QUEEN'S UNIVERSITY BELFAST

DMA-based Prefetching for I/O-Intensive Workloads on the Cell Architecture

Mustafa Rafique, M., Butt, A. R., & Nikolopoulos, D. (2008). DMA-based Prefetching for I/O-Intensive Workloads on the Cell Architecture. In *Proceedings of the Fifth ACM International Conference on Computing Frontiers (CF)* (pp. 23-32). Association for Computing Machinery. https://doi.org/10.1145/1366230.1366236

Published in:
Proceedings of the Fifth ACM International Conference on Computing Frontiers (CF)

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.

Open Access
This research has been made openly available by Queen's academics and its Open Research team. We would love to hear how access to this research benefits you. – Share your feedback with us: http://go.qub.ac.uk/oa-feedback

Download date:24. Mar. 2025

DMA-based Prefetching for I/O-Intensive Workloads on the Cell Architecture

M. Mustafa Rafique, Ali R. Butt, and Dimitrios S. Nikolopoulos
Dept. of Computer Science, Virginia Tech
Blacksburg, Virginia, USA
{mustafa, butta, dsn}@cs.vt.edu

ABSTRACT

Recent advent of the asymmetric multi-core processors such as Cell Broadband Engine (Cell/BE) has popularized the use of heterogeneous architectures. A growing body of research is exploring the use of such architectures, especially in High-End Computing, for supporting scientific applications. However, prior research has focused on use of the available Cell/BE operating systems and runtime environments for supporting compute-intensive jobs. Data and I/O intensive workloads have largely been ignored in this domain. In this paper, we take the first steps in supporting I/O intensive workloads on the Cell/BE and deriving guidelines for optimizing the execution of I/O workloads on heterogeneous architectures. We explore various performance enhancing techniques for such workloads on an actual Cell/BE system. Among the techniques we explore, an asynchronous prefetching-based approach, which uses the PowerPC core of the Cell/BE for file prefetching and decentralized DMAs from the synergistic processing cores (SPE's), improves the performance for I/O workloads that include an encryption/decryption component by 22.2%, compared to I/O performed naïvely from the SPE's. Our evaluation shows promising results and lays the foundation for developing more efficient I/O support libraries for multi-core asymmetric architectures.

Categories and Subject Descriptors

C.1.2 [**Processor Architecture**]: Multiple Data Stream Architecture; D.4.4 [**Operating Systems**]: Input/output

General Terms

Design, Experimentation, Performance

Keywords

Cell Broadband Engine, High-Performance Computing, I/O Intensive Workloads

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CF'08, May 5–7, 2008, Ischia, Italy.
Copyright 2008 ACM 978-1-60558-077-7/08/05 ...$5.00.

1. INTRODUCTION

Asymmetric multi-core processors are widely regarded as a viable path to sustaining high performance, without compromising reliability. Given a fixed transistor budget, asymmetric multi-core processors invest heavily on many simple, tightly coupled, accelerator-type cores. These cores are typically designed with custom Instruction Set Architectures (ISAs) and features that enable acceleration of computational kernels operating on vector data. Researchers have collected mounting evidence on the superiority of asymmetric multi-core processors in terms of performance, scalability, and power-efficiency [5, 24, 40, 42, 43]. Application-specific asymmetric multi-core architectures have previously been used extensively in network processors [51]. The recent advent of the Cell Broadband Engine (Cell/BE) processor [45] as a high-performance computing and data processing engine [3, 7, 8, 19, 23, 41, 48], further attests to the potential of emerging asymmetric multi-core architectures.

This paper explores the use of the Cell/BE, arguably a dominant asymmetric multi-core processor, in I/O-intensive applications. With modern high-performance computing applications generating and processing exponentially increasing amounts of data, the scalable parallel processing capabilities, large on-chip data transfer bandwidth, and aggressive latency overlap mechanisms of the Cell/BE render it an attractive platform for high-performance I/O. Although several recent efforts have demonstrated the potential of the Cell/BE for high-speed computation on data staged through its accelerator cores (SPE's) [19, 23], there is little understanding of how I/O operations interact with the architecture of the Cell/BE. The implications of such Cell/BE characteristics as asymmetry, DMA latency overlap, and software management of disjoint address spaces, on the design and implementation of the I/O software stack have not been explored. This paper addresses these important questions and makes the following contributions:

- A study of the I/O path in the currently available Cell/BE operating system and in the accelerator support library;
- An exploration of various alternative I/O methods that can be applied to the Cell/BE architecture;
- An investigation of the impact of data prefetching techniques on improving I/O performance for the Cell/BE architecture; and
- An evaluation and recommendation of appropriate methods for handling I/O intensive workloads.

Our evaluation reveals that allowing individual accelerators to perform direct I/O faces the bottleneck of all I/O requests routed through the main core. Thus, we argue that (i) if the current OS and library support is not to be extended, the most efficient technique of performing I/O is to allow the main core to pre-stage (prefetch) the data for the accelerator cores, and (ii) the performance can be improved if the accelerator support library is extended to do direct I/O, hence removing the said bottleneck. We explore several I/O optimization schemes on the Cell/BE, involving prefetching and staging of data between cores. An asynchronous prefetching scheme which combines prefetching from the PowerPC core (PPE) with asynchronous DMAs from the synergistic processing cores (SPE's), improves the performance of I/O workloads by up to 22.2%, compared to naïve I/O from the SPE's. We also re-affirm the intuition that the Cell SPE's have significant acceleration capabilities, which can be leveraged in compute-intensive components of I/O software stacks, such as encryption/decryption and compression.

The rest of this paper is organized as follows. Section 2 provides background and motivation for the research presented in the paper. Section 3 describes the Cell/BE architecture in detail. Section 4 describes our experimental setting and workloads, followed by a presentation and evaluation of several schemes to improve I/O performance on the Cell/BE. Section 5 discusses related work. Section 6 concludes the paper.

2. MOTIVATION AND BACKGROUND

In this section, we describe the background of this work, and outline the enabling technologies for this research.

We are concerned with the implementation of efficient I/O schemes for data-intensive applications on asymmetric multi-core processors. We assume processors with heterogeneous cores, heterogeneous ISAs, and disjoint address spaces between cores of different technology. This organization provides for a simplified hardware design which supports high raw computational speed and data transfer bandwidth, at the cost of increased programming complexity. Applications leverage the processor by offloading their time-consuming computational kernels to accelerator-type cores and by using the on-chip interconnection network to efficiently stage ("stream") data to the local storage space of the accelerators. Clearly, acceleration capabilities are relevant to I/O-intensive applications with significant I/O processing components such as encryption and compression.

