**QUEEN'S
UNIVERSITY
BELFAST**

Rearchitecting MapReduce for Heterogeneous Multicore Processors with Explicitly Managed Memories

Papagiannis, A., & Nikolopoulos, D. (2010). Rearchitecting MapReduce for Heterogeneous Multicore Processors with Explicitly Managed Memories. In *Proceedings of the 39th International Conference on Parallel Processing (ICPP)* (pp. 121-130). Institute of Electrical and Electronics Engineers Inc.. <https://doi.org/10.1109/ICPP.2010.21>

Published in:

Proceedings of the 39th International Conference on Parallel Processing (ICPP)

Queen's University Belfast - Research Portal:

[Link to publication record in Queen's University Belfast Research Portal](#)

General rights

Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy

The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person's rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.

Rearchitecting MapReduce for Heterogeneous Multicore Processors with Explicitly Managed Memories

Anastasios Papagiannis* and Dimitrios S. Nikolopoulos*

Institute of Computer Science (ICS)

Foundation for Research and Technology – Hellas (FORTH)

GR-70013, Heraklion, Crete, GREECE

{apapag,dsn}@ics.forth.gr

Abstract—This paper presents a new design and an implementation of the runtime system of MapReduce for heterogeneous multicore processors with explicitly managed local memories. We advance the state of the art in runtime support for MapReduce using five instruments: (1) A new multi-threaded, event-driven controller for task instantiation, task scheduling, synchronization, and bulk-synchronous execution of MapReduce stages. The controller improves utilization of control-efficient cores, minimizes control overhead in the runtime system, and overlaps task instantiation with task scheduling on compute-efficient cores. (2) An implicit partitioning scheme which eliminates redundant memory copies. (3) An adaptive memory management scheme which combines efficient memory preallocation for applications with statically known output volume with dynamic allocation using runahead tasks for applications with statically unknown output volume. (4) An optimized quick-sort/merge-sort scheme which reduces the critical path length of merge-sort. (5) An optimized execution scheme which avoids redundant data transfers to and from local stores in applications that emit keys with the same value. Put together, these techniques accelerate representative MapReduce workloads by a factor of $1.81\times$ (geometric mean) compared to a reference design that represents the state of the art.

Keywords—MapReduce; Heterogeneous multicore processors; Resource management; Runtime systems; Operating Systems; Parallel Programming Models.

I. INTRODUCTION

MapReduce is a parallel programming model for processing large amounts of raw data. Introduced by Google in 2004 [1], MapReduce transparently distributes data between processing nodes, performs data processing and data aggregation, and provides support for transparent failure recovery. The key advantage of MapReduce as a programming framework is that it hides the complexity of data distribution, parallelization, load balancing, and failure recovery from the programmer. Borrowing from functional languages, MapReduce exposes only two simple primitives to programmers, a *map* primitive for processing raw data

to generate intermediate (key, value) pairs and a *reduce* primitive, which merges intermediate values associated with the same key. Both map and reduce execute user-defined processing and reduction functions respectively.

The MapReduce programming model is widely deployed in distributed applications ranging from simple user tasks such as searching and sorting, to large-scale web applications such as link-graph traversal, processing of access logs, construction of term vectors, and document clustering, to scientific applications based on machine learning and data mining algorithms [2], [3], [4]. Due to the market value, societal value, and computational requirements of large-scale distributed data processing applications, MapReduce has also attracted considerable research interest.

Massively parallel data processing tasks in MapReduce can exploit multicore processors and several implementations of the MapReduce runtime system on a diverse range of multicore architectures have been presented to date [5], [6], [7], [8], [9], [10], [11]. Due to the embarrassingly parallel structure of most MapReduce applications, MapReduce can also take advantage of processors that expose large amounts of data parallelism via wide SIMD units or multithreading, as well as the ample on-chip bandwidth available on these processors. MapReduce ports on the Cell [12], GPUs [13] and Larrabee [14], exhibit the aforementioned architectural strengths.

In this work we revisit the design and implementation of MapReduce for heterogeneous multicore processors with explicitly managed local memories. These processors represent a sweet spot in multicore processor architecture for two reasons: First, heterogeneity can deliver higher performance than homogeneity in parallel computations with a non-negligible sequential component, or significant control overhead for task scheduling and synchronization [15]. Heterogeneous architectures devote many simple cores to data-parallel computation, using SIMD or multithreading datapaths, while reserving a significant portion of silicon real estate for few relatively complex sequential cores that run efficiently control-intensive code. Second, the use of memory hierarchies managed explicitly in software yields a simple hardware design with high performance, via aggressive data

*Also with the Department of Computer Science, University of Crete, Heraklion, Greece. Work supported by the European Commission, through grants FP7-PEOPLE-224759, FP6-IP-27648, FP7-ICT-217068, FP6-IST-004408, FP7-ICT-248647.

prefetching and application-specific data caching, at the cost of additional effort by programmers and additional complexity in the compiler and runtime system.

