CHARM++ ON THE CELL PROCESSOR

BY

DAVID M. KUNZMAN

B.S., University of Toledo, 2002

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

Abstract

The Charm++ programming model has mainly been used to develop applications in the realm of High Performance Computing (HPC). As such, many Charm++ applications have tremendous need for computational power to perform their respective calculations/simulations in a reasonable amount of time. IBM, Sony, and Toshiba have recently developed the Cell Broadband Engine Architecture (CBEA or just *Cell* for short). Cell-based platforms are attractive platforms for Charm++ applications because of the enormous amount of computational power a single Cell processor can deliver. However, because the design of the CBEA differs from traditional processor architectures, the Cell is difficult to program using traditional programming methods and languages.

The work presented here represents the initial efforts in porting Charm++ applications to the Cell processor. An interface, called the Offload API, has been developed which allows Charm++ applications to *offload* work onto the Cell's SPEs. The Offload API has been designed to be an easy-to-use interface which will increase the productivity of programmers without sacrificing performance. By using the Offload API, Charm++ applications can utilize the SPEs. While the Offload API has been developed with the Charm++ programming model in mind, it can be used by any C/C++ application to *offload* work to the SPEs. Example programs which use the Offload API, both C/C++ based and Charm++ based, have been written and shown to correctly execute.

To my family and friends for all their love and support.

Acknowledgments

I would like to thank my advisor Prof. L. V. Kalé. He has provided me with guidance and support that has helped make this work possible. I'd also like to thank Gengbin Zheng and Eric Bohm for their help and advice with many aspects of this work. To all of my colleagues in the Parallel Programming Lab (PPL) who have been quite helpful, willing to share their time, knowledge, and insights. My colleagues in PPL have made my experience at UIUC both wonderful and unforgettable. I'd also like to acknowledge the people at IBM who have provided their knowledge and advice in regards to the Cell (especially Hema Reddy and Bob Szabo). And of course, my family and friends, without their support none of this would have been possible. Thank you to all.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Charm++ is an object-oriented message driven parallel programming model. Charm++ has mainly been used to develop High Performance Computing (HPC) application which require tremendous amounts of computational power. This need for computational power makes the Cell Broadband Engine Architecture (CBEA or *Cell* for short) very attractive for Charm++ applications. The Cell processor, jointly developed by IBM, Sony, and Toshiba, has enormous computational power. However, because the Cell's architecture varies so much from that of traditional processor architectures, it is also hard to program using traditional programming models. By applying the Charm++ programming model to the Cell and developing the Offload API for the Cell, the work presented here aims to remove many of the difficulties in programming for Cell-based platforms while still efficiently utilizing the processor.

1.1 Motivation and Goals

The Charm++ programming model [8] is mainly used in the realm of High Performance Computing (HPC) applications. This fact leads to the main motivation behind the work presented here. The main motivation of this work is to allow applications written using the Charm++ programming model to take advantage of the enormous processing power of the Cell processor. Results which demonstrate the power of the Cell processor in terms of application speedup include performance results for the Terrain Rendering Engine [11], calculating large FFTs [3], and in the area of Medical Imaging [12]. The performance speedups seen by these applications are encouraging and provide additional motivation to adapt the Charm++ Runtime System to Cell-based platforms so Charm++ applications may also benefit from the Cell. However, harnessing the power of the Cell is easier said than done. The architecture of the Cell processor [6] is fairly different from that of other commodity processors. This architectural difference is both the source of the Cell's computational power and the reason programming for the Cell is difficult. A programmer must manage certain resources, explicitly move data, perform synchronizations, and so on. This leads to the second motivation of this work, to create an easy-to-use interface that can be used by programmers to make writing applications for the Cell processor easier. A desire exists in the community to create a programming model that simplifies programming of the Cell processor. Previous work in creating programming models for the Cell include Cell Superscalar (CellSs) [1], Sequoia [5], the MultiCore Framework [2], and so on.

Another motivation behind the work presented here is the suitability of the Charm++ programming model to the Cell processor's architecture. Many features of the Charm++ programming model, such as data encapsulation, virtualization, and the ability to peek-ahead in the message queue (as described in Section 3.1.1), make Charm++ a good fit for the Cell processor.

The main goal of the work presented here is the creation of an interface which will allow the Charm++ Runtime System to utilize the SPEs. This interface, called the Offload API [9], is a set of libraries which allows a C/C++ application to utilize the SPEs. While this interface has been design with the Charm++ programming model in mind, it can be used by any C/C++ application. The design of this interface tries to incorporate the features of the Charm++ programming model that fit well with the Cell's architecture as described in Section 3.1.1. The bulk of the work performed so far, and thus presented here, has been the implementation of this interface. The Offload API should help make programming for the Cell easier while still maintaining the high levels of performance achievable on Cell. By removing the need for *Cell specific code* within the application code, using the Offload API can increase the productivity of programmers. The programmers are then free to focus on the application code itself rather than concerning themselves with the numerous details and complications of the Cell architecture.

The long-term direction of this work is to adapt the Charm++ Runtime System so that it can utilize the Cell processor. Because Charm++ has been used to implement several scientific HPC applications, the computational power of the Cell processor is quite attractive. Allowing Charm++ applications to utilize the Cell processor efficiently could potentially increase the performance of these already existing applications. Another long-term goal of this work is to allow Charm++ applications to utilize both Cell-based and non-Cell-based platforms with little to no code modification. Because there are already several Charm++ applications, it would be advantageous of the Charm++ tools and Charm++ Runtime System could automatically handle as many aspects of executing code on the SPEs as possible. This would allow Charm++ applications to be portable between Cell-based and non-Cell-based platforms.

1.2 Challenges for Parallelism on Cell

There are two types of parallelism that an application can exhibit. The first is *fine-grained parallelism*. Fine-grained parallelism refers to the parallelism present between a handful of instructions. That is to say, when there are multiple instructions within the pipeline that do not have any data dependencies, the instructions can be processed concurrently by the hardware. Because these instructions are operating on mutually exclusive sets of data, the instructions can be executed simultaneously by different functional units. Fine-grained parallelism can also refer to a specific class of instructions called SIMD (single instruction multiple data) instructions. SIMD instructions use *packed* data as input and produce *packed* data as output. Here, *packed* refers to the idea of having multiple values contained within one register, four 32-bit integers in a single 128-bit register, for example. This allows a functional unit to perform some operation on all four integers in parallel. This is particularly important to the Cell processor since the peak performance of an SPE is eight times greater when using SIMD instructions compared to when it is not using SIMD instructions.

The presence of fine-grained parallelism helps an application fully utilize a processor's pipeline. However, in recent years, many of the commodity processors are moving to architectures where, instead of enhancing the pipeline(s) in a single core, multiple cores are being placed on a single chip. Additionally, many scientific applications in use today utilize many processors in a single execution. To fully utilize multiple cores, an application needs to have large sections of code that can be executed by the various cores independently of each other. This is the second type of parallelism that an application can exhibit, called *coarse-grained parallelism*. This is the type of parallelism that this document will be addressing. Coarse-grained parallelism refers to multiple *threads of execution* that can be executed concurrently. Here, a thread of execution refers to a set of instructions that can be executed serially by a single processor core. Each of the cores executes a separate thread of execution. These threads of execution will typically execute hundreds, thousands, or more instructions before performing some type of synchronization or communication operation (a barrier, passing data between the threads, and so on). Applications that exhibit coarse-grained parallelism and execute on multiple processors and/or cores are typically referred to as *parallel applications* or *parallel programs*.

For an application to fully utilize the Cell processor (and, generally speaking, many of the recent processor architectures), it must exhibit both fine and coarse-grained parallelism. However, expressing an application/computation with various levels of parallelism can be quite difficult depending on the nature of the application/computation. This becomes even more difficult when performance considerations are taken into account. There are the same performance considerations that single-threaded applications have: load-store latencies, cache effects, optimizations such as software pipelining, etc. In addition to these considerations, there are also other considerations for parallel applications, such as concurrent access of data (which may require the addition of locks or other synchronization primitives), data access patterns and the resulting cache effects (data continuously being accessed by different processors and thus thrashing between multiple caches), communication costs between the different threads of execution, load-balancing work across the processors so some processors do not sit idle while other processors have too much work, and so on.

Debugging parallel applications can also be rather difficult. One of the main causes of this difficulty are *timing bugs*. Because there are multiple cores operating asynchronously to each other within the same application, the exact ordering of what happens in the application may change from execution to execution. Timing bugs refer to errors that may occur if the ordering of events within the application happens in just the right sequence to allow the bug to occur. Timing bugs tend to be quite elusive since only a small change to the program (adding a printf statement for debugging purposes) or the environment (the presence of debugger) can change the ordering of events within the application and cause the timing bug to disappear.

The architecture of the Cell processor further complicates these issues. The Cell processor has great potential for providing large amounts of computational power to applications [1] and thus is an attractive platform for many applications. However, the architecture of the Cell processor differs from that of conventional processors in many important regards (see Section 2.1). While these architectural differences are the source of the Cell's enormous computation power, it is also the reason that the Cell processor is considered both difficult to program and difficult to debug. Furthermore, performance tuning an application on the Cell processor can be quite tricky as there are several operations (multiple cores executing different code, data transferring between locations, etc.) that occur asynchronously. Timing these operations for optimal performance can prove to be quite difficult at times. Even small changes to code can potentially have large effects on the overall performance of the

[1] The SPEs alone can deliver 204.8 GFlop/s peak with a 3.2 GHz clock. In comparison, an Itanium2 can deliver a peak rate of 6.4 GFlop/s with a clock rate of 1.6 GHz.

application.

Another aspect of Cell architecture that further complicates issues is the requirement for *Cell specific code* within an application written for the Cell processor. When moving between platforms based on different processor architectures, many applications written in C/C++ can simply be recompiled and executed on the new platform. This is not the case for the Cell architecture. For an application to utilize the SPEs on the Cell, it must include Cell specific code to perform various tasks such as starting the SPE threads, taking care of moving data between system memory and the SPEs' Local Stores, synchronization, and so on (see Section 2.1).

