

Scheduling Dynamic Parallelism on Accelerators

Filip Blagojević, Costin Iancu,
Katherine Yelick

Lawrence Berkeley National Laboratory
1 Cyclotron Road
Berkeley CA 94720
{fblagojevic, cciancu,
kayelick}@lbl.gov

Matthew Curtis-Maury

NetApp, Inc 7301 Kit Creek
Rd, RTP, NC 27709
mcm@netapp.com

Dimitrios S. Nikolopoulos,
Benjamin Rose

Virginia Tech
Blacksburg, VA 24060
{dsn,bar234}@cs.vt.edu

ABSTRACT

Resource management on accelerator based systems is complicated by the disjoint nature of the main CPU and accelerator, which involves separate memory hierarchies, different degrees of parallelism, and relatively high cost of communicating between them. For applications with irregular parallelism, where work is dynamically created based on other computations, the accelerators may both consume and produce work. To maintain load balance, the accelerators hand work back to the CPU to be scheduled. In this paper we consider multiple approaches for such scheduling problems and use the Cell BE system to demonstrate the different schedulers and the trade-offs between them. Our evaluation is done with both microbenchmarks and two bioinformatics applications (PBPI and RAXML). Our baseline approach uses a standard Linux scheduler on the CPU, possibly with more than one process per CPU. We then consider the addition of cooperative scheduling to the Linux kernel and a user-level work-stealing approach. The two cooperative approaches are able to decrease SPE idle time, by 30% and 70%, respectively, relative to the baseline scheduler. In both cases we believe the changes required to application level codes, e.g., a program written with MPI processes that use accelerator based compute nodes, is reasonable, although the kernel level approach provides more generality and ease of implementation, but often less performance than work stealing approach.

Categories and Subject Descriptors

D.4.1 [Operating Systems]: Process Management — Concurrency, Scheduling, Synchronization; C.1.3 [Processor Architectures]: Other Architecture Styles — Heterogeneous (hybrid) systems

General Terms

Performance, Design

Copyright 2009 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
CF'09, May 18–20, 2009, Ischia, Italy.
Copyright 2009 ACM 978-1-60558-413-3/09/05 ...\$5.00.

Keywords

Cooperative Scheduling, Cell BE

1. INTRODUCTION

In recent years accelerator based parallel architectures have been resurrected as a viable alternative for system design and there is a large body of work [10, 19, 9, 16, 22, 25, 17, 24] indicating that these architectures are able to provide high performance with very low power consumption. The Cell BE is an exponent of such architecture and Cell based clusters such as the Roadrunner system at Los Alamos National Laboratory are used in application settings such as high performance scientific computing or financial data analysis.

Accelerator based architectures are asymmetrical, contain both general purpose processors and specialized hardware and often exhibit different degrees of hardware parallelism inside each hierarchy. The Cell BE contains one PowerPC core (PPE) with two hardware execution contexts and eight specialized processors (SPEs). GPU based solutions have a very deep level of parallelism (thousands) and are often integrated into systems with generic multi-core processors with limited parallelism (tens). A common execution model for these architectures assumes a cooperation between the general purpose processors and the accelerator hardware: the ported applications usually have a “driver” running on the main processor side and offload parts of the computation to the specialized hardware. One of the open research problems is mapping applications to the two independent execution hierarchies that exhibit different degrees of parallelism and providing efficient synchronization and coordination.

Previous work [5, 7] investigates the mapping of parallel computation to the Cell BE and shows the importance of the careful management of the PPE ↔ SPE interaction: explicitly interacting with the Linux scheduler and yielding the PPE whenever processes are waiting for SPE tasks to finish is able to achieve many-fold application performance improvement. Yielding the PPE processors improves the attentiveness of the computation to asynchronous events generated by SPEs which results in a higher overall SPE utilization, but it is still subject to the implicit Linux kernel scheduling policies. The Linux’s scheduler lack of knowledge about the asynchronous nature of the SPE computation is likely to increase the latency of any response to the SPE generated events.

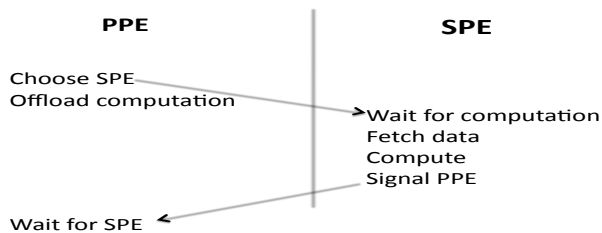


Figure 1: Generic structure of code implementing offloading and PPE↔SPE interaction.

In this paper we examine kernel and user-level scheduling techniques to minimize the latency of PPE processes response to SPE generated events. In both implementations we use an event-driven cooperative scheduling approach; SPE tasks indicate which PPE process is required to serve their requests. The kernel level implementation extends the Linux kernel with data structures and system calls which allow SPEs to request activation of a specific process on the PPE side. For the user-level approach we use shared memory abstractions and a work stealing strategy. We evaluate the performance of the proposed techniques on the Sony PlayStation3 using micro-benchmarks and two bioinformatics applications: PBPI and RAXML. When compared to the performance of the default Linux scheduling, the results indicate that using our kernel modifications we can reduce SPE idle time by up to 30%, while work-stealing provides improvements of 70%. The reduction of SPE idle time translates directly into application performance improvements: we obtain an average speedup of 5%, with maximum performance improvements of 10% using kernel scheduling. Using work-stealing we get speedups of 12% and 21% respectively.

The principles described in this paper are widely applicable and of interest to several classes of software developers on accelerator based systems: application and framework developers, runtime implementors (e.g. MPI, OpenMP, Charm++, RapidMind) and developers of communication libraries for clusters of accelerator based systems.

2. CELL BE PROGRAMMING

Besides having to address the architectural idiosyncrasies (alignment restrictions for data transfers, explicitly managed local storage), the main challenges in achieving good performance on the Cell BE are choosing the right execution model and parallelization strategy. The offloading execution model assumes frequent PPE↔PPE and PPE↔SPE interaction but little SPE↔SPE interaction: there is a driver application executing on the PPE that offloads SPE computations that are independent of each other. This approach has been shown to provide good performance in practice and many application studies on the Cell BE employ it. The “offloading” approach is of particular interest when considering porting existing parallel applications in a Cell cluster environment or when using a compiler for parallelizing the Cell code in the OpenMP fashion. The principles described in this paper are directly applicable in both situations.

Several software development infrastructures for the Cell

BE, including ours, provide offloading APIs, e.g. IBM ALF [2], Charm++ Offloading API [1], Map/Reduce for Cell [12], RapidMind [20]; sometimes ad-hoc implementations are provided by developers. We provide a generic offloading API for the Cell BE that allows programmers to select SPEs to perform computations, move data between the local store and main memory and in particular to allocate and manage synchronization objects in main memory.