Implementing MapReduce on heterogeneous multicore processors is challenging due to the inherent requirement of providing transparent data distribution, scheduling, and load balancing while taming heterogeneity and managing data transfers in software. Although these challenges have been identified in earlier work [7], we find that current implementations of MapReduce for heterogeneous multicore processors still suffer from inefficiencies, including (1) excessive control overhead for task scheduling and synchronization on the control-efficient cores, (2) redundant memory copies and redundant data transfers for partitioning intermediate data once this data is generated from the map function, (3) redundant data transfers for MapReduce computations producing keys with the same value, (4) sub-optimal sorting and merging algorithms and (5) memory management overhead. We address these five problems in a new design and an implementation of MapReduce, which we evaluate on the IBM-Sony-Toshiba Cell processor. Our specific contributions are:

- A new multi-threaded controller for scheduling and synchronization of MapReduce tasks. The controller implements an event-driven model for task initiation, task scheduling, and detection of task completion, and a bulk-synchronous execution model of the parallel MapReduce stages that are offloaded to compute-efficient cores. The two techniques combined minimize overhead on the control-efficient cores, utilize effectively the memory hierarchy of the control-efficient cores, and increase available task parallelism for compute-efficient cores.
- A new implicit partitioning scheme for data emitted from the map stage, that avoids redundant memory copies and redundant data transfers between cores.
- A new scheme for implementing the quick-sort and merge-sort stages, that reduces the critical path of merge-sort by eliminating both redundant data transfers and redundant computation.
- A new memory management scheme which combines efficient memory preallocation for map tasks with statically known output volume, with dynamic allocation using runahead tasks to estimate the output volume of map tasks with statically unknown output volume.
- An execution scheme that avoids redundant data transfers to and from local stores in computations that emit keys with the same value in the map stage.

Performance analysis of our runtime system on Cell demonstrates that it improves performance by a factor of $1.81\times$ (geometric mean) for typical MapReduce workloads and up to $2.3\times$ for partition-dominated and sort-dominated MapReduce workloads, compared to a reference design that

Figure 1: Cell Broadband Engine organization.

represents the state of the art [7]. Furthermore, the new runtime system scales better than the reference design.

The rest of this paper is organized as follows: Section II provides an overview of MapReduce. Section III presents our design of the MapReduce runtime for multicores with explicitly managed local memories. Section IV presents our experimental analysis and results. Section V discusses related work and Section VI concludes the paper.

II. BACKGROUND

A. MapReduce Programming Framework

MapReduce [1] processes an input of (key, value) pairs to produce an output of (key, value) pairs. A MapReduce program executes in three stages, a *map* stage that produces a set of intermediate (key, value) pairs for each input pair, a *group* stage that groups all intermediate (key, value) pairs associated with the same key, and a *reduce* stage that merges the values associated with each key. The *map* and *reduce* stages are user-provided and application-specific.

B. Cell Broadband Engine Architecture

Figure 1 illustrates a block diagram of the Cell processor [16]. The Cell is a heterogeneous multicore processor with one PowerPC SMT core (the Power Processing Element – PPE), and eight short SIMD cores (the Synergistic Processing Elements – SPEs). The PPE is a control-efficient simultaneous multithreading core that implements the PowerPC ISA and supports general-purpose operating systems and virtualization. The PPE integrates a two-level cache hierarchy and includes advanced instruction execution capabilities such as multiple instruction issue, branch prediction, and SIMD instructions. The SPEs implement a wide datapath for executing vector instructions in a dual pipeline. They enable high-performance vectorization of data-parallel

code. Instead of caches, each SPE core includes a relatively large register file with wide registers (128, 128-bit registers) and a software-managed local store (scratchpad) memory of size 256 KB and a 6-cycle load latency.

III. MAPREDUCE DESIGN

We present a new design of MapReduce on the Cell and use the design presented by de Krujif and Sanakralingam [7] as a reference for qualitative comparisons. We refer to the latter as the *reference* design. We conduct a quantitative evaluation and comparison of the two designs in Section IV.

Our runtime system implements a four-stage execution scheme for MapReduce: *Map*, *quick-sort*, *merge-sort* and *reduce*. By contrast, the reference design implements a five-stage scheme, including the four stages in our design and a *partition* stage which follows the *map* stage. Our design implements partitioning implicitly while emitting data from the *map* stage. Figure 2 illustrates our design (right) and the reference design (left) of MapReduce.