Charm++ (see Section 2.2) has the potential to help solving some of these issues. Charm++ is a parallel programming model based on C/C++ that has generally been used to develop High Performance Computing (HPC) applications. In the Charm++ model, an application/computation is broken down into a set of objects that interact with each other by passing messages between one another. Because these objects, called *chares*, tend to be both small and *self-contained*, they are well suited to be passed to SPEs for *execution*. Furthermore, these objects are designed to be executed concurrently with dependencies being handled by the passing of messages between the objects. There are several features of the Charm++ programming model that make Charm++ a good fit for the Cell processor, including data encapsulation, virtualization, the ability to peek-ahead in the message queue, and so on. These features of the Charm++ programming model will help Charm++ applications more fully utilize the Cell processor. The way in which programs are writing in the Charm++ programming model will also help applications to be moved between Cell-based and non-Cell-based platforms with little to no modification of the application code. Further information on the suitability of Charm++ to the Cell processor can be found in Section 3.1.

Chapter 2

Background

2.1 The Cell Processor

2.1.1 Overview

The Cell Broadband Engine Architecture (CBEA), commonly referred to simply as "Cell", was jointly developed by IBM, Sony, and Toshiba. The design the Cell is different than that of other commodity processors. The Cell is a heterogeneous multicore chip. It has nine cores in total. There is one *main* core called the Power Processor Element (PPE). There are also eight additional cores called Synergistic Processor Elements (SPEs). The following paragraphs describe the architecture of the Cell, depicted in Figure 2.1, in greater detail.

2.1.2 Power Processor Element (PPE)

The PPE can be thought of as the *main* core. The PPE has a similar structure and operates in mainly the same way as a standard commodity processor with a single core. The PPE has a cache hierarchy through which data accessed by the pipeline must pass. The load and store operation issued by the pipeline access system memory just like in a standard commodity processor architecture. The PPE can handle two simultaneous hardware threads. These threads can access standard programming elements usually available to code running on commodity processors (C/C++ library methods, OS functionality, and so on).

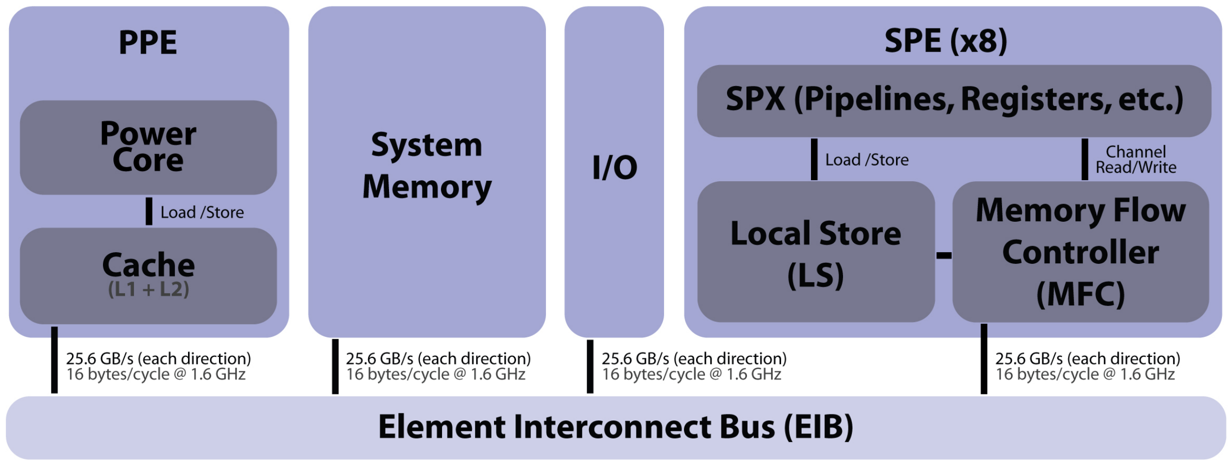

Figure 2.1: Architecture of the Cell Processor

2.1.3 Synergistic Processor Element (SPE)

The SPEs are specially designed to do large amounts of computation quickly. To this end, the SPEs have a different Instruction Set Architecture (ISA) from the PPE. The SPEs have greater support for SIMD instructions allowing more fine-grained parallelism to take place within the pipeline on the SPE. However, having a different ISA also means that code compiled for the PPE will not execute on an SPE and vice versa. Instead, any code that is to execute on an SPE must be compiled separately from the code that is to execute on the PPE. Once the SPE code has been compiled, the resulting SPE executable must be *embedded* within a PPE object file and linked to the PPE executable.

The SPEs do not have direct access to system memory like the PPE. Instead, each SPE has a dedicated Local Store (LS) memory. When the SPE's pipeline performs load/store operations, the load/store operations access data contained within the Local Store. The Local Store must contain all data (code, data, and stack) needed by the SPE. At 256 KB of total memory, fitting all of this into the Local Store can be difficult for most applications. Programmers, in many cases, will likely have to stream data through the Local Stores. The Local Stores are completely under the control of the programmer (for example, dereferencing NULL is a valid operation on the SPE and will not cause the program to segfault). Data must be moved explicitly between the Local Stores and system memory by using Direct Memory Access (DMA) transactions.

Also unlike the PPE, the SPEs do not have an Operating System (OS) executing on them. The OS operates on the PPE. When an application creates a thread of execution on one of the SPEs, that thread controls the SPE until either the user's application code running on the PPE or the SPE itself does something to relinquish control of the SPE.

Each SPE can issue up to two instructions per clock cycle. There are two pipelines, the even and the odd. Some instructions are directed to the odd pipeline (such as load/store operations) while other instructions are directed to the even pipeline (such as floating-point instructions).

2.1.4 Local Store (LS) and Memory Flow Controller (MFC)

Because of the limited size of the Local Store (256 KB), many applications may tend to treat the Local Store as a temporary holding place for data. Instead of keeping the data in the Local Store, data would be streamed through the Local Store. The SPE would perform some type of operation on this data and then direct it to its next location (either system memory or another SPE's Local Store).

The movement of data between the Local Stores and system memory is controlled by the Memory Flow Controller (MFC). Each SPE has an MFC. Both the owning SPE and the PPE can issue Direct Memory Access (DMA) commands to the MFC. These DMA commands instruct the MFC to move data between the Local Store and another location in the system (another Local Store, system memory, memory mapped I/O, and so on). These DMA commands are issued directly by the application code. The MFC moves the data asynchronously to code executing on any of the cores. The application code must later *check* to see if the DMA transaction has completed and then take the appropriate action. The DMA transactions are cache coherent with the PPE's caches.

2.1.5 Element Interconnect Bus (EIB)

The Element Interconnect Bus (EIB) is the on chip interconnect that connects all of the components together (PPE, SPEs, system memory, and I/O). At its peak, the EIB can transfer 200 GB/s. This provides tremendous on-chip bandwidth between the cores. The link connecting the EIB to the system memory (the off-chip link to system memory) can carry 25 GB/s.

2.2 Charm++

In the Charm++ Programming Model, the overall computation is broken up into a set of objects called *chares* (also sometimes referred to as *virtual processors*). The chares communicate with each other by asynchronously invoking special member functions called *entry methods* (also referred to as *sending messages between chares*). From the point of view of the calling chare, A, the code simply calls the member function on the receiving chare, B. This call returns immediately, with no return value, and chare A continues executing code. Sometime in the future, chare B will receive the message. Once the message arrives to the processor chare B is located on, the Charm++ Runtime System is free to begin execution of the entry method (member function) and passes the message's data in to the entry method via parameters. When a programmer is writing an application in the Charm++ model, they needgg not concern themselves with the processor's architecture, the number of processors available at runtime, the type or topology of the interconnect used, etc. Instead, the programmer defines the application in terms of the chares and allows the Charm++ Runtime System to map the chares to the processors available at runtime. As the application executes, the Charm++ Runtime System can *migrate* chare objects between processors. This allows the runtime to move chare objects off of heavily-loaded processors and on to other processors which are under-loaded. This processes is called *load-balancing*. Load-balancing increases the overall performance of the application by making better use of all the processors available. For reference purposes, a simple Charm++ program has been included in Appendix B.

Charm++ is based on C/C++ (that is, code is written in C/C++). The chare objects are C++ objects which are child classes of classes provided by Charm++. These parent classes provide the hooks needed by the Charm++ Runtime System. The functionality of the Charm++ Runtime System, and the other features of Charm++, is provided by linking the user's application code to one or more of the libraries make available through the Charm++ distribution. Charm++ is open source and freely available for download.

Traditionally, the Charm++ programming model has been used to create High Performance Computing (HPC) applications. The most popular application is the NAnoscale Molecular Dynamics (NAMD) application used to perform simulations of bio-molecules. Other applications include cosmology simulations, crack propagation simulations, rocket simulations, and so on. [1] Typically, Charm++ applications use anywhere from one to several thousands of processors for a single application execution.

2.2.1 Charm++ Runtime System

When a Charm++ application executes, it executes on top of the Charm++ Runtime System. The runtime takes care of many aspects of the application's execution automatically for the application. The runtime takes care of mapping chare objects to physical processors, send messages, routing messages, receiving messages, scheduling of entry methods, dynamic load-balancing of chare objects, and many other tasks needed by Charm++ applications. For more information on the Charm++ Runtime System, please consult the Charm++ Manual [2] [4].

[1]For more information regarding Charm++ and its applications, please visit *http://charm.cs.uiuc.edu/*

[2]The Charm++ Manual, along with other documentation on Charm++, can be found at the Charm++ website (*http://charm.cs.uiuc.edu/*).

2.2.2 Converse

Beneath a Charm++ application, there are several layers of software that handle various tasks and provide various services to the application, such as sending messages, scheduling entry methods, load-balancing, and so on. At the lowest level, for any given platform, there is a corresponding *Machine Layer* which has the appropriate code to interact directly with the platform/hardware. Each Machine Layer, at a minimum, is required to support a certain set of services (sending messages, for example). When the Charm++ Runtime System is compiled, the user is required to specify the Machine Layer which will be used. Converse [7] is a higher level layer which abstracts these features so that layers *above* Converse do not have to have to be concerned about which Machine Layer is in use. Instead, the higher level software calls into Converse which will either perform the required task and/or call into the Machine Layer to perform the task. In addition to being an abstraction for Machine Layers, Converse also provides some services of its own. One such services which is relevant to this work is the ability to create light-weight threads called *Converse Threads*. Like all threads, Converse Threads are used to execute code, can be suspended, and can later resume execution. For more information on Converse, please consult the Converse Programming Manual [10].