For parallel applications designed for execution in a cluster environment, partial PPE execution (and oversubscription) is almost required, as communication libraries are not designed to run on accelerators. Also, oversubscription with offloading is an execution model that requires minimal changes for already existing applications to run on Cell. Offloading execution requires significant PPE↔SPE communication. The generic structure of the PPE↔SPE code is presented in Figure 1. The asynchronous interaction between PPE and SPE is handled in the function `wait_for_SPE` and various mechanisms can be used for its implementation. In the rest of this paper we discuss the implementation and performance trade-offs of different approaches to implement fast PPE↔SPE synchronization and cooperation on the Cell BE. We compare asynchronous event handling using IBM Cell SDK primitives (callbacks and mailboxes) with kernel extensions for cooperative scheduling and with user level implementations for work stealing. Figure 2 illustrates the importance of fast synchronization and cooperative scheduling in RAXML (application described in Section 4) running on a cluster of Sony PlayStation3 consoles, when the PPE is oversubscribed with 6 MPI processes, each off-loading on 1 SPE (the Cell processor on the PS3 exposes only 6 SPEs to the application). The results show that compared to a scheduling policy which is oblivious to PPE ↔ SPE co-scheduling (labeled as Linux in Figure 2), cooperative scheduling (labeled as yield-if-not-ready in Figure 2) achieves a performance improvement of 1.7–2.7×. The optimal balance and granularity of parallelism is not easy to determine in most application settings and high degree of oversubscription is likely to provide the best performance in many scenarios [8, 6]. To illustrate benefits of oversubscription, Figure 3 presents the performance of RAXML for various configurations of assignments of SPEs to PPE processes. Note that the performance of configurations with the highest degree of oversubscription, that use 6 PPE processes with one SPE assigned to each, is two to three times faster than configurations that assign 6 SPEs to one PPE process. In this case, mechanisms to increase the responsiveness of PPE processes to asynchronous SPE requests are likely to improve performance.

3. HANDLING ASYNCHRONOUS EVENTS

Achieving good performance on the Cell BE requires careful orchestration of the PPE ↔ SPE interaction. Co-scheduling of PPE ↔ SPE parallelism is a technique discussed in [5] that has been shown to significantly improve the performance of several applications. The main objective of co-scheduling is to maximize SPE utilization, i.e. minimize the waiting time whenever a thread off-loaded to an SPE needs to communicate or synchronize with its originating thread on the PPE. This is achieved by increasing the chance that the required PPE thread will run as soon as an SPE request is generated. Figure 4 illustrates how different implementations of the PPE ↔ SPE synchronization function can have a significant impact on co-scheduling utilization. In Fig-

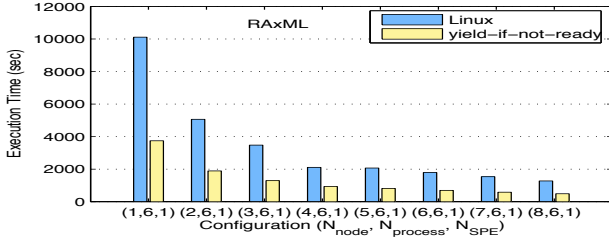


Figure 2: Performance of yield-if-not-ready policy and the native Linux scheduler in RAxML. N_{node} - number of nodes, $N_{process}$ - number of processes per node, N_{SPE} - number of SPEs per process.

ure 4(a), PPE threads are busy-waiting for the corresponding off-loaded threads to return results from SPEs. The time quantum allocated to each PPE thread by the OS can cause continuous mis-scheduling with respect to SPE threads.

Busy-waiting can be combined with explicit yielding in order to eliminate some of the mis-scheduling. Figure 4(b) illustrates a *yield-if-not-ready* implementation, where PPE threads explicitly yield the processor whenever a corresponding off-loaded SPE thread is pending completion. As discussed in the previous section, *yield-if-not-ready* is able to achieve many-fold application performance improvement. In the rest of this paper, for brevity we refer to the yield-if-not-ready policy as YNR.

Although YNR reduces the PPE response time compared to the busy waiting, the mis-scheduling can still occur, as presented in Figure 4(b). If p is the total number of active PPE threads, the YNR policy bounds the slack by the time needed to context switch across $p - 1$ PPE threads. On a Sony PlayStation3, the upper bound on slack when considering 6 active processes (as many as there are available SPEs) is around $15\mu s$, as determined by the duration of context switches using `sched_yield()` calls under congestion. In the ideal implementation, a PPE thread is run as soon as an SPE request has been generated. Figure 4(c) illustrates the optimal scheduling situation. In order to achieve the optimal scheduling, the thread scheduler needs information as to what PPE thread should run to serve the SPE requests. The efficiency of any cooperative scheduling approach is ultimately determined by the system response time – activation of the threads able to handle any outstanding events.

Besides hand coded libraries for fast synchronization using variables in memory, asynchronous PPE \leftrightarrow SPE interaction can be implemented using two mechanisms provided by the IBM Cell SDK. A process can register an event handler (callback) to respond to SPE requests. An SPE that triggers an event handler will generate a PPE interrupt and it will block until the handler completes. Processes executing on the PPE side have to observe the interrupt and respond to the SPE request. The latency of this response time is a combination of the software overhead of event handling and the time for a process to be scheduled on the PPE, given the default Linux scheduling policy. In order to decrease this latency, the synchronization code on the PPE has to be implemented using YNR. The second signaling mechanism uses interrupt mailboxes. A PPE process can perform blocking calls to read a mailbox, the process is suspended until data becomes available. In general, given the fact that in

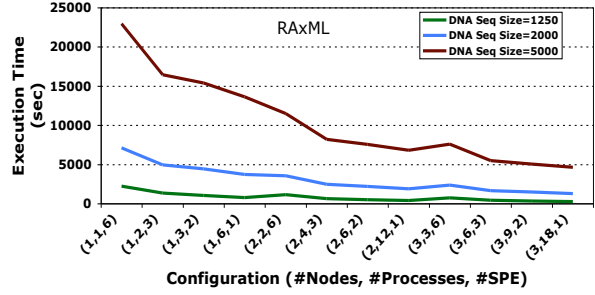


Figure 3: Performance improvements with process over-subscription in RAxML. N_{node} - number of nodes, $N_{process}$ - number of processes per node, N_{SPE} - number of SPEs per process.

Linux events are serviced only when a process is scheduled on the CPU, both approaches have to use a YNR scheme and their performance is likely to be lower than a pure YNR scheme using the local store for PPE \leftrightarrow SPE signaling.

