Cellule: Lightweight Execution Environment for Accelerator-based Systems

Vishakha Gupta
Georgia Institute of Technology
vishakha@cc.gatech.edu

Jimi Xenidis
IBM Corporation
jimix@us.ibm.com

Karsten Schwan
Georgia Institute of Technology
schwan@cc.gatech.edu

Priyanka Tembey
Georgia Institute of Technology
ptembey3@mail.gatech.edu

Ada Gavrilovska
Georgia Institute of Technology
ada@cc.gatech.edu

ABSTRACT

The increasing prevalence of accelerators is changing the high performance computing (HPC) landscape to one in which future platforms will consist of heterogeneous multi-core chips comprised of both general purpose and specialized cores. Coupled with this trend is increased support for virtualization, which can abstract underlying hardware to aid in dynamically managing its use by HPC applications while at the same time, provide lightweight, efficient, and specialized execution environments (SEE) for applications to maximally exploit the hardware.

This paper describes the Cellule architecture which uses virtualization to create high performance, low noise SEEs for accelerators. The paper describes important properties of Cellule and illustrates its advantages with an implementation on the IBM Cell processor. With compute-intensive workloads, performance improvements of up to 60% are attained when using Cellule's SEE vs. the current Linux-based runtime, resulting in a system architecture that is suitable for future accelerators and specialized cores irrespective of whether they are on-chip or off-chip. A key principle, coordinated resource management for accelerator and general purpose resources, is shown to extend beyond Cell, using experimental results obtained on a different accelerator platform.

General Terms

Virtualization, accelerators, lightweight execution, service processor

Keywords

Special execution environment, rHype, Controller, OS noise

1. INTRODUCTION

To accelerate the execution of a large class of workloads, general purpose CPUs are being enhanced with additional asynchronous processing units, such as the Cell processor [42], graphics processors [10, 37], cryptographic units, and network processors [27]. While the high performance achieved by configurations ranging from GPU-accelerated desktops to the Cell-based RoadRunner [40] supercomputer has established the market for accelerator-based heterogeneous multicore systems, there remain several technical challenges to be addressed, particularly in the systems software. They include increased programming complexity in addition to difficulties in extracting predictably high levels of performance when running carefully programmed codes on accelerator cores. One issue concerning the latter is that the commodity cores directly associated or interacting with accelerators typically run general purpose operating systems. This can perturb the execution of parallel codes with undue levels of OS noise[5]. Furthermore, general OS interfaces may hide (through, say drivers [34]) or make it difficult to efficiently exploit accelerator hardware. Finally, when the commodity cores directly associated with accelerators are designed as 'service processors', they are less capable than full-featured cores for running complex operating systems like Linux. This is the case for the Power core associated with the Cell processor[17], the ARM-based service core on the IXP network processor[25], and the Pentium M on Larrabee[37].

The *Cellule* system presented and evaluated in this paper targets accelerator-based high performance platforms. Cellule addresses the problems mentioned above by running accelerator codes in Specialized Execution Environments (SEEs). Each SEE is customized for a certain accelerator to efficiently run its applications, presenting a complete 'container' for a custom operating system, accelerator libraries (e.g., the Cell's libSPE libraries), and other support software. Cellule uses virtualization to implement SEEs, so that any number of them can be created for any number of accelerators attached to a high performance machine. In fact, Cellule's SEEs could even be customized to meet specific application requirements, such as data streaming vs. graphics vs. numerically intensive classes of applications. This is because each SEE is created per application by a hypervisor that ensures isolation between multiple SEEs, makes resource allocation decisions, and multiplexes them based on scheduling policies that can be adapted to suit the underlying platform and its use. Applications using SEEs also benefit from virtualization technology's improved portability and reduced development and management costs [13].

The Cellule system described and evaluated in detail in this paper was developed for the Cell accelerator, as shown in Figure 1, but

its principles extend to other accelerators, including those used in networking, as demonstrated in Section 4.2 of this paper.

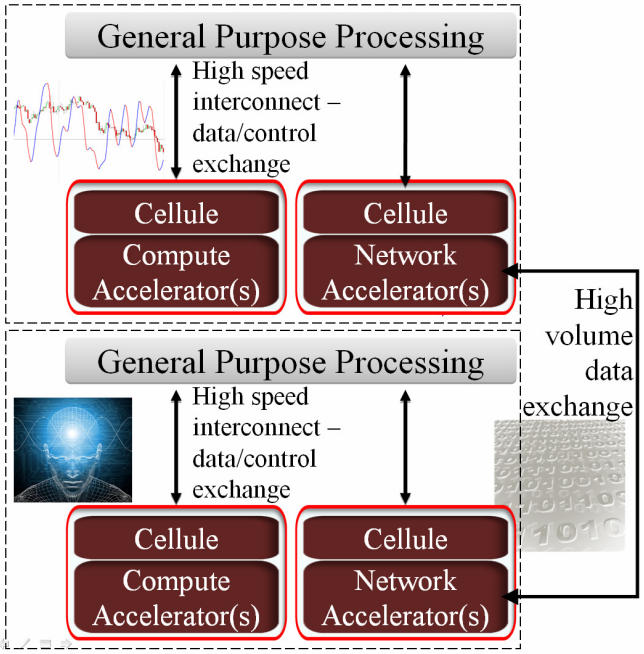


Figure 1: Potential Cellule-based High Performance Systems

Cellule contributes to broader efforts in our research that target future integrated accelerator-based systems, examples including Intel's Tolapai and AMD's Fusion platforms [24, 2]. Our goal is to better understand how to harvest future combined commodity and accelerator resources, i.e., heterogeneous multicore systems, to efficiently deal with their (i) general purpose and their specialized processing cores, (ii) diverse memory units and memory management support, (iii) communication media connecting them, and (iv) specific architectural features used to accelerate certain application requests or tasks. In such settings, Cellule's notion of a SEE can be used to provide fine-grain control over heterogeneous cores and their capabilities, while at the same time, making it possible to develop and experiment with higher level programming models[34, 14, 26] that hide underlying hardware complexities from application programmers. The Cellule SEE, for instance, can run libSPE-based Cell codes with high performance and low noise, as shown with experimental results presented in this paper. Experimental evaluations do not consider, however, the well-known problems arising from the unfortunate fact that current accelerators are not always tightly integrated with host cores, using PCI interconnects, instead.

The low-level runtime support and interfaces [34, 22, 23] as well as higher level libraries [43, 21] used by accelerator codes currently rely on general purpose operating systems and drivers for support. This is also the case for ongoing work to develop uniform accelerator APIs with associated language features and runtimes [26, 9, 39]. Cellule complements such efforts by exploring and providing system-level abstractions and principles that enable the efficient execution of programs on both general purpose and accelerator cores. The approach taken by Cellule uses the virtualization technologies that are increasingly present in modern high end hardware in order to create 'lightweight' execution environments for running accelerator codes. A prototype implemented on IBM's Cell B.E processor permits the creation of a Cell SEE that replaces an existing Linux implementation, offering performance benefits and reductions in the variability of application execution times for Cell codes. To demonstrate generality of the approach, selected SEE functionality is also implemented on Intel's IXP network processor.

