waiting for all SPEs to complete. Cellgen relies on a similar method to collect the result from SPEs during reduction operations, while Sequoia relies on DMAs and barriers among SPEs.

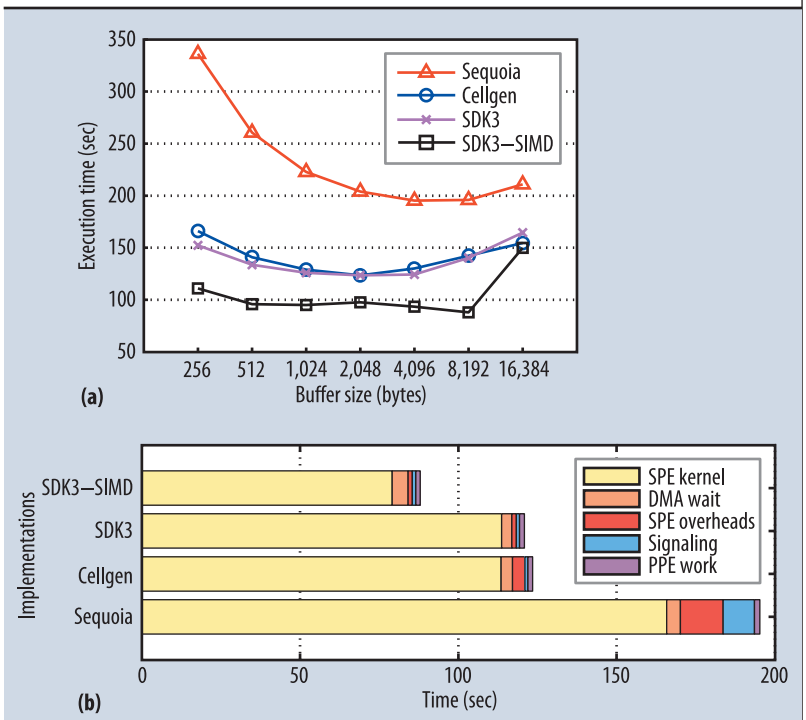

Figure 4. PBPI implementation performance using six SPEs. (a) Total execution time as a function of DMA buffer size. (b) Comparison of the best cases from each implementation. "SPE kernel" accounts for the time to run hand-coded or generated the SPE kernel. "DMA wait" is the DMA data transfer cost that is not overlapped with computation in addition to the time for checking the completion of DMA commands. "SPE overheads" account for DMA preparation, barriers, and other programming-model-specific overheads, which vary depending on their implementations. "Signaling" accounts for overhead from signaling between PPE and SPE.

Our study revealed that implicit management of locality can produce code with performance comparable to code generated from explicit management of locality. Generating such code requires adequate compiler and runtime support, but it also reduces the programming effort as measured by lines of code.

Cellgen, a programming model that uses private/shared data classification clauses as the sole mechanism for managing locality, demonstrates this point. The compiler and runtime system can be extended to integrate more scheduling algorithms—such as dynamic, interleaved, or work stealing—to further the programmer's task in managing granularity and scheduling. Nevertheless, tuning of data transfers by hand is still necessary for optimization of specific data access patterns, and models for which the programmer explicitly manages locality help in this regard.

As expected, we found that programming models are sensitive to data transfer overheads imposed by the implementation of their abstractions. The implementations need to mask both overheads incurred by abstractions and the

actual data transfer overheads with proper scheduling and data distribution.

Acknowledgments

This research was supported by grants from the NSF (CCR-0346867, CCF-0715051, CNS-0521381, CNS-0720750, CNS-0720673), the US Department of Energy (DE-FG02-06ER25751, DE-FG02-05ER25689), IBM (VTF-874197), and the European Commission (SARC-IP-27648, MCF-IRG-224759).

References

1. T. Chen et al., "Cell Broadband Engine and Its First Implementation—A Performance View," *IBM J. Research and Development*, Sept. 2007, pp. 559-572.
2. J.D. Owens et al., "GPU Computing," *Proc. IEEE*, May 2008, pp. 879-899.
3. P.H. Wang et al., "EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-core Multithreaded System," *Proc. 2007 ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 07), ACM Press, 2007, pp. 156-166.
4. T. Chen et al., "Optimizing the Use of Static Buffers for DMA on a CELL Chip," *Proc. 19th Int'l Workshop on Languages and Compilers for Parallel Computing* (LCPC 06), LNCS 4382, Springer, 2007, pp. 314-329.
5. A.M. Aji et al., "Cell-SWat: Modeling and Scheduling Wavefront Computations on the Cell Broadband Engine," *Proc. 5th Conf. Computing Frontiers* (CF 08), ACM Press, 2008, pp. 13-22.
6. X. Feng, K.W. Cameron, and D.A. Buell, "PBPI: A High Performance Implementation of Bayesian Phylogenetic Inference," article no. 75, *Proc. 2006 ACM/IEEE Conf. Supercomputing* (SC 06), ACM Press, 2006.
7. J.C. Linford and A. Sandu, "Optimizing Large Scale Chemical Transport Models for Multicore Platforms," *Proc. 2008 Spring Simulation Multiconference* (SpringSim 08), Soc. for Computer Simulation Int'l, 2008, pp. 369-376.
8. K. Fatahalian et al., "Sequoia: Programming the Memory Hierarchy," article no. 83, *Proc. 2006 ACM/IEEE Conf. Supercomputing* (SC 06), ACM Press, 2006.
9. OpenMP Architecture Review Board, "OpenMP Application Program Interface, v. 3.0," May 2008; www.openmp.org/mp-documents/spec30.pdf.
10. J. Balart et al., "A Novel Asynchronous Software Cache Implementation for the Cell-BE Processor," *Proc. 20th Int'l Workshop on Languages and Compilers for Parallel Computing* (LCPC 07), LNCS 5234, Springer, 2007, pp. 125-140.
11. D. Nuzman, I. Rosen, and A. Zaks, "Auto-Vectorization of Interleaved Data for SIMD," *ACM SIGPLAN Notices*, June 2006, pp. 132-143.

Scott Schneider *is a PhD candidate in the Computer Science Department at Virginia Tech and a member of the Parallel Emerging Architecture Research Laboratory (PEARL). His research interests include high-performance computing, systems, and programming languages. Schneider received an MS in computer science from The College of William and Mary. He is a member of the ACM and the IEEE. Contact him at scschnei@cs.vt.edu.*

Jae-Seung Yeom *is a PhD candidate in the Computer Science Department at Virginia Tech and a member of PEARL. His research interests include high-performance computing, intelligent systems, and information security. Yeom received an MS in information networking from Carnegie Mellon University. He is a member of the IEEE ComSoc Communications and Information Security Technical Committee. Contact him at jyeom@cs.vt.edu.*

Dimitrios S. Nikolopoulos *is an associate professor of computer science at the University of Crete and a researcher at the Institute of Computer Science (ICS) of the Foundation for Research and Technology—Hellas (FORTH). His research interests include software, hardware, and tools for scalable and energy-efficient parallel computation. Nikolopoulos received a PhD in computer engineering from the University of Patras. He is a member of the ACM and the IEEE. Contact him at dsn@ics.forth.gr.*