Kernel support can be used to implement efficient cooperative scheduling and decrease response latency. We have experimented with the real-time scheduling features available in Linux and implemented versions of PBPI and RAxML where we force a process to run by increasing its priority. This approach requires a very thorough understanding of the application characteristics¹ in order to avoid process starvation. In order to avoid the drawbacks of real-time scheduling we extend the Linux kernel scheduler with a `_yield_to()` system call and use the main memory for PPE \leftrightarrow SPE signaling. For brevity, we refer to this approach as SLED (Slack-Minimizing Event Driven synchronization) and its design is presented in the rest of this section. Our final solution does provide the best performance for micro-benchmarks and applications, and we believe it to be the most intuitive to developers.

3.1 Fast Synchronization Using `_yield_to()`

The overview of the SLED implementation is illustrated in Figures 5 and 6. The basic data structure for PPE-SPE signaling is a `ready_to_run` “list”. After offloading, PPE processes explicitly call the `SLEDS_wait_for_SPE()` function. Each SPE thread upon completing the assigned task writes the PID of the parent process to the shared `ready_to_run` data structure: the presence of a particular PID in any entry indicates that the corresponding SPE is waiting for a response from that particular process. Providing cooperative scheduling requires two distinct steps: 1) choosing a process to run from the list and 2) scheduling the chosen process on a PPE context. For the latter, the Linux kernel scheduling code has to be extended to give priority to a given process.

`SLEDS_wait_for_SPE()` can be implemented either at user-level or as a system call. The organization of the `ready_to_run` data structure determines whether the selection has to be made inside a critical section. A kernel level implementation will perform this operation inside a critical section by default while a user level implementation might have to provide its own mutual exclusion. In order to increase system responsiveness, the duration of these critical sections has to be minimized. In the rest of this section we discuss the de-

¹For example the RAxML application uses a master-worker approach and avoiding live-lock using real-time scheduling required a significant effort.

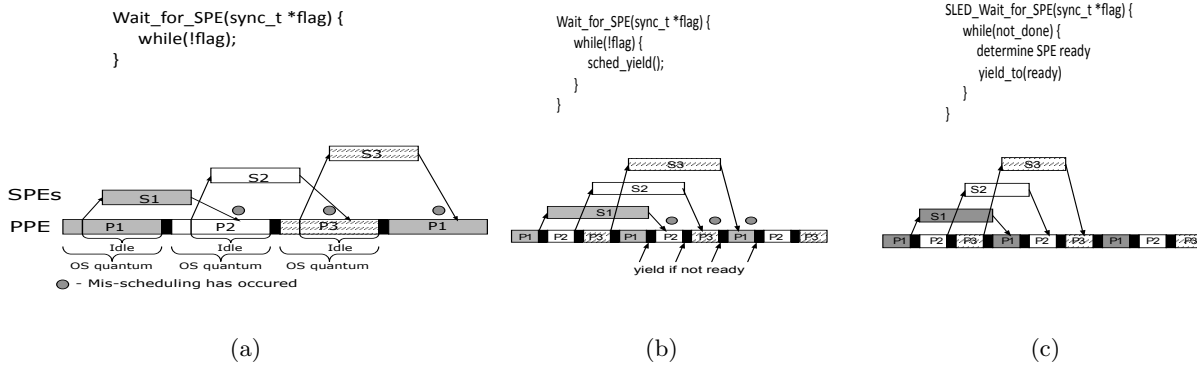


Figure 4: Three cases illustrating the importance of co-scheduling PPE threads and SPE threads. PPE (P) threads poll shared memory locations directly to detect if a previously off-loaded SPE (S) thread has completed. Dark intervals indicate context-switching. Dots mark cases of mis-scheduling.

sign trade-offs for 1) list organization and management; 2) implementing `SLEDS_wait_for_SPE()` as a user-level function or a system call.

Figure 6 presents the implementation of the `SLEDS_wait_for_SPE()` call; same implementation can be used at user-level as well as for a system call. As a first step, `SLEDS_wait_for_SPE` scans the `ready_to_run` list in order to determine the next process to run. Depending on the implementation choice, i.e. kernel or user level, the duration of the scan (parameter N) determines the system responsiveness. On a PlayStation3, which allows access to only 6 SPEs, and using the split list implementation described in Section 3.1.1, the `ready_to_run` list contains only 3 entries. We will discuss the influence of the parameter N in Section 4.1. The function `_yield_to()` is a system call which schedules a specific process to run; this is an extension to the standard Linux kernel.

3.1.1 The Synchronization Data Structure: `ready_to_run`

We have experimented with several data structures for the implementation of the `ready_to_run` “list”. A FIFO data structure would ensure that the process corresponding to the SPE thread which was the first to finish processing would be the first to run on the PPE side; it would also provide some fairness of the SPE allocation and avoid starvation. The downside of a FIFO data structure is that maintaining its consistency requires critical sections which introduce additional overhead. In order to avoid the locking overhead required by FIFO, a data structure with one entry per SPE can be used, as shown in Figure 5. Consistency is easily maintained using atomic instructions but the scheduler has to repeatedly scan the entire list in order to choose the next candidate. Scanning the entire list introduces additional overhead which influences the system responsiveness. User level implementations are efficient but there exists the potential for unfairness and starvation.

The Linux kernel maintains per CPU running queues, while the approaches described so far provide one data structure shared between all running processes. With this design choice, the implementation has to provide CPU context awareness as illustrated by the following scenario. The off-loaded task which belongs to the process P_1 has finished processing on the SPE side (the PID of the process P_1 has been written to the `ready_to_run` list). Process P_1 is bound to CPU_1 , but the process P_2 which is running on CPU_2 off-loads, and initiates the context switch by passing the PID of process P_1 to the kernel. Since the context switch

occurred on CPU_2 and P_1 is bound to run on CPU_1 , the kernel needs to migrate process P_1 to CPU_2 . This situation can be solved by explicitly performing process migration inside our extended kernel scheduling (`_yield_to()`) or by ensuring that processes scanning the `ready_to_run` list will never select a process with different CPU affinity. The situation occurs frequently in practice and both solutions introduce additional scheduling overhead. We have experimented with both approaches and found the migration solution the most undesirable since it creates an uneven distribution of processes across available CPUs. Depending on the implementation of the next process selection, affinity checks are performed either at user level or inside the kernel. User level implementations have to perform two additional system calls to obtain affinity information and this leads to significant performance degradation when compared to the kernel implementation.