Besides acceleration of vector data processing, the design of the I/O subsystem on asymmetric multi-core processor merits further investigation. A design consideration of particular importance is the distribution of the I/O processing path between the cores of the processor. Current designs run the operating system on the conventional "host" cores (e.g. the PowerPC PPE of the Cell/BE) of the processors and route all I/O requests made from the "accelerator" cores (e.g. the SPE's of the Cell/BE) through the host cores. While this design simplifies the system software architecture it imposes bottlenecks. In particular, parallel I/O from the accelerator-type cores, which are typically many more than the host cores, may suffer from serialization and queuing at the host cores.

In contrast to conventional processors, asymmetric multi-core processors delegate more control of the memory hierarchy to software. The Cell/BE maps the local store of each accelerator core to the virtual address space, and enables direct transfers to and from the local stores of any core, through a DMA mechanism. The DMA mechanism further enables overlap of multiple DMA requests with computation on each core. This capability extends naturally to the I/O subsystem, which should properly stage data from the disk, to off-chip memory, to on-chip memory, so that the non-overlapped data transfer latency is minimized. The design space for data staging in the I/O system involves tuning of the unit of data transfer between layers of the memory hierarchy, synchronous and asynchronous prefetching algorithms to stage data timely in the local store of accelerator cores for further processing, and synchronization and communication mechanisms between host cores and accelerator cores to coordinate I/O requests.

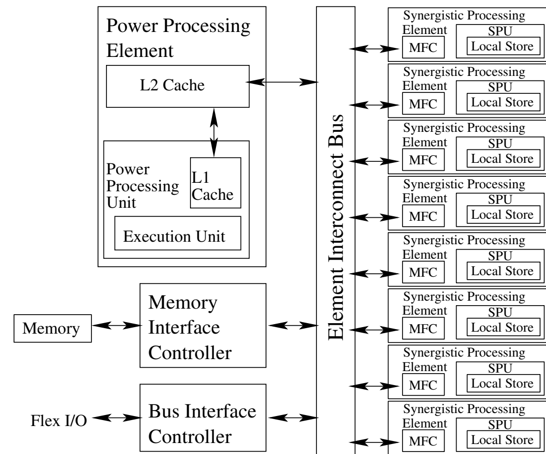

Figure 1: Cell Broadband Engine system architecture.

3. CELL ARCHITECTURE

We now present details of the Cell/BE architecture. Given the focus of this work, we also describe the path that is taken by application I/O requests before being delivered to the disk. Figure 1 shows the high level system components making up the Cell/BE, namely, the PowerPC Processor Element, a number of Synergistic Processing Elements, the Memory Interface Controller, two non-coherent I/O Interfaces, and the Element Interconnect Bus.

3.1 PowerPC Processor Element

The execution "controller" of the Cell/BE is a general purpose 64-bit PowerPC Processor Element (PPE), currently operating at 3.2 GHz [25, 29]. The PPE is a dual issue, two-way multithreaded core. The PPE boasts a 32 KB instruction and 32 KB data Level-1 cache, and a 512 KB Level-2 cache. The PPE can theoretically execute two double precision or eight single precision operations per clock cycle. In current installations, the PPE runs Linux, with Cell/BE-specific extensions that provide access to the accelerator-type cores of the processor to user-space libraries. The PPE itself implements superscalar architecture, out-of-order and SIMD (AltiVec) instruction execution, as well as non-blocking caches [45].

3.2 Synergistic Processing Elements

The intended use of the PPE on the Cell/BE is mainly execution control, running the operating system and supporting legacy applications. The major portion of computational workload is handled by multiple Synergistic Processing Elements (SPE's) [22]. SPE's are optimized vector engines and operate at the global processor frequency of 3.2 GHz. Each SPE consists of a Synergistic Processing Unit (SPU), and a Memory Flow Controller (MFC). Each SPE also has an embedded software-managed SRAM referred to as "Local Store". The local store is similar to scratch-pad memory. The SPE has exclusive access to its local store, and the local store holds both the executable running on the SPE and the data needed by the executable. SPE's can access external DRAM and memory-mapped remote local stores exclusively through DMA operations. The PPE can also access the SPE local stores through DMAs. PPE accesses to an SPE local store are processed with higher priority than local load and stores issued by the SPE. The size of the local store on current processor models is 256 KB.

SPE's execute code read directly from their local stores and may issue a very limited number of system calls, including I/O calls. These calls utilize a stop-and-signal instruction and are routed to the PPE for kernel-level processing. The PPE and SPE's have different instruction sets, therefore applications running on the Cell/BE are divided into two executables. The main executable runs on the PPE and uses a POSIX-like interface for creating and triggering threads on the SPE's. The SPE threads can utilize high-level vector processing library operations, expressed using directives, to leverage the SIMD execution units, as well as a get/put interface to execute DMAs and access main memory and/or the local stores of other SPE's. SPE thread management, vector intrinsics and high-level primitives for DMA transfers are provided by a user-level runtime library (`libspe2`). The PPE and SPE may execute threads in parallel and synchronize through either DMAs, or a "mailbox" mechanism. SPE's are expected to run through completion, as operating system support for preemptive time-slicing of SPE's is currently at an experimental stage.

3.3 Memory Interface Controller

The Memory Interface Controller (MIC) is responsible for providing the PPE and SPE's access to the main system memory. MIC supports a dual channel Rambus XIO macro that interfaces to external XDR Rambus DRAM. The XIO operates at the maximum frequency of 3.2 GHz. Each XIO channel can have eight memory banks with a total memory size of 256 MB, making the total memory size of a single-processor system limited to 512 MB. Observed peak raw memory bandwidth is stated to be 25.6 GB/s at 3.2 GHz with both XIO channels [29], however such estimates for the peak bandwidth assume that all the banks are fully engaged by incoming request streams, and all the requests are made up of only reads or writes of 128 bytes. In the more typical case of blended reads and writes, the estimated effective bandwidth is 21 GB/s [45].