Chapter 3

Charm++ on Cell

3.1 Affinity of Charm++ to the Cell Processor

It has already been mentioned that Charm++ has traditionally been used for the development of High Performance Computing (HPC) applications. As such, when a Charm++ application is executed, it generally makes use of many processors at once. Generally speaking, to date, Charm++ applications use anywhere from one to several thousands of processors in a single application execution. Because of this, the adaptation of the Charm++ Runtime System must not only consider the case where a single Cell processor is being used, but rather, the more general case where the chare objects are spread out across many Cell processors.

3.1.1 Suitability of Charm++ to the Cell Architecture

There are several features of the Charm++ programming model that allow it to map well to the architecture of the Cell processor. These features include data encapsulation, virtualization, peek-ahead in the message queue, and so on.

Data Encapsulation

The first feature is data encapsulation. In this context, data encapsulation refers to the idea that the data accessed by an executing entry method is highly localized (spatially). In particular, entry methods tend to only access data that is located in the chare object itself and/or data located within the message that triggered the execution of the entry method.

Additionally, to increase the amount of parallelism within Charm++ applications (to allow for scaling to a greater number of processors) the chare objects tend to be somewhat small in size. The exact size of chare objects is highly dependent on the nature of the application itself. That said, a single entry method tends to take approximately tens to hundreds of microseconds to execute. The amount of data a chare contains tends to reflect this small execution time.

Data encapsulation is an important feature for Cell for two reasons: size of the LS and layout of data. Because the data needed by any single entry method has a high degree of spacial locality, the data is a good target for the DMA transactions that are required to move the data into and out of the SPEs' Local Stores. Having the data encapsulated within the chare objects also forces the programmer to consider both data location and data movement as the application executes. In this context, *data location* refers to the logical grouping of the data within the chare objects (not necessarily where the data is located within a particular processor's memory). *Data movement* refers to the passing of data between objects via messages.

Virtualization

The second feature is virtualization. Here, virtualization refers to the presence of many chares on each processor. By placing many chares on each processor, at any given moment, it is likely that there will be at least one chare with a message waiting for it (and thus, there is an entry method that needs to be executed). Additionally, while one entry method is executing, other chares that are waiting on messages to arrive will likely receive their messages. Once the entry method that is currently executing has completed, another message that has since arrived will immediately trigger the execution of another entry method. This effectively hides the latency of passing messages between processors by overlapping the execution of entry methods (useful computation) with the latency of the messages (communication overhead).

This form of virtualization has been important for the performance of Charm++ applications because of its ability to hide the latency of sending messages over the interconnect (also for hiding the latency of migrating objects between processors and data movement in general). It will also be important for Charm++ applications executing on the Cell processor by hiding the latency not only over the interconnect, but also by hiding the latency of the DMA transactions that are required to move data between the SPEs' Local Stores and system memory.

Message Queue Peek-Ahead

A third feature, which follows from the idea of virtualization, is the ability to peek-ahead in the message queue. As messages arrive to a processor, they are queued by the Charm++ Runtime System until the runtime system is ready to execute the entry method that the message is destined for. The runtime system can take advantage of this message queue by peeking ahead in the message queue at the messages that will need to be processed in the near future. It is *known* that this message needs to be processed in the near future and thus any pre-processing of the message will be useful. The message (data) has an associated chare object (data) and an associated entry method (code). The triplet (message:read-only data, chare:read/write data, entry method:code) can be used by the runtime system to push data and/or code down to the Local Store of an SPE while the same SPE is currently processing another entry method. This way, when the entry method that is currently executing on the SPE finishes, the data and/or code will already be located within the SPE's Local Store and the SPE can immediately start execution of the next entry method.

The ability to peek-ahead in the message queue is an important feature for the Cell's architecture because it allow the Charm++ Runtime System to accurately identify future work (and the associated data). This adds an element of predictability to the application's execution which is important for the Cell's architecture because of the need to DMA data and/or code into an SPE's Local Store before the SPE can begin processing the data. Furthermore, in the case of Charm++, the predictability is not speculative. Messages which have arrived to a processor for a particular chare object ***will*** need to be processed (execution of entry method).

3.1.2 Portability

Another area in which the Charm++ programming model may apply well to the Cell's architecture is portability of code. Current applications written using the Charm++ model are already portable between various platforms (including, but not limited to, IBM's Blue Gene/L, Cray's XT3, SGI's Altix, and clusters built using commodity hardware). The current approach (not including Cell-based platforms, for the moment) simply requires the end user to recompile the Charm++ Runtime System for the specific platform (creating the libraries needed by the Charm++ application) and then recompiling the Charm++ application itself. The same approach that is used to accomplish portability of code between various non-Cell-based platforms could be used for Cell-based platforms as well. When on a Cell-based platform, the Charm++ Runtime System would be compiled in such a way as to incorporate the mechanisms needed for *offloading* or *redirecting* entry method execution onto the SPEs. When the Charm++ application is being compiled, any entry methods which are *safe* (see next paragraph) to execute on the SPEs will be compiled twice, once for the PPE and once for the SPE. The Charm++ Runtime System can then decide dynamically during runtime where a particular entry method should be executed. For example, if it is determined that the execution time for a particular entry method is highly correlated to the size of the message that it receives (amount of data received), then the Charm++ Runtime System may execute the entry method on the PPE if a small message arrives or execute it on the SPE if a large message has been received.

However, Cell-based platforms have an additional constraints that is brought on by the Cell architecture itself. One constraint is that load/store instructions on an SPE access the SPE's Local Store and not system memory. This means that entry methods have to fit certain criteria (be *safe* to execute on SPEs) before they can be *offloaded* to the SPEs.

For instance, the entry methods cannot make arbitrary accesses into system memory if they are going to execute on the SPEs. Entry methods that need to make arbitrary access to system memory can still be executed on the PPE at the risk of reduced performance (since this increases the amount of code that ***must*** be executed on the PPE itself). Initially, this constraint will be addressed by using a keyword in the *.ci* file (a file used by the Charm++ tools during the application build process to identify chares and their entry methods). The user will identify which entry methods are *safe* to execute on the SPEs. On a non-Cell-based platform, this keyword would be ignored.

Another constraint that comes from the architecture of the Cell is the size of the SPEs' Local Stores. With each Local Store being 256 KB in size, there is only so much room in an SPE's Local Store to hold data. If a large message is received and directed to an entry method which is *safe* to execute on an SPE, the Charm++ Runtime System may still be forced to execute the entry method on the PPE if no SPE has enough remaining memory (after code and stack requirements) to accommodate the message.

3.2 Approach

3.2.1 Offloading Computation to the SPEs

Currently, all commodity processors on which Charm++ applications can execute have direct access from each of the cores to system memory. Because of this, chare objects are simply kept in main memory (belonging to one of the cores) and entry methods for that chare object are executed by the core *owning* the chare object. This approach is infeasible for Cell due to the limited size of the SPEs' Local Stores. With only 256 KB of memory, each Local Store would only be able to hold a very limited number of chare objects (i.e. - only a limited number of chare objects could reside on each SPE). Most of the chare objects would then reside on the PPE since the PPE has direct access to system memory and thus much more memory to hold more chare objects.

In the case of the Cell processor, the same streaming method that can be used to stream data through an SPE's Local Store can be applied to the chare objects. Instead of having multiple chare objects *reside* on each SPE, a chare object will be moved to the SPE temporarily while one of its entry methods executes. Once the entry method has finished executing, the chare object will then be moved back to system memory. All other tasks performed by the Charm++ Runtime system, such as network activity, file I/O, and so on, will be handled by the PPE.

There is an obvious optimization for this approach. When a single chare object has multiple messages waiting for it, the chare object can be moved into an SPE's Local Store to start with. Once in place, the SPE can then execute multiple entry methods (after receiving the required input messages) before finally moving the chare object back into main memory.

There are also some complications to this method. First, this may increase the latency for migrating chare objects. Consider the case where the PPE has already passed a chare object down to an SPE for execution of one of its entry methods. The SPE executes asynchronously to the PPE and thus the chare object's state may be dirty in the SPE's Local Store. Because of this, the Charm++ Runtime System has two choices. The first choice is to wait until the entry method has finished executing. Once the chare object has been copied back into system memory, the Charm++ Runtime System could then migrate the object to a new processor (and forward any messages that have since arrived for that chare object). A second option would be to migrate the chare object (and forward any messages in the message queue for it) immediately and simply disregard the updated chare object state received back from the SPE. This will require the message that was currently being processed to also be sent to the new processor and thus re-execution of the entry method the message was destined for. However, it would allow the Charm++ Runtime System to migrate the object immediately which might be advantageous (for example, if failure of the processor/node is imminent).

The same complication also presents itself in the case of the another message arriving to a processor for a chare object which is currently executing an entry method on one of the SPEs. The Charm++ Runtime System cannot simply create another copy of the chare object's state on another SPE and then execute the other entry method concurrently. The Charm++ Runtime System has to have some idea of where each chare object currently resides (in system memory or currently being streamed through one of the SPEs) and take action accordingly. In the case where the object is currently being streamed through one of the SPEs, the Charm++ Runtime System will potentially have to stall the processing of another message which has since arrived until the chare object has been passed back to system memory. An alternative would be to send the new message to the same SPE and to force the SPE to not copy the chare object to system memory until the second message has also been processed. However, this would be tricky since the SPEs (and, more specifically, the SPEs' MFCs) operate asynchronously to the PPE and, therefore, the object might already be in the process of being copied back into system memory when the new message arrives.