The reference design uses a multithreaded controller running on the PPE for the purposes of scheduling tasks – which includes task instantiation on the PPE and offloading of task arguments to SPEs–, processing task completion notifications, and executing selected tasks natively without off-loading them on SPEs. More specifically, the reference design uses a total of 20 threads on a Cell processor with 8 SPEs. These threads include:

- 1 PPE *main thread* that spawns all other threads and initiates the map stage.
- 8 SPE *worker threads* that perform map, quick-sort, merge-sort and reduce tasks on SPEs.
- 8 PPE *scheduler threads*, one for each SPE thread, that schedule tasks on SPEs and process task completion notifications on a per-SPE basis.
- 2 PPE *worker threads* that perform the partition stage and the last step of merge-sort. These two threads utilize the PPE’s 2-way SMT core.
- 1 PPE *event thread* that controls the execution flow and processes memory allocation requests for output buffers on behalf of SPEs. The same thread synchronizes with the SPEs to prevent races during buffer allocation.

Our runtime uses two threads for task instantiation, scheduling, processing of completion notification, memory allocation, and execution flow control on the PPE. We leverage simultaneous multithreading on the PPE core to overlap task instantiation with task scheduling. However, we minimize the degree of multithreading using one thread for instantiating all tasks in all MapReduce stages and one thread for scheduling and processing all task completion notifications from SPEs. Each thread executes an event-driven controller implementing the stages illustrated in Figure 2 (right). This design minimizes control overhead on the PPE and avoids bottlenecks due to the low computational density of the PPE compared to SPEs.

Specifically, our design replaces the 12 PPE threads of the reference design with 1 PPE *main thread* that spawns the other threads and instantiates all tasks, and 1 PPE *scheduler thread* which schedules tasks on SPEs and processes task completion notifications. We use multithreading on the PPE only to decouple work instantiation from work scheduling in the runtime system. This enables overlap of the two operations, leading to a more scalable work creation scheme and more task-level parallelism available to SPEs due to reduced control overhead on the PPE. However, we avoid oversubscribing the PPE core and the associated context switching overhead on the PPE by having a single thread instantiating all tasks and a single thread scheduling all tasks. Reducing context switching overhead has the side-effect of reducing the critical path of processing task completion notifications [17] in the scheduler thread. Both the main thread and the scheduler thread use an event-driven execution scheme instead of multithreading to process requests and replies from and to SPEs. In addition to task instantiation, the main thread implements memory allocation on behalf of SPEs using either static preallocation or runahead map tasks, a topic which we discuss in the following paragraphs. The scheduler thread implements dynamic task scheduling on SPEs, using per-SPE task queues. Each SPE self-schedules assigned tasks from its queue.

The two PPE threads implement a bulk synchronous execution model for MapReduce stages, using barriers between stages. The scheduler thread signals barrier releases as soon as all tasks in a stage have completed execution. The reference implementation overlaps partially tasks across adjacent MapReduce stages, by using per-SPE buffers and one thread per SPE to detect completion of one stage’s tasks on the specific SPE and initiate the next stage, while completion of tasks from the previous stage may still be pending on other SPEs. We adopt a bulk synchronous design, potentially losing some parallelism, to preserve single-threaded task initiation and single-threaded completion detection on the main and scheduler thread respectively and to minimize stage control overhead. We make these choices since the dynamic scheduling of tasks in each stage of our runtime system balances the load across SPEs effectively, thereby leaving little, if any, time for overlapping stages. In practice, we find that the control overhead for overlapping stages jeopardizes performance in all cases (Section IV).

Figure 3 shows our dual-threaded PPE runtime and the reference multithreaded PPE runtime. We use arrows to represent communication during the execution of MapReduce. In the rest of this section we analyze further the implementation of each MapReduce stage in our runtime.

A. Map

During the Map stage, the reference design uses the PPE to divide the input data with a simple heuristic between all available SPEs. The SPEs then execute the user-defined map

Figure 2: MapReduce reference (left) and new (right) design on Cell.

Figure 3: MapReduce reference (left) and new (right) multithreaded runtime on the PowerPC core (PPE).

function. The PPE allocates one output buffer of size 64 KB per SPE, however the runtime system can not determine statically the total size of the intermediate output emitted by each SPE in the map stage. Therefore, when SPEs need to emit more output volume than 64 KB they request a new buffer from main memory. Given a request from an SPE, the PPE thread controlling the requester processes the request by invoking the memory allocator on another PPE thread, with which it needs to synchronize. Since input and output buffers can be larger than the size of the local store of each SPE, the runtime system streams data from and to buffers in main memory using DMAs and multi-buffering to overlap memory access latency. SPEs map keys and values to separate output buffers. Each key contains also a pointer to its values and a hash value which is used in the partition stage. The reference design implements PPE-

SPE communication and synchronization using the Cell's mailbox registers and signal notification mechanisms [18].