To avoid some of the drawbacks discussed in this section we use an implementation for the `ready_to_run` data structure that contains one entry per SPE. The list is statically split in two halves, each PPE hardware execution context being responsible for managing one half; only processes sharing the execution context are accessing the same `ready_to_run` list. In order to eliminate the need for affinity awareness, the application is required to pin its processes to CPUs at job start-up time. This approach reduces most of the overhead associated with the data structure maintenance but it has some potential for load unbalance.

3.1.2 `_yield_to()` Implementation

The standard scheduler used in the Linux kernel, starting from version 2.6.23, is the Completely Fair Scheduler (CFS). For each process in the system, the CFS records the amount of time that the process has been waiting to be scheduled on the CPU. Based on the amount of time spent waiting in the run queue and the number of processes in the system, as well as the static priority of the process, each process is assigned a dynamic priority. The dynamic priority of a process is used to determine when and for how long the process will be scheduled to run. The data structure used by the CFS for storing the active processes is a red-black tree. The processes are stored in the nodes of the tree, and the process with the highest dynamic priority (which will be the first to run on the CPU) is stored in the left most node in the tree. The SLED scheduler passes the information from the `ready_to_run` list to the kernel through the `_yield_to()` system

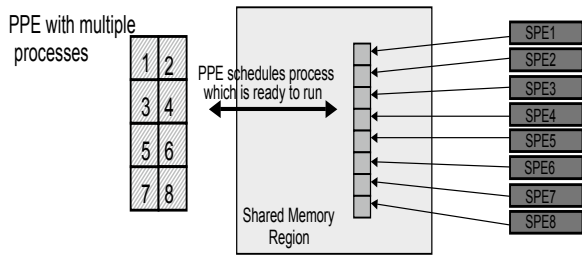


Figure 5: Upon completing the assigned tasks, the SPEs send signal to the PPE processes through the ready-to-run list. The PPE process which decides to yield passes the data from the ready-to-run list to the kernel, which in return can schedule the appropriate process on the PPE.

call, which extends the standard kernel scheduler `sched_yield()` system call by accepting an integer parameter `pid` which represents the process that should be the next to run. First, the process which should be the next to run is pulled out from the tree, and its static priority is increased to the maximum value. The process is then returned to the running tree, where it will be stored in the left most node (since it has the highest priority). After being returned to the tree, the static priority of the process is decreased to the normal value. Besides increasing the static priority of the process, we also increase the time that the process is supposed to run on the CPU. Increasing the CPU time is important, since if a process is artificially scheduled to run many times, it might exhaust all the CPU time that it was assigned by the Linux scheduler. In that case, although we are capable of scheduling the process to run on the CPU using the `_yield_to()` function, the process will almost immediately be switched out by the kernel. Before it exits, the `_yield_to()` function calls the kernel-level `schedule()` function which initiates context switching. We have measured the overhead in the `_yield_to()` system call caused by the operations performed on the running tree and we found it to be approximately 8% compared to the standard `sched_yield()` system call.

3.2 User-Level Work-Stealing

The SLED scheduler requires kernel modifications to implement the `_yield_to()` call. In addition, applications relying on SLED or YNR have to perform repeated system calls and introduce additional overhead in order to achieve the desired scheduling strategy. We investigate an implementation of the same principles using a user level approach based on work-stealing. In the work-stealing approach a PPE process which is currently running checks the user-level shared work queue and picks the right SPE to respond to, even if that SPE does not belong to the currently running PPE process. To enable the work-stealing execution, we need to provide "any-to-any" communication between the PPE processes and SPE threads.

On the Cell BE, each SPE local storage is memory mapped into the address space of the PPE process that had spawned the computation. Execution model where "any-to-any" communication (between the PPE processes/threads and SPEs) is allowed, requires a shared address space between the execution entities on the PPE side. Thread libraries such as `pthread`s can easily provide the shared memory abstraction,

```

1: void SLED_wait_for_SPE(){
2:   int next, i, j=0;
3:   if(!ready_to_run[mySPE]) {
4:     while(next == 0 && j < N){
5:       i=0;
6:       j++;
7:       while(next == 0 && i < 3){
8:         next = ready_to_run[i];
9:         i++;
10:      }
11:    }
12:    _yield_to(next);
13:  }
14: }

```

Figure 6: Outline of `SLED_wait_for_SPE`. Line 4-11: selection of next process. Line 12: `_yield_to()` if "local" SPE not ready.

however our applications of interest have been parallelized on the PPE side using MPI² and most MPI implementations are not thread safe. Furthermore, any existing scientific applications which is MPI based, is likely to be ported to Cell using the MPI communication. Inserting another (thread) level of parallelism into the application is possible, but requires significant programming effort.

In the interest of portability and correctness, we had to provide a shared memory space between all MPI processes running on the PPE. In order to achieve this we extend an experimental implementation of the Berkeley UPC [11] compiler, enabling a usage of shared memory across distinct processes. This implementation will be made available in the near future. UPC is a Partitioned Global Address Space language that provides a shared memory abstraction, uses a SPMD programming model, allows control over data layout and can easily interoperate with MPI programs.

We use application specific implementations for work stealing in both PBPI and RAXML, described in Section 4. In order to enable work stealing we had to ensure that all data structures are allocated in a shared memory region which is obtained at program start-up. We do provide memory allocators for the management of this region. The basic abstraction we provide is a shared work queue. Work is described by the parameters required for offloads and by variables necessary to identify the state of the computation for each PPE process. In order to ensure termination, these control variables are allocated in the shared memory region.

Both applications are written in a recursive style. Since the depth of the recursion is not apriori known, computations in the original code are offloaded only from the terminal branch of the recursion. In order to simplify the management of the shared work queue and avoid a full continuation passing style implementation, we manually eliminated recursion in both applications. The modified applications precompute all the work descriptors and fill in the shared work queue. PPE processes repeatedly scan the `ready_to_run` list and spawn the next work belonging to the process requested by an SPE. Unfolding recursion has the additional benefit of increasing the granularity of the offloaded task. The results reported for the work-stealing implementation use the same granularities as the original implementation in order to provide a fair comparison.

²There are Linux processes running on the PPE.

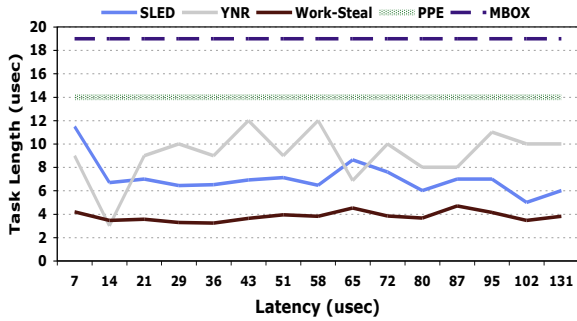


Figure 7: Variation of the latency of PPE-SPE response time for YNR, SLED and work-stealing with increasing task lengths.

