Computer Sciences Department

MapReduce for the Cell B.E. Architecture

Marc de Kruijf
Karthikeyan Sankaralingam

Technical Report #1625

October 2007

MapReduce for the Cell B.E. Architecture

Marc de Kruijf   Karthikeyan Sankaralingam
Vertical Research Group
Department of Computer Sciences, University of Wisconsin–Madison
{dekruijf,karu}@cs.wisc.edu

Abstract

MapReduce is a simple and flexible parallel programming model proposed by Google for large scale data processing in a distributed computing environment [4]. In this paper, we present a design and implementation of MapReduce for the Cell architecture. This model provides a simple machine abstraction to users, hiding parallelization and hardware primitives. Our runtime automatically manages parallelization, scheduling, partitioning and memory transfers. We study the basic characteristics of the model and evaluate our runtime's performance, scalability, and efficiency for micro-benchmarks and complete applications. We show that the model is well suited for many applications that map well to the Cell architecture, and that the runtime sustains high performance on these applications. For other applications, we analyze runtime performance and describe why performance is less impressive. Overall, we find that the simplicity of the model and the efficiency of our MapReduce implementation make it an attractive choice for the Cell platform specifically and more generally to distributed memory systems and software-exposed memories.

1. Introduction

The Cell processor is capable of an order of magnitude performance improvement over conventional processors, with a peak performance rating of 256 GFLOPS at 4 GHz [8, 9, 10]. However, harnessing this potential is challenging for three reasons. First, programmers must write explicitly multi-threaded code to utilize the eight SPE (Synergistic Processing Element) cores in addition to the main PPE (Power Processing Element) core. Second, SPE local memory is software-managed, requiring programmers to explicitly orchestrate all reads and writes to and from the global memory space. Third, the SPEs are statically scheduled SIMD cores, requiring data-level parallelism in the application to achieve high performance.

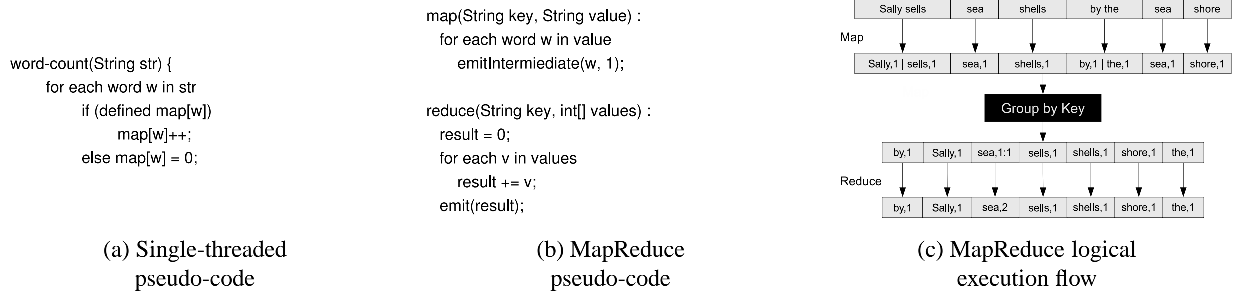
To address these programming challenges, we present a design and implementation of MapReduce for the Cell processor. MapReduce was initially proposed by Google for large scale data processing in a distributed computing environment [4] and the model has recently been ported to shared memory multiprocessor systems [22]. The MapReduce model provides a simple programming abstraction that hides low-level details from the programmer using a runtime tuned to the underlying architecture. The runtime system we implement for the Cell processor takes care of partitioning the input data, scheduling the program's execution across the cores, managing memory transfers between global and SPE memory, and handling inter-core communication and synchronization. Users simply specify a *Map* function that processes data to generate a set of intermediate key/value pairs, and a *Reduce* function that processes all intermediate values associated with the same intermediate key. This allows programmers without any experience with parallel systems to easily program for the Cell processor.

Implementing an efficient MapReduce runtime for the Cell processor has three main challenges compared to a homogeneous, shared memory multi-processor implementation. First, memory must be explicitly managed and transferred between global and SPE memory. Second, since the SPEs, which execute the Map and Reduce functions, have software-managed memories, the runtime is responsible for overlapping computation with memory transfers as much as possible. Third, between the Map and Reduce phases, there is a logical grouping phase that must be effectively distributed across the SPEs.

Our particular implementation tackles the memory management problem by pre-allocating Map and Reduce output regions for bulk DMA transfers. We parallelize computation with memory transfers by double-buffering and streaming data whenever possible. Finally, we implement the logical grouping using a two-step process of partitioning and sorting. Overall, we decompose the execution flow into work units, or tasks, and schedule dependent work units as others work units complete. We use a priority work queue to implement load balancing and use the priorities to control the execution flow to avoid bottlenecks.

In this paper, we provide a detailed discussion on the design of the runtime and evaluate performance. The three main contributions of this paper are:

- A runtime that allows programmers to focus solely on the computation component of their application, hiding communication, buffer management, synchronization, and scheduling complications. The runtime is available for



(a) Single-threaded pseudo-code

(b) MapReduce pseudo-code

(c) MapReduce logical execution flow

Figure 1. MapReduce example illustration.

public download at `sourceforge.net/projects/mapreduce-cell`.

- A characterization of the efficiency and scalability of the runtime across the application space using a suite of micro-benchmarks.
- A demonstration of the productivity and performance benefits of the model and runtime. A set of complete applications implemented using the model are no more complex than their corresponding serial implementations, and applications perform on average 2.7 times better and up to 8.6 times better on the Cell processor than on an optimized Intel Core2[1].

The remainder of this paper is organized as follows. In section 2, we explain the MapReduce model. Section 3 presents our API and design, and discusses implementation. Section 4 presents results covering performance, scalability, and productivity. Section 5 discusses future enhancements. Section 6 presents related work, and section 7 concludes.

2. The MapReduce Model

Dean and Ghemawat describe the MapReduce programming model as follows [4]:

> The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the MapReduce library expresses the computation as two functions: *Map* and *Reduce*.
>
> *Map*, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key and passes them to the *Reduce* function.
>
> The *Reduce* function, also written by the user, accepts an intermediate key and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per *Reduce* invocation.

The canonical MapReduce application example is that of counting the number of words in a set of documents. Figures 1a–c provide a simple illustration. Using MapReduce, the input is divided into regions, and the Map function is performed on each region in parallel. The Map function outputs each word in the given region (the key) and the number of times the word occurs in that region (the value). Numbers (values) corresponding with the same word (key) are then grouped together and passed to the Reduce function. The Reduce function sums the number of occurrences of a word in each region to output a final count. Similar to Map, Reduce invocations are independent and can proceed in parallel.