The difference in our design compared to the reference design is that the implementation attempts to precompute the size of the output that each SPE emits, thereby minimizing the communication and synchronization needed between the SPEs and PPE for posting and processing memory allocation requests. To compute the volume of output data that each SPE emits we measure statically and symbolically in a preprocessor, when feasible, the number of *MapReduce_emitIntermediate* calls in the user-specified Map function. Alternatively, the programmer can provide the same information using the *MapReduce_Specification_setEmitNum* function. In case neither the preprocessor can extract nor the programmer can provide this information, we instantiate and schedule *runahead map* tasks on SPEs. These

tasks are clones of the application’s real map tasks, with the computation of actual output values stripped off. The runahead map tasks execute only the number of *MapReduce_emitIntermediate* calls made from each SPE. Let N_i be the number of bytes that the runtime system schedules to send to the i -th SPE during the entire the map stage. The runtime system uses a fixed size datatype for all input (key, value) pairs and all intermediate (key, value) pairs. Let in_width be the size of input (key, value) pairs and im_width be the size of intermediate (key, value) pairs. Also, let $\#emit_calls_i$ be the number of *MapReduce_emitIntermediate* calls issued by the i -th SPE. Let $im_total_size_i$ be the total output size from the i -th SPE during the map stage. We compute this size as:

$$im_total_size_i = \frac{N_i}{in_width} \cdot im_width \cdot \#emit_calls_i \quad (1)$$

Besides this modification, our runtime uses self-scheduling of queued tasks on each SPE and multi-buffering to overlap task execution with the transfer of arguments of waiting tasks from DRAM to local stores and the write-back of the output of completed tasks from local stores to DRAM.

B. Partition

The reference design redistributes the data emitted from the map stage into a user-specified number of partitions, by hashing the keys of the emitted (key, value) pairs. This partitioning step executes on the PPE and requires memory copies. The PPE iterates through each key and copies the key to the corresponding partition, according to a hash value that SPEs precompute during the map stage. The partition phase guarantees that intermediate output in each partition is processed in key order and it is present in MapReduce runtimes to potentially accelerate the sorting stages. Due to memory copies and a relatively compute-inefficient control core, the partitioning stage creates a bottleneck.

On multicore processors with relatively fast access to shared memory from all cores, the runtime system can sort emitted data efficiently without data preprocessing in a partition stage. This is not necessarily true in loosely coupled MapReduce setups, such as clusters of workstations [1]. Contrary to the reference design, our runtime does not perform explicit hashing and partitioning of emitted data, thus eliminating the partition stage. Instead, our runtime uses a unified output buffer for emitted data (Figure 2) and implements implicit partitioning by setting the bounds of the output region of each SPE in the unified buffer. This design eliminates the memory copies of the partition stage, which the reference design implements on the PPE, and reduces PPE-SPE synchronization.

Furthermore, our runtime system does not split keys from values. Separation of keys from values improves spatial locality while streaming data to the SPE local stores and processing data from local stores using SIMD instructions [7].

In practice, several applications (e.g. Word Count, which is discussed in Section IV) use the same value for all emitted keys. Our runtime system avoids streaming values when these are identical for all keys, relying instead on key counting via address comparisons on SPEs to reduce rapidly the values of chunks of emitted keys with the same value. This reduces memory latency by avoiding redundant DMA transfers and achieves faster reduction of values from SPE local stores. In cases when keys are associated with different values, we do not find appreciable performance gains from splitting keys and values in terms of non-overlapped memory latency and accelerated key processing on the SPE, since splitting (key, value) pairs requires merging them back after the sort stages, a step that becomes needless in our design.

C. Quick-sort

The reference design sorts 64KB lists produced by the partition stage. The reference runtime system uses a buffer of size 128KB in the local stores and double-buffering, to perform in-place quick sort of 64 KB chunks on SPEs. The sorted lists are written back to memory with DMAs. Our runtime system uses the entire local store space made available for data (128KB) as a single buffer for in-place quick-sort in the local store sizes. Avoiding double-buffering sacrifices a potential reduction in memory latency (through prefetching) for the purpose of accelerating the following merge-sort stage. Specifically, using larger buffers in quick-sort implies the execution of less steps in the following merge-sort stage. The savings in memory latency from double-buffering in quick-sort do not outweigh the overhead of the additional step in merge-sort, which amounts to a parallel sort of at least as many 64 KB buffers as the number of SPEs. Our runtime system still uses multi-buffering in merge sort to overlap the write-back of merged buffers with their processing on SPEs.

D. Merge-sort

The merge-sort stage completes the sorting process by performing an external binary merge-sort, which is parallelized across SPEs. In each step of merge-sort, each SPE retrieves two sorted buffers from memory and merges them. The process continues until it produces two sorted buffers in memory, which are merged by the PPE. In the reference design, this step includes also the merging of keys and values in pairs, a step that is not needed in our design.