3.4 I/O Controller

The I/O controller is an off-chip component that provides interface to external network, disk, and other I/O devices. The Cell/BE I/O controller, called FlexIO, is also based on Rambus. The FlexIO has twelve one-byte wide links, five of which are point-to-point inbound paths to the Cell/BE, and the remaining seven are outbound transmit links [14]. The links are configured in two logical interfaces, referred to as Input/Output Interfaces (IOIF). Each link operates at 5 GHz and the IOIF provides raw bandwidth of 35 GB/s outbound and 25 GB/s inbound. However, the actual data and commands are transmitted as packets, which incur an overhead due to presence of metadata such as command identifier, data tags, and data size [45]. As a result, the effective bandwidth that can be attained is reduced to between 50% and 80% of the raw bandwidth. The operating system running on the PPE supports transparent application access to the I/O controller. The I/O requests from SPE's are handled by the PPE operating system. Currently, the applications do not have direct access to the controller.

3.5 Element Interconnect Bus

All the components making up the Cell/BE, i.e., the PPE, the SPE's, the off-chip I/O interfaces, and the MIC, communicate through a shared Element Interconnect Bus (EIB) [28] and using DMA transfers, supported by the MFCs. The EIB operates at half the system clock rate. The EIB is designed as a circular ring comprised of four 16-bytes wide unidirectional data channels. Two of the data channels run in the clockwise direction, and the other two run in the anti-clockwise direction. Each channel is capable of conveying up to three concurrent transactions.

Both the PPE and SPE's use EIB to transfer data to and from the main memory. The PPE accesses main memory with normal load and store instructions through the EIB. The PPE can also issue DMA put and get commands to and from the local storage of SPE's, which can be mapped to the virtual address space. An SPE accesses both main memory and the local storage of other SPE's exclusively with DMA commands. The MFC of each SPE runs at the same frequency as the EIB, supports naturally aligned DMA transfers of 1, 2, 4, 8, or a multiple of 16 bytes, and includes a DMA list that can be used to execute up to 2048 DMA transfers with a single DMA command. The maximum supported size of a single DMA request from SPE's is 16 KB.

Before sending data on to the EIB, each requesting unit sends out a small number of initial command requests. Each request on the EIB uses one command credit. The number of credits reflects the size of the command buffer of the EIB for that particular request. The EIB returns the credit back to the requesting unit when a slot becomes available in the command buffer due to a previous request moving ahead in the request pipeline.

Different elements utilize the EIB by issuing a request which is queued by a bus arbiter process. In case of contention for the bus by multiple elements, the arbiter strives for an optimal allocation of data channels to the requesters. In order to avoid stalling any read requests, highest priority is given to the memory controller, while all other components are treated equally with their requests served in a round-robin fashion. Furthermore, the data ring is not granted to a requester if the requested transfer would interfere with any other data transfer, or if it would have to travel more than halfway around the ring to reach its destination.

Each unit can simultaneously send and receive 16 bytes of data on every bus cycle on the EIB. The maximum data bandwidth of the entire EIB is limited by the maximum rate at which addresses are snooped across all units in the sys-

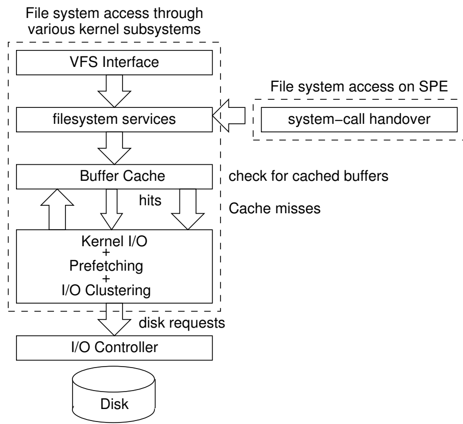

Figure 2: I/O requests path on the PPE and SPE.

tem, which is one per bus cycle. Since each address request can transfer up to 128 bytes, the theoretical peak bandwidth on the EIB at 3.2 GHz is calculated to be 128 Bx(3.2/2) GHz = 204.8 GB/s. However, the location of source and destination relative to each other, interference between existing and new transfers, number of the Cell/BE chips in the system, and whether the data transfer is to local stores or to main memory, are some of the factors [45] that reduce sustained data bandwidth from its theoretical peak. Moreover, if all the data requests are in the same direction, half of the rings will be idle, thus reducing the data bandwidth on EIB to at least half of its peak value.

3.6 PPE/SPE I/O Path

Figure 2 shows the path that application I/O requests from the PPE and SPE's follow before being serviced by the disk. The PPE supports a full operating system, and the I/O path on the PPE is a standard one. All I/O requests through the VFS layer are first sent to the buffer cache. In case of miss in the buffer cache, a request to read the data from the disk is issued. The kernel clustering mechanisms combine multiple requests for contiguous blocks, and the kernel prefetching algorithm detects and prefetches blocks to reduce execution stalls due to synchronous disk requests. Interested readers are referred to [9] for a detailed explanation of the standard Linux I/O path.

Since the SPE does not support a native operating system, there is no kernel context on SPE and all system calls issued by SPE's are handled as *external assisted library calls*. We now discuss how external calls are supported. Note that the same process is used to service all system calls from SPE's on the PPE, not just the I/O related calls.

The SPE uses special stop-and-signal instructions [26] to hand over control to the PPE for handling external service requests. In order to perform an external assisted library call, the SPE allocates local store memory to hold the input and output parameters of the call and copies all the input parameters (from stack or registers) into this memory. It then combines a special function opcode corresponding to the requested service with the address of this input/output memory image to form a 32-bit message. The SPE then places this message into the local store memory, immediately followed by a `stop` instruction. It then signals the PPE to execute the library function on behalf of SPE. In response to the signal, the PPE reads the assisted call message from SPE's local store, and uses the stop and signal type and opcode to dispatch the control (PPE context) to the specified assisted call handler. The handler on the PPE retrieves the input parameters for the assisted call from the local-store memory pointed to by the assisted call message, and executes the appropriate system call on the PPE. On completion of the system call, the return values are placed into the same local store memory, and the SPE is signaled to resume execution. Upon resumption, the library on the SPE reads the input value from the memory image and places them into the return registers, hence, completing the call. Thus all I/O calls on the SPE's are routed through the PPE operating system.