3. MapReduce for Cell

The MapReduce model has been implemented for two architectures. Google's MapReduce runtime targets distributed clusters [4] and Stanford's Phoenix runtime targets shared memory multi-processors [22]. Since the Cell architecture is a distributed memory architecture, our high-level design resembles Google's design, while the granularity of operations is similar to Phoenix.

In Google's MapReduce, a partition is sorted and reduced all on the same node. Our implementation decouples sorting and reduce to enable better load balancing. This makes sense on the Cell platform since local store sizes are small, communication is fast, and bandwidth is abundant. Beyond that, specifics of Google's design are unavailable, and our detailed runtime description contributes uniquely to the literature.

In Phoenix, the Map and Reduce work is distributed across the different cores and hardware caching exploits locality. For grouping values by key, output keys are first hashed and pointers to keys and values are then inserted into a hashed binary search tree, thereby eliminating the need for sorting and copying memory. These techniques are not viable for the Cell processor since manipulating large, recursive graph structures is extremely complex and awkward with software-managed memories. We develop a set of novel techniques to implement the runtime for the Cell exploiting its strengths: global coordination via the PPE and fast DMA transfers with overlapping computation.

Our runtime uses extensive multi-threading, running primarily on the PPE, while the Map and Reduce functions map naturally to the SPEs. The PPE runtime schedules work for the different SPEs, manages data structures, and performs all memory management. The main responsibility of the SPEs

[1] Applications are not tuned for SIMD on either platform.

`void MapReduce_exec(MapReduce_Specification specification);` The exec function initializes the MapReduce runtime and executes MapReduce according to the user specification.

`void MapReduce_emitIntermediate(void **key, void **value);`
`void MapReduce_emit(void **value);`
These two functions are called by the user-defined *Map* and *Reduce* functions, respectively.

Table 1. The MapReduce for Cell API.

is to run user code and assist in the grouping step, which is accomplished by a two-step process of partitioning and sorting. In this section, we describe the API, the runtime design, and discuss implementation highlights.

3.1 The Cell MapReduce API

Our Cell MapReduce implementation is written in C. Table 1 shows the interface presented to the user. There are also three MapReduce data structures – a runtime specification structure, a key/value array structure, and a list of arrays structure – and associated functions for creating, manipulating, and destroying these data structures[2]. The programmer simply specifies the program, including the number of SPEs, using this interface, and the runtime completely handles parallelization to the different SPEs and memory management.

The primary difference from other APIs is that users do not dynamically allocate output in the Map and Reduce functions. Instead, the runtime provides pointers to local output regions, providing three key benefits: (1) it shifts the burden of explicit memory management away from the programmer and entirely to the runtime, (2) it enables high performance since the runtime can perform correct memory alignment and efficient transfers, and (3) it relieves the overhead of dynamic memory allocation in the Map and Reduce functions. As a result, `emit` and `emitIntermediate` take references to pointers as arguments, as shown in Table 1, and modify the pointers to point to pre-allocated memory. It is then the responsibility of the application to provision this memory. This is a key enhancement in our API compared to the Phoenix API.

3.2 Design Overview

Our MapReduce design consists of five stages: *Map*, *Partition*, *Quick-sort*, *Merge-sort*, and *Reduce*. Figure 2a shows the overall flow of execution, and Figure 2b shows the flow applied to the example word count application. Program execution starts with the Map stage, triggered by a main runtime control thread. This stage takes as input a single, logical array of key/value pairs and produces one logical array of values, and one logical array of keys with pointers to values. The Partition, Quick-sort, and Merge-sort stages then group identical keys together to produce a set of partitions sorted by key. When complete, each partition contains a contiguous array of intermediate values for each key. Finally, the Reduce stage takes as input a partition and applies the user-defined Reduce function to each key and associated values to produces one logical output array of key/value pairs.

The Map and Reduce stages, which execute user code, run exclusively on the SPEs. The three intermediate stages execute on both the SPEs and the PPE. We describe each stage in detail below.

Stage 1: Map

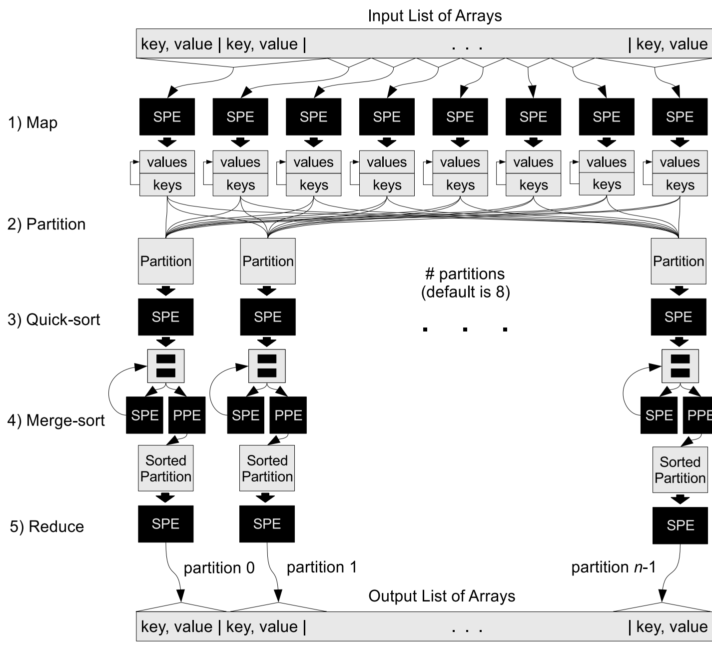
During Map, the SPEs execute the user-specified Map function on the input data and produce a set keys and a set of values, with keys containing pointers to their values. Runtime code on the PPE initiates the execution of this phase and orchestrates data movement to feed the SPE local memories. The input is divided into eight logical groups (one per SPE) of approximately equal size using a simple heuristic. Output buffers (one per SPE) are then pre-allocated in global memory, and pointers to the input and output buffers are passed to each SPE. The SPEs stream data into their local memories using DMA, process the data, and write results into a local output buffer. Since the size of output produced by Map cannot be statically determined, runtime monitoring code is appended to the Map function to check when this buffer fills up. Similarly, the input to an SPE can be larger than the local input buffer available in the SPEs local memory, so SPE runtime code also orchestrates the batched transfer of data into the local memories. When an SPEs local output buffer is full, the contents are transferred to the pre-allocated output buffer in shared memory. Figure 2c shows the memory transfers. The SPE notifies the PPE that the output buffer is ready and receives a pointer to a newly allocated output buffer to continue processing. All communication and synchronization is orchestrated through the Cell's mailbox and signal notification mechanisms.