E. Reduce

Both the reference design and our design use the same strategy for executing the *reduce* stage, therefore we do not discuss this stage further.

IV. EXPERIMENTAL ANALYSIS

We evaluate our runtime system using application workloads implemented with the standard MapReduce API and

Application	Class	Input size
Word Count	partition-dominated	10MB
Histogram	partition-dominated	50MB
Linear Regression	map-dominated	128MB
Distributed Sort	sort-dominated	16MB

Table I: MapReduce application workloads.

Application	S_{self}		S_{best}		$S_{new,vs.old}$
	ref	new	ref	new	
Word Count	1.34 ₆	2.88 ₆	1.22 ₆	2.88 ₆	2.35×
Histogram	1.65 ₆	5.43 ₆	1.65 ₆	3.37 ₆	2.04×
Linear Regression	5.81 ₆	5.88 ₆	5.81 ₆	5.80 ₆	1.00×
Distributed Sort	1.32 ₆	2.61 ₆	1.15 ₆	2.61 ₆	2.27×
GeoMean	2.03	3.94	1.92	3.48	1.81×

Table II: Maximum speedup (number of SPEs on which it is reached in subscript) relative to the execution time with one SPE using the same MapReduce implementation (S_{self}), maximum speedup relative to the shortest execution time with one SPE between the two MapReduce implementations (S_{best}), and improvement in minimum execution time between the old and new implementations ($S_{new,vs.old}$). Higher is better in all cases.

used in an earlier evaluation of the reference design on Cell [7], with no modifications to their source code. Our experimental platform is a Sony PS3 with one Cell processor running at 3.2 GHz. The processor makes 6 SPEs available to user-level code. The PS3 has 256 MB of XDRAM and runs Linux Fedora Core 7 (kernel version 2.6.27). We performed experiments in single-user mode with all daemons deactivated to minimize noise from the operating system.

A. Application Workloads

Table I lists the MapReduce application workloads that we use to evaluate the new and reference runtime systems. Following conventions from [7], we classify applications into map-dominated, partition-dominated and sort-dominated, according to the phase of MapReduce where these applications spend most of their execution time. *Histogram* counts the frequency of occurrence of each RGB color component in an image file. The map function emits the occurrences of each color component in pixels and the reduce function produces the sum of occurrences of each component. *Word Count* counts the number of occurrences of each word in a text file. The map function splits the input into words, whereas the reduce function sums the number of occurrences of each word to produce a final count. *Kmeans* creates clusters from a set of data points, by finding the closest cluster for each data point in the map function and computing the cluster means in the reduce function. *Distributed Sort* sorts a set of quad-word integers in the sort stage of MapReduce. The map function is the identity function and the reduce function is empty.

B. Scalability and Performance

Table II provides indicators of the scalability of our MapReduce runtime system and the reference runtime sys-

Figure 4: Execution time breakdown of the sort stage.

tem on Cell [7]. These indicators are the maximum speedup of each implementation relative to the execution time of itself when using the PPE and one SPE core (S_{self}), the maximum speedup of each implementation relative to the fastest execution time between the two implementations when they use the PPE and one SPE core (S_{best}), and the absolute speedup of our implementation relative to the minimum execution time with the reference implementation.

The new design exhibits better scalability both with respect to itself and with respect to the fastest between the two designs when using one SPE. The geometric mean of S_{self} is 3.94× vs. 2.03× in the reference design, while the geometric mean of S_{best} is 3.48× versus 1.92× in the reference design. In terms of absolute performance, our design reduces the minimum execution time on average by 1.81× (geometric mean) compared to the reference design. Scalability and speedups are markedly better than the reference design in partition-dominated and sort-dominated applications (Word Count, Histogram, Distributed Sort), where the new design achieves 2.3× performance improvement on average. The scalability and performance of the new design is close to that of the reference design in map-dominated applications. Figure 4 illustrates the execution time for sorting in the Distributed Sort benchmark with and without double-buffering in the quick-sort stage. The result motivates the choice of eliminating double-buffering while sorting. In particular, double-buffering makes the quick-sort stage slightly faster but the merge-sort stage significantly slower. Figure 5 illustrates the execution time of all benchmarks with different input sizes. The results verify that our runtime system remains scalable across input sizes.

C. Analysis

Word Count: Figure 6 illustrates a breakdown of the execution time of Word Count on SPEs. We break execution time down into busy time that SPEs spend executing the compute portions of map, sort, and reduce tasks, and time that SPEs spend waiting for data from DRAM or notifications from the PPE for DMA completion, task completion, or availability

Figure 5: Execution time with varying input size.

Figure 6: Word Count execution time breakdown (SPE).

of new tasks. Each bar plots the average time across a given number of SPEs, ranging from 1 to 6. Although busy time is roughly equal from 3 SPEs and beyond and scales similarly in both implementations, our implementation reduces idle time on SPEs by a factor of $3.3\times$.