4. EXPERIMENTAL SETUP

In all experiments we use the Cell processor embedded in the Sony PlayStation3 console and the variant of the 2.6.23 Linux kernel version specially adapted for it. This is the kernel we have extended with SLED support. We use the IBM Cell SDK3.0 to compile all applications and an experimental version of the Berkeley UPC software for the work-stealing implementation. The PS3 allows user level access to only six SPEs and this represents the upper bound on PPE oversubscription we have explored in all experiments. We investigate the impact of our techniques using micro-benchmarks and two real-world bioinformatics applications: RAXML and PBPI.

4.1 Micro-benchmark Results

To understand the implementation trade-offs we use a micro-benchmark where multiple MPI processes repeatedly offload tasks to SPEs. The PPE only initiates the off-loading of variable length tasks and waits for their completion. All results described in this section were obtained with the PPE oversubscribed with 6 MPI processes.

Figure 7 presents the variation of the SPE idle time with increasing lengths of the offloaded task. The SPE idle time is directly determined by the latency of the signaling mechanisms employed. Work-stealing exhibits the lowest overhead and the average SPE idle time is around $3\mu s$. With SLED the average SPE idle time is around $7\mu s$, while the YNR experiments show $10\mu s$. These results indicate the upper bounds on the SPE idle time reduction: Work-Steal can reduce idle time by at most 70% when compared to YNR, while SLED will reduce it by at most 30%. The SLED experiments shown correspond to the implementation where the selection of the next process is implemented at the user level and processes are pinned to CPUs. For comparison, we include the overheads due to contention on the PPE side: PPE and MBOX. These represent an upper bound for the SPE idle time when using their associated scheme for signaling. PPE presents the overhead ($14\mu s$) of repeatedly calling `sched_yield()` when oversubscribing with 6 processes. The lower overhead of the YNR scheme is explained by lower PPE contention. MBOX presents the SPE idle time ($19\mu s$) when repeatedly offloading empty tasks and coordinating using register interrupt mailboxes. Synchronization using callbacks is an order of magnitude slower.

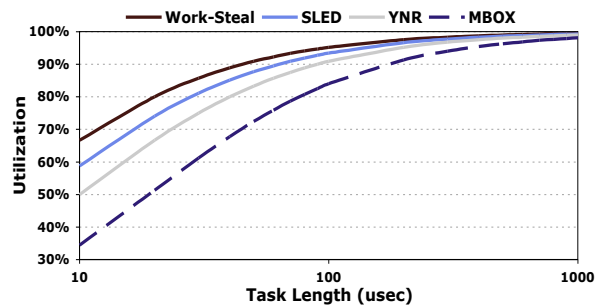


Figure 8: Expected SPE utilization with different task lengths. Idle time been chosen as the average SPE idle time across experiments. Work-Steal- $3\mu s$, SLED- $7\mu s$, YNR- $10\mu s$, MBOX- $19\mu s$.

Figure 8 presents the expected improvements in SPE utilization for different task lengths. These improvements are an upper bound on achievable application speedup for any given scheme. For Work-Steal, SLED and YNR we have used the average idle times determined in experiments, $3\mu s$, $7\mu s$ and $10\mu s$ respectively; for MBOX we use $19\mu s$. The expected speedup decreases with the task length and cooperative scheduling is most effective for short to medium tasks. For example, the expected speedup of Work-Steal compared to YNR is $\approx 18\%$ for $30\mu s$ tasks and $\approx 3\%$ for $300\mu s$ tasks. The ordering of lines in Figure 8 indicates which implementation strategy will perform better in practice. A comparison of the SLED and YNR idle times in Figure 7 shows that the latter exhibits lower latency for task lengths shorter than $\approx 15\mu s$. The additional implementation overhead of SLED causes this performance degradation for very short “synchronization” intervals. This behavior is consistent and YNR outperforms SLED in micro-benchmarks and applications for these very short tasks.

The performance of SLED is determined by several implementation choices and in the rest of this section we will discuss the influence of polling length (parameter N in Figure 6) when `SLED_wait_for_SPE` is implemented at user-level or kernel level respectively. Figures 9 and 10 present the performance trends observed when `SLED_wait_for_SPE` is implemented at kernel level. Figure 9 presents the evolution of performance with scan length normalized to the best execution time for a given task length; for example a point inside the (1-1.1) region indicates performance at most 10% worse than the observed minimum. The shorter the polling interval, the closer the performance gets to the observed best. Longer polling inside the kernel decreases the responsiveness to other processes. Figure 10 presents the evolution of SPE idle time per offload. This is a measure of performance degradation caused by PPE contention. The overall performance when polling is implemented at the user level is relatively insensitive to the duration of the polling interval. The average SPE idle time with user level polling is $7\mu s$, while kernel level polling shows $\approx 9\mu s$; this indicates that user level polling is preferable to the kernel level. The micro-benchmark results indicate that a short scan length of 50-100 passes over the `ready_to_run` list is likely to provide good performance regardless of the implementation choice.

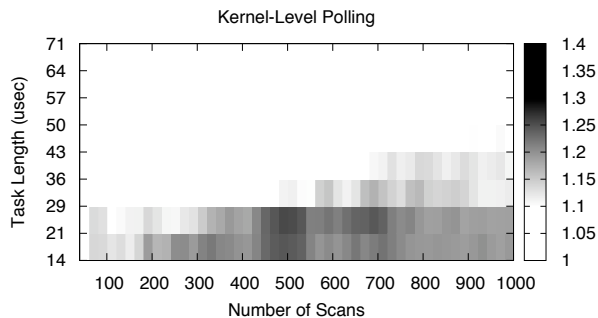


Figure 9: Variation of the relative speed of SLED with kernel level polling as a function of task length and polling duration. The graph shows the time per experiment normalized to the shortest time for a given task length.

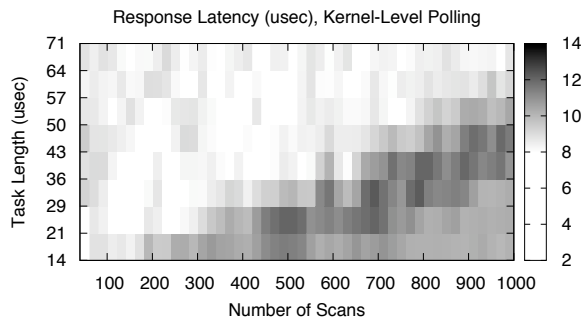


Figure 10: Variation of SPE idle time as a function of task length and polling duration. Polling is implemented at kernel level.