	PPE Time	SPE Time
SPE Context Creation	-	1.46
Program Loading on SPE	-	0.16
Thread Creation on SPE	-	0.104
Buffer Allocation	0.012	0.015
File Reading	48688	48414
Buffer Deallocation	0.012	0.016
Total SPE execution time	-	48806
Total time	49664	49241

Table 1: Average time (in msec.) required by major tasks while reading a 2 GB file from the disk on the PPE and a SPE.

4. EVALUATION OF I/O IMPROVING TECHNIQUES

In this section, we present and evaluate a number of I/O improving techniques for the Cell/BE architecture. For our evaluation, we use different I/O workloads executed on a Sony Play Station 3 (PS3). The PS3 is a hypervisor-controlled platform. It has 6 active SPE's with 256 KB local storage, 256 MB of main memory of which about 200 MB is directly accessible to the operating system (OS), and a 60 GB hard disk. Although the Cell/BE has 8 SPE's, on the PS3, one SPE is reserved for running the hypervisor and another SPE is deactivated. Accesses to storage devices, including the disk, are routed through the hypervisor with dedicated hypercalls and their completion is communicated to the OS through virtual interrupts. Due to the proprietary nature of the PS3 hypervisor, it is not possible to assess its imposed overhead on accesses to storage devices for the purpose of this work.

In the following, we first evaluate the characteristics of our experimental platform by running simple workloads, then we explore how the SPE's can be used to handle I/O intensive tasks such as data encryption. Finally, to account for experimental errors the presented numbers represent averages over three different runs unless otherwise stated.

4.1 Identity Tests

In the first set of experiments, we created a workload that reads a large file of size 2 GB. We refer to this experiment as the Identity Test. The goal of this Test is to determine the maximum I/O bandwidth available on our experimental platform for the PPE and SPE's.

Table 1 shows the timing break down for the Identity Test both on the PPE and a SPE using a block size of 16 KB. Note that context creation, loading, and thread creation are only

block size	4 KB		16 KB	
	PPE	SPE	PPE	SPE
Time (msec.)	48674	53779	48688	49414
Throughput (MB/s)	41.09	37.19	41.08	40.48

Table 2: The average time and observed throughput for reading a 2 GB file from disk on the PPE and a SPE using different block sizes.

needed when running the Test on the SPE. It is observed that the time to read the file on the SPE is similar to that on the PPE. The table also shows that the cost of the context loading steps on the SPE is relatively insignificant. However, this cost can become crucial if SPE workloads are repeatedly loaded or if the execution time of the SPE program is small.

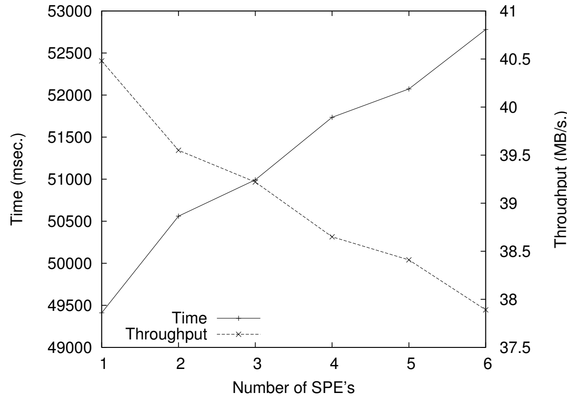

Figure 3: Average time and observed throughput for simultaneously reading a 2 GB file using 16 KB blocks from one to six SPE's.

Next, we modified the block size to 4 KB, and determined the overall time it would take to perform the Identity Test both on the PPE and the SPE. Table 2 shows the results, and comparison with the previous case. We observe that while changing block sizes does not have a significant effect on the PPE I/O throughput, the large block size gives better throughput on the SPE. The reason for this improved throughput is that data transfers between SPE and the memory is done via DMA, and DMA is optimized by using the maximum transfer size per DMA operation, which for the Cell/BE is 16 KB. For this reason, in our remaining experiments we set the buffer size to 16 KB.

Next, we repeated the Identity Test while increasing the number of SPE's from one to six. Figure 3 shows the result. For this experiment, the PPE invokes one thread on each of the available SPE's, however, the total size of data read is same as before, i.e., 2 GB. A different file is read at each SPE so that unique requests are issued and any caching at the I/O controller and/or memory does not come into play. As seen in the figure, the average observed throughput decreases as the number of SPE's reading the file increases. The average observed I/O throughput is reduced by 6.4% when all the six available SPE's are used, compared to the case of using a single SPE for the same amount of data. This is due to increased contention for the EIB, and indicates that simply offloading I/O intensive jobs to multiple SPE's is unlikely to yield the best use of resources.

Num. SPE's	Buffer size				
	1 KB	2 KB	4 KB	8 KB	16 KB
1	21750	13952	9427	7828	6394
6	95410	56953	37031	26168	21206

Table 3: Time measured (in msec.) at PPE for sending data to SPE through DMA under varying buffer sizes, and for using one and six SPE's.

4.2 Workload-based Tests

In this section, we present the results of running an I/O intensive workload on the Cell/BE architecture. We first describe our workload, followed by detailed investigation of techniques to support the workload on the PS3.

4.2.1 Workload Overview

The workload that we have chosen is a 256-bit encryption/decryption application. Our choice is dictated by the computation intensive component of the encryption and decryption along with the need to do large I/O transfers for reading the input and writing the output. The workload reads a file from the disk, encrypts or decrypts it, and then writes back the results. Given that the PS3 has only about 200 MB of main memory available to user programs, we chose to encrypt a 64 MB file. This allows us to keep the entire file in memory if so needed and isolate the effects of buffer caching etc. We also vectorized the computation phase of our workload to achieve high performance on SPE's, which improved the time taken in the computation phase by 42.1%.

4.2.2 Effect of DMA Request Size

Our evaluation requires that the computation be offloaded to specific SPE's. Therefore, we first evaluate the effect of DMA buffer sizes on such offloading. Table 3 shows the time of computation offloading as we varied the buffer size used for DMA communications between the PPE and SPE. Note that these buffers are different from the file I/O block size of the previous experiments (which is fixed at 16 KB). We focused on the decryption phase of our workload for this experiment. In this case, all I/O is performed at the PPE, which after reading a full buffer of data from disk, passes its address in the main memory to a SPE. The SPE uses the passed address to do a DMA transfer and brings the contents of the buffer to its local store. The SPE then processes the

	Time (msec.)	
Buffer size	4 KB	16 KB
No. of times SPE is loaded	16384	4096
SPE loading (excluding execution) time	1787	823
SPE execution (including loading) time	8014	4273
CPU time used by SPE	5200	1850
Disk read time	450	497
Disk write time	1191	1221
CPU time for disk read operations	400	570
CPU time for disk write operations	330	250
Execution time of program	10176	6565
CPU time used by PPE	6050	2890

Table 4: Breakdown of time spent (in msec.) in different portions of the code when data is exchanged between a SPE and the PPE through DMA buffer sizes of 4 KB and 16 KB.