The reduction of idle time exhibits cumulatively the effects of our design. The new runtime reduces idle time by 1) preallocating buffers, therefore eliminating idle time that SPEs wait for the PPE to allocate buffers before emitting data after the map stage, 2) minimizing context switching and the associated overhead and cache miss penalties on the PPE, thereby reducing the critical path of instantiating tasks, scheduling tasks, and processing task completion notifications, 3) minimizing synchronization time for accessing buffers from the PPE, since only one thread processes buffers on the PPE side, 4) eliminating the redundant memory copies of the partition stage, 5) eliminating the data transfers and computation overhead of the first parallel step in merge-sort.

Figure 7 illustrates a breakdown of execution time on the PPE core. Specifically, we illustrate the execution time of

Figure 7: Word Count execution time breakdown (PPE).

Figure 8: Histogram execution time breakdown (SPE).

each component of MapReduce on the PPE. We decompose execution time into map, partition, quick sort, merge sort, and reduce components. Note that the time of each of these components on the PPE subsumes and overlaps with busy time (work) and idle time on SPEs. Our runtime system eliminates the overhead of partitioning, including buffer allocation, processing of keys for partitioning, and DMA transfers. It further reduces the execution time of the map stage by 1) reducing the overhead on the PPE for processing completion notifications and scheduling map tasks from and to SPEs respectively, 2) reducing DMA and synchronization time further by not sending requests to the PPE to allocate new output buffers in main memory, 3) eliminating the transfers of key values to and from local stores. Map scales significantly better in our implementation, indicating that reducing control overhead on the PPE is critical. No stage benefits from stage overlapping in the original design, due to control overhead. Our event-driven controller design for the main thread and the scheduling thread addresses this problem.

The new runtime reduces total sort time (quick-sort and merge-sort) by a factor of $2.1\times$. The new implementation accelerates sorting by 1) minimizing wait time on the PPE to schedule sort tasks on SPEs and process their completion notifications, 2) eliminating synchronization between threads on the PPE for concurrent buffer processing, 3) accelerating merge-sort by eliminating its first parallel step. Reduction time is negligible in Word Count.

Histogram: Figure 8 and Figure 9 show SPE and PPE breakdowns of execution time in Histogram, respectively.

Figure 9: Histogram execution time breakdown (PPE).

Figure 10: Distributed Sort execution time breakdown (SPE).

Although in Histogram the new runtime exhibits the same trends that we observe for Word Count in terms of scaling, reducing idle time on SPEs, and reducing overall execution time (due to acceleration of the partitioning and sorting stages), the execution time of tasks increases significantly during the map stage. Furthermore, total busy time on SPEs is constantly higher in our implementation. Histogram is the only application where our runtime uses runahead map tasks to compute the size of output buffers for intermediate data. The increased busy time reflects the overhead of runahead tasks. Reduction time is negligible in Histogram.

Distributed Sort: Figure 10 shows the breakdown of execution time on SPEs and Figure 11 shows the breakdown of execution time on the PPE for Distributed Sort. As expected, Distributed Sort obtains the maximum benefit from acceleration of the quick-sort and merge-sort stages. Overall, the quantitative and qualitative behavior of Distributed Sort is identical to that of Word Count, except from a sizable acceleration of map in our runtime system, attributed to the elimination of value transfers and buffer allocation overhead.

Linear Regression: Figure 12 and Figure 13 show breakdowns of execution time of Linear Regression on SPEs and the PPE respectively. Linear regression is map-dominated and both the reference design and our design of the MapReduce runtime perform similarly.

V. RELATED WORK

Several related research efforts focus on porting MapReduce to prominent hardware platforms for high-performance

Figure 11: Distributed Sort execution time breakdown (PPE).

Figure 12: LinearRegression execution time breakdown (SPE).

computing, including multicore processors [8], [10], [11], GPUs [6], [19] the Cell processor [7], [9] and FPGAs via direct software to hardware translation [20].

Throughout this paper, we compare our runtime system design and implementation against the design and implementation of MapReduce for the Cell proposed by de Kruijff and Sankaralingam [7]. As discussed in Section III, our design revamps the multithreading control code for task instantiation, task scheduling, synchronization, memory allocation, and processing of completion notifications. This reduces operating system and runtime system overhead and exposes more parallelism to the compute-efficient SPE cores of the Cell. Furthermore, our runtime system replaces explicit partitioning with implicit partitioning and precomputation of partition boundaries to avoid redundant memory copies, avoids segregation of values and keys, and reduces the critical path of merge-sort. Other related work on porting MapReduce on Cell [9] focuses on optimizing communication between Cell processors and external control nodes, rather than specific optimizations on the Cell processor.