Optimizations: In our implementation, the SPE's input and output buffers are double-buffered. Furthermore, there are always two pre-allocated output buffer in shared memory so that as soon as one buffer is full, a pointer to the next one is ready.

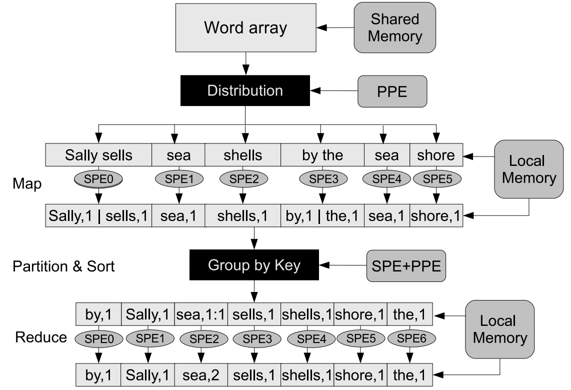
Stage 2: Partition

Partition groups identical keys into the same partition by distributing Map output keys into a set of partitions as shown in Figures 2a and 2c. As SPE output buffers are written to shared memory, the PPE iterates through each key and

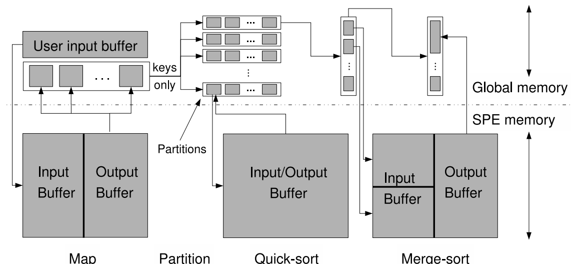
[2] The full API is documented in the header files `include/mapReduce.h` and `include/mapReduce_spu.h` distributed with the runtime.



(a) MapReduce execution flow.



(b) Execution flow applied to example of Figure 1c.



(c) MapReduce memory management.

Figure 2. MapReduce for the Cell architecture.

copies it into the partition pointed to by the key's hash value, previously computed by the SPE.

Optimizations: First, to exploit locality, the SPE output buffers are segregated into separate key and value regions. Second, partitions are linked-lists of buffers sized to fit in the local stores of the SPEs for in-place sorting, allowing Quick-sort to proceed in parallel with Partition as individual buffers become full. Third, while hashing is significantly more efficient at grouping keys than sorting, providing user control over partitioning (the number and the hashing) is desirable since different applications will partition optimally in different ways. Hence, the API allows the user to specify a custom hash function and the number of partitions.

Stage 3: Quick-sort

We complete the grouping of keys by sorting. Our runtime uses a two-phase sorting strategy to distribute the sort to the SPEs. As individual partition buffers are filled, they are streamed to available SPEs for in-place quick-sorting. The sorted output is then copied back to its original location in shared memory, overwriting the unsorted contents, as shown in Figure 2c. A partition's buffers are sort-merged, if necessary, in the next phase.

Stage 4: Merge-sort

Due to memory limitations, an in-place sort of an entire partition is not always possible. In such a case, we merge all quick-sorted buffers into a single sorted buffer using an external binary merge-sort. In Merge-sort, pairs of previously sorted buffers within a partition are scheduled for merging on the SPEs. In contrast to Map, the output size can be statically determined, so the SPE outputs are DMAed into a single pre-allocated output buffer in shared memory, as shown in Figure 2c.

The runtime continues merge-sorting on the SPEs until only two buffers remain in the partition. The final merge-sort takes place on the PPE. The primary reason is that we now need to bring the keys and their remotely located values together into a contiguous block of memory for Reduce. For this, direct and uncomplicated access to the entire shared memory space is highly desirable. In the final merge sort, all duplicate keys are eliminated so that a single unique key is associated with multiple values to be passed on to Reduce.

Optimizations: Merges are scheduled for optimal bandwidth utilization rather than greedily as illustrated by Figure 3. This yields $n/log(n)$ better bandwidth utilization in the limit as n approaches infinity, where n is the number of quick-sort buffers in a partition. For high-volume applications, we measured an average 20% performance improvement using this strategy.

Stage 5: Reduce

During Reduce, the SPEs execute the user-specified Reduce function on all the values for a given key. The PPE passes information about the sorted partition and a pointer to a pre-

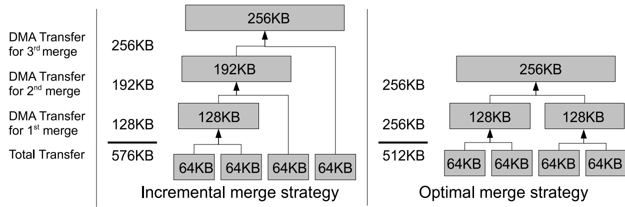


Figure 3. Bandwidth-optimal merge strategy.

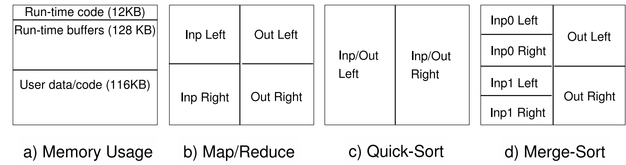


Figure 4. Double-buffering in SPEs. Left and Right differentiate the two buffers.

allocated output buffer to the SPE. Output buffers are arrays of key/value pairs. This stage is otherwise identical to Map. An output list maintained by the PPE contains a pointer to all the output buffers, and is the output structure returned to the user.

3.3 Implementation Analysis

Challenges to an efficient implementation are multi-threading the runtime, scheduling of stages, and memory management. We describe our solution to each below.

3.3.1 Multi-threaded Runtime

Since a large number of events occur concurrently, we implemented the runtime using extensive multi-threading with dedicated threads, using a total of 20 threads. All threads are spawned once, and remain active through the entire execution of the program.

- 1 PPE *main thread* spawns all other threads, creates the necessary data structures, and initiates the Map stage.
- 8 SPE *worker threads* perform the work of the Map, Reduce, Quick-sort stages, and SPE portion of the Merge-sort stage.
- 8 PPE *scheduler threads*, one for each SPE thread, notify an SPE when a work unit is ready for prefetching through the Cell's mailbox mechanism.
- 2 PPE *worker threads* perform the partitioning by distributing keys into partitions as well as the final merge-sort. There are two such threads to utilize the PPE's two-way SMT capability.
- 1 PPE *event thread* responds to output buffer memory allocation requests and completion notifications from the SPEs. It controls the execution flow by scheduling dependent work units on the PPE scheduler and worker threads as well.