There are some special cases where it would be possible for the Charm++ Runtime System to execute multiple entry methods (each executing on different SPEs) for the same chare object. First, consider the case where the chare object does not contain any state (which is the case for compute objects in NAMD). Another example would be the case where one entry method, A, simply reads the data contained within the chare object while the other entry method, B, reads and/or writes the chare object's state. In this case, since the ordering of messages are not guaranteed in the Charm++ model, the result would be the same as if the entry methods were executed serially on the same core with entry method A being executed first followed by entry method B. This is possible since there would be two separate copies of the chare object's state (one copy in the Local Store of the SPE executing entry method A and another copy in the Local Store of the SPE executing entry method B). Since there are separate copies of the chare object's state, the writes performed by entry method B will not effect the execution of entry method A. Since entry method A would only read the chare object's state, the Work Request would not commit the unmodified state back to system memory (thus avoiding the race condition of which Work Request's output data, specifically the buffer containing the chare object's state, is written last; only the Work Request for entry method B updates the chare object's state in system memory).

3.2.2 Compiling Charm++ Applications

Charm++ applications are written in C++. The Charm++ distribution contains tools that do pre-processing of the user's application code to generate the necessary *hooks* needed by the Charm++ Runtime System. In addition to the standard *.h* and *.cpp* files used by C++ applications, Charm++ also requires a *.ci* file. The contents of the *.ci* file basically describes what chare objects exist (not by instances of the chares, the contents provide information about the classes themselves). For instance, the *.ci* file for a chare class will list all of the member functions of the class that function as entry methods. For an example of a *.ci* file, please refer to the example Charm++ program in Appendix B.

3.2.3 Current Status of Charm++ on Cell

For a Charm++ application to *offload* work onto the SPEs, a few changes must be made to it. First, any entry method that should be offloaded to an SPE should be marked as *[threaded]* in the corresponding *.ci* file. Marking an entry method as threaded causes the Charm++ Runtime System to create a light-weight Converse Thread which will execute the entry method.

The entry method must then call the Offload API directly (the Offload API is discussed more in Chapter 4) to create a Work Request. When issuing the Work Request, no callback method is to be specified and the user data should be the ID of the Converse Thread. When the Charm++ Runtime System initializes on a Cell-based platform, it initializes the Offload API and installs a default callback function. The default callback function will awaken the Converse Thread that created the Work Request which triggered the callback. After calling into the Offload API to send the Work Request, the entry method must then suspend itself by calling CthSuspend(). Once the Work Request that was issued by this entry method completes, the entry method (the Converse Thread) will be awoken by the default callback function and execution of the entry method will continue on the PPE. Any out-bound messages created by the entry method should be sent in the PPE portion of the entry method since the SPEs do not currently have a mechanism for creating and sending messages to other cores.

Example Charm++ applications that are written in this manner have already been created. These example programs are located within the Charm++ distribution. [1]

[1]The example applications are located in the directory *[charmDir]/examples/charm++/cell/* where *[charmDir]* is the base directory of the Charm++ distribution.

Chapter 4

The Offload API

The Offload API has been developed to allow the Charm++ Runtime System to utilize the SPEs. The Offload API has been structured with the Charm++ programming model in mind. Similar to the way Charm++ decomposes an application into a set of chares, a programmer using the Offload API will decompose the program (fully or partially) into *Work Requests* (more on Work Requests later). However, it has also been developed in such a way as to be independent of the Charm++ Runtime System. That is to say, any C/C++ application can use the Offload API independently of Charm++.

There are two goals for the Offload API. The first goal is to allow the Charm++ Runtime System to utilize the SPEs in a natural manner. As was already discussed, the entry methods of the chare objects will be offloaded to the SPEs. To accomplish this, the API should be structured in a manner that allows the chare, the incoming message, and any out-going messages to be easily passed to and from the SPEs along with specifying the code that needs to execute (in this case, which entry method). There is a high degree of spatial locality for the data involved (self-contained chare object and self-contained message(s)) and the Offload API should be able to take advantage of this.

The second goal is to create an efficient API that will allow C/C++ applications to take advantage of the computational power of the SPEs in an easy-to-use manner. One of the difficulties that arises from the Cell architecture is the ability to program an application in a straight forward manner. The application programmer must write a good deal of Cell specific code to accomplish this (DMA transactions, double-buffering code from DMAs, and so on). It is also important not to severely confine the programmer to this model. While

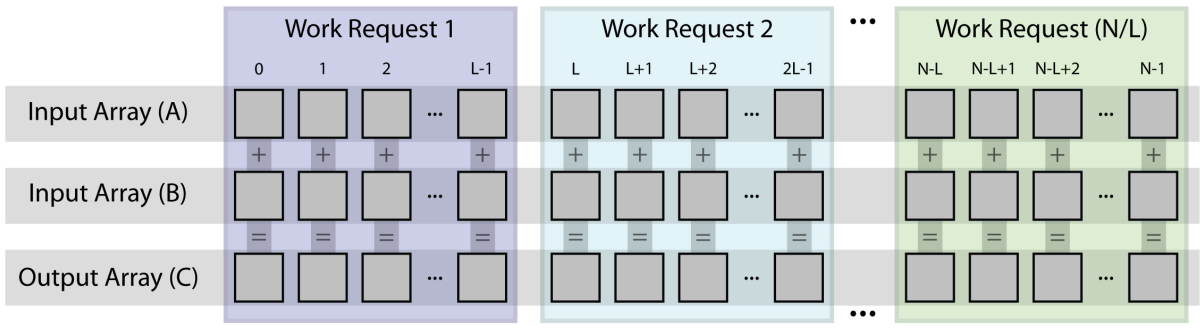

Figure 4.1: Work Request Example
Example breakdown of computation into a set of Work Requests. Two arrays of floats, A and B, are added together with the results being stored in another array of floats, C. Each Work Request is responsible for L of the operations. Appendix A contains the source code for a C/C++ application which performs this calculation using the Offload API.

the Offload API does provide a general model in which a program should be written, it does not stop the programmer from doing Cell specific programming. For example, within a Work Request, a programmer can still issue and wait for DMA requests, use all of the SPE intrinsics provided with the Cell SDK, etc. This allows the Offload API to be used to create a program but does not hinder the programmer from making optimizations to the program later.

4.1 Application Decomposition

When using the Offload API, the application is broken down into subcomputations. These subcomputations are referred to as *Work Requests*. For example, consider a simple application that simply adds the contents of two integer arrays of length N (see Figure 4.1). A Work Request may be defined as a portion of these additions. That is, one Work Request may add elements 0 through $L-1$, while another Work Request may add elements L through $2L-1$, and so on.

There are two key points to keep in mind when defining each Work Request. To illustrate these points, the addition of two arrays example given above will be used. First, it is the Work

Requests that allow for coarse-grained parallelism within the application. In the example above, the choice of Work Request definition (what work will be performed by each Work Request) allows for all of the Work Requests to execute concurrently. That is, no Work Requests share input or output data in such a way as to create data dependencies or race conditions between any of the Work Requests. This is trivial in the example given, however, in a real application, maximizing the number of concurrent Work Requests can play a vital role in the amount of parallelism present in the application. Second, the input and output data is clearly defined. In this example, each Work Request needs consecutive elements from the three arrays. In this particular case, because the elements of the arrays needed by any given Work Request are in contiguous memory, array sections lend themselves well to DMA transactions. This may not be the case for all applications. Consider applications that traverse tree like data structures where many memory access are made in order to perform a particular calculation.

Another aspect to consider, for performance reasons, is the cost of moving the data (transfer overhead) vs. the amount of computation performed. The Offload API will also introduce a certain amount of overhead (API overhead). The transfer overhead can be hidden by overlapping it with useful computation. However, the API overhead requires the use of one of the processor cores and thus cannot be hidden by overlapping it with useful computation (at least not on the same processor core). Thus, it is important that the API overhead is minimized as much as possible. The transfer overhead, however, simply needs to be small enough that the computation can mask its presence.

4.1.1 Structure of a Work Request

The structure of a Work Request is fairly simple (see Figure 4.2). A Work Request is comprised of input and output buffers and a function to execute. The function that will be executed is written by the user and assigned a unique *Function Index* by the user. When the user issues the Work Request to the Offload API, this Function Index is passed along with the

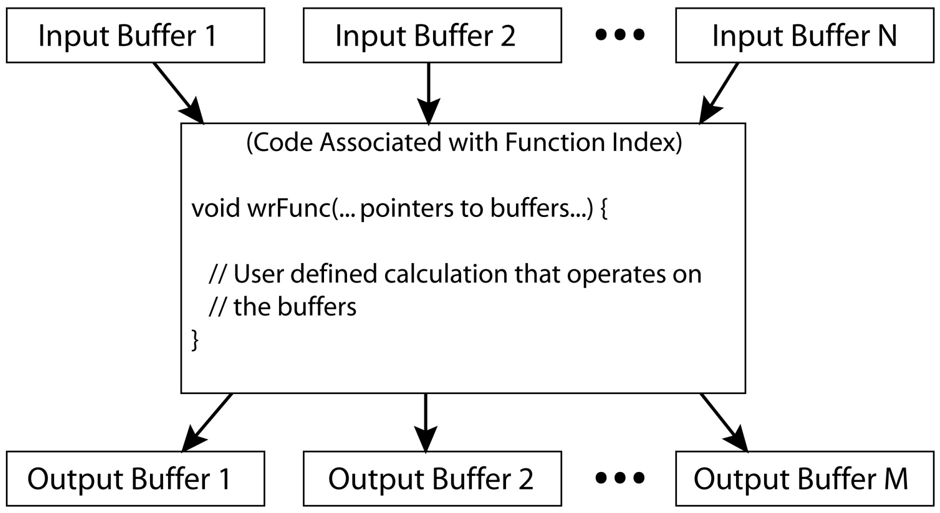
Input Buffer 1
Input Buffer 2
•••
Input Buffer N
(Code Associated with Function Index)
void wrFunc(... pointers to buffers...) {
// User defined calculation that operates on
// the buffers
}
Output Buffer 1
Output Buffer 2
•••
Output Buffer M

Figure 4.2: Structure of a Work Request

Work Request. The user is also required to write a *funcLookup()* (function lookup) function. The purpose of this function is to map Function Indexes to the actual functions within the SPE code. The funcLookup() function is basically a C/C++ *switch* statement which uses the Function Index as the input. The individual *cases* then make the corresponding function call. The funcLookup() function is used as the *entry point* into the user's code by the SPE Runtime. For reference, the code in Appendix A contains an example funcLookup() function.