4.2 Application Results

The input sets for both applications contain a number of species with a given DNA sequence length. The number of species determines the memory footprint of the application, while the length of the input DNA sequence determines the duration of the off-loaded tasks. For PBPI we use a data set with 107 species, while for RAxML the set contains 40 species. For both applications we vary the length of the DNA sequence from ≈ 100 to $\approx 5,000$. Figure 13 shows the distribution of the duration of the offloaded tasks for sample “short”, “medium” and “long” DNA sequences. Most tasks in PBPI are shorter than $100\mu s$, while in RAxML long DNA sequences will generate much longer tasks.

Figure 11 presents the speedup for PBPI, achieved by the SLED and work-stealing implementations when compared to YNR. The x -axis shows the DNA sequence length. For PBPI, work-stealing consistently outperforms YNR and speedup for the whole workload is 12%, with a maximum speedup of 21%. YNR outperforms the SLED scheme for small task sizes. For task sizes larger than $15\mu s$ SLED achieves an average speedup of 5% for the whole workload, with a maximum of 10%. The performance improvements for RAxML are more modest due to the larger task length. The average improvements for the workload are $\approx 3\%$, with a maximum of 5%. Work-stealing for RAxML is consistently slower than the YNR implementation with an average slowdown of 23% for the workload considered. In RAxML, the same off-loaded region can be reached from various parts of the code, and also different branches in the code can be taken after off-loading completion. To perform work-stealing upon off-loading the processes need to exchange the information about branches which will be taken and complete control over program counters and stacks. Our work migration implementation is not a full Continuation Passing Style and we use only “partial” migration, i.e. at certain point in the code each process will be returned its original data (if the work migration was performed). We found that the partial work-stealing requires significant synchronization which results in significant performance degradation.

Figure 12 shows that cooperative scheduling (SLED) improves the SPE idle time by $\approx 20\%$ for both applications. All application performance results are consistent with the trends reported for micro-benchmarks. The results for both

applications indicate that a judicious implementation of cooperative scheduling is required for best performance. IDLE graph in Figure 12 represents the SPE idle time – time that SPEs are waiting for work as a percentage of total execution time. For brevity we report only the general trends observed for our design decisions for each implementation. Distributed data structures (`ready_to_run` list) and process pinning are required for good performance. For the YNR implementations pinning processes to hardware contexts did not affect performance, this might change in the presence of application level load imbalance. Implementations with a shared data structure require³ affinity awareness: 1) SLED (unpinned processes) with checking for affinity at the user level produces performance inferior to YNR; and 2) SLED with kernel level affinity checking produces performance inferior to SLED and pinning, but better than YNR. For SLED and pinning the performance is determined by the choice of implementing `SLED_wait_for_SPE` at the user or kernel level. Figure 14 presents the execution time of PBPI with SLED and polling at the kernel level. While for both PBPI and RAxML performance is relatively insensitive to the polling duration in all other cases, PBPI exhibits a many-fold performance degradation in this case. After each offload stage PBPI performs a `MPI_Allreduce` operation and we believe that spending a longer time inside the kernel for polling adversely affects the progress of the other processes. We therefore advocate for implementing synchronization functions such as `SLED_wait_for_SPE` as user-level library calls.

The benefits of cooperative scheduling are observable when the PPE hierarchy is oversubscribed and this might not be the best performing parallelization scheme for a given application. The parallelization trade-offs for each application are analyzed in [6]. Note that extracting loop level parallelism required several months of development effort for each application while using task-level parallelism derived directly from the MPI code required less coding effort to achieve similar performance. For RAxML, oversubscription with 6 MPI processes produces best performance results when compared with more aggressive implementations that use loop level parallelism. These differences range from many-fold to tens of percent and our approach will only improve the performance. Extending PBPI with loop level parallelism yields

³In order to avoid process migration.

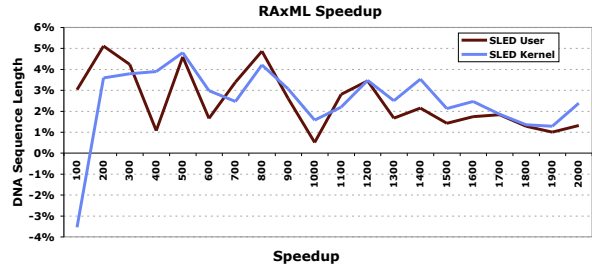
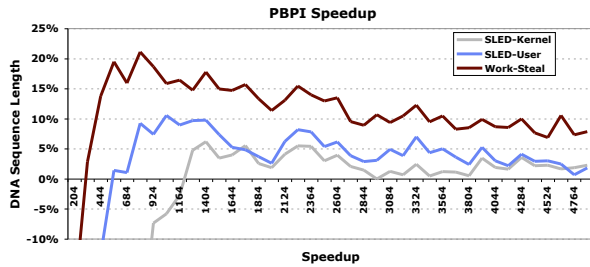


Figure 11: PBPI and RAXML speedup of SLED and work-stealing compared to YNR. Work-stealing for RAXML produces a 23% slowdown for the workload.

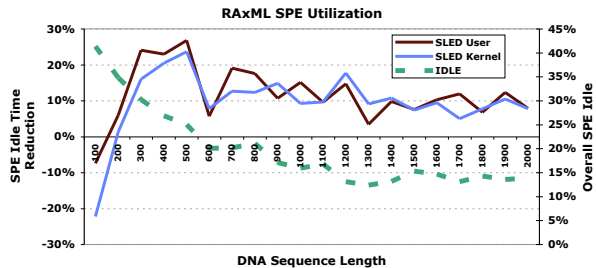
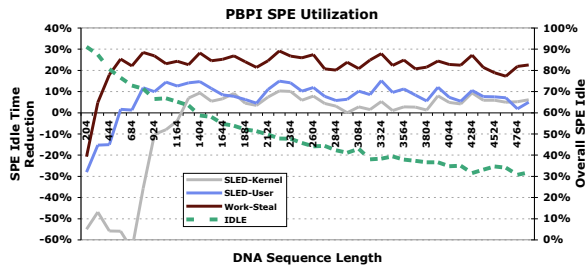


Figure 12: Improvements in SPE utilization in RAXML. IDLE (*right-hand y-axis*): time when SPEs are waiting for work as a percentage of total execution time. SLED-user (*left-hand y-axis*) represents the reductions of the idle time when compared to the YNR idle time.

an implementation where the best performance is obtained with different PPE processes/SPE ratios on a per dataset basis. The resulting implementation achieves a performance improvement that varies from many-fold for very short tasks to at most 30% better for medium and large tasks, when compared to the implementation with task-level parallelism only. For short to medium task sizes our scheduling strategy recuperates a significant fraction of the performance differences and alleviates the need for a lengthy development process to coarsen the loop level parallelism.