3.3.2 Stage Scheduling

Figure 2a shows the logical execution flow of the runtime. However, stages 2 through 5 can execute concurrently as long as data dependences within a partition are correctly enforced. In fact, since the final merge occurs on the PPE, Merge-sort becomes a serialization bottleneck if the logical flow is maintained, as each partition would complete quick-sorting and SPE merge-sorting at approximately the same time, and all the SPEs would grind to a waiting for the PPE to complete each final merge.

To avoid this PPE bottleneck, stages 3 and 4 are executed in partition order. That is, the runtime gives priority to outstanding work on the leftmost partition of Figure 2a. This way, execution reaches the PPE merge-sort for partition 0 as soon as possible, at which point priority shifts to partition 1 and the SPEs transition to sorting that partition. This pattern then repeats, maximizing concurrency between the PPE and SPEs.

3.3.3 Memory Management

Managing the SPE local memories efficiently is crucial for high performance. Of the available 256KB, the runtime uses around 12KB for the SPE runtime code and 128KB for its buffers, leaving a little less than half of the memory for user code, data, and stack space. The runtime buffers are double-buffered differently in each phase as shown in Figure 4.

- **Map and Reduce** each allocate 32KB for input and 32KB for output per double-buffer as shown in Figure 4b. The input and output buffering must be managed independently because the input to output ratio of the phases is data-dependent and cannot be statically determined.
- **Quick-sort** is an in-place sort, requiring only a single buffer for both input and output. Two 64KB buffers are used as shown in Figure 4c.
- **Merge-sort** allocates two 16KB regions for the two input streams and 32KB for output as shown in Figure 4d. Due to the nature of the merge operation, each region drains or fills at a different rate, so each region is double-buffered independently.

Our implementation overlaps memory transfers with computation as much as possible within a given stage. Across stages, prefetching could potentially reduce the start-up time. However, when an SPE transitions from one stage to another, it is not always possible to pre-reserve space for the next phase as the previous phase is completing. In future work, we will explore techniques for such prefetching.

3.4 Support for Application Varieties

The execution flow previously described assumes a full MapReduce execution. However, applications may not require all stages of the model.

Map only: Certain data-parallel applications require only a Map stage, such as DCT, and can output directly to a known global output region. For such applications stages 2 through 5 are unnecessary overheads. If the user does not call `emitIntermediate` inside the Map function, the

runtime will not schedule any further work past Map, thus removing these overheads.

Map only with sorted output: If `emitIntermediate` is called but no Reduce function is defined, the runtime completes after Merge-sort; Reduce is never scheduled and the output list contains one sorted array of pairs per partition.

Chaining Invocations: Sometimes an application will chain multiple MapReduce invocations. Hence, the output of Reduce must be naturally compatible with the input of Map. Due to multiple, fixed size Reduce output buffers, output is a list of buffers. To allow such chaining efficiently, we define our API such that both inputs and output are lists of buffers.

Parameter	Description	Micro-benchmark Configuration
map intensity	Execution cycles per input byte to Map[a]	`[*,0,.01,512,256,16MB]`
reduce intensity	Execution cycles per input byte to Reduce	`[0,*,1.0,1  ,256,16MB]`
map fan-out	Ratio of input size to size of keys output in Map[b]	`[0,0,*  ,512,1  ,16MB]`
reduce fan-in	Number of values per key in Reduce	`[0,128,1.0,*,256,16MB]`
partitions	Number of partitions	`[0,0,1.0,512,*  ,16MB]`
input size	Input size in bytes	`[0,0,1.0,512,1  ,*   ]`

[a] We do not consider the sizes of keys and values within the input, as this only affects the number of Map function invocations, and can often be optimized within an application.

[b] As a simplification, we assume that all intermediate keys are less than 16 bytes in size and quadword aligned; hence, effectively 16 bytes in size. This was true for all evaluated applications.

Table 2. Application parameters and micro-benchmark configuration to isolate effects of each parameter.

4. Evaluation

This section presents an evaluation of our MapReduce runtime. We first characterized the MapReduce model and derived a set of parameters that capture application behavior. We then developed a parameterizable micro-benchmark for performance analysis. In this section, we present a discussion of these parameters, a detailed evaluation using our parameterizable micro-benchmark, and a derivation of application categories based on specific parameter configurations. We conclude with an analysis of complete applications.

4.1 Tools and Metrics

We used a 3.2 GHz Cell QS20 Blade Server running Fedora Core Linux to execute our micro-benchmark and applications. Using the Cell SDK v2.0, SPE code was compiled with the IBM XLC compiler [6], and PPE code with the GCC compiler, both at optimization level 3.

We instrumented our runtime and used processor performance counters to measure execution cycles. We analyze runtime and application performance using the following three metrics: execution time, runtime overhead, and scalability as the number of the SPEs is increased from 1 to 8. We measure overhead as percentage of time the SPEs are idle, which can occur for two reasons. First, SPEs may have to wait for work or synchronization from the PPE. Second, SPEs may be stalled waiting for completion of outstanding DMA transfers. Runtime code executing on the SPEs accounts for less than 1% of execution time, and hence we ignore this source of overhead. For complete applications, we also analyze execution rate by examining BIPS (Billions of Instructions Per Second). To measure BIPS, we estimated dynamic instruction count for the map, reduce, hash, and sort execution kernels using the Cell simulator (we did not have access to processor performance counters for instruction count), and divided by total application execution time measured using the Cell blade.

4.2 MapReduce Model Characterization

Table 2 shows the different parameters we use to characterize MapReduce programs. The model is simple enough that these six parameters completely capture behavior of applications at the API level. We developed a parameterizable micro-benchmark which we used to isolate each parameter and study its effect. We carefully tune the parameters for these experiments such that the studied parameter becomes the control variable and dictates performance. The third column in Table 2 shows the settings of other parameters, as a 6-tuple, as each parameter is isolated. The next four sections discuss execution time, a classification of application types, overheads, and the scalability effects of the parameters.

4.2.1 Execution Time

Figure 5 shows how execution time scales as each application parameter is varied linearly. In general, execution scales linearly or near-linearly for most parameters, showing that the runtime effectively implements the model. We discuss each individual parameter in detail below.

map intensity and **reduce intensity:** The amount of user computation performed in the Map and Reduce functions directly controls performance. As expected, increasing the intensity of the kernels scales execution time proportionally.

map fan-out: Altering the number of key/value pairs emitted in the Map stage affects the computation and memory transfer that occurs in all subsequent stages – one less key/value pair means one less element to partition and sort, and, if the key is unique, also one less reduction to perform. We see a super-linear ($nlog(n)$) effect on execution time as the fan-out is scaled. Fan-out increases the number of values to be sorted, and to isolate map fan-out we set map intensity, reduce fan-in and reduce intensity to low values. As a result, sorting, which scales as $nlog(n)$, dictates performance. Proportionally increasing the number of partitions can make this growth linear as described further below.