	PPE	PPE	SPE	SPE
Time	3403	205	714	640

	SPE	SPE	PPE	PPE
Time	4174	329	217	217

Table 5: Time (in msec.) for reading workload file at PPE/SPE followed by access from SPE/PPE.

data in the local store, and upon completion of the computation issues another DMA to transfer the processed contents back to the main memory. Finally, the PPE can write the updated buffer in the main memory back to the disk. Note that the maximum size of a single channel DMA that can be sent on the EIB is 16 KB, thus the maximum DMA size of our experiments is limited to that. The whole experiment is repeated for two cases: using a single SPE, and using all six SPE's. These results show that increasing the buffer size improves the execution times of our workload.

Timing breakdown for the cases of 4 KB and 16 KB DMA buffers.

For the previously described experiment, we also performed a detailed timing analysis for 4 KB and 16 KB DMA's using a single SPE. Table 4 shows the results. This experiment was conducted to see the effect of different DMA sizes on the time spent on various parts of the program. For the same input file, when the DMA size is increased from 4 KB to 16 KB, the number of times the PPE has to invoke a thread on an SPE is reduced by a factor of 4, thus reducing SPE loading time. The number of times the SPE is loaded to perform the same task also affects the total execution time, since it cuts down the number of times initialization is required on the SPE. Table 4 shows that the total execution time for the same workload is less when SPE and the PPE communicate with each other through DMA operations and a block size of 16 KB, than using a block size of 4 KB for the same data set. Observe that the total execution time is significantly less when using 16 KB blocks compared to 4 KB blocks. This is due to the fact that the total time also includes the time required at SPE to fetch the data into its local store through DMA operations, and the number of DMA operations done by SPE for 16 KB blocks is 4 times less than that for 4 KB blocks for the same data set.

4.2.3 Impact of File Caching

As discussed in Section 3, the I/O system calls from the SPE are handed over to the PPE for handling. This implies that once a file (or portion of a file) is accessed by the PPE it may be in memory when subsequent access for the file are issued from a SPE or the PPE, and these accesses can be serviced fast. In this experiment, we aim at confirming this empirical observation. First, we flushed any file cache by reading a large file (2 GB). Then we read the 64 MB workload file on the PPE, followed by reading the same file at a SPE. Table 5 shows the result for reading a file cold first on the PPE, followed by reading at SPE. The same experiment is repeated for first reading the file at a SPE, followed by at the PPE. From the table, we conclude that the caching effect is noticeable, and can help in reducing I/O times both on the PPE and on the SPE's, by first reading a file on the PPE. We also notice that file reading on the SPE is slower due to the I/O being routed through the PPE.

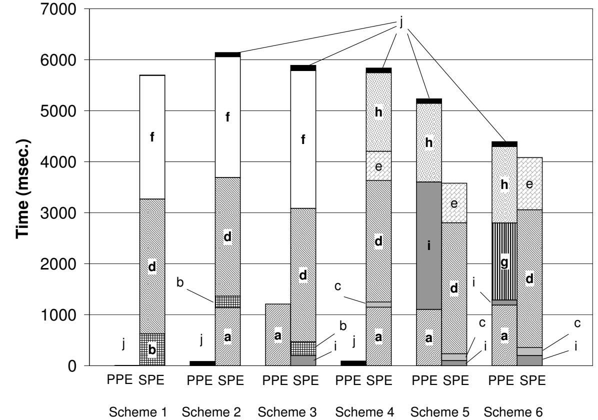

Figure 4: Timing Breakdown of different tasks for all the six schemes. The bars are labeled as follows: (a) File read at PPE, (b) File read at SPE, (c) DMA read at SPE, (d) encryption, (e) DMA write at SPE, (f) File write at SPE, (g) Write wait at PPE, (h) File write at PPE, (i) Waiting time, and (j) Miscellaneous.

Given the effectiveness of file caching, we now explore a number of schemes to improve I/O performance of our workload. For the following experiments, we utilize the encryption phase of our workload. Figure 4 shows the results. In some schemes tasks are executed in parallel at the PPE and SPE's. This is shown as two side-by-side bars for a scheme, with the total execution time dictated by the higher of the two bars. The breakdown for various steps is also shown.

Scheme 1: SPE performs all tasks.

Under this scheme, we perform all the tasks of our workload, i.e., reading the input file (b), processing it (d), and writing the output file (f), on the SPE. Note, however, that we still utilize the PPE to invoke the tasks as a single program on the SPE.

Scheme 2: Synchronous File Prefetching by the PPE.

In this scheme, we attempt to improve the overall performance of our workload by allowing the PPE to prefetch the input file in memory. This scheme is driven by the above observation that subsequent accesses by SPE's to a file read earlier by the PPE improves I/O times due to file caching. For this purpose, the PPE first pre-reads the entire file causing it to be brought in memory. Then the program from Scheme 1 is executed as before. Results in Figure 4 show that the *File read at SPE* (b) is much faster for this scheme, compared to Scheme 1. However, the time it takes to read the file on the PPE (a) is 81.6% longer compared to *File read at SPE* (b) in Scheme 1. We believe this is due to the PPE flooding the I/O controller queue, and lack of overlapping opportunities between computation and I/O in a sequential read compared to the read and process cycle of Scheme 1. Hence, Scheme 2 shows promise in terms of improving SPE read times, but suffers from slow I/O times on the PPE. The overall workload execution time is longer in Scheme 2 than Scheme 1.

	Scheme 1 Time	Scheme 6 Time
Read at 1 SPE	346	-
Read at PPE	-	1719
Process at 1 SPE	1082	858
Write at 1 SPE	1357	-
DMA read at 1 SPE	-	499
DMA write at 1 SPE	-	287
DMA wait at 1 SPE	-	25
Write wait at PPE	-	99
Write at PPE	-	1448
Total (6 SPE's)	3735	3410

Table 6: Timing breakdown (in msec.) of various tasks for scalability tests for Scheme 1 and Scheme 6 by using six SPE's.