5. DISCUSSION

The results indicate that a cooperative scheduling approach is able to improve the performance of Cell applications and based on our experience, adding this behavior to existing codes is not a very time consuming task. The results obtained on the Sony PlayStation3 are very positive and we expect our techniques to have an even higher impact on the Cell blades (QS20, QS21 and QS22) which contain 2 PPEs (4 hardware execution contexts) and 16 SPEs and are likely to exhibit higher contention.

The duration of the busy waiting involved in the selection of the next process to run determines the performance of the cooperative scheduling approach. The results indicate that short polling periods are beneficial to performance, polling should be performed outside critical sections and distributed data structures are required for performance. Polling inside critical sections can result in many-fold performance reduction (Figure 14) even for an architecture like Cell where the degree of active contention for the critical section is small⁴. The optimal duration of busy waiting is system dependent and our experiments indicate that different granularities might be required on the Cell blades. For our Sony PlayStation3 experiments we use 100 list iterations.

⁴There are only two active hardware contexts.

There are several points of concern related to what type of applications can benefit from an approach like ours. The work-stealing approach lowers the overhead per offload by avoiding system calls and it is able to respond better to PPE load unbalance at the expense of increased development time and complexity. Work-stealing without compiler support for a continuation passing style is also best suited for SPMD applications where all processes repeatedly offload the same computation on all SPEs. The SLED approach can easily accommodate more irregular applications. Load balance and fairness are of concern: as future work we will consider providing user level APIs to mark cooperative scheduling regions and extending the implementation of `_yield_to` to penalize non-cooperative processes.

The length of offloaded tasks is a parameter of concern for the performance of any cooperative scheduling scheme. This length determines the frequency of the synchronization events and increasing this length will yield diminishing returns for cooperative scheduling. In our experiments and applications we have observed task lengths up to several hundred of microseconds. We believe this granularity will appear very often in applications. Private communications with the authors of [25] revealed medium (less than $300\mu s$) granularities for their computational kernels implementations. Analyzing the machine balance of the Cell BE architecture provides more support for our assumption. The Cell BE embedded in a PS3 console has a peak rate of 21.03 double precision Gflops (230 single precision Gflops) and each SPE has access to a local store of 256KB. The limited size of the local store combined with the fast processing rate allows the SPEs to perform a very large number of operations per data point before exhausting the local store and maybe requiring PPE/SPE synchronization. A single precision computation can perform one operation per every data point in the local store in under $3\mu s$. Algorithms with complexity less than $O(N\sqrt{N})$ are likely to process the whole

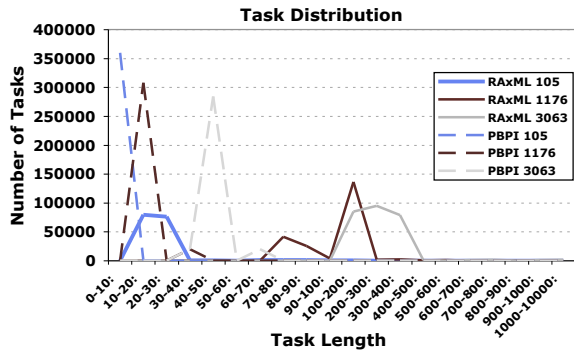


Figure 13: Application sample task distribution for “short”, “medium” and “long” DNA sequences.

local store within relatively short time frames. Furthermore, as IBM continues to release enhanced Cell processors (the latest was announced in June 2008 for the Cell blade QS22), the length of the SPE tasks will only decrease, which will in return increase the SPE idle intervals.

The Linux kernel mailing lists contain multiple discussion threads related to the implementation and the desirability of a `yield_to()` function system call. The position of the main kernel developers seems to be that such a function is against the Linux “spirit”. Their position is defended for homogeneous multi-processor systems such as workstations or servers that are required to accommodate very diverse workloads. We believe adding this functionality to kernels destined for accelerator based heterogeneous systems is of practical value.

6. RELATED WORK

The Cell processor has drawn significant attention in industry and academia. A large number of evaluation studies have been conducted to determine suitability of the Cell BE for scientific computation. As an example we list several contributions: Bader et al [3] examine the implementation of list ranking algorithms on Cell; Petrini et al [22] reported experiences from porting and optimizing Sweep3D on Cell; the same author presented a study of graph explorations algorithms on Cell [23].

Several high-performance programming models and compilers have recently emerged as a part of an effort to alleviate application design difficulty for the Cell BE. CellSs [4] is a compiler and a runtime system which reduces programming effort for the Cell processor by enabling usage of a single code module, i.e. an optimized SPE executable is automatically generated from the main code module. The CellSs runtime system detects data dependencies across SPE tasks and uses the obtained information to efficiently schedule tasks. Eichenberger et al [14] present compiler techniques targeting automatic generation of highly optimized code for Cell. To spread the parallel code across multiple SPEs they use OpenMP thread-level parallelization, which is based on the off-loading model. It is likely that both approaches will experience high volume of data exchange and synchronization between the PPE and SPE sides of the processor. Zhao and Kennedy [26] present a dependence-driven compilation framework for simultaneous automatic loop-level parallelization and SIMDization on Cell. Sequoia [15] is a programming language which enables automatic communication across vertical memory hierarchies. Although it does not support a single code module programming on Cell,

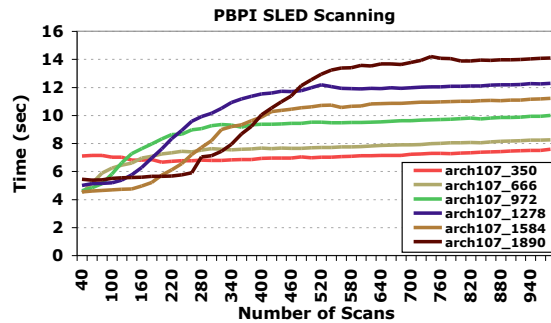


Figure 14: PBPI performance with SLED and kernel level polling.

Sequoia performs automatic memory management on the SPE side which significantly reduces programming effort. CorePy [21] is a runtime system which allows programmers to execute the SPE code from Python applications. All of the aforementioned programming models are based on the “off-loading” approach, where the application is split among the PPE and SPEs, and can benefit from efficient PPE \leftrightarrow SPE synchronization. The Charm++ framework provides a Cell offloading API which requires efficient PPE-SPE signaling. All these efforts can benefit from efficient PPE \leftrightarrow SPE synchronization using our cooperative scheduling approach. We consider our work to be of special interest to any compiler or programming framework on Cell which can potentially suffer from high PPE \leftrightarrow SPE synchronization overhead. Using cooperative scheduling strategies we target to reduce the PPE response time when a request is initiated from the SPE side. Consequently, we reduce the idle time on the SPE side while it is waiting for the PPE reply.