When issuing a Work Request to the Offload API, the programmer must also specify the input and output data buffers that are to be transferred to and from the Local Store of the SPE where the Work Request will be executed. There are three types of buffers that can be used with a Work Request (see Table 4.1), including read/write, read-only, and write-only. The contents of read/write buffers are transferred to the SPE's Local Store prior to the execution of the Work Request and transferred back to system memory after the execution of the Work Request. For read-only data buffers, the contents of the buffer are only transferred to the SPE's Local Store prior to the execution of the Work Request. While the Work Request can still modify the contents of these buffers, the contents are discarded after execution of the Work Request completes (not transferred back to system memory).

Buffer Type	Description
Read/Write	The buffer contents are transferred before and after execution of the Work Request.
Read-Only	The buffer contents are only transferred before the execution of the Work Request.
Write-Only	The buffer contents are only transferred after the execution of the Work Request.

Table 4.1: Work Requests: Data Buffer Types

Write-only data buffers are only copied out of the SPE's Local Store after the execution of the Work Request. Prior to the user code issuing a Work Request, buffers should be set aside in system memory to be targets of all the write-only buffers associated with that Work Request. The SPE Runtime will set aside memory in the SPE's Local Store prior to executing the Work Request which can be written to during the execution of the Work Request. The initial contents of a write-only buffer are undefined (or, optionally the SPE Runtime will zero the memory prior to Work Request execution). Once the Work Request has finished executing, the contents of the write-only buffer will be transferred to the specified location in system memory. The purpose of having these different data buffer types is to reduce the amount of data that needs to be transferred over the EIB for any given Work Request.

4.1.2 Types of Work Requests

There are two types of Work Requests supported by the Offload API. The first is a *scatter/-gather* Work Request. This type of Work Request takes advantage of the DMA list capability of the Cell architecture. For this type of Work Request, there can be numerous input and output buffers of each type. The second type of Work Request is just a simplification of the first type, a *standard* Work Request. The idea behind the standard Work Request type is to alleviate the programmer from having to create a list of buffers. Instead, the standard Work Request type is limited to have only a single buffer of each type simply passed as parameters to the function call (though, flags passed to the sendWorkRequest() function can be used to have two buffers of one type at the expense of not having any buffers of another type, i.e., make the read/write buffer behave as a read-only buffer so the Work Request has two read-only buffers and zero read/write buffers).

4.2 Work Request Groups

Work Requests that are issued to the Offload API can also be collected together into groups called *Work Request Groups*. This allows Work Requests to be logically grouped together. By grouping Work Requests, the application code can be notified once when all of the Work Requests in the group have completed instead of being notified when each individual Work Request completes. As an example usage, consider the arrays being added in Figure 4.1 from Section 4.1. It may only be important for the application to *know* when all elements in the arrays have been added together instead of *knowing* when each of the corresponding sections of the arrays have been added. In this case, each of the N/L Work Requests could be added to the same Work Request Group. The application would then be notified only once, after the entire array C has been computed. The example code in Appendix A does exactly this.

To create a Work Request Group, first, the user's application code makes a call into the Offload API to create the Work Request Group. Once the group has been created, the Offload API will pass back a WRGroupHandle. This WRGroupHandle is later used by the application code to add Work Requests to the Work Request Group. As each Work Request is issued to the Offload API, if the optional WRGroupHandle is specified, the Work Request being issued is added to the Work Request Group associated with the WRGroupHandle provided. Once all Work Requests have been added to the Work Request Group, another call is made to the Offload API which indicates to the API that no more Work Requests will be added to the Work Request Group. The example code in Appendix A illustrates the usage of Work Request Groups.

Work Request Groups are constructed in this manner so the initial Work Request that are added to the group can immediately issue to the SPEs with all the required information. If Work Requests were created and then later added to an existing Work Request Group, there would be a race condition where a Work Request could potentially finish before the application code had a chance to add the Work Request to the desired Work Request Group. If a Work Request finishes and has a callback function, the WRHandle associated with the Work Request is immediately recycled by the Offload API. By specifying the Work Request Group a Work Request belongs to upon creation of the Work Request, the Work Request is free to finish as soon as possible. Once finished, the WRGroupHandle associated with the Work Request is used to update the state of the Work Request Group the Work Request belongs to and the WRHandle can immediately be recycled by the Offload API.

After all of the Work Requests in the Work Request Group have been completed, the application code is notified via the one of the same methods that are available for individual Work Requests. One method is to specify a callback function. The application code can either install a default callback function to be used for all Work Request Groups and/or each individual Work Request Group can have a specific callback function specified. The other method, if no callback function is specified, requires the user to use the returned WRGroupHandle to check whether or not the Work Request Group has completed. Like Work Requests, there is a blocking Offload API function which will not return until the all Work Requests in the Work Request Group associated with the specified WRGroupHandle have completed. There is also an Offload API function that will simply poll the Work Request Group to check if it has finished or not, returning the result to the caller immediately (i.e. - does not block).

4.3 Work Request Tracing

To help with debugging, the Offload API also has a mechanism that allows a Work Request to be marked for *tracing*. As a Work Request which has been marked for tracing passes through the Offload API and the SPE Runtime, various pieces of information about that Work Request are output. This information includes which states the Work Request passes through, pieces of information about the Work Request as it is in each state (such as the number of buffers associated with the Work Request), and so on. Additionally, the user's code on the SPE can also check to see whether or not the currently executing Work Request has been marked for tracing. If so, the user's code can take the appropriate action. The ability to trace all Work Requests, trace a set of Work Requests, and to only trace a single Work Request has been useful in both the development of sample programs and debugging of the Offload API itself.

4.4 Definition of Interface

The Offload API interface is fairly straight forward. The description in Table 4.2 is provided as a reference for the reader. [1] A more detailed description can be found within the doxygen documentation on the Charm++ website. [2] This doxygen generated documentation can also be generated directly from the Charm++ distribution. [3]

4.5 Structure of the Offload API

The Offload API is contained within two libraries. One library contains the interface that is linked to and used by the user PPE application code (the interface described in Table 4.2).

[1]The Offload API is still under active development and thus, in the future, may change from the description given in this document.

[2]Available at http://charm.cs.uiuc.edu/doxygen/charm/group__OffloadAPI.html

[3]Available at http://charm.cs.uiuc.edu/download/

Functions for Offload API Control

- InitOffloadAPI() - Initializes the Offload API. Once called, the API is ready to accept Work Requests.
- CloseOffloadAPI() - Closes the Offload API and frees any resources associated with the API.
- OffloadAPIProgress() - Causes the Offload API to make progress (check for completed Work Requests, issue pending Work Requests, etc.)

Functions for Issuing Work Requests

- sendWorkRequest() - Issues a single *standard* Work Request. Returns a WRHandle.
- sendWorkRequest_list() - Issues a single *scatter/gather* Work Request. Returns a WRHandle.
- isFinished() - Indicates whether or not the Work Request associated with the provided WRHandle has completed.
- waitForWRHandle() - Once called, this function does not return until the Work Request associated with the provided WRHandle has completed.

Functions for Work Request Groups

- createWRGroup() - Creates a Work Request Group returning an WRGroupHandle. The WRGroupHandle can be used when issuing Work Requests to associate the individual Work Requests with the Work Request Group.
- completeWRGroup() - Completes the Work Request Group (lets the Offload API know that no more Work Requests will be added to this Work Request Group).
- isWRGroupFinished() - Indicates whether or not the Work Request Group associated with the provided WRGroupHandle has completed.
- waitForWRGroupHandle() - Once called, this function does not return until the Work Request Group associated with the provided WRGroupHandle has completed.

Table 4.2: Offload API Description

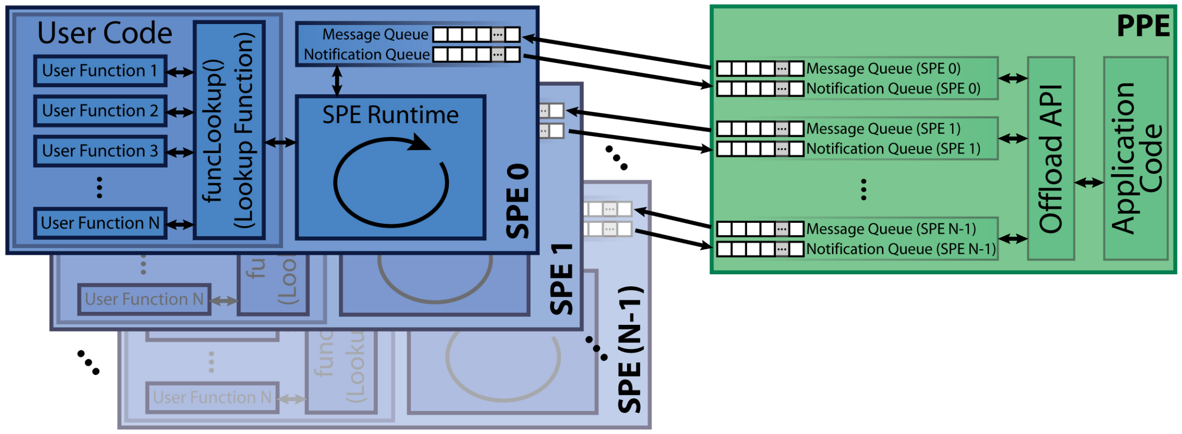

Figure 4.3: Interaction of Offload API, SPE Runtime, and User Code

The second library, which is linked to the SPE user's SPE code, contains the SPE Runtime.

Each SPE has a small runtime that is continuously running called the *SPE Runtime*. The API, located on the PPE, is responsible for receiving Work Requests issued by the user's code, deciding which SPE should service the Work Request, and then issuing the Work Request down to one of the SPEs. It then falls upon the SPE Runtime to take care of everything else related to the Work Request (except for the final notification to the user's code indicating that the Work Request has completed which is also handled by the PPE). The SPE Runtime is responsible for moving all data needed/produced by a Work Request, setting aside memory in the SPE's Local Store for the Work Request, executing the Work Request, notifying the PPE that the Work Request has completed, and so on.