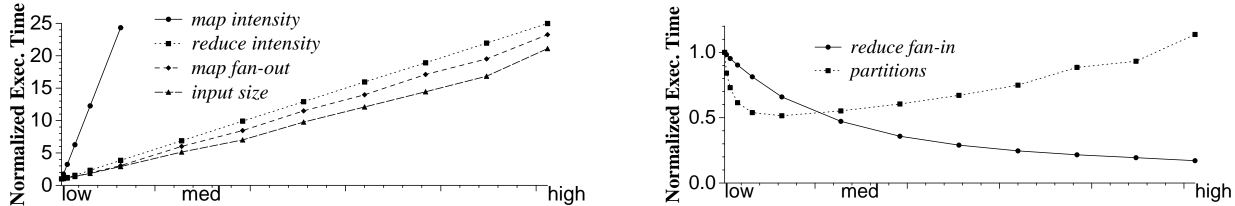


Figure 5. Execution time normalized to execution time at the lowest setting for the given parameter. The individual curve are not meaningfully compared since the weighting of parameters across micro-benchmark configurations varies.

reduce fan-in: The number of reduce invocations is equivalent to the number of key/value pairs emitted during Map divided by the *reduce fan-in*. Thus, there is an inverse relationship between the *reduce fan-in* and the execution time.

partitions: Execution time initially decreases rapidly as the number of partitions is reduced, and then increases gradually after reaching a minimum. As the number of partitions increases, the size of each partition reduces. At first, execution time also reduces because sorting is limited to a partition, and sorting many small sized partitions is less work than sorting a few large partitions. However, when the size of partitions drops below the size of a SPE quick-sort buffer, performance degrades. An increase in the number of buffers, and a subsequent decrease in the amount of SPE work performed per buffer, cause the PPE to be overburdened with buffer management and synchronization tasks, and unable to handle and respond to requests in a timely manner.

input size: Application performance scales slightly faster than linearly with the input size. This is because the amount of key comparisons per input element in sorting grows by $nlog(n)$. Proportionally increasing the number of partitions will make this growth linear.

4.2.2 Application Types

In the above discussion and in Table 2 we show only the isolated effect of each parameter, purposefully overlooking parameter inter-dependencies. In real applications however, these parameters do interact and affect the runtime and overall performance. Rather than exhaustively analyze all possible combinations, we examine each stage of the runtime and determine a set of *application types* that capture the dominant interactions between the parameters.

Map-dominated: The Map stage's performance is to first-order dictated by *map intensity*, but is also related to *map fan-out*, which dictates the amount of partitioning work. Partition runs on the PPE concurrently with Map, and must complete within some proximity of Map to ensure unobstructed sorting with no serialization bottleneck. Hence, to avoid this bottleneck, the ratio of *map intensity* to *map fan-out* must be above some threshold value. Those applications that meet this threshold we label *map-dominated*. For our micro-benchmark configuration representative of this application type, we set this ratio very high.

Partition-dominated: Conversely, if this ratio is below the threshold value, partitioning dominates and we label these applications *partition-dominated*. For this application type, our representative micro-benchmark configuration has the ratio set very low.

Sort-dominated: The criticality of sorting is dictated by the *map fan-out* and the number of *partitions*. While many real-world applications set *partitions* to its performance-optimal value, others desire a single sorted partition, and still others define a custom hash function and set the number of partitions to divide output into logically separate, sorted groupings. We label applications with a high *map fan-out* and very few *partitions* as *sort-dominated*. For our representative micro-benchmark configuration, we set *partitions* to one and *map fan-out* high. This configuration stresses the Merge-sort stage of the runtime, and entirely bypasses partitioning as there is only one partition.

Reduce-dominated: Although applications dominated by Reduce are in theory a possibility, all real-world applications we examined have simple Reduce computations, with low *reduce intensity* and high *reduce fan-in*. Reduce thus comprises only a small percentage of overall execution time. Moreover, Reduce has minimal effect on the execution of other stages. Hence, we do not study this as a separate application type in this paper.

Table 3 lists the application types we study and their micro-benchmark parameter configurations. In examining real-world applications, we observed that most applications align strongly with one of these application types. *Map-dominated* applications are traditional, computationally-intense data-parallel applications that utilize the Reduce phase only for coalescing the results from individual Map phases. In contrast, *partition-dominated* applications rely heavily on the partitioning, sorting, and Reduce phases of the runtime to group many key/value pairs together, such as the word counting example in section 2. Finally, *sort-dominated* applications are like *partition-dominated* applications, but with sorted output.

Application Name	Application Type	Lines of Code		Speedup vs. Core2			BIPS		
		MapReduce	Serial	1-SPE	8-SPEs	8-SPE Ideal	1-SPE	8-SPEs	8-SPE Ideal
histogram	partition-dominated	345	216	0.16	0.15	2.44	1.56	1.51	24.49
kmeans	partition-dominated	324	318	0.91	3.00	6.92	2.08	7.35	17.01
linearRegression	map-dominated	279	114	0.34	2.59	2.67	1.47	11.32	11.70
wordCount	partition-dominated	226	324	0.87	0.96	10.26	1.52	1.74	18.64
NAS_EP	map-dominated	264	112	1.08	8.62	8.62	2.00	15.93	15.95
distributedSort	sort-dominated	171	93[c]	0.41	0.76	5.48	1.28	2.38	17.15

[c] The serial implementation of distributedSort uses the C standard library qsort().

Table 4. Applications characteristics and performance.

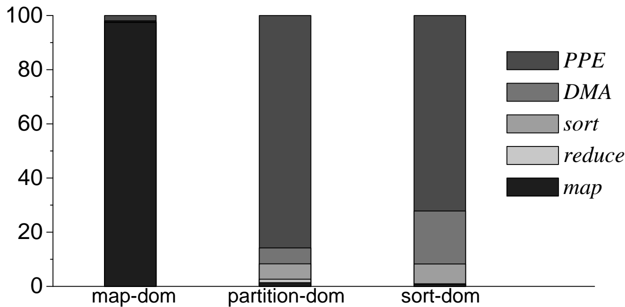


Figure 6. Execution breakdown for application types.