In addition, several studies consider Cell execution models which do not involve heavy PPE \leftrightarrow SPE communication. Kudlur et al [18] developed a streaming code generation template. Their method is capable of automatically mapping a stream program to the Cell processor. Another example is the MapReduce framework [12] which provides support for MapReduce [13] execution model on Cell.

7. CONCLUSION

In this paper we explore the design and performance trade-offs of cooperative scheduling implementations for the accelerator based systems. The experimental architecture we use is the Cell Broadband Engine. Programs written for the accelerator based systems often contain two disjoint parallel hierarchies executing on hardware with different degrees of native parallelism. Achieving good performance requires in many cases oversubscribing the general purpose core. For these cases, mechanisms able to increase the attentiveness of the processes running on the general core to asynchronous requests incoming from the accelerators will improve performance. We evaluate two such mechanisms: an integration of cooperative scheduling in the Linux kernel and a work-stealing user level approach. Both approaches improve application performance in the presence of oversubscription. The kernel level approach provides more generality and ease of implementation at the expense of performance when compared to the work stealing approach. The amount of effort required to incorporate our techniques into existing applications is not prohibitive and we believe their adoption can increase programmer productivity by alleviating the need for

complicated program transformations to extract loop level parallelism. Cooperative scheduling can be easily integrated into MPI and OpenMP runtimes for a performance boost on accelerator based system architectures. Furthermore, it is likely that cooperative scheduling will significantly improve the communication performance among accelerators in a cluster environment since this type of communication is commonly performed through the general core.

8. REFERENCES

- [1] Charm++ on the Cell Processor. Available at <http://charm.cs.uiuc.edu/research/cell/>.
- [2] IBM Accelerated Library Framework for Cell Programmer's Guide and API Reference.
- [3] D. A. Bader, V. Agarwal, and K. Madduri. On the design and analysis of irregular algorithms on the cell processor: A case study of list ranking. In *Proc. of the 21st International Parallel and Distributed Processing Symposium*, pages 1–10, 2007.
- [4] P. Bellens, J. M. Pérez, R. M. Badia, and J. Labarta. Memory - cellss: a programming model for the cell be architecture. In *Proc. of Supercomputing'2006*, page 86, 2006.
- [5] F. Blagojevic, M. Curtis-Maury, J.-S. Yeom, S. Schneider, and D. S. Nikolopoulos. Scheduling Asymmetric Parallelism on a PlayStation3 Cluster. In *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 146–153, Washington, DC, USA, 2008. IEEE Computer Society.
- [6] F. Blagojevic, X. Feng, K. Cameron, and D. S. Nikolopoulos. Modeling Multi-grain Parallelism on Heterogeneous Multi-core Processors: A Case Study with the Cell BE. In *Proc. of the 2008 HiPEAC Conference*, Jan. 2008.
- [7] F. Blagojevic, D. Nikolopoulos, A. Stamatakis, and C. Antonopoulos. Dynamic Multi-grain Parallelization on the Cell Broadband Engine. In *Proc. of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 90–100, Mar. 2007.
- [8] F. Blagojevic, A. Stamatakis, C. Antonopoulos, and D. Nikolopoulos. Raxml-cell: Parallel phylogenetic tree inference on the cell broadband engine. In *Proc. of the 21st International Parallel and Distributed Processing Symposium*, Long Beach, CA, Mar. 2007.
- [9] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.
- [10] M. Charalambous, P. Trancoso, and A. Stamatakis. Initial Experiences Porting a Bioinformatics Application to a Graphics Processor. In *Panhellenic Conference on Informatics*, pages 415–425, 2005.
- [11] W. Chen, D. Bonachea, J. Duell, P. Husbands, C. Iancu, and K. Yelick. A Performance Analysis of the Berkeley UPC Compiler. In *Proceedings of the 17th International Conference on Supercomputing (ICS)*, June 2003.
- [12] M. de Krujif and K. Sankaralingam. MapReduce for the Cell B.E. Architecture.
- [13] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. pages 137–150.
- [14] A. Eichenberger, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. Gschwind, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. Oden, D. Prener, J. Shepherd, and B. So. Optimizing Compiler for the CELL Processor. In *Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, Saint Louis, MO, Sept. 2005.
- [15] 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. Memory - sequoia: programming the memory hierarchy. In *Proc. of Supercomputing'2006*, page 83, 2006.
- [16] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast Computation of Database Operations Using Graphics Processors. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 215–226, New York, NY, USA, 2004. ACM.
- [17] M. Kistler, M. Perrone, and F. Petrini. Cell Multiprocessor Interconnection Network: Built for Speed. *IEEE Micro*, 26(3), May-June 2006. Available from <http://hpc.pnl.gov/people/fabrizio/papers/ieemicro-cell.pdf>.
- [18] M. Kudlur and S. Mahlke. Orchestrating the Execution of Stream Programs on Multicore Platforms. *SIGPLAN Not.*, 43(6):114–124, 2008.
- [19] E. S. Larsen and D. McAllister. Fast Matrix Multiplies Using Graphics Hardware. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 55–55, New York, NY, USA, 2001. ACM.
- [20] M. Monteyne. RapidMind Multi-core Development Platform. Available from <http://www.rapidmind.net/case-rapidmind.php>.
- [21] C. Mueller, B. Martin, and A. Lumsdaine. CorePy: High-Productivity Cell/B.E. Programming. In *Proc. of the First STI/Georgia Tech Workshop on Software and Applications for the Cell/B.E. Processor*, June 2007.
- [22] F. Petrini, G. Fossom, M. Kistler, and M. Perrone. Multicore Surprises: Lesson Learned from Optimizing Sweep3D on the Cell Broadband Engine.
- [23] F. Petrini, D. Scarpazza, O. Villa, and J. Fernandez. Challenges in Mapping Graph Exploration Algorithms on Advanced Multi-core Processors. In *Proc. of the 21st International Parallel and Distributed Processing Symposium*, Long Beach, CA, Mar. 2007.
- [24] D. P. Scarpazza, O. Villa, and F. Petrini. Peak-Performance DFA-based String Matching on the Cell Processor. In *IPDPS*, pages 1–8. IEEE, 2007.
- [25] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The Potential of the Cell Processor for Scientific Computing. *ACM International Conference on Computing Frontiers*, May 3-6 2006.
- [26] Y. Zhao and K. Kennedy. Dependence-based Code Generation for a Cell Processor. In *Proc. of the 19th International Workshop on Languages and Compilers for Parallel Computing*, New Orleans, LA, Nov. 2006.