Scheme 3: Asynchronous Prefetching by the PPE.

In the next scheme, we try to remove the file reading bottleneck of Scheme 2. For this purpose, we created a separate thread to prefetch the file into memory. Simultaneously, we offloaded the program of Scheme 1 to the SPE. The goal is to allow the prefetching by the PPE to overlap with computation on SPE, thus any data accessed by SPE will already be in memory and the overall performance of the workload will improve. Note that we do not have to worry about synchronizing the prefetching thread on the PPE with the I/O on SPE. In case the PPE thread is ahead of SPE, no problems would arise. However, if the SPE gets ahead of the PPE thread, the SPE's I/O request will automatically cause the data to be brought into memory, which in turn will make the PPE read the file faster, thus once again getting ahead of the SPE. The integrity of data read by SPE will not be compromised.

It is observed from the results in Figure 4 that although the I/O times (a) for individual steps increased, better I/O/computation overlapping resulted in an overall improvement of 4.7%, compared to Scheme 2. This shows that the PPE can facilitate I/O for SPE's and doing so results in improved performance.

Scheme 4: Synchronous DMA by the SPE.

So far, we have attempted to improve SPE performance by indirectly bringing the file in memory and implicitly improving the performance of the SPE workload. However, such schemes are prone to problems if the system flushes the file read by the PPE from the buffer cache before it can be read by SPE, hence negating any advantage of a PPE-assisted prefetch.

In this scheme, we explicitly prefetch the file on the PPE and give the SPE the address of memory where the file data is available. The SPE program is modified to not do direct I/O, rather use the addresses provided by the PPE. Hence, the PPE will read the input file in memory, give its address to the SPE to process, the SPE will create the output in memory, and finally the PPE will write the file back to the disk. The SPE will use DMA to map portions of the mapped file to its local store and send the results back. Figure 4 shows the results. Here, we observe that the *DMA read at SPE* (c) takes 55.0% and 62.0% less time than *File read at SPE* (b) in Scheme 2 and Scheme 3, respectively. However, the synchronous reading of file in this scheme takes long, causing the overall times to not improve as much: 4.9% and 0.2% compared to Scheme 2 and Scheme 3, respectively.

Scheme 5: Asynchronous DMA by the SPE.

To mitigate the effect of blocking read, we once again try the approach of Scheme 3 and utilize a separate thread to read the input file asynchronously. The DMA handover and processing at SPE are similar to that of Scheme 4. Figure 4 shows the results. A caveat here is that there is no automatic syncing of the prefetch thread and the SPE process, as was the case in Scheme 3. If the SPE process got ahead of the PPE prefetch thread, it will process junk data from memory where the input file has not been loaded. Hence, although this scheme is promising, it cannot guarantee the correctness of the operation.

Scheme 6: Asynchronous DMA by the SPE with Signaling.

The main shortcoming in Scheme 5 is the lack of a signaling mechanism between the prefetching thread producing the data (reading into memory) and the SPE consuming the data. One way to address this to use the *mailbox* abstraction supported by the Cell/BE. However, documentation [27] advises against using *mailboxes* given their slow performance. Therefore, we used DMA-based shared memory as a signaling mechanism to keep the prefetching thread synchronized with the SPE's. The PPE starts a thread to read the input file, and simultaneously also starts the SPE process. The difference from Scheme 5 is that the prefetching thread continuously updates a *status location* in main memory with the offset of the file read so far, and uses this location to determine how much of the data has been produced by SPE for writing back to the output file. Moreover, the SPE process, instead of blindly accessing memory assuming it contains valid input data, periodically uses DMA to access a pre-specified memory *status location*. In case the prefetching thread is lagging, the SPE process will busy-wait and recheck the *status location* until the required data is loaded into memory. Finally, the SPE can also use the shared location to specify the amount of processed output. This allows the PPE to simultaneously write back the output to the disk, and achieve an additional improvement over Scheme 5 where output was written back only after the entire input was processed. Thus, Scheme 6 achieves both reading of the input file and writing of output file in parallel with the processing of the data. Figure 4 shows the results, which are quite promising. Scheme 6 achieves 22.2%, 24.1%, and 24.0% improvement in overall performance compared to Scheme 1, Scheme 3, and Scheme 4, respectively.

Scalability Test.

In order to test the scalability of Scheme 6, we tested it by fully parallelizing it to 6 available SPE's on the Cell/BE, and compared the result with the scaled version of Scheme 1, where all the I/O is being managed by the SPE's. For this experiment, we made the following changes to the workload of Scheme 1 and Scheme 6, while keeping the total input size unchanged (i.e. 64 MB).

For Scheme 1, the PPE starts one thread for each of the six SPE's. Each SPE thread reads an input file of 10.67 MB, encrypts it, and writes the resulting buffer back to the disk. The total input size across all the SPE's remains 64 MB.

For Scheme 6, we still read the 64 MB file on the PPE, but instead of giving the entire workload to a single SPE, it is evenly distributed among the six SPE's. Each SPE processes its portion of the buffered data as follows. The

first 16 KB block in the input buffer is processed by one SPE, the next 16 KB block in the same buffer is processed by another SPE, and so on. Once the PPE has read the file completely in main memory, it waits for the output to be produced by the SPE's before writing it back to the disk.

Table 6 shows the results of this experiment. The total time for Scheme 1 also includes the time spent by PPE to wait for all the six SPE threads to complete their execution (i.e. barrier time). Note that parallelizing read and write operations among SPE's provides considerable speedup, 44.7%, 44.0% respectively, compared to Scheme 1 in Section 4.2 where the same amount of data is read and written by a single SPE. Also note that average time to read the file at each SPE is less than the total file reading time on the PPE because each SPE reads only a fraction of the file read by the PPE. The results show that Scheme 6 performs better (8.7%) than Scheme 1, when all available SPE's are utilized in the Cell/BE, albeit by a narrower margin compared to the case where a single SPE is used.

Scheme 6 improves performance only by about 23.9% when it is scaled from one to six SPE's. This result is attributed to several reasons. First, the file reading time for the scaled version of Scheme 6 is considerably more than the original version because here the PPE also has to compute the reserve status locations based upon the block number that has just been read from the disk.