The important concepts derived from *Cellule* and its SEEs are highlighted below and explained in greater detail in the following sections:

- *accelerator-specific memory model*: offering reduced overheads in terms of memory allocation and use;
- *simple task model*: appropriate for accelerator use rather than for the general applications targeted by operating systems like Linux;
- *smaller code base*: for ease of development, debugging, and efficient operation;
- *improved interrupt and signaling mechanisms*: for improving predictability and response time for HPC applications; and
- *finer grained scheduling and resource management*: adapted to the idiosyncrasies and special features of target accelerators.

The remainder of this paper articulates the basic principles and abstractions inherent in Cellule, and evaluates them on prototypes of future heterogeneous multicore nodes. Section 2 focuses on the Cellule architecture and introduces the specific design and implementation for our example accelerator based system based on the Cell B.E processor in Section 3. Section 4 evaluates Cellule's Cell-based adaptation and certain functionalities on an IXP-based adaptation. We compare our work with other related efforts in Section 5 and present conclusions and future work in Section 6.

2. CELLULE ARCHITECTURE

The Cellule principles outlined in the previous section are achieved with the software architecture and stack described in Figure 2. The figure also shows relevant hardware-software and software-software interactions.

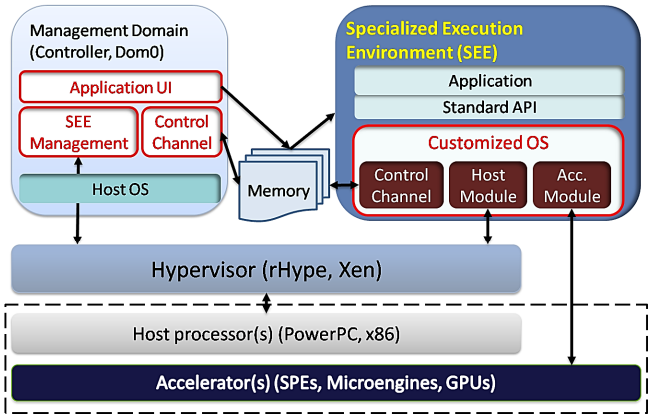


Figure 2: *Cellule* Architecture

Cellule is comprised of several components:

- *Hypervisor*, which controls the underlying hardware, supports the creation and resource allocation for lightweight VMs running SEEs and provides isolation between different SEEs. It also implements efficient SEE scheduling and interrupt delivery mechanisms.

- *Specialized Execution Environment (SEE)*, in which accelerator-based applications are executed. This is the lightweight container supporting higher level programming interface(s) or abstraction(s) in order to enable a wide range of applications, with an underlying runtime adapted to the given accelerator architecture.
- *Controller*, or a management domain, responsible for driving configuration and management actions and for execution of non-critical, heavy-weight or less-frequently needed functionality on behalf of the SEE, e.g., IO functionality in case of compute-intensive accelerators.
- *Control Channel*, used for communication between the SEE and the controller domain, realized via shared memory or message passing, as best suited for the target accelerator platform.

The hypervisor and the controller (or management domain) work together to create specialized execution environments (SEEs). In order to provide a user-friendly interface, a UI in the management domain (e.g., command line in Linux) allows the invocation of desired applications. These applications, linked against the SEE library, can be run in a new partition (or domain in Xen [4] parlance) created with the help of the SEE Management module present in the management domain. As mentioned earlier, the SEE is expected to support the standard API used by its applications and targeted accelerator (e.g., libSPE in the case of the Cell processor). The operating system (OS) in the new partition can be viewed as the runtime customized to support the standard API, with special distinction between the host part and the accelerator part (see the host and accelerator modules shown in the figure). The Control Channel functions as a channel for offloading certain functions as explained above from the SEE or for passing control messages between the SEE and management domain. The hypervisor multiplexes/demultiplexes interrupts and other system level requests/resources between multiple SEEs and schedules them according to policies resident in the hypervisor.

SEEs make it possible to run accelerator codes with low noise and high performance, for several reasons:

- *Accelerator-specific memory model:* for accelerators expected to perform efficient data transfers multiple times and in large blocks, the memory management scheme used by the execution environment can make a significant difference. For instance, the SEE for the Cell processor creates a 'flat' address space model using large page sizes. This results in fewer SLB and TLB misses compared to the standard Linux-based SEE. Similarly, the SEE running on the IXP network processor maps memory in DMA regions, thereby avoiding unnecessary data copying.
- *Simple task model:* the SEE should provide the capability of loading the required executable(s), triggering their execution, and coordinating data movement to and from the execution units within the accelerator(s) in a lightweight manner. Therefore, seeking deterministically high levels of performance and the avoidance of OS noise [5], the SEE task model involves only a single main task or coordinator (which usually runs on the host core) that is designed to immediately respond to communication or computational events from the accelerators. Hypervisor calls are required only for requesting and releasing system resources, thereby limiting virtualization overheads during application execution. The program fragments running on accelerator cores are moved there without the need to traverse the multiple levels of hierarchy imposed by a general purpose execution environment.
- *Smaller code base for ease of development, debugging and efficiency:* the entire Cellule code running on the Cell board is less than 10KLOC (thousand lines of code). This has several benefits, the primary ones being: (1) ease of debugging and (2) reduced complexity. Smaller and less complex systems are more easily loaded (i.e., remote loading for Cellule involves a footprint of only ∼600K when compressed), they have fewer potential sources of errors and side-effects, and they are typically easier to test and debug.
- *Potential for improved interrupt and signaling mechanisms:* depending on the frequency of interrupts or signals from the accelerator, efficient interrupt delivery and signaling can play a significant role in achieving expected performance and timing guarantees. For *Cellule*, predictable, efficient interrupt delivery is enabled by the facts that (1) the SEE runs the application in supervisor mode and (2) that it adjusts the code path for fast interrupt delivery wherever possible. Since the current implementation of Cellule on Cell does not involve IO and its applications do not depend on interrupts, we have not evaluated this potential. We do evaluate the utility of lightweight signaling mechanisms for better resource co-ordination on the IXP in Section 4.
- *Finer grained scheduling, resource management and isolation:* because Cellule virtualizes accelerator resources, the scheduling of SEEs can be coordinated with the manner in which host cores are scheduled. This is shown to be important for concurrency across host and accelerator tasks in related work by our group using graphics accelerators [15]. It also makes it possible to schedule SEEs to match specific application needs and/or overall system requirements, an example of the latter being shared accelerator use by multiple host-level applications. Isolation properties enforced by, i.e., resource allocation managed by, Cellule can ensure that each such application receives the share of accelerator resources it needs for timely execution.

Adapting the Generic Architecture. The previous discussion has presented a general view of SEEs and their realization. We next describe in additional detail the SEE implemented for the Cell B.E processor, where the Power core acts as host to the multiple co-processors (i.e., SPEs). For brevity, less information is provided about the architectural and implementation details of the SEE functionality constructed for the IXP network processor, mentioning only some of the high level insights gained from our evaluation of an early SEE prototype on that platform (see Section 4). The following sections first present a brief overview of the Cell B.E. hardware, to highlight its unique characteristics that give rise to the specific adaptation of *Cellule* constructed for it.