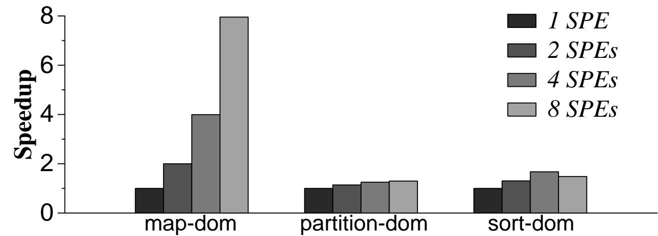


Figure 7. Speedup as number of SPEs is scaled from 1 to 8.

4.2.3 Runtime Overhead

Figure 6 shows the breakdown of execution time in the SPEs for the three application types running with 8 SPEs. *Map*, *reduce*, and *sort* denote execution time in the respective computation stages. *PPE* and *DMA* represent runtime overhead. *PPE* denotes the percentage of cycles SPEs are waiting for work and synchronization from the PPE, including the absence of work due to initialization, finalization, and workflow dependences. *DMA* denotes the percentage of cycles SPEs must wait for DMA completion in order to proceed. The contribution of runtime code executing on the SPEs is less than 1% and is included in the *map*, *reduce*, and *sort* times.

Map-dominated applications are ideally suited for the architecture and the model, and the runtime is very efficient in executing these applications. For these applications, the SPEs spend more than 95% of their time in the Map and Reduce kernels.

Partition-dominated applications frequently have runtime overhead around 90%. These applications emit a large volume of intermediate data, and typically have minimal Map, or Reduce work. *PPE* contribues 85% and *DMA* contributes 5% overhead. 40% of *PPE* is caused by serialization on the PPE's substantial partitioning task. The remaining 45% of *PPE* is caused by the low computation demands on the SPEs, which cause a high turnover rate for the PPE's buffer management and scheduling tasks, overwhelming the PPE. To potentially alleviate these PPE overheads, section 5 discusses techniques to parallelize Partition and enhance PPE thread scheduling for improved response times.

Sort-dominated applications also have high overheads around 90%, but for different reasons. *DMA* contributes 20% and *PPE* contributes 70% overhead. 40% of *PPE* is due to the inherent serialization of Merge-sort and Reduce, given only one partition. The remaining 30% of *PPE* is due to low computation demands on the SPEs, similar to *partition-dominated* applications. The 20% *DMA* overhead is caused primarily by merge-sort output transfers.

Application Type	Micro-benchmark Configuration
map-dominated	`[128,0 ,.01,8,8 ,16MB]`
partition-dominated	`[0 ,0 ,1.0,8,256,16MB]`
sort-dominated	`[0 ,0 ,1.0,8,1 ,16MB]`

Table 3. Application types.

4.2.4 Scalability

Figure 7 shows the relative speedup of the three application types as the number of SPEs is increased. *Map-dominated* applications scale perfectly. *Partition-dominated* applications experience both partitioning and the final merge-sort as serialization points. Together, these two tasks dominate execution time, leaving little other work that can scale with more processors. *Sort-dominated* applications scale better, but still poorly due to the Merge-sort and Reduce serialization points; when the number of processors is increased from 4 to 8, there is contention for DMA and PPE resources, causing a performance degradation.

4.3 Application Performance

We now discuss the performance of complete applications, focusing again on execution time, overheads, and scalability.

4.3.1 Applications

We implemented six real-world applications, four of which are applications from the Phoenix evaluation suite [22] well suited to the Cell architecture. We also implemented one application from the NAS benchmark suite, *EP* [1], easily adapted to the model, and one application, *distributedSort*, as representative of data processing applications where sorted output is often desired. The application names, their type, and the lines of code are shown in Table 4. Equivalent micro-benchmark configurations are shown in Table 5. A detailed description of each application follows.

histogram: The frequency of occurrence of each RGB color component for a given bitmap image file is counted. Map counts the occurrences of each color component and Reduce gathers the intermediate sums to produce a final sum for each component.

kmeans: This application clusters a set of data points. Map takes as input a point, finds the distance between the point and each cluster, and assigns the point to the closest cluster. Reduce computes the new cluster means by averaging the coordinates of all points assigned to the given cluster. The algorithm iterates until it converges.

linearRegression: This applications computes a line-of-best-fit for a set of 2D coordinates. Coordinates are grouped into 32KB chunks for bulk processing. Map computes intermediary summary statistics for a chunk and reduce gathers all the data for a given statistic and calculates the best fit.

wordCount: The frequency of occurrence of each word in a file is counted for a given text file. For compatibility, our implementation truncates all words to 16 characters.

NAS_EP: Map generates pairs of Gaussian random deviates and counts the number of pairs that lie within successive square annuli, along with the sum of all pairs generated. Reduce computes the final sums for each attribute.

distributedSort: This application performs an integer sort. Map selects the key on which to sort. Reduce is omitted.

Application Name	Micro-benchmark Configuration
histogram	[6 ,<1,0.38 ,4700,512,100MB]
kmeans	[793,13,2.0 ,100 ,100,1.6MB]
linearRegression	[61 ,<1,<0.01,8192,8 ,128MB]
wordCount	[8 ,<1,1.0 ,25 ,512,10MB]
NAS_EP	[145,<1,<0.01,8 ,8 ,512MB]
distributedSort	[8 ,0 ,1.0 ,- ,1 ,32MB]

Table 5. Measured parameter values for applications.

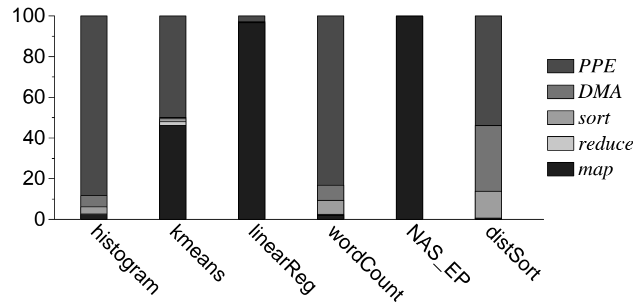


Figure 8. Execution breakdown for applications.

Four other applications from the Phoenix suite were not implemented for the following reasons. *Reverse Index* and *String Match* require variable length structures (strings) which our current runtime does not support. Providing such support with software-managed memories is complex and highly challenging. *Matrix Multiply* and *PCA* require direct access to a large global data structure and have no Reduce work. MapReduce therefore provides limited productivity benefits. Particularly for the Cell, implementing them to accommodate arbitrary data sizes also requires explicit user memory management, defeating a key benefit of our runtime.

Our baseline for performance comparison is another best-of-breed system, an Intel Core2 Duo with a 4MB L2 cache operating at 2.4 GHz. We implemented single-threaded versions of the applications in C and compiled them using GCC at optimization level 3. While optimized multi-threaded Cell implementations of these applications could serve as a baseline (or upper-bound), they require extensive application-specific tuning and reveal little intuition about the applications themselves. Instead we measure runtime overheads to quantify inefficiencies and compare against this baseline system.