File reading time also increases because of EIB contention since now all six available SPE's along with the PPE are using the EIB to read and write data in main memory. Secondly, the DMA wait time at each SPE increases significantly in the scaled version of Scheme 6 (25 msec.) as compared to Scheme 6 using a single SPE (0.07 msec.), although each SPE in the former case is required to process 1/6 of the total data processed in the latter case. This happens also because of EIB contention and suboptimal routing of the DMA requests on the EIB rings.

4.3 Discussion

Our evaluation has shown that to achieve good I/O performance, the I/O block sizes and the DMA buffer sizes should be matched to the maximum DMA channel size of 16 KB. Further, we observed a clear benefit of prefetching a file using the PPE and then offloading it to SPE, rather than letting the SPE's do the I/O directly. One observed bottleneck is that all I/O from SPE's is sent to the PPE for handling, which results in performance degradation. Our DMA based approach using signaling provided best performance for our workload. We recommend using similar techniques for I/O intensive workloads with the current OS implementation on the Cell/BE.

An important observation is that by allowing SPE's to do DMA to a prefetched file in memory, the bottleneck of doing centralized I/O is removed: each SPE directly goes to the memory through DMAs rather than going through the PPE for I/O. This indicates that incorporating I/O functionality in the SPE library code rather than relying on the PPE OS can yield promising results. The trade-off lies in the fact that loading full I/O capabilities onto the SPE's reduces the space available in SPE local storage for running other compute-intensive tasks. We plan to explore this avenue, by developing direct I/O functionality in `libspe` and investigating the aforementioned trade-off in the context of realistic I/O workloads.

5. RELATED WORK

We discuss related research on the Cell/BE and on prefetching for improving I/O performance.

Cell/BE The Cell/BE has been the subject of several application studies, including particle transport codes [41], numerical kernels [1, 3], irregular graph algorithms [4], and algorithms for sequence alignment and phylogenetic tree construction [7, 44]. More recent studies explores the potential of the Cell/BE for accelerating the processing of large data volumes and used the Cell/BE to implement fast sorting [19], query processing [23], and data mining [8] algorithms. Our contribution departs from earlier work by focusing on the implementation of I/O operations in the Cell/BE system software stack.

The Cell/BE has also spurred several efforts for developing high-level programming models and supporting environments for simplifying code development and optimization. These efforts include Sequoia [17], Cell SuperScalar [6], CorePy [37] and PPE-SPE code generators from single-source modules [16, 49]. Our research is based on the generic Linux I/O interfaces, however it is conceptually related to programming models that explicitly manage the memory hierarchy by staging data vertically through the machine and localizing computation to specific layers of the memory hierarchy [17].

Prefetching A key technique for improving I/O performance of workloads is prefetching, which dates back to as early as Multics [18]. A large amount of work on I/O prefetching utilizes hints about an application's I/O behavior, e.g., programmer-inserted hints [12, 39], compiler-inferred hints [36], and hints prescribed by a binary rewriter [13]. Alternatively, dynamic prefetching has been proposed that detects applications' reference patterns at runtime, e.g., prediction using probability graphs [21, 50], and time series modeling [47]. Prefetch algorithms tailored for parallel I/O systems also have been studied [2, 30, 31]. Speculative prefetching at the level of whole files or database objects has been proposed by many works [15, 20, 34, 35, 38].

The interaction between prefetching and caching has also been identified [9, 10]. Based on these interactions, a number of works have proposed integrated caching and prefetching schemes [2, 11, 30, 31, 32, 39, 46] that simultaneously identify and handle temporal and spatial I/O access patterns. FlexiCache [33] provides a new flexible interface that allows easy modification of disk cache management decisions using OS-level modules.

In this paper, we explore how basic prefetching techniques can be employed to improve the performance of I/O intensive workloads on the Cell/BE architecture. To the best of our knowledge, this is the first exploration of such techniques in the Cell/BE setting.

6. CONCLUSION

We investigated prefetching-based techniques for supporting I/O intensive workloads involving significant computation components on the Cell/BE architecture. We observe that the current operating system facilities for performing I/O directly on accelerator cores (SPE's) are limited, and do not provide judicious use of the available resources. A particular concern is that currently, I/O on SPE's is redi-

rected to the PPE, hence creating a central bottleneck. We have presented an asynchronous prefetching-based approach that partially breaks up this bottleneck, utilizes decentralized DMA to achieve 22.2% better performance for our workload compared to the case where all I/O is handled at the SPE. However, we argue that a fundamentally better approach would be to extend SPE support libraries with I/O functionality, thus removing the dependence on the PPE, simplifying the SPE program design for I/O intensive workloads, and improving overall performance. We are currently investigating the feasibility of such library support.

Acknowledgment

This research is supported by the NSF (grants CCF-0346867, CCF-0715051, CNS-0521381, CNS-0720673, CNS-0709025, CNS-0720750, NSF CAREER Award CCF-0746832), the DOE (grants DE-FG02-06ER25751, DE-FG02-05ER25689), and by IBM through an IBM Faculty Award (Virginia Tech Foundation grant VTF-874197). In addition, M. Mustafa Rafique is supported by a Scholarship from the Fulbright Foreign Student Program of the U.S. Department of State, funded in part by the Government of Pakistan.

7. REFERENCES

[1] S. Alam, J. Meredith, and J. Vetter. Balancing Productivity and Performance on the Cell Broadband Engine. In *Proc. IEEE CLUSTER*, 2007.

[2] S. Albers and M. Büttner. Integrated prefetching and caching in single and parallel disk systems. In *Proc. ACM SPAA*, 2003.

[3] D. Bader and V. Agarwal. FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine. In *Proc. IEEE HiPC*, 2007.

[4] 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. IEEE IPDPS*, 2007.

[5] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. In *Proc. ISCA*, 2005.

[6] P. Bellens, J. M. Pérez, R. M. Badia, and J. Labarta. CellSs: a Programming Model for the Cell BE Architecture. In *Proc. SC*, 2006.

[7] F. Blagojevic, A. Stamatakis, C. Antonopoulos, and D. Nikolopoulos. RAxML-CELL: Parallel Phylogenetic Tree Construction on the Cell Broadband Engine. In *Proc. IEEE IPDPS*, 2007.

[8] G. Buehrer and S. Parthasarathy. The Potential of the Cell Broadband Engine for Data Mining. Technical Report TR-2007-22, Department of Computer Science and Engineering, Ohio State University, 2007.