3. CELLULE ON CELL B.E

Several aspects of the Cell architecture make it a suitable vehicle for prototyping Cellule and substantiating our earlier claims.

3.1 The Cell Processor Architecture

The IBM Cell B.E processor is a heterogeneous chip multiprocessor consisting of multiple elements. Its service processor is an IBM 64-bit dual-threaded Power Architecture core, called Power Processing Element (PPE). This somewhat simplified Power core is augmented with eight specialized co-processors based on a single-instruction multiple-data (SIMD) architecture, called Synergistic Processor Element (SPE) acting as the platform's workhorses. The PPE and SPEs are integrated via a coherent, high speed, on-chip

bus. Figure 3 shows the architecture's various components. Additional details about the performance and power properties of the chip appear in [17, 31].

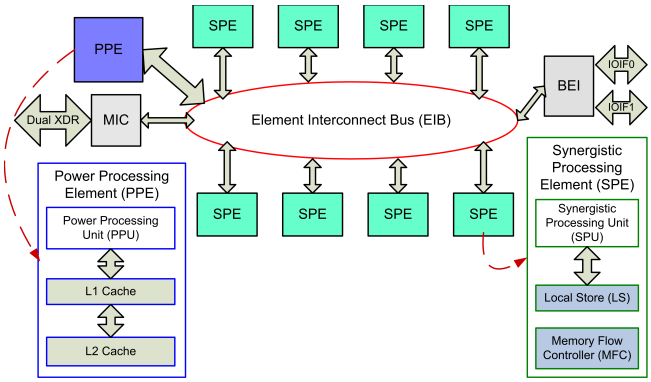


Figure 3: Cell B.E Architecture

Asynchronous operation is important, because Cell performance is driven by the SPEs, with the PPE primarily acting as a coordinator. As with other co-processor architectures [28], this design choice makes it important to decouple PPE and SPE operations, to limit PPE overheads by using a lightweight execution environment, and to constrain PPE operations to those needed for coordination and data staging.

Architectural features for high performance include fast asynchronous DMA from main memory to SPE local stores initiated by the SPEs to reduce the load on the PPE, support for large pages to avoid SLB and TLB misses, and a lower capability PPE to reduce power consumption and die area while accelerating computation. SPEs use small mailbox registers in order to indicate execution status, communicate flags and other such control information and avoid interrupts. The PPE polls these mailboxes as necessary. By scheduling the PPE side of the application as frequently as possible, it is possible to reduce the latency between when a mailbox is written and read. By pre-loading all data in memory, the number of SLB and TLB misses can also be reduced.

Isolation and customization through virtualization are critical elements of Cellule. The PowerPC architecture inherently supports virtualization, thereby making it a viable technology. Specifically, for the Cell platform, there is a distinction between the SPE privilege 1 memory area, which is exclusively for Hypervisor use, and the privilege 2 problem state and local store memory area, which is accessible to privileged mode software and can be made available to the application. This implies that the application never has to go through the hypervisor to access the SPEs once they are acquired, since the DMA queues and mailboxes used for PPE-SPE communication are part of the problem state. Thus, the hypervisor does not intervene in the normal execution of the application, except (1) when interrupts or signals are received from the SPEs or IO devices and (2) in cases where it has to switch between different SEEs (i.e., partitions). Further, the presence of IOMMU presents us with the possibility of isolating external IO device communications if required. Most importantly, virtualization makes it possible to customize the execution environment to match the Cell architecture and its properties without sacrificing the isolation guarantees.

3.2 Cellule Adaptation on Cell

The Cellule adaptation for Cell looks very similar to the architecture shown in Figure 2. The corresponding software components used on Cell are as follows:

Hypervisor:. IBM's rHype [20] (research hypervisor), is a small, low-latency, modular hypervisor suitable for high performance computing and architecture validation. Due to its prior use as a high performance validation platform for Cell chips, it constitutes an appropriate hypervisor code base for realizing the Cellule prototype.

Specialized Execution Environment (SEE):. the SEE provides the base mechanisms needed to run arbitrary Cell applications. A SEE is created on demand by the hypervisor upon the controller's request. The SEE for Cell runs (1) Cell applications and (2) a runtime exposed to these applications via the libSPE [22] interface (the standard library [22] used by PPE programs to access and manage SPEs in a Linux-based environment). The SEE permits Cell applications to run unmodified compared to the Linux environment, necessitating the presence of the libSPE interface which, otherwise, can be replaced with any other programming standard since the basic access mechanisms remain constant. The libSPE-based applications running within the SEE on the PPE are structured to request SPE resources from the execution environment (e.g., SEE or the Linux OS which handles such requests in a fashion it deems best) before using them. In Cellule, this part of the application runs in the supervisor privilege mode in the SEE and interacts directly with rHype. rHype implements the policies for SPE as well as SEE management and guarantees isolation between them. The SPE part of the application runs directly on SPEs once they are acquired.

Controller:. this is the management partition that collaborates with rHype in creating, managing and destroying SEEs. This partition can run Linux or any operating system that can be booted on rHype. It provides (1) the Cell Application UI – which allows the user to select Cell applications to run, (2) SEE Management – which is responsible for enabling the execution of Cell applications by requesting rHype to create an SEE, and for running the application in this SEE, and (3) a Control Channel – which is a communication channel for handling I/O or other such requests that are rarely used and hence not supported in the SEE.

Control Channel:. to reduce the complexity of the SEE, I/O actions and other infrequent calls in an application can be executed in a different partition, preferably in the Controller. Our current design designates the Controller as an endpoint by directing SEE I/O calls to communication protocols run in the Controller. One option for implementing such protocols is to use 9P from Plan 9 [35], due to its simplicity and its provision of a single interface to any form of communication, ranging from Ethernet to shared memory. SEE/Controller interactions for I/O can then be carried out by a 9P Client within the SEE and a 9P Server in the controller. This server is responsible for performing the I/O requests on behalf of a SEE. To users running SEE applications, it appears as if their applications are executing in the controller partition. As described in Section 6, it would be interesting to explore the use of a separate, specialized I/O partition for such purposes. Control Channel operation and performance are not evaluated in this paper, as they are not germane to the technical points being made.

3.3 Current Implementation

We have ported rHype to the Cell platform from the earlier PowerPC 970 base. Its current implementation does not over-commit SPEs. The memory model implemented is a flat one, with support for large pages. The SEE implements exactly the functions required for a standard Cell application, and it provides a wrapper that handles common libSPE function calls so as to run unmodified SPE applications. The wrapper currently supports most interfaces from libSPE2.1. The PPE source of an application must be linked against the SEE library, but the SPE executable is loaded unmodified and operates just as it does in the current Linux environment. A typical usage scenario for Cellule is as follows:

1. Initially, rHype owns all SPEs and arranges for a controller to boot.
2. The controller provides options to invoke the desired SPE application.
3. The chosen Cell application (already linked with the SEE library) is mapped into the address space of a new partition, which is created by the controller's management component.
4. The application now running in the new SEE consists of a main task that interacts with rHype for allocation of SPEs and subsequently, uses them without any intervention from rHype. Any signals and/or interrupts received from the SPEs are received by rHype and directly passed to the partition.
5. Console input or output use Thinwire, a simple utility capable of demultiplexing IO for the different partitions onto different channels. We did not implement the 9P client and server for offloading IO requests.