The general structure of the Offload API, the SPE Runtime, and how the two interact is depicted in Figure 4.3. As can be seen in the figure, the application code executing on the PPE interacts directly with the Offload API. For each SPE, the Offload API maintains two queues, the *Message Queue* [4] and the *Notification Queue*. The Message Queue is a one-way queue leading from the PPE to the SPE. That is to say, the PPE is the only producer writing data into the queue and the corresponding SPE is the only consumer reading data from the

[4]Here, the term *queue* is a bit of a misnomer. The entries in this *queue* can be processed by the SPE Runtime in any order as decided by the SPE Runtime. The term *Message Queue* is used because of the strong correlation between this *queue* and the Message Queue in the Charm++ Runtime System.

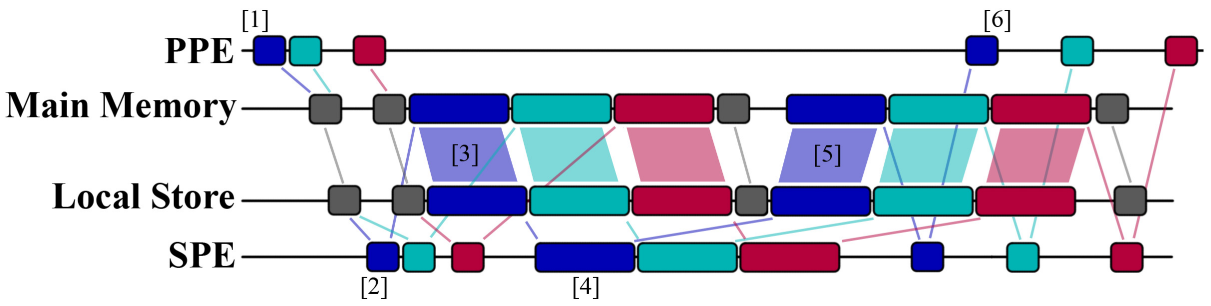

Figure 4.4: Work Request Flow

queue. The Message Queue is used by the PPE to pass Work Requests to the SPE. Each queue entry (a single Work Request) is processed/executed by the SPE Runtime. Once the Work Request has finished, the SPE Runtime notifies the PPE that the Work Request has completed via the corresponding entry in the Notification Queue. Like the Message Queue, the Notification Queue is a one-way queue leading from the SPE to the PPE. Once the PPE has been notified that the Work Request has completed, the Offload API will notify the user's application code of the completion of the Work Request.

Each of the SPEs has an SPE Runtime which is responsible for processing the Work Requests received via the Message Queue. The SPE Runtime continuously loops through the Message Queue entries doing any processing needed. If the entry was previously empty, the SPE Runtime checks the queue entry for a new Work Request. If a new Work Request is found, or the queue entry already contains a Work Request, the SPE Runtime does any processing needed by the Work Request. Once a Work Request is ready to execute, the SPE Runtime calls into the user's code via the funcLookup() function which is also provided by the user (as discussed in Section 4.1.1). The funcLookup() function then calls the appropriate user defined function thus *executing* the Work Request.

4.5.1 The Life of a Work Request

Figure 4.4 depicts the *life* of a Work Request. *[1]* The user code on the PPE issues a Work Request to the Offload API. The Offload API then decides which SPE the Work Request should execute on and passes the Work Request to that SPE via that SPE's Message Queue. *[2]* Periodically, the SPE Runtime checks each of the Message Queue entries. The SPE Runtime will find the new Work Request, allocate memory for it, and then issue a DMA-Get to the SPE's MFC to bring the input data needed by the Work Request into the SPE's Local Store. *[3]* The SPE's MFC moves the input data asynchronously. The SPE is free to do other work (process/execute another Work Request) while this Work Request's input data is being moved into the Local Store. *[4]* Once the data has arrived in the SPE's Local Store, the SPE Runtime is free to execute the Work Request. Once the Work Request has finished executing, the SPE Runtime issues a DMA-Put to place any output data into system memory. *[5]* The output data is moved asynchronously by the MFC. Once again, the SPE Runtime is free to do other work while this is taking place. Once this DMA-Put has completed, the SPE then notifies the PPE that the Work Request has finished and the output data is in system memory. *[6]* The PPE then notifies the user's application code that the Work Request has completed.

There are a few things in Figure 4.4 that should be pointed out. First, the SPE Runtime takes care of most activities related to the Work Requests passed to an SPE. Because of this, the PPE is able to do other work (as defined by the application code) while the SPE is executing the Work Request(s). This is also important because the SPEs run at the same clock speed as the PPE. If the PPE were to do a significant portion of the processing needed by each Work Request (issue DMAs, etc.) then the PPE would quickly become saturated and thus a bottleneck for the system. Second, the movement of data (input and output data for the Work Requests) can be overlapped with useful computation (processing/execution of other Work Requests). Third, the image itself is not to scale. Ideally, one would want the overhead of the data transfers (*[3]* + *[5]*) to be less than the time spent executing the Work Request (*[4]*).

The processing of each Work Request by the SPE Runtime is controlled by passing the Work Request through a small state machine (see Figure 4.5). As the Work Request is

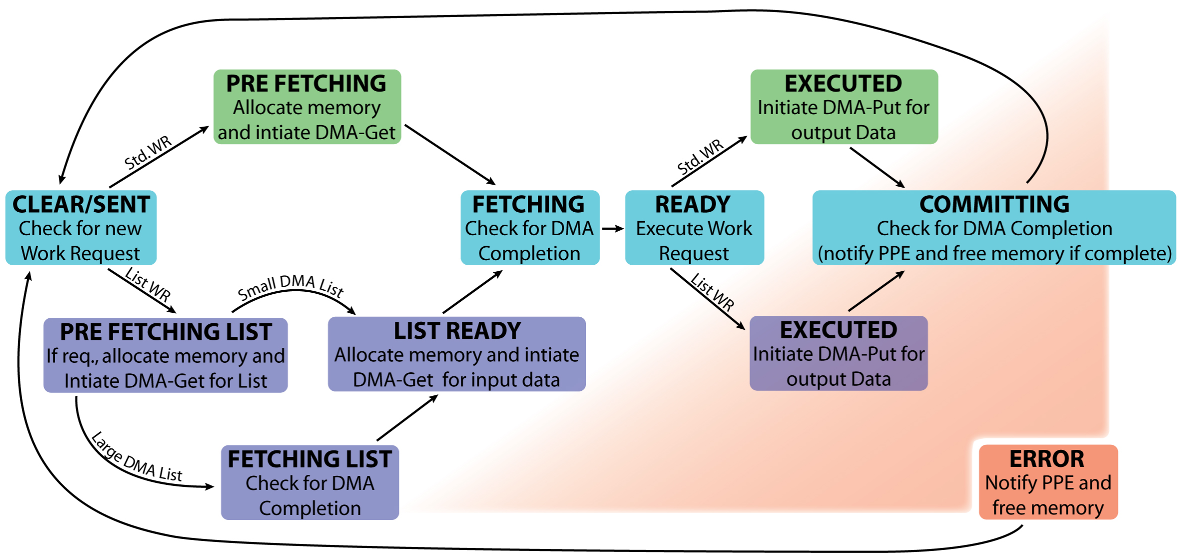
PRE FETCHING
Allocate memory and intiate DMA-Get
EXECUTED
Initiate DMA-Put for output Data
Std.WR
CLEAR/SENT
Check for new Work Request
FETCHING
Check for DMA Completion
READY
Execute Work Request
Std.WR
COMMITTING
Check for DMA Completion (notify PPE and free memory if complete)
List WR
Small DMA List
List WR
PRE FETCHING LIST
If req., allocate memory and Intiate DMA-Get for List
LIST READY
Allocate memory and intiate DMA-Get for input data
EXECUTED
Initiate DMA-Put for output Data
Large DMA List
FETCHING LIST
Check for DMA Completion
ERROR
Notify PPE and free memory

Figure 4.5: Work Request State Machine
Note: If a recoverable error occurs, such as not being able to allocate memory in the SPE's Local Store, the Message Queue entry will remain in the same state. Only if an unrecoverable error occurs, not enough memory in the Local Store to service a Work Request, for example, will a Message Queue entry transition into the ERROR state.

processed, it passes through a series of states. To be clear, when a Work Request is issued to one of the SPEs, it is placed in (associated with) one of the Message Queue entries for that SPE. While a Work Request is on an SPE, the terms *Message Queue entry* and *Work Request* can be used interchangeably, however, they are not the same thing as is described in the next paragraph. The exact path the Work Request takes through the state machine is dependent on the type of Work Request (standard or scatter/gather) and, if it is a scatter/gather Work Request, the size of the DMA List (i.e. - the number of buffers associated with the Work Request). When the SPE Runtime processes a Work Request, it executes a chunk of code corresponding to the state the Work Request is in. For example, if a Work Request is in the FETCHING state, the SPE Runtime will execute a chunk of code that checks to see if the DMA-Get for the input data has completed. If so, the Work Request will transition into the READY state. Otherwise, the Work Request will remain in the FETCHING state and the SPE Runtime will move on to the next Work Request (Message Queue entry). For some states, the SPE Runtime will continue processing the Work Request. For example, consider the case where a Message Queue entry is initially in the CLEAR state. If a standard Work Request is passed to the SPE via this Message Queue entry, the next time the SPE Runtime processes that Message Queue entry, the SPE Runtime will transition the Message Queue entry (Work Request) into the PRE FETCHING state and then continue processing the same Message Queue entry (Work Request). Then, assuming that the SPE Runtime was able to issue the DMA transaction, the SPE will transition the Message Queue entry into the FETCHING state. At this point, however, the SPE Runtime will no longer continue working on this Message Queue entry since it is unlikely that processing the FETCHING state (checking for completion of the DMA-Get) will succeed since the DMA-Get was just issued to the MFC. Instead, the SPE Runtime will move on to the next Message Queue Entry (Work Request).