[9] A. R. Butt, C. Gniady, and Y. C. Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. *IEEE ToC*, 56(7):889–908, 2007.

[10] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. *SIGMETRICS Performance Evaluation Review*, 23(1):188–197, 1995.

[11] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM TOCS*, 14(4):311–343, 1996.

[12] P. Cao, E. W. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *Proc. USENIX OSDI*, 1994.

[13] F. W. Chang and G. A. Gibson. Automatic I/O hint generation through speculative execution. In *Proc. USENIX OSDI*, 1999.

[14] K. Chang, S. Pamarti, K. Kaviani, E. Alon, X. Shi, T. Chin, J. Shen, G. Yip, C. Madden, R. Schmitt, C. Yuan, F. Assaderaghi, and M. Horowitz. Clocking and Circuit Design for a Parallel I/O on a First-Generation Cell Processor. In *Proc. IEEE ISSCC*, 2005.

[15] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *Proc. ACM SIGMOD*, 1993.

[16] A. E. Eichenberger et al. Using Advanced Compiler Technology to Exploit the Performance of the Cell Broadband Engine Architecture. *IBM Systems Journal*, 45(1):59–84, 2006.

[17] K. Fatahalian et al. Sequoia: Programming the Memory Hierarchy. In *Proc. SC*, 2006.

[18] R. J. Feiertag and E. I. Organik. The Multics input/output system. In *Proc. ACM SOSP*, 1972.

[19] B. Gedik, R. Bordawekar, and P. S. Yu. Cellsort: High performance sorting on the cell processor. In *Proc. VLDB*, 2007.

[20] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *Proc. USENIX Summer Technical Conf.*, 1994.

[21] J. Griffioen and R. Appleton. Performance measurements of automatic prefetching. In *Proc. PDSC*, 1995.

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

[23] S. Heman, N. Nes, M. Zukowski, and P. Boncz. Vectorized Data Processing on the Cell Broadband Engine. In *Proc. DaMoN*, 2007.

[24] M. Hill and M. Marty. Amdahl's Law in the Multi-core Era. Technical Report 1593, Department of Computer Sciences, University of Wisconsin-Madison, Mar. 2007.

[25] H. P. Hofstee. Power efficient processor architecture and the cell processor. In *Proc. IEEE HPCA*, 2005.

[26] IBM Corp. Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification, Version 1.0, November 2005.

[27] IBM Corp. Cell Broadband Engine Software Development Kit 2.1 Programmer's Guide (Version 2.1). 2006.

[28] IBM Corp. Cell Broadband Engine Architecture (Version 1.02). 2007.

[29] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy Introduction to the cell microprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, 2005.

[30] M. Kallahalla and P. J. Varman. Optimal prefetching and caching for parallel I/O sytems. In *Proc. ACM SPAA*, 2001.

[31] T. Kimbrel and A. R. Karlin. Near-optimal parallel prefetching and caching. *SIAM J. Comput.*, 29(4):1051–1082, 2000.

[32] T. Kimbrel, A. T. R. H. Patterson, B. Bershad, P. Cao, E. Felten, G. Gibson, A. Karlin, and K. Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proc. USENIX OSDI*, 1996.

[33] P. Konanki and A. R. Butt. Flexicache: A flexible interface for customizing Linux file system buffer cache replacement policies. In *Work-in-Progress, USENIX FAST*, 2007.

[34] K. Korner. Intelligent caching for remote file service. In *Proc. ICDCS*, 1990.

[35] D. Kotz and C. Ellis. Practical prefetching techniques for parallel file systems. In *Proc. ICPADS*, 1991.

[36] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proc. USENIX OSDI*, 1996.

[37] C. Mueller, B. Martin, and A. Lumsdaine. CorePy: High-Productivity Cell/B.E. Programming. In *Proc. 1st STI/Georgia Tech Workshop on Software and Applications for the Cell/B.E. Processor*, 2007.

[38] M. Palmer and S. Zdonik. FIDO: A cache that learns to fetch. Technical Report CS-90-15, Brown University, 1991.

[39] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proc. ACM SOSP*, 1995.

[40] M. Pericàs, A. Cristal, F. Cazorla, R. González, D. Jiménez, and M. Valero. A Flexible Heterogeneous Multi-core Architecture. In *Proc. PACT*, 2007.

[41] F. Petrini, G. Fossum, J. Fernández, A. L. Varbanescu, M. Kistler, and M. Perrone. Multicore surprises: Lessons learned from optimizing sweep3d on the cell broadband engine. In *Proc. IEEE IPDPS*, 2007.

[42] K. R., K. Farkas, N. Jouppi, P. Ranganathan, and D. M. Tullsen. Processor Power Reduction via Single-ISA Heterogeneous Multi-core Architectures. *Computer Architecture Letters*, 2, 2003.

[43] K. R., D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance. In *Proc. ISCA*, 2004.

[44] V. Sachdeva, M. Kistler, W. E. Speight, and T.-H. K. Tzeng. Exploring the viability of the cell broadband engine for bioinformatics applications. In *Proc. IEEE HiCOMB*, 2007.

[45] 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*, 51(5):559–572, 2007.

[46] A. Tomkins, R. H. Patterson, and G. A. Gibson. Informed multi-process prefetching and caching. In *Proc. ACM SIGMETRICS*, 1997.

[47] N. Tran and D. A. Reed. ARIMA time series modeling and forecasting for adaptive I/O prefetching. In *Proc. ACM ICS*, 2001.

[48] J. A. Turner. Roadrunner: Heterogeneous Petascale Com-puting for Predictive Simulation. Technical Report LANL-UR-07-1037, Los Alamos National Lab, Las Vegas, NV, Feb. 2007. ASC Principal Investigator Meeting.

[49] A. L. Varbanescu, H. J. Sips, K. A. Ross, Q. Liu, L.-K. Liu, A. Natsev, and J. R. Smith. An effective strategy for porting c++ applications on cell. In *Proc. ICPP*, 2007.

[50] V. Vellanki and A. L. Chervenak. A cost-benefit scheme for high performance predictive prefetching. In *Proc. ACM/IEEE CS*, 1999.

[51] B. Wun and P. Crowley. Network I/O Acceleration in Heterogeneous Multicore Processors. In *Proc. IEEE HOTI*, 2006.