4.3.2 Execution Time and Runtime Overhead

Columns 5 through 7 of Table 4 show the relative speedup of the applications compared to an Intel Core2 serial implementation, and Figure 8 shows the breakdown of execution cycles. Ideal speedup is calculated by removing all the overhead execution cycles from the execution time of the program. A couple of applications perform worse than the baseline, a few perform better, while NAS_EP excels – its achieved speedup is close to ideal. An explanation of each application follows.

linearRegression and *NAS_EP* are *map-dominated* and significantly out-perform our baseline, with total runtime overhead under 4% and 1%, respectively, and achieving close to ideal speedup at 2.6X and 8.6X, respectively. Note that neither application is hand-optimized for SIMD execution on the Cell, which could further boost performance. *NAS_EP* in particular performs computation on double-precision floating-point numbers, to which the Cell is not

well suited, but which is required by the benchmark specification. For the sake of fairness, we used the C standard math library routines on both the baseline system and the Cell system, but our experiments showed that using SIMD single-precision operations yields a minimum 7X *additional* performance improvement for the Cell MapReduce implementation.

Among *partition-dominated* applications, while *histogram* is naturally data-parallel, implementing it using MapReduce makes it reliant on the PPE for partitioning and runtime orchestration with 92% overhead and only 0.15X speedup. *wordCount* performs similarly with 90% overhead, but is more competitive with the baseline showing 0.96X speedup. The speedup difference is caused by an extremely straightforward and very fast single-threaded implementation of *histogram*, compared to relatively a more complex and slower single-threaded implementation of *wordCount*, which requires a binary search tree for data storage. Finally, *kmeans* has very high map-intensity, allowing useful work to occur concurrently with partitioning. Partitioning still dominates as a serialization bottleneck, however, preventing the transition into the *map-dominated* category, despite strongly outperforming our baseline with 3.0X speedup. This bottleneck is evidenced by the modest performance scaling from 1 SPE to 8 SPEs.

distributedSort is the only *sort-dominated* application, although any partition-dominated application is likely to become *sort-dominated* if a single sorted output is required. *distributedSort* has 28% overhead due to Merge-sort serialization, 32% due to lengthy DMA stalls, and 26% due to idle cycles waiting for work and synchronization from the PPE, resulting in 86% total overhead. Speedup relative to the baseline is only 0.76X. Although the ideal speedup is 5.6X, it is not achievable in practice as DMA, synchronization, and the inherent serialization of merge-sorting constrain any sorting application.

4.3.3 Scalability

One of the benefits of the model is its inherent scalability. While, our performance analysis used all 8 SPEs, by simply configuring the number of SPEs in the runtime, an application can be scaled up or down. We study scalability of the runtime by varying the number of SPEs from 1 to 8 and examining the execution rate (BIPS). Columns 8 and 9 of Table 4 show BIPS for 1 and 8 SPEs, respectively. The Cell processor's theoretical peak is 51.2 BIPS when using all 8 SPEs at 3.2 GHz.

Applications that suit the model and architecture (*kmeans*, *linearRegression*, and *NAS_EP*), show near-linear speedup. For example, *NAS_EP* performance increases from 2.0 to 15.93 BIPS. *Partition-dominated* applications with little map work like *histogram* and *wordCount* are bottlenecked by the PPE and hence adding more SPEs provides little speedup, at best going from 1.52 to 1.74 BIPS. *Sort-dominated* applications scale better, but eventually DMA and PPE overheads dominate and the SPEs go under-utilized. Our representative *sort-dominated* application, *distributedSort*, scales from 1.28 to 2.38 BIPS.

Ideal BIPS is calculated by isolating execution of only the Map, Reduce, and sorting kernels, and is provided in column 10 of Table 4. As shown, if the SPEs can be perpetually fed with data, all applications can sustain high performance. For applications that map well to the model and architecture, our runtime is able to match this ideal BIPS. For other applications, it is less successful, but the ideal BIPS will be hard to achieve even with application-specific tuning, as much of the DMA and PPE overheads are inherent to the applications themselves.

5. Discussion

Our performance results show that computationally intense applications scale well and sustain high performance. In addition to the performance benefits, the MapReduce model provides a large programmability benefit, as indicated by the lines of code listed in the third and fourth columns of Table 4. While the runtime itself is quite sophisticated, requiring more than 5000 lines of code, the MapReduce programs were no more cumbersome to develop than their single-threaded counterparts – we had to perform no application-specific tuning, partitioning, scheduling, or memory management.

However, our results show that computationally weak applications perform relatively poorly. Unfortunately, it is unclear whether this is a limitation of the architecture or the model since these applications are primarily PPE bound, as opposed to bandwidth or computation bound. As future work, we propose two implementation enhancements that have the potential to greatly improve the scalability of the runtime executing these types of applications.

Distributing Partition to the SPEs: The serialization bottleneck of partitioning on the PPEs is a substantial deterrent to efficient processing of computationally weak applications. For such applications, decentralizing the partitioning work across the SPEs is clearly desirable. The obvious approach is to replicate the number of partition buffers for each SPE, and have each SPE partition directly into its own set of buffers. However, this complicates selecting of the optimal number of partitions, and also forces DMA transfers at the granularity of key/value pairs. Nevertheless, our experiments indicate that this enhancement could yield up to a 2X performance improvement for *partition-dominated* applications running with all 8 SPEs.

A single OS thread: Upon close examination of our performance data, we discovered that our runtime suffers from multi-threading overheads on the PPE. As an example, we observed that for computationally weak Map tasks, the time taken by the PPE to respond to an output allocation request is twenty times the time taken for SPE computation. The relatively large (coarse) time-slice granularity of the OS is

primarily responsible for the slow PPE response time. Thus, the solution that we propose is to implement the PPE portion of the runtime using only a single, event-driven OS thread, while making all communication asynchronous. This presents a challenge as currently the PPE scheduler threads communicate work items to the SPEs using blocking mailbox calls.

We include some other enhancements to be considered for future work below.

Arbitrary memory accesses in Map and Reduce: Extensions to the model and the run-time to efficiently support arbitrary memory accesses can enhance the range of applications supported by the runtime. Our current design has the limitation that (1) the input, intermediate, and output element data structures must be of bounded size, and that (2) they must occupy a contiguous region of memory. Supporting either or both of these features for the Cell platform is challenging. For our runtime, they also require a substantial API and design re-architecture.

Hierarchical MapReduce: With the Cell processor being used in clusters and the growing requirement of high performance computing in servers, a hierarchical MapReduce model at the large-scale and chip-level is a natural future enhancement we will explore.