With the previous examples in mind (along with Figure 4.5), it would be advantageous to point out the difference between a *Work Request* and a *Message Queue entry.* While, in most cases they can be thought of as the same thing, they are not. When a Work Request is issued to the SPE by the PPE, it is placed in one of the Message Queue's entries. It is actually the Message Queue entry that passes through the state machine on the SPE Runtime. The Message Queue entry, when it is empty (has no associated Work Request), is in the CLEAR state which indicates to the PPE that the PPE can place a Work Request into that Message Queue entry. When a Message Queue entry has an associated Work Request, it is in one of the other states (FETCHING, READY, COMMITTING, and so on). It does not make logical sense for a Work Request to be in the CLEAR state. Additionally, the states as shown in Figure 4.5 are only relevant when talking about a Work Request being processed by the SPE Runtime. This represents only a portion of a Work Request's life span as Work Requests also exist on the PPE.

Table 4.3 contains more information on the individual states along with some preliminary timing information for processing a Message Queue entry in a given state. The times presented represent the amount of time it takes to process a Message Queue entry which is in the given state. For the FETCHING and COMMITTING states, the timings listed represent the amount of time taken to check for DMA completion, not the amount of time taken to transfer the data. A Message Queue entry may stay in a particular state if the conditions to move onto the next state are not satisfied. Thus, the timings listed may occur multiple times for a given Message Queue entry. The function that processes a Message Queue entry in the FETCHING state, for example, may be called multiple times on the same Message Queue entry if the data had not arrived by the first time the function was called.

4.6 Benefits of the Offload API

4.6.1 Ease-of-Programming

One of the motivations behind the development of the Offload API is to create an easy-to-use interface which programmers can use to write programs for the Cell processor. Appendix A contains an example C/C++ application written using the Offload API. For brevity, no programs that use the Cell SDK directly are included in this document. However, there are numerous example available for comparison, including examples which are include within the Cell SDK [5] itself. One can see that the example program contains no Cell specific code. With the exception of the of the *funcLookup()* function and the calls into the Offload API, all of the code is specific to the application itself. This allows the programmer to focus their efforts on writing application code instead of Cell specific code and thus increasing the productivity of the programmer. Also note that, with a simple modification to the code, when not on a Cell-based platform, the calls to *sendWorkRequest()* (on the PPE) could be replaced by calls directly to *add()* (in the SPE code) making porting of this code from a Cell-based platform to a non-Cell-based platform fairly straight forward.

[5]The Cell SDK is available from IBM's *Cell Broadband Engine Resource Center* located at *http://www-128.ibm.com/developerworks/power/cell/downloads.html.*

State	Function Time (spuDC[1])	Description
CLEAR/SENT[5]	0.22 (no WR found) 0.86 (WR found)	There is no Work Request in this Work Request Queue slot.
PRE FETCHING (Std.)	9.28	A standard Work Request has arrived in this slot. Allocate memory and initiate DMA transaction to bring in the Work Request's input data.
PRE FETCHING LIST (List)	0.47[2] (Sm List[4]) — (Lg List[4])	If the DMA List is large[4], issue a DMA-Get to retrieve the DMA List for this Work Request from system memory. Otherwise, copy small[4] DMA List directly from Message Queue (skipping DMA-Get for DMA List).
FETCHING LIST (List)	—	DMA-Get has been issued for the large DMA List. Wait for the DMA-Get to complete.
LIST READY (List)	22.44[2] (Sm List[4]) — (Lg List[4])	DMA List is ready. Issue DMA-Put to get input data for the Work Request.
FETCHING	0.63	DMA-Get for data has been issued. Wait for the DMA-Get to complete.
READY	+0.99[3]	Input data has arrived and the Work Request is ready to execute (the Work Request is executed here)
EXECUTED (Std.)	2.27	The standard Work Request has been executed. Initiate the DMA-Put that will place output data into system memory.
EXECUTED (List)	10.65[2] (Sm List[4]) — (Lg List[4])	The list Work Request has been executed. Initiate the DMA-Put that will place the output data into system memory.
COMMITTING	1.13	The DMA-Put has been issued. Wait for the DMA-Put to complete.
ERROR	—	An error has occurred during the processing of the Work Request. Notify the PPE and clean up SPE Runtime data structures.

—:	Currently no data.
[1]:	In units of SPU Decrementer Cycles (approx. 70ns). (These are only preliminary results with very few code optimizations in the SPE Runtime code.)
[2]:	Time depends on length of DMA List (more time for longer lists). Numbers based on DMA Lists with 129 entries (128 read-only + 1 write-only).
[3]:	The time listed is in addition to the time required by application code.
[4]:	After reaching a certain threshold, the size of a DMA list becomes *large*. Small lists use a preallocated buffer (in the SPE's Local Store) to hold the list while large lists need to have memory allocated in the SPE's *heap*.
[5]:	The CLEAR/SENT state is special in that the transition of a Message Queue entry from CLEAR to SENT is controlled by the PPE (it is how the PPE triggers the SPE Runtime to start processing a Work Request).

Table 4.3: Work Request States

4.6.2 Automatic Management of the SPE's Local Store

Another advantage to using the Offload API is that the SPE Runtime will automatically manage the memory in the SPEs' Local Stores for the programmer. The programmer does not have to be concerned with setting aside memory in the LS. Instead, the programmer simply indicates which buffers in system memory are associated with the Work Request when the Work Request is created on the PPE. The Offload API will take care of finding a location in the SPE's Local Store for the data. As far as the programmer is concerned, a pointer to the buffer is simply passed to the Work Requests associated function on the SPE which can be used directly by the user's SPE code. Please refer to the Offload API code example in Appendix A. The same holds true for output data. Once the user's PPE code is notified that a Work Request has completed, the data is present in system memory and ready to be used.

4.6.3 Automatic Handling of DMA Transactions

Similar to how the Offload API takes care of managing the SPE's Local Store for the programmer, the Offload API will also issue the required DMA commands for Work Requests. The programmer does not have to write any code to issue DMA commands to the MFC, double/triple buffer the data for performance reasons, or have any code that checks for DMA completions (and has the ability to do some other work if the DMA has not completed so the SPE does not go idle). When using the Offload API, the programmer simply defines the work that is to be done (the Work Request functions in the SPE code) and then issues Work Requests to the Offload API with the appropriate buffers. All data movement is handled by the Offload API on behalf of the user.

4.6.4 Automatic Overlap of Data Movement with Computation

As can be seen in Figure 4.4, the movement of data between main memory and the an SPE's Local Store is automatically overlapped by the SPE Runtime. As Work Requests arrive to the SPE Runtime, the SPE Runtime starts issuing DMA commands to bring input buffers for the Work Requests into the SPE's Local Store. Once the DMA commands have been issued to the MFC, the SPE Runtime begins execution of any Work Requests that already have all of their input data in the Local Store. Once execution of one or more Work Requests completes, data from the previously issued DMA commands will have completed (as long as the execution time of the Work Request is sufficiently long enough to fully encompass the data transfer time). This frees the programmer from being concerned about many of the particular details of the DMA transactions and, in particular, setting up multiple buffers to handle double or triple buffering to stream data through the Local Store.

4.6.5 Load-Balancing

By removing the requirement of specifying a specific SPE on which each Work Request is to run, the Offload API is free to distribute the Work Requests to any of the SPEs as it sees fit. The result is that load-balancing schemes can be applied by the API to help distribute the workload (Work Requests) across all of the SPE more evenly. This will increasing the overall performance of the application by ensuring that the SPEs do not become idle. Currently, a simple round-robin scheme is used. When one of the SPE's Message Queue's becomes filled, the Offload API will try another SPE's Message Queue.

4.7 Limitations of the Offload API

The Offload API cannot hide all of the architectural details of the Cell's architecture from the programmer. There are four items in particular that need to be considered by the programmer.

The first architectural detail is the limited size of the SPE's Local Store. It is important for the programmer to realize that the buffers used by a Work Request cannot be arbitrarily large. The buffers need to fit within the Local Store which also contains the SPE code, the SPE's stack, and other data buffers being used by other Work Requests also present on the SPE.

The second architectural detail is the alignment of the data buffers and DMA lists. The Cell architecture requires that all buffers that are involved with DMA transactions (source or target) need to be 16 byte aligned at a minimum (128 byte alignment is preferred). It is up to the programmer to ensure that the buffers passed to the Offload API are aligned properly.

The third architectural detail is more for performance considerations rather than a requirement imposed by the Cell's architecture. The DMA transactions issued by the SPE's Memory Flow Controllers (MFCs) are cache coherent with the PPE's caches. Because of this, if a DMA transaction targets data that is contained with the PPE's cache, the current copy of the data contained in the cache will be accessed instead of the the DMA transaction accessing system memory. This will cause more pressure to be placed on the PPE's cache (i.e. the cache being accessed more). Additionally, the link connecting the PPE to the EIB will be placed under a greater load as it must now be used to transfer the data associated with the DMA transaction along with the normal traffic generated by the PPE (fetching instructions, accessing data, accessing SPE's state, and so on). Generally speaking, this means that it is better for data being accessed by Work Requests should not be accesses by the PPE directly (or vice versa) for the sake of better performance.

The fourth detail involves the compilation processes. The Offload API does not hide the fact that the PPE and the SPEs have different ISAs. Because of this, the codes that are executed on each of the core types have to be compiled separately. This also holds true for the Offload API. The SPE code must be compiled, linked with the library containing the SPE Runtime, and then the resulting SPE executable must be embedded into an object file for the PPE. The PPE code must be compiled and then linked to both a library containing the Offload API and the object file containing the SPE executable.

In addition to not hiding all of the architectural details of the Cell's architecture, the Offload API is not well suited for all types of computations. Computations involving a large data structure which must be traversed by making many arbitrary accesses into system memory (such as a graph or tree traversal with very little work per node visit) are not likely to perform well using the Offload API (or on the Cell in general). The Offload API is instead designed to play to the strengths of the SPEs, namely, streaming large amounts of data broken down into continuous chunks through the SPEs for processing.