3.4 Cellule vs. General Purpose OS

In **Linux on Cell**, the Power PC Linux implementation requires additional kernel modifications in order to use the SPEs from an application, since the controlling registers are only accessible from the PPE in privileged mode. Toward that end, a virtual file system named "spufs" [6] is used to externalize the SPUs. Unlike device-backed file systems, these do not need a partition to store data, but instead, keep all their resources in RAM while using regular system calls like open, read, or getdents to communicate between user space and the kernel functionality. The libSPE calls encapsulate file operations and make them invisible to the user, unless a user wants specific optimizations like enabling the use of large pages and memory mapped access to SPE data structures. Large page support can be enabled for certain applications, but this has to be done on a per application basis, and the number of large pages might be restricted based on the setup (unless a user has sudo privileges on the system). The current scheduler is from the mainline Linux kernel, but can be modified to adapt to Cell, requiring significant knowledge of the operating system. The Cellule features described above, namely (1) lightweight execution environment, (2) flat address space, (3) simple task model to avoid threading overheads and (4) its ease of customization and small footprint for debugging, make it a more attractive alternative to a Linux based general purpose environment for this processor.

4. EXPERIMENTAL EVALUATION

The evaluation of Cellule demonstrates the effects of our design choices, motivating the architecture proposed in this paper.

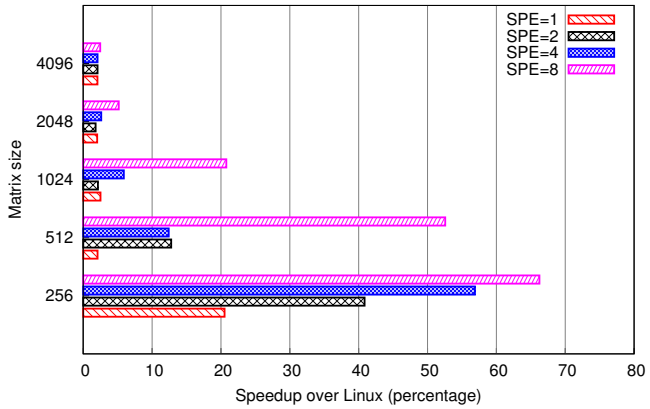
4.1 Evaluation of Cellule on Cell

Cellule on Cell runs standard workloads provided in the the Cell B.E SDK and provided by other groups at IBM with performance similar or superior to the performance attained with the current Linux-based execution environment. These evaluations are conducted on a QS20 blade with 1GB of main memory. Since Cell boards used as accelerators currently have 1PPE with 8SPEs and rHype is currently incapable of SMP behavior, the maximum CPU count in Linux has been set to 1. This enables us to characterize performance in the base case leading to better indicators even when we have all PPE threads enabled. We have used the Linux kernel 2.6.30 for Fedora 7. All tests are performed with large page support enabled. Time is measured in terms of the number of timebase [19] 'tick' increments for Cellule and Linux for greater accuracy. While creating an SPE context [22] in Linux, there is an option to specify whether a user wants the Problem State area to be mapped into user space so that later calls to mailbox read, write, etc do not have to go through the range of system calls for reading and writing files in the spufs. All benchmarks use this mapping for fair comparison. The execution time for our benchmarks encapsulates the time spent in loading the SPE(s), triggering their execution and the computation stage along with PPE-SPE communication to coordinate operations. The benchmarks used above represent the kinds of applications used by IBM when promoting Cell, including scientific and financial codes. They are listed below.

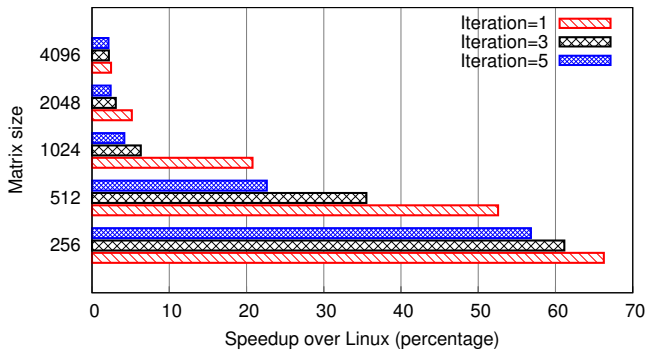
Matrix Multiplication:. performs an optimized version of single precision floating point matrix multiplication ([MM]), where SPEs perform DMA in blocks within a matrix. Matrix sizes (m) can be specified as 128x128, 256x256 and so on. It is possible to tune the number of SPEs (s) involved and the number of iterations (i) performed by each SPE. SPEs perform DMA from the memory to fetch one block of data at a time, then DMA the results back and repeat this sequence if there are more blocks. The implementation is thus characterized by its extreme computational and data intensity and multiple DMAs depending on the size of matrices. It also forms a good example given matrix multiplication is a common step in a lot of applications e.g. image processing, scientific etc. Figures 4.a) and 4.b) show the percentage change of execution times for Cellule vs. Linux ($=\frac{Linux-Cellule}{Linux} * 100$) by varying the number of SPEs used and the number of iterations respectively.

As seen from the figures, programs running in Cellule's SEE perform better than Linux, particularly in case of smaller workload sizes, i.e., when the overall SPE computation time is small compared to the total time of setup and execution. As the size of the workload increases, the time spent in the SPEs for execution becomes larger, making the difference in overall execution time less obvious.

Black Scholes:. [12] is an option pricing formula to calculate European call or put options and and has been adapted by a large number of financial applications. The option pricing model is an example of an intrinsically parallel application suitable for multi-core platforms, with good speedups attained by its port to the Cell platform. The implementation splits the entire data among the number of SPEs to be used and then sends a mailbox message to all SPEs to start the timing runs. The variable parameters are the number of SPEs, the number of cycles for which the computation should be run and the size of data in multiples of 64KB blocks. Figure 5 shows the percentage change of execution times for Cellule vs. Linux. We show the results by varying only the number of cycles because changing number of spes has a similar effect as seen with the pervious benchmark.