MapReduce is included as a programming API in Intel's Merge programming framework [8]. Merge focuses on providing compiler and runtime support for selecting between variants of map and reduce functions when targeting different core architectures on the same heterogeneous processor. Compared to Merge [8], [21], which is based on a shared virtual address space, our design provides a scalable implementation for multicore processors with private address spaces and explicit communication between cores and DRAM.

Figure 13: LinearRegression execution time breakdown (PPE).

Phoenix, a port of MapReduce for cache-coherent shared-memory multicore systems [10], [11], exploits locality implicitly by controlling the granularity of tasks and the assignment of tasks to cores. Similarly to our port on Cell, Phoenix performs dynamic assignment of map and reduce tasks to cores. Phoenix controls task sizes so that the working set of each task fits in the L1 cache. Phoenix also provides an option to perform prefetching in the L2 data cache. Our runtime controls locality explicitly, using DMAs and software prefetching via multi-buffering in map and merge-sort. Contrary to Phoenix, our runtime system does not hash and partition keys in per-core buffers, thereby eliminating memory copies, while still allowing a balanced distribution of work during the sort and reduce stages. Furthermore, our implementation addresses heterogeneity, via extensive optimization of the code running on control-efficient cores.

Implementations of MapReduce on GPUs also consider implications of explicitly-managed local memories [5], [6], [19]. Similarly to our runahead map tasks, Mars [6] uses mock map tasks to compute the sizes of buffers needed by each core for emitting results of real map tasks. Contrary to Mars, our runtime performs efficient memory preallocation for map tasks for which we can compute statically the volume of emitted data. Furthermore, the dynamic memory management scheme in Mars is specific to the GPU memory system, whereas our memory management scheme is general enough to be applied to any multicore processor, including homogeneous processors with coherent caches. Other optimizations of MapReduce on GPUs focus on achieving fine-grain interleaving of memory accesses from threads on the GPU, to utilize the GPU memory bandwidth. Our implementation focuses instead on reducing control overhead and avoiding redundant data transfers and memory copies, that cost dearly on state of the art processors.

Recent work on programming models for heterogeneous multicore processors with explicitly managed memories includes several proposals for hiding the complexity of managing local stores and heterogeneity in the programming model, through compiler directives [22], [23] or language constructs [24]. MapReduce hides the memory hierarchy and heterogeneity in a library API and our implementation preserves these desirable properties.

VI. CONCLUSIONS

Efficient large-scale parallel processing of inordinate amounts of data is essential in numerous application domains, ranging from Internet-scale enterprise applications to petascale computing. These demanding applications have no option but to leverage high-performance multicore processors and run them at near peak performance, to deliver adequate throughput to users. In this work we presented a new design and an implementation of MapReduce, a high-level parallel programming model for large-scale data processing, targeting multicore processors with heterogeneous cores and explicitly managed memory hierarchies.

We demonstrated methods to minimize control overhead by overlapping task instantiation with task scheduling, using event-driven controllers and a bulk-synchronous execution scheme. These methods replace prior state of the art that relies on expensive software multithreading that oversubscribes cores, to overlap MapReduce tasks. Our results motivate us to argue that runtime systems should favor simplicity in their control structures whenever applicable. Complexity in control should be favored if adequate control-efficient resources are available on the processor to make control overhead insignificant. We find that this is not the case on Cell, without discounting that we may have to revisit this conclusion for future processor technologies.

We further demonstrated several MapReduce-specific optimizations including a new scheme for distributing and sorting intermediate data and a memory management scheme for minimizing overhead in applications with both statically known and statically unknown demand for memory. Put together, our techniques improved substantially performance and scalability compared to the state of the art.

Our work is amenable to further improvements through integration with processor-specific code optimization frameworks and libraries. Our runtime system will benefit from automatic or semi-automatic vectorization frameworks, better load-balancing in all execution stages, improved OS schedulers and memory management schemes [17], and further fine-tuning of sorting algorithms [25]. Furthermore, while our work is processor-centric, it is by no means a complete treatment of a scalable MapReduce runtime framework. Scaling the runtime system to multi-processors with multicore processors and clusters of heterogeneous components still presents important challenges [9]. Providing efficient processor-specific schemes for fault tolerance in MapReduce is also essential, since the state of the art relies on simple solutions such as timeouts [11]. The aforementioned topics are subjects of ongoing investigation.