6. Related work

Our related work discussion covers Cell-specific programming tools, high-performance applications on the Cell processor, programming model extensions for parallelism that can be applied to Cell, and extensions to the MapReduce model.

Eichenberger et al. describe a variety of techniques in the optimizing Cell compiler for extracting parallelism and simplifying memory management [6]. Topics include SIMD parallelism extraction, automatic code and data partitioning across the cores using OpenMP pragmas [20], and the implementation of a software-managed cache to support a single shared memory abstraction on the SPEs. IBM Research has also proposed message passing *microtasks* based on the Message Passing Interface (MPI) standard [19]. Finally, Blagojevic et al. describe scheduling techniques for different granularities of parallelism for the Cell processor [3].

Williams et al. used a mathematical model to examine the potential of the Cell processor for scientific computation and identify memory management as a key challenge [25]. Application-specific tuning has been used to map several applications to the Cell processor, including solving linear equations [14], H.264 decoding [24], speech recognition [16], and ray tracing [2]. Nanda et al. describe software and application tuning for Cell servers focusing on system-level design and application tuning across Cell processors [18]. The hierarchical MapReduce model we proposed as an enhancement can be applied in such systems.

The stream programming model [12] is well suited for the Cell architecture, and several runtimes support this model [17, 23]. Fatahalian et al. introduce a new programming model called Sequoia targeted at explicitly memory managed machines [7]. The MapReduce model is similar to streaming, but the automatic grouping enables more sophisticated applications than streaming alone. Deitz et al. develop techniques for user-defined scans (similar to *Map*) and formulation of user-defined reductions that can be applied in Chapel and MPI [5]. Knight et al. describe general compilation techniques for explicitly managed memories [13]. Kandemir et al. describe compiler techniques for scratchpad memory space which can be applied to the SPE local memories [11].

Lämmel presents a formal description of the MapReduce model [15]. The Sawzall language captures the MapReduce model and provides an interface to a set of aggregators that can express many common data processing and data reduction problems [21]. Finally, Yang et al. add a *merge* function to extend MapReduce for processing multiple related heterogeneous datasets [26].

7. Conclusions

The Cell processor provides a unique and innovative architecture with vast performance potential but a challenging programming platform. This paper presented a MapReduce runtime for the Cell processor. The runtime provides a simple machine abstraction, hiding parallelization and hardware primitives from the user. The programming model is well suited to several data-parallel programs, and it maps naturally to the Cell architecture. The simplicity of the model provides enormous productivity benefits and makes the architecture accessible to many domains and types of users. The scalability and performance of the runtime make it attractive for high-performance computation as well.

Applications suited to the model profit greatly from the productivity, performance, and efficiency benefits, yet the model is not universally applicable to all types of applications. Future architectures must embrace such "domain-specific" solutions, as one universal paradigm or implementation is unlikely to be efficient and provide high performance. Looking forward, this work is applicable to all chip-multiprocessor systems of the future that adopt a distributed memory design or software-exposed memories. High performance and programmer productivity will both be equally important constraints for such architectures. The simplicity of the model and the efficiency of our MapReduce design make it an attractive choice for programming these systems.

References

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan,

and S. K. Weeratunga. The NAS parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[2] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich. Ray tracing on the CELL processor. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 25–23, 2006.

[3] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, and C. D. Antonopoulos. Dynamic multigrain parallelization on the Cell broadband engine. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 90–100, New York, NY, USA, 2007. ACM Press.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, December 2004.

[5] S. J. Deitz, D. Callahan, B. L. Chamberlain, and L. Snyder. Global-view abstractions for user-defined reductions and scans. In *PPoPP '06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 40–47, New York, NY, USA, 2006. ACM Press.

[6] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing compiler for the CELL processor. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, 2005.

[7] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 83, New York, NY, USA, 2006. ACM Press.

[8] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

[9] H. P. Hofstee. Power efficient processor architecture and the Cell processor. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262, 2005.

[10] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D.Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4):589–604, July 2005.

[11] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *DAC '01: Proceedings of the 38th Conference on Design Automation*, pages 690–695, New York, NY, USA, 2001. ACM Press.

[12] B. Khailany, W. Dally, U. Kapasi, P. Mattson, J. Namkoong, J. Owend, B. Towles, A. Chang, and S. Rixner. Imagine: media processing with streams. *IEEE Micro*, 21(2):35–46, 2001.

[13] T. J. Knight, J. Y. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan. Compilation for explicitly managed memory hierarchies. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 226–236, New York, NY, USA, 2007. ACM Press.

[14] J. Kurzak, A. Buttari, and J. Dongarra. Solving Systems of Linear Equations on the CELL Processor Using Cholesky Factorization - LAPACK Working Note 184. Technical Report UT-CS-07-596, University of Tennessee Computer Science, 2007.

[15] R. Lämmel. Google's MapReduce programming model – revisited. Accepted for publication in the Science of Computer Programming Journal, 2006.

[16] Y. Liu, H. Jones, S. Vaidya, M. Perrone, B. Tydlitat, and A. K. Nanda. Speech recognition systems on the Cell Broadband Engine processor. *IBM Journal of Research and Development*, 51(5), 2007.

[17] Mercury MultiCore Plus SDK. http://www.mc.com/products/productdetail.aspx?id=2826&ProductTypeFolder=60.

[18] A. K. Nanda, J. R. Moulic, R. E. Hanson, G. Goldrian, M. N. Day, B. D. D'Amora, and S. Kesavarapu. Cell/B.E. blades: Building blocks for scalable, real-time, interactive, and digital media servers. *IBM Journal of Research and Development*, 51(5), 2007.

[19] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. MPI microtask for programming the Cell Broadband Engine processor. *IBM Systems Journal*, 45(1):85–102, January 2006.

[20] Official OpenMP specifications. http://www.openmp.org/specs/mp-documents/cspec20.pdf.

[21] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming, Special Issue Dynamic Grids and Worldwide Computing*, 13(4):277–298, 2006.

[22] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, February 2007.

[23] RapidMind. http://www.rapidmind.com.

[24] R. Sakai, S. Maeda, C. Crookes, M. Shimbayashi, K. Yano, T. Nakatani, H. Yano, S. Asano, M. Kato, H. Nozue, T. Kanai, T. Shimada, and K. Awazu. Programming and performance evaluation of the Cell processor. In *HOT CHIPS 17*, 2005.

[25] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd Conference on Computing Frontiers*, pages 9–20, New York, NY, USA, 2006. ACM Press.

[26] H. Yang, A. Dasdan, R. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 1029–1040, New York, NY, USA, 2007. ACM Press.