(a) $[Iterations = 1]$



(b) $[SPEs = 8]$

Figure 4: Matrix multiplication - Cellule vs. Linux

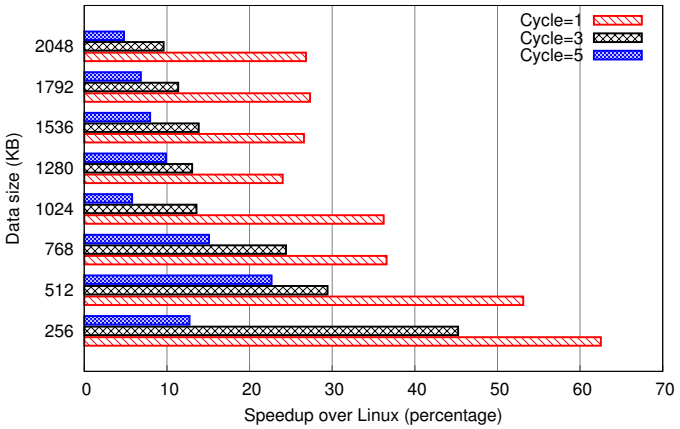


Figure 5: Black Scholes - Cellule vs. Linux $[SPEs = 8]$

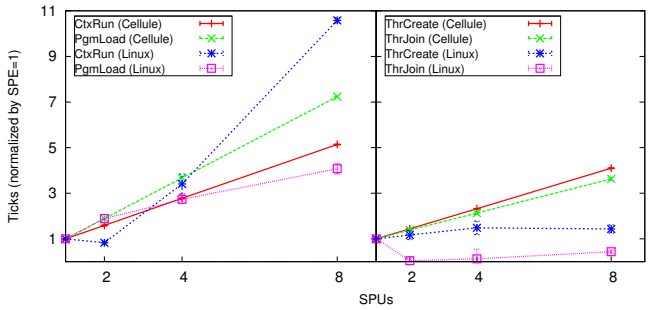
Libspe2 Common Calls	Cellule Time (μsec)	Linux Time (μsec)	Cellule Std. Dev. (μsec)	Linux Std. Dev. (μsec)
CtxRun	2.2	1361.9	0.03	21.82
ThrCreate	3.6	1433.5	0.0	23.97
ThrJoin	0.6	323.9	0.03	18.3
PgmLoad	77.8	104.2	0.06	12.14
MapPS	0.8	1.1	0.03	0.13
MBoxWr	1.0	57.1	0.0	10.02
MBoxRd	9.6	1.5	0.18	0.06

Table 1: DMA Microbenchmark: Function call overhead with SPE=1

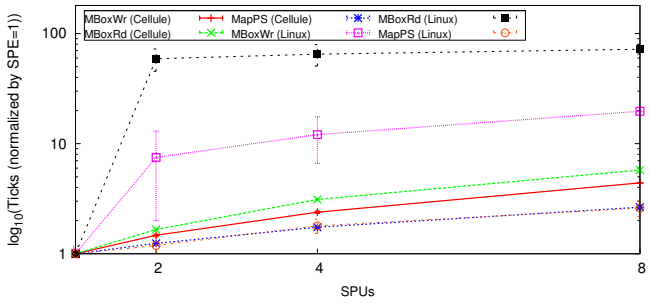
BlackScholes performance follows a similar trend as that seen for matrix multiplication. With increases in the work to be performed on the SPE (e.g., with increase in the number of iterations), we see a smaller percentage gain in Cellule vs. Linux.

Although the Linux implementation has been fine-tuned to handle the nuances of the Cell architecture, the general purpose nature of the operating system struggles to compete with the memory and execution model principles implemented in Cellule, as seen from the results above. The following results justify this statement as well as the Cellule design and implementation principles described in Section 2.

1. *Insights from microbenchmark:* In order to explain the results shown for our benchmarks, we first evaluate the overhead seen on a per call basis for the most common calls [22] used by a Cell application. The microbenchmark used for this purpose is a simple DMA example from the SDK in which a given number of SPEs DMA a 16KB block, perform a simple addition on it, and then DMA it back to PPE memory. While this does not reflect the nature of real applications with substantial SPEs execution times, it does help us evaluate the responsiveness of the two systems as well as their respective implementation overheads.
   Figures 6.a) and 6.b) show timebase comparison and standard deviation on a per-call basis for Cellule and Linux. The numbers have been normalized with respect to times seen when the number of SPEs utilized is 1. The absolute time values with SPE=1 are shown in Table 1 for reference.
   As can be seen from the table and figures, Cellule numbers grow at a predictably linear rate with increases in the number of SPEs and show little or no standard deviation. On the other hand, Linux demonstrates high deviation for certain values, e.g., MBoxWr (write to SPE's mailbox), or it shows exponential increases with increases in the number of SPEs, e.g., MBoxRd (read from SPE mailbox), as illustrated in Figure 6.b (the y-scale is logarithmic). While this does imply an increase in the execution time for applications using these calls in Linux, the implementations of these functions may not always be the reason for these overheads (value for MBoxRd in Table 1). Specifically, we observe that every time the application is swapped out or the kernel delays the delivery of interrupts from the SPEs when they finish computing (due to the switch time involved), the functions show large variations. The exceptions here are the ThrCreate and ThrJoin calls where the Cellule implementation takes longer than what is experienced in Linux. This is due to the difference in the amount of work being done. In general, the smaller execution time recorded by Cellule for most other calls leads to its overall better performance. These results highlight the principles of a lightweight environment described earlier.
2. *Security and Isolation for SEEs:* The previous results do not show the timing comparison between calls to acquire and release SPEs. The Cellule implementation of acquire and release (spe_context_create() and spe_context_destroy() in libspe2) involves several conservative steps to ensure isolation and security of data for the previous and the new owner. Hence, the calls are an order of magnitude slower in Cellule compared to Linux. However, in a typical host-accelerator



(a) SPU Execution and Pthread Calls

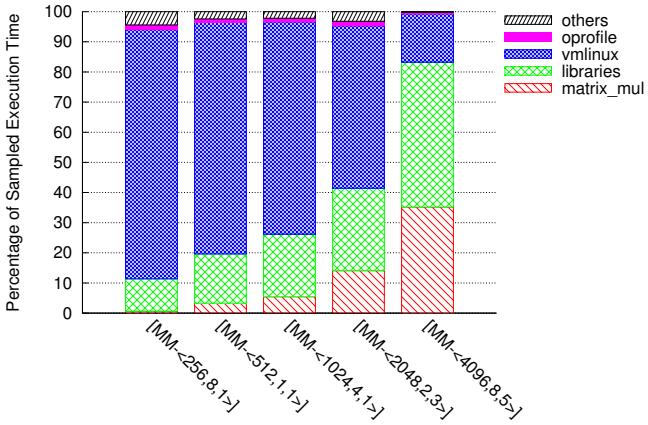


(b) Libspe2 Calls for PPE-SPE Communication

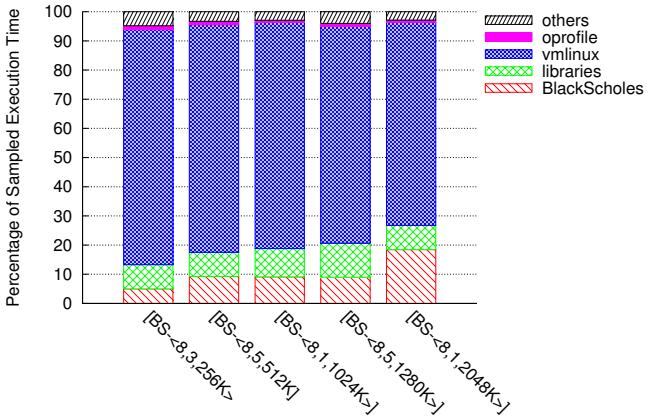
Figure 6: Comparison (with std. deviation) in Timebase Ticks for Common Function Calls

environment where the Cell processor will be the accelerator as shown in Figure 1, SEEs will be created relatively rarely and only on-demand from the host, where the controller running on the PPE will be responsible for acquiring and releasing all of the associated resources, including the SPEs. This should result in moderate to non-existent performance penalties for relatively long running applications.