REFERENCES

- [1] J. Dean and S. Ghemawat, "Mapreduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

- [2] D. Robinson, *Amazon Web Services Made Simple: Learn how Amazon EC2, S3, SimpleDB and SQS Web Services enables you to reach business goals faster*. London, UK, UK: Emereo Pty Ltd, 2008.
- [3] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, "Map-Reduce for Machine Learning on Multicore," in *NIPS'06: Proc. of the 20th International Conference on Neural Information Processing Systems*, Vancouver, Canada, Dec. 2006, pp. 281–288.
- [4] Adam Pisoni, "Skynet," Apr. 2008, <http://skynet.rubyforge.org>.
- [5] B. Catanzaro, N. Sundaram, and K. Keutzer, "A Map Reduce Framework for Programming Graphics Processors," in *STMCS'08: Proceedings of the Third Workshop on Software and Tools for Multicore Systems*, Boston, MA, USA, Apr. 2008.
- [6] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a MapReduce Framework on Graphics Processors," in *PACT'08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Toronto, Ontario, Canada, Oct. 2008, pp. 260–269.
- [7] M. de Krujif and K. Sankaralingam, "Mapreduce for the Cell B.E. Architecture," *IBM Journal of Research and Development*, vol. 53, no. 5, pp. 10:1–10:12, Sep. 2009.
- [8] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a Programming Model for Heterogeneous Multi-core Systems," in *ASPLOS XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, Seattle, WA, USA, Mar. 2008, pp. 287–296.
- [9] M. M. Rafique, B. Rose, A. R. Butt, and D. S. Nikolopoulos, "Supporting mapreduce on large-scale asymmetric multi-core clusters," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 25–34, Apr. 2009.
- [10] 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 IEEE International Symposium on High Performance Computer Architecture*, Phoenix, AZ, USA, Feb. 2007, pp. 13–24.
- [11] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, Austin, TX, USA, Oct. 2009, pp. 198–207.
- [12] P. Hofstee and M. Day, "Hardware and Software Architectures for the CELL Processor," in *CODES+ISSS'05: Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, Jersey City, NJ, USA, Sep. 2005, pp. 1–1.
- [13] M. Harris, "Many-core GPU Computing with NVIDIA CUDA," in *ICS'08: Proceedings of the 22nd ACM International Conference on Supercomputing*, Island of Kos, Greece, Jun. 2008, pp. 1–1.
- [14] L. Seiler *et al.*, "Larrabee: a Many-core x86 Architecture for Visual Computing," in *SIGGRAPH'08: Proceedings of the 35th ACM International Conference and Exhibition on Computer Graphics and Interactive Techniques*, Los Angeles, CA, USA, Aug. 2008, pp. 1–15.
- [15] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," *IEEE Computer*, vol. 41, no. 7, pp. 33–38, Jul. 2008.
- [16] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell Broadband Engine Architecture and its First Implementation - A Performance View," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 559–572, Sep. 2007.
- [17] F. Blagojevic, C. Iancu, K. Yelick, M. Curtis-Maury, D. S. Nikolopoulos, and B. Rose, "Scheduling Dynamic Parallelism on Accelerators," in *CF'09: Proceedings of the 6th ACM Conference on Computing Frontiers*, Ischia, Italy, May 2009, pp. 161–170.
- [18] IBM Corp., "Cell Broadband Engine Software Development Kit 2.1 Programmer's Guide (Version 2.1)," 2006.
- [19] W. Ma and G. Agrawal, "A Translation System for Enabling Data Mining Applications on GPUs," in *ICS'09: Proceedings of the 23rd ACM International Conference on Supercomputing*, Yorktown Heights, NY, USA, Jun. 2009, pp. 400–409.
- [20] J. H. Yeung *et al.*, "Map-Reduce as a Programming Model for Custom Computing Machines," in *FCCM'08: Proceedings of the 16th IEEE Symposium on Field-Programmable Custom Computing Machines*, Palo Alto, CA, Apr. 2008, pp. 149–159.
- [21] P. H. Wang *et al.*, "EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-Core Multithreaded System," in *PLDI'07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, USA, Jun. 2007, pp. 156–166.
- [22] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: a Programming Model for the Cell BE Architecture," in *SC'06: Proceedings of the ACM/IEEE Supercomputing'2006: Conference on High Performance Computing, Networking, Storage and Analysis*, Tampa, FL, USA, Nov 2006, p. 86.
- [23] S. Schneider, J. Yeom, B. Rose, J. C. Linford, A. Sandu, and D. S. Nikolopoulos, "A Comparison of Programming Models for Multiprocessors with Explicitly Managed Memory Hierarchies," in *PPoPP'09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, USA, Feb. 2009, pp. 131–140.
- [24] K. Fatahalian *et al.*, "Sequoia: Programming the Memory Hierarchy," in *SC'06: Proceedings of the ACM/IEEE Supercomputing'2006: Conference on High Performance Computing, Networking, Storage and Analysis*, Tampa, FL, USA, Nov. 2006, p. 83.
- [25] B. Gedik, R. R. Bordawekar, and P. S. Yu, "Cellsort: High Performance Sorting on the Cell Processor," in *VLDB'07: Proceedings of the 33rd International Conference on Very Large Data Bases*, Vienna, Austria, Sep. 2007, pp. 1286–1297.