3. *OS noise:* we have used oprofile on Linux to attain an estimate of the kernel vs. application proportion seen by Cell applications. Figures 7.a) and 7.b) show some sample datasets used earlier for Matrix Multiplication (as MM-⟨Size, #*SPEs*, #*Cycles*⟩) and BlackScholes (as BS-⟨#*SPEs*,#*Cycles*,*Size*⟩). These data sets have been chosen either because Linux performs worse than expected or because it suddenly shows a smaller overhead and hence, better performance.
   The benchmark + libraries shown in the figures are actually used by the application, but they are shown separately because the percentage of time occupied by libraries captures the input data allocation and initialization costs. As seen from the figures, there is a correlation between an increase in kernel activity and a decrease in performance noticed for the application. While a proportion of the kernel time, especially in case of larger datasets, is spent waiting for the SPEs to finish computation (e.g., from 1-38% for matrix multiplication samples profiled), a large portion of this time is due to the latency involved in switching back to the application after the SPEs finish computation. Therefore, the relatively larger switching overheads experienced by a general purpose OS like Linux can directly affect application performance.
4. *Need for sophisticated scheduling policies:* the partition scheduler in rHype is a simple Round Robin scheduler. Lower latencies and negligible variation in execution time for applications are obtained from the simple task model used in the SEE, thereby avoiding threading overheads. We are confident that with further optimizations to the memory management module and some of the SPE management functions, SEE performance can be improved further. However, Cellule scheduling is still prone to perturbations, as shown in Figure 8 where the performance of Cellule is shown to vary. The result is that Cellule can perform better or worse than Linux, depending on data sizes and the system interactions that ensue. This raises the importance of future work on improved hypervisor-level scheduling and the need for better scheduling across the hypervisor and the SEE partitions[15].



(a) Example profile data - Matrix Multiplication



(b) Example profile data - Black Scholes

Figure 7: Profile results on Linux show more kernel activity especially for smaller data sizes

4.2 Cellule Principles on the x86-IXP Network Processor

We comment on some of the Cellule principles by describing an implementation of select SEE functionality on the Intel IXP network processor [30].

Background:. The IXP network processor has an architecture similar to Cell in terms of its use of both commodity and specialized compute engines, which in this case, consist of a coordinator XScale core and supporting co-processors, called microengines, that work as accelerators for networking tasks. Each microengine provides a number of hardware threads, and these can be associated with different processes or virtual machines. The allocation of these threads determines the difference in network performance

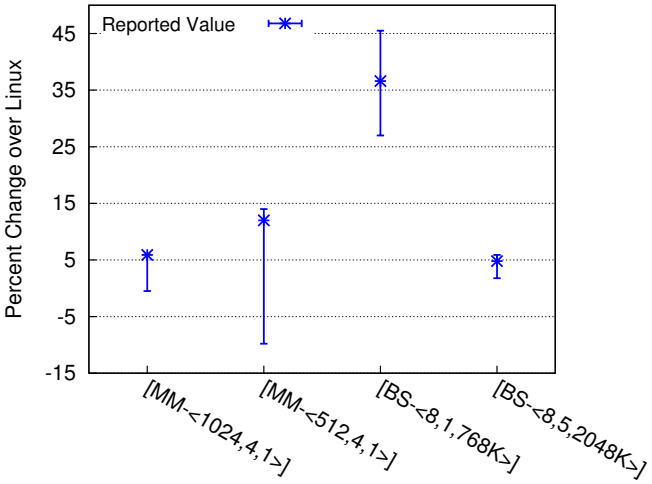


Figure 8: Variation in Cellule performance due to Perturbations

experienced by the processes or virtual machines. The Cellule principle of fine-grained runtime resource management can be demonstrated with the change in the allocation of these hardware threads in synchronization with the requirements of the processes or virtual machines using them. With the XScale used primarily as a setup and coordinator core and x86 cores used as the host, the Cellule implementation on the IXP also proves the applicability of its design for machines with future accelerators like Larrabee [37].

Evaluation:. The x86-IXP Cellule experimental prototype is implemented on a platform consisting of a Dell Precision 390 server host machine running with an Intel Core 2 Duo processor and 1GB physical RAM. The IXP network accelerator platform is the Netronome NFE-i8000 board [32] comprised of an IXP2855 Network Processor, a 600MHz Xscale ARM processor, 256 MB off-chip DRAM, and 256 MB of off-chip SRAM and 4 GigE ports. The NFE-i8000 connects to the x86 over a PCIe interconnect. The software includes Xen 3.3 as our hypervisor, and the IXP Cellule prototype SEE guest domain (DomU) runs Kubuntu 8.04 Hardy with the 2.6.24 Linux Kernel. 384MB of this guest domain memory is reserved for DMA of packet payload and data buffers to/from the IXP. The Netronome NFE-i8000 uses an MSI to interrupt the host, which is directly routed to the guest domain avoiding any hypervisor interference.

Netperf: The Netperf TCP-STREAM benchmark [16] is used to evaluate the Cellule-IXP prototype. The TCP-STREAM test is a bulk data transfer test and lets us test the data streaming capabilities from the SEE DomU through the IXP Network processor. We vary the number of active netperf sessions and their application process priorities, always dedicating one egress GigE port on the IXP platform to one netperf session to avoid contention on the port capacity. At the same time, we test the accelerator's scalability to support high-throughput streams with varying resource requirements and different priorities.

Categorizing virtualization overheads. Table 2 shows the performance comparison of running the netperf benchmark as one application process on a native Linux host with no virtualization capability vs. running it inside a Linux SEE guest domain over Xen.

Host-App	Virtualized	Non-virtualized
Netperf 1	940 Mbps	943 Mbps
Netperf 2	938 Mbps	939 Mbps

Table 2: Netperf performance comparison of Virtualized / Non-virtualized case

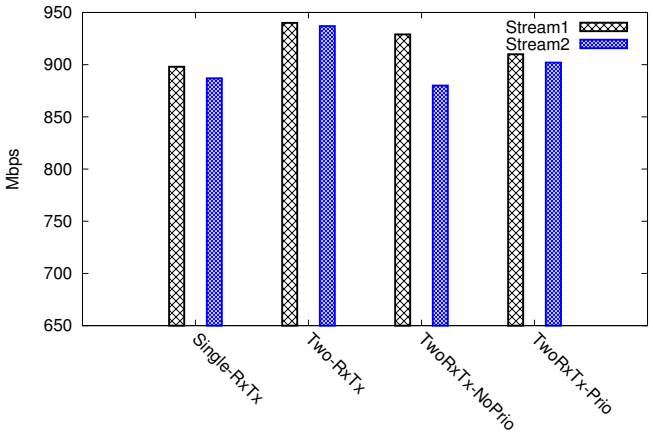


Figure 9: Netperf performance comparison with Cellule optimizations

Table 2 columns correspond to two separate runs of the netperf session pinned to different egress ports on the IXP. In both test runs, we observe negligible virtualization overheads (less than 0.5%) in terms of throughput degradation. This suggests that the I/O path between the host and device does not suffer for virtualized accelerators, enabling them to perform as well as in the non-virtualized case.

Cellule optimizations for Fine-grained accelerator resource management. We mention in Section 2 that SEEs enable accelerator resource at fine grain and in ways that co-ordinate those management actions with host-side processing. The feasibility and utility of such fine-grained management actions are supported via the following sets of experiments, represented in Figure 9.

Test SingleRxTx: the first setup uses one micro-engine for Rx/Tx to/from the host through the IXP. Compared to a Broadcom NIC throughput mark of 940 Mbps, this throughput suffers a maximum degradation of close to 6%. The two bars again correspond to two separate runs of netperf sessions pinned to different egress ports. Due to the round-robin polling of Tx queues on the host side, the second throughput measure will always be slightly lower than the first. We are currently working on increasing the number of kernel threads that service these application queues and their priority scheduling, as opposed to the naive round robin scheduling. The goal is to get fairer results.

Test TwoRxTx: This configuration uses separate micro-engines on the IXP for the Rx and Tx paths. As observed, throughput numbers are improved and are closer to the native Broadcom NIC throughput measurements. This shows that increasing and isolating the number of threads to service the packet-flow queues on the IXP translates to more packet-descriptors processed per unit time and shows an increase in Tx throughput.

Test TwoRxTx-NoPrio: this configuration has two netperf sessions simultaneously running inside the guest and sends out traffic to the two micro-engine Rx/Tx setup on the IXP, but to two separate egress ports. Due to the aforementioned round-robin scheduling effects, the throughput degradation is more pronounced in the second application case (close to 6.5%). These netperf sessions are assigned priorities in the ratio 4:3, but neither the host service thread nor the IXP polling threads have any notion of host process priorities. Hence, the bandwidth distribution suffers as the polling scheduling is naive.

Test TwoRxTx-Prio: here, we change the number of Tx micro-engines to 2, where one micro-engine is dedicated to every egress port and host process. The same netperf sessions with assigned priorities are run, and corresponding numbers of threads are 8:6 on the two Tx micro-engines in proportion to process priorities. As can be seen, the throughput differentiation resulting from this action is also in proportion with the process priorities. Achieving this priority conformity spells the need for such fine-grained coordination actions.

4.3 Discussion of Results

Insights from the experimental evaluations shown in this section support the concept of a specialized execution environment. The improved performance for benchmarks on the Cellule-Cell prototype support our argument of a customized memory model coupled with a simple task model. Additional experimental results attained on the IXP network processor suggest the utility of lightweight signaling mechanisms in the SEE that can co-ordinate fine-grained resource management actions on the accelerator with host-level actions. We are exploring the use of such mechanisms in trying to eliminate some of the noise issues pointed out earlier, to help coordinate between the asynchronous components executing on host and the accelerator cores.

A more general insight from measurements on the Cell B.E. is that the importance of SEEs diminishes somewhat with increasing data-sizes and thus, increasing sizes of work units running on accelerator cores. This holds for individual accelerators, but we note that for larger scale accelerator-based machines like Roadrunner, even small amounts of noise can result in substantial overall performance degradation, thus further bolstering the relative advantages experienced by SEE- vs. commodity OS-based execution environments.

5. RELATED WORK

Previous work dealing with virtualized devices and accelerators differs from the Cellule approach by forcing applications to treat accelerators as devices rather than as processors[29, 15, 36]. While the device model works for accelerators with limited access, limited capability or proprietary models, it can make it difficult and sometimes impossible to completely exploit the functionalities provided by their hardware. With Cellule, we run VMs on both commodity and accelerator cores, each containing different code and runtimes but treated and manipulated by the hypervisor like standard virtual machines. A SEE's specialized runtime deals with architectural differences of accelerator vs. general purpose cores, and its relatively smaller code base eases performance tuning.

In contrast to Exokernel [11], Cellule uses virtualization methods to provide I/O and memory isolation. Further, while the SEE is similar to library operating systems for Exokernel or to earlier work on custom run-times for real-time systems [7, 38], an original contribution of SEE is the set of lessons learned concerning efficient accelerator use as mentioned in Section 1. Another interesting aspect of Cellule is its provision of a general model for efficiently using heterogeneous multi-core platforms, comprised of a Hypervisor and controller that can run different implementations of a SEE (for different accelerators) in their own partitions and separating general I/O functions from the custom communication functionality needed within the SEE. Finally, interesting lessons for how to construct efficient I/O functionality for the next release of Cellule may be drawn from past work on communication-centric OS kernels [18] and from recent work on self-virtualizing I/O [36].

The *Cellule* architecture on Cell borrows heavily from that of Libra [3], which is a specialized execution environment for IBM's J9 Java virtual machine. The key difference to Libra, of course, is Cellule's focus on accelerators and heterogeneous multi-core platforms. Also important is the fact that Cellule concepts extent to other accelerators, as demonstrated with our work with the IXP network processor. Cellule also allows for modifications in low level scheduling, interrupt and signaling mechanisms. In the domain of high performance computing, IBM's Blue Gene/L system and Cray's Catamount operating systems are examples where specialized kernels are directly deployed on the nodes to support applications. On both machines, nodes are split into two types: the I/O nodes, which run the Linux operating system, and compute nodes which run a custom kernel, e.g. BLRTS [1] on the Blue Gene machine. The Catamount OS has similar characteristics. However, these systems do not specifically address the needs of computational accelerators and of the applications that run on them, nor do they support important notions like the need to coordinate scheduling between the host and the accelerator in order to provide the differential services demanded by end users.

P.R.O.S.E [41] is an extension of the exokernel idea to a virtualized system, which then leads to the Libra paper and further motivates our work. We also use virtualization to create SEEs, but in contrast to PROSE, we target accelerator platforms, exploring low overhead data sharing, task models, and lightweight communications. An earlier paper, termed "Specialized Execution Environments" [8], motivates the relevance of SEEs and mentions that they will be useful for special hardware, but that position paper neither provides an actual implementation of its ideas nor an evaluation.

Research on accelerator-based multicore systems has brought forth recent systems like Helios [33]. Helios satellite kernels appear similar to SEEs, but Cellule follows an exokernel approach that overcomes some of its shortcomings through the use of virtualization, whereas satellite kernels are microkernels. SEEs are customized for faster execution based on the idiosyncrasies of the accelerator and support the standard programming models preferred by the developers for a particular class of accelerators. Also, SEEs are created one per application in order to isolate noise issues, which could potentially arise when running components from multiple processes within the same satellite kernel. Due to a tighter coupling between the host and accelerator present in the systems considered, we have not dealt with the issue of message communication between widely separated components.

6. CONCLUSIONS AND FUTURE WORK

Cellule offers a general model for efficient program execution environments for heterogeneous multi-core platforms. Insights attained by creating and experimenting with a concrete instance of *Cellule* for the Cell accelerator include an identification of sev-

eral properties important to such environments. These properties are (1) a memory model customized for the accelerator, i.e., the specialized cores, in question and (2) a simple task model that avoids the overhead of full-featured thread implementations like POSIX threads. Additional work with a second accelerator, the IXP network processor, demonstrates potential advantages derived from (3) lightweight signaling mechanisms to the partition running accelerator applications. Experimental results attained with *Cellule* are encouraging even for our relatively unoptimized *Cellule* prototype. Further, such performance gains are not attained by compromising generality or portability, as *Cellule*'s SEE specialized execution environment is constructed to run arbitrary LibSPE-based Cell applications and unmodified socket applications for the Cellule-IXP prototype. In summary, our work on Cellule not only encompasses the actual implementation presented, but also a set of concepts and architecture suggestions for future accelerator-based systems. It is a first working prototype of a SEE for accelerators that in comparison to a standard general purpose heavy-weight OS, has better performance and provides the benefit of accelerator-customized methods for resource management.

Future work with *Cellule* concerns the stated longer term goal of our work, which is to develop efficient architectural models for heterogeneous multi-core platforms. From the point of view of general purpose processors, *Cellule* implements efficient run-time for the offload engines being used, connected via I/O links (e.g., the PCI links used in architectures like Roadrunner) or more tightly coupled (e.g., via shared memory). Initial results from the Cellule-IXP prototype highlight the importance of better coordination mechanisms for the scheduling of general purpose vs. accelerator cores.

7. REFERENCES

[1] G. Almasi, R. Bellofatto, J. Brunheroto, et al. An overview of the blue gene/l system software organization, 2003.

[2] AMD Corporation. Amd white paper: The industry-changing impact of accelerated computing. `http://www.amd.com/us/Documents/`, 2009.

[3] G. Ammons, J. Appavoo, et al. Libra: A library operating system for a jvm in a virtualized execution environment. In *VEE*, 2007.

[4] P. Barham, B. Dragovic, K. Fraser, et al. Xen and the art of virtualization. In *SOSP*, 2003.

[5] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan. The influence of operating systems on the performance of collective operations at extreme scale. 2006.

[6] A. Bergmann. Spufs: The cell synergistic processing unit as a virtual file system, 2005.

[7] T. E. Bihari and K. Schwan. Dynamic adaptation of real-time software. *ACM Trans. Comput. Syst.*, 9, 1991.

[8] M. Butrico, D. Da Silva, O. Krieger, et al. Specialized execution environments. *SIGOPS Oper. Syst. Rev.*, 42(1), 2008.

[9] G. Diamos and S. Yalamanchili. Harmony: An execution model and runtime for heterogeneous many core systems. In *HPDC Hot Topics*.

[10] T. Dokken, T. R. Hagen, and J. M. Hjelmervik. The gpu as a high performance computational resource. In *SCCG*, 2005.

[11] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *SOSP*, 1995.

[12] B. Fischer and M. Scholes. The pricing of options and corporate liabilities. In *Journal of Political Economy*, 1973.

[13] A. Gavrilovska, S. Kumar, H. Raj, K. Schwan, et al. High-performance hypervisor architectures: Virtualization in hpc systems. In *HPCVirt*, 2007.

[14] A. Ghuloum, T. Smith, G. Wu, et al. Future proof data parallel algorithms and software on intel multi-core architecture. *Intel Technology Journal*, 11(4), December 2007.

[15] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, et al. Gvim: Gpu-accelerated virtual machines. In *HPCVirt*, 2009.

[16] Hewlett Packard. Netperf - users' manual.

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

[18] N. C. Hutchinson and L. L. Peterson. The x-kernel: An ar5chitecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1), 1991.

[19] IBM Corporation. Cell broadband engine programming handbook. `http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/`.

[20] IBM Corporation. The research hypervisor. `http://researchweb.watson.ibm.com/hypervisor`.

[21] IBM Corporation. SIMD math library specification for cell broadband engine architecture. `http://tinyurl.com/c2z4ze`.

[22] IBM Corporation. SPE management library. Part of Cell Broadband Engine SDK Documentation.

[23] IBM Corporation. Accelerated library framework for cell broadband engine programmer's guide and api reference. `http://tinyurl.com/c2z4ze`, October 2007.

[24] Intel Corporation. Enabling consistent platform-level services for tightly coupled accelerators. `http://tinyurl.com/quick-assist`.

[25] Intel Corporation. Intel ixp 2800 network processor hardware reference manual.

[26] Khronos OpenCL Working Group. The opencl specification, December 2008.

[27] M. Koehler. NP complete. *Embedded Systems Programming*, November 2000.

[28] R. Krishnamurthy, K. Schwan, R. West, and M.-C. Rosu. A network co-processor-based approach to scalable media streaming in servers. In *ICPP*, 2000.

[29] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara. Vmm-independent graphics acceleration. In *VEE*, 2007.

[30] e. a. Matt Adiletta. The next generation of intel ixp network processors. *Intel Technology Journal*, 6(3), Aug 2002.

[31] P. H. Michael Gschwind et al. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(2), March 2006.

[32] Netronome. Netronome flow driver - users' guide.

[33] E. B. Nightingale, O. Hodson, R. McIlroy, et al. Helios: heterogeneous multiprocessing with satellite kernels. In *SOSP*, 2009.

[34] NVIDIA Corporation. NVIDIA CUDA programming guide. `http://tinyurl.com/cudapghd`.

[35] R. Pike, D. Presotto, S. Dorward, et al. Plan 9 from Bell Labs. *Computing Systems*, 8(3), Summer 1995.

[36] H. Raj and K. Schwan. High performance and scalable i/o virtualization via self-virtualized devices. In *HPDC*, 2007.

[37] L. Seiler, D. Carmean, E. Sprangle, et al. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3), 2008.

[38] D. Silva, K. Schwan, and G. Eisenhauer. CTK: Configurable object abstractions for multiprocessors. *Software Engineering*, 27(6), 2001.

[39] J. Stratton, S. Stone, and W. mei Hwu. Mcuda: An efficient implementation of cuda kernels on multi-cores. Technical Report IMPACT-08-01, University of Illinois at Urbana-Champaign, March 2008.

[40] J. A. Turner. The los alamos roadrunner petascale hybrid supercomputer: Overview of applications, results, and programming, March 2008.

[41] E. Van Hensbergen. P.r.o.s.e.: partitioned reliable operating system environment. *SIGOPS Oper. Syst. Rev.*, 40(2), 2006.

[42] S. Williams, J. Shalf, L. Oliker, et al. The potential of the cell processor for scientific computing. In *CF*, Ischia, Italy, 2006.

[43] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Boston, MA, USA, 1999.