4.8 Current Status of the Offload API

While the Offload API is still under active development, it can currently be used to create applications. Example C/C++ programs have been written using the Offload API. The Offload API along with the example C/C++ programs can be found in the Charm++ distribution. [6] The Charm++ Runtime System has some hooks that allow it to do basic interactions with the Offload API. Because of this, Charm++ applications can currently take advantage of the SPEs. However, at the current time, this still does require modification to the Charm++ application's source code. As was already mentioned, example Charm++ applications using the Offload API already exist and are available as part of the Charm++ distribution.

[6]Available for download at *http://charm.cs.uiuc.edu/*

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The main focus of the work presented in this document has revolved around the creation of the Offload API with the requirements of the Charm++ programming model in mind. As discussed, the Charm++ programming model fits will with the Cell's architecture for many reasons. The Offload API has been designed and implemented to facilitate offloading entry methods to the SPEs by the Charm++ Runtime System. Additionally, the Offload API has been designed so that it can be used separately from Charm++ (i.e. - by any C/C++ program). The Offload API also simplifies programming for the Cell processor and increases programmer productivity by removing the need for programmers to concern themselves with many of the details of the Cell architecture (as described in Section 4.6). Instead, programmers can focus on their application code. Example programs, including both C/C++ applications that directly use the Offload API and Charm++ applications where entry methods directly use the Offload API, already exist. The Offload API is open source and freely available for download (included within the Charm++ distribution which is also open source and freely available for download).

5.2 Future Work

While much progress has been made, a great deal of work still remains. The most immediately pressing work is to gain performance data on the Offload API and applications (both

C/C++ and Charm++).

5.2.1 Performance Evaluation

Performance of the Offload API

There are many aspects of the Offload API that need to be evaluated for performance reasons. Some evaluation of the performance of the Offload API is already underway. Because the Offload API is the mechanism which will be used by the Charm++ Runtime System to offload entry methods to the SPEs, it is important that the overhead imposed by the Offload API is minimized.

Future work along these lines will include reducing the overhead associated with each of the state functions (i.e. - the states in which each Work Request passes through when on the SPE as listed in Table 4.3). Reducing this overhead will not only increase the amount of time that the SPE can spend executing user code, it will also allow for finer-grained Work Requests.

Performance of C/C++ Applications using the Offload API

Another area in which performance data will be useful is the actual performance as seen by end C/C++ applications. This will give insight to various characteristics of the Offload API, including the required frequency of progress calls to optimize performance (how much *attention* should the PPE pay to the SPEs vs. the time the PPE can spend executing application code). This testing will also give insight as to what size the buffers used in Work Requests should be to optimize performance. Also, how much DMA latency can be hidden in practice and how much time should each Work Request spend executing on the SPE (time in user code and associated data buffer sizes) to help hide the overhead imposed by the Offload API.

Performance of Charm++ Applications on Cell

Performance information on Charm++ applications executing on Cell-based platforms will give insight as to how the Charm++ Runtime System should be tuned to optimize performance of the SPEs. Questions to be asked include:

1. How much of a performance hit is incurred when several entry methods are executed on the PPE vs. the SPEs (since the PPE is already a potential bottleneck in that it must coordinate all the SPEs along with handle network activity)?

2. How does a long latency entry method executed on the PPE effect the performance of the Offload API (assuming the Charm++ Runtime System calls *progress* on the Offload API rather than *progress* being called directly from application code)?

3. What is the latency of a message when it is generated on an SPE on one Cell processor and received by an SPE on another processor?

4. Load-Balancing concerns: How should load-balancers be modified to take SPEs into account? For example, consider the case where a processor has been given many chares, however, none of the chares are utilizing the SPEs. Could a load-balancer detect this and move some chares which use the SPEs onto that processor (which may increase the entry method throughput without noticeably increasing the total execution time since the otherwise idle SPEs are now doing the additional work in parallel with the already heavily-loaded PPE). More generally, will requiring a load-balancer to take the type of core/work into consideration make a difference in its ability to effectively load-balance the chare objects across the heterogeneous processors and/or the heterogeneous platform.

5.2.2 Extending the Offload API

More Work Request Types

Another avenue of future work would be to create additional Work Request types (as directed by the needs of applications). One possible type of Work Request could be a *Streaming Work Request*. For a Streaming Work Request, the user would specify a starting address, *section* size, *section* count, and *stride* for each buffer. The SPE Runtime would allocate enough memory in the SPE's Local Store to double, or even triple, buffer the data buffer sections. A single section of each buffer would be brought into the Local Store and execution of the Work Request would begin on this set of sections. While the Work Request is executing on one set of sections, the SPE Runtime can be *fetching* the next section of each data buffer (the next set of sections). This way, once one set of sections have been processed by the user's code, the next set of sections will be ready for processing. This processes, of overlapping processing of one set of sections while *fetching* the next set of sections, would continue until the data buffers have been processed entirely. This type of Work Request not only plays to the strengths of the SPE's architecture (streaming of data through the Local Store), it also decreases the overhead created by the Offload API and SPE Runtime (SPE only given one Work Request, memory allocated only once, PPE notified only once, and so on). Additionally, this allows a single Work Request to refer to more data than can be stored in an SPE's Local Store since the Local Store would only have to store two or three sets of sections at any given moment.

As an illustration of the usage of a Streaming Work Request, consider once again the computation done in Figure 4.1 (the addition of two arrays, A and B). Here, a single Streaming Work Request could be issued that would have two read-only buffers and one write-only buffer. The starting address of the first read-only buffer would be the starting address of the array A. The starting address of the second read-only buffer would be the starting address of the array B. The starting address of the write-only buffer would be the starting address of the array C. For all the buffers, both the size of each section and the stride between sections would be *L * the size of each array element*. The advantage here is that the application code would only have to issue a single Work Request (and receive a single notification) to perform the entire operation regardless of the size of the arrays.

It should be mentioned that this type of Work Request would limit the ability for the Charm++ Runtime System to execute multiple entry methods for the same chare objects as an optimization (as mentioned in Section 3.2.1). Because the buffers are not copied in their entirety before the computation begins, it would be possible for two entry methods to interfere with each other even with the conditions stated in Section 3.2.1.

Movement of Code

The Offload API, currently, only allows for the movement of data between system memory and an SPE's Local Store when a Work Request is issued. Due to the limited size of the SPE's Local Store, it would also be advantageous if the code that is to be executed by the Work Request could also be free to move between system memory and the SPEs' Local Stores. This would leave only the SPE Runtime which would have to permanently remain in the SPE's Local Store. The remaining memory could be used to stream Work Requests.

5.2.3 Modification of Charm++ Tools

Currently, for a Charm++ application to take advantage of the SPEs on a Cell processor, it must explicitly call into the Offload API from the entry methods of the chare objects. However, this hinders the possibility of Charm++ applications being portable between Cell-based and non-Cell-based platforms. By modifying the Charm++ tools (charmc and charmxi), it should be possible to have the tools generate the code necessary to *offload* entry methods to the SPEs. As has already been discussed, the tools, at least to start with, would check the *.ci* file for a keyword that indicates the particular entry method in question is eligible for execution on one of the SPEs. The Charm++ tools would then compile the entry method twice (once for PPE and once for SPE). The Charm++ Runtime System would then execute the entry method on either the PPE or the SPE as it sees fit during runtime.

Extend to GPUs and FPGAs

Both Graphical Processor Units (GPUs) and Field Programmable Gate Arrays (FPGAs) have recently been used to boost the performance of applications. Allowing the Charm++ Runtime System to harness the computational power of both of these technologies would possibly increase the performance of Charm++ applications. However, both of these technologies, like the Cell, are considered hard to program. Perhaps the same techniques that have been applied here to both hide programming complexity and *offload* chunks of computation to the accelerators (in the case of Cell, the SPEs) can also be applied to both GPUs and FPGAs. In a truly ideal case, when compiling a Charm++ application, the Charm++ tools could detect the presence of SPEs, GPUs, and/or FPGAs. Upon detection of each type of accelerator, the Charm++ tools could compile a version of the entry methods (once again, as long as the entry methods are suitable for the particular accelerator) for each of the accelerators. Then, during runtime, the Charm++ Runtime System could dynamically make decisions about where each each of the chare object's entry methods should execute (thus *offloading* the computation to the accelerator of choice).

Appendix A

Offload API Code Example

The code listed in this appendix performs the calculation that is depicted in Figure 4.1. Three files are listed with this example (arrayAdd.cpp, arrayAdd_shared.cpp, arrayAdd_shared.h) along with the output from an execution of the program. The file arrayAdd.cpp contains the code that is executed on the PPE. The file arrayAdd_shared.cpp contains the code that is executed on the SPE. The file arrayAdd_shared.h is a header file that is included by both the PPE and the SPE code.

A.1 File: arrayAdd.cpp *(PPE)*

```
#include <stdio.h>
#include <string.h>
#include <spert_ppu.h>
#include "arrayAdd_shared.h"

volatile float A[N] __attribute__((aligned(128)));
volatile float B[N] __attribute__((aligned(128)));
volatile float C[N] __attribute__((aligned(128)));

void errorHandler(int errorCode,
                  void* userData,
                  WRHandle wrHandle) {
  fprintf(stderr, "--- ERROR HANDLER CALLED!!! ---\n");
}
```

```
int main(int argc, char* argv[]) {

  // Initialize the arrays
  for (int i = 0; i < N; i++) {
    A[i] = i;  B[i] = 3;  C[i] = 0;
  }

  // Display the arrays
  printf("Arrays (pre-calculation)...\n");
  for (int i = 0; i < N; i++)
    printf("%d:  %f + %f = %f\n", i, A[i], B[i], C[i]);

  // Initialize the Offload API
  InitOffloadAPI(NULL, NULL, errorHandler);

  // Create a Work Request Group
  WRGroupHandle wrGroup = createWRGroup();

  // Create all of the Work Requests (adding each
  //   to the Work Request Group)
  register int chunkByteCount = L * sizeof(float);
  for (int i = 0; i < (N / L); i++) {

    // Calculate the start of this Work Request's array sections
    register void* a = (void*)(A + (i * L));
    register void* b = (void*)(B + (i * L));
    register void* c = (void*)(C + (i * L));

    // Create the Work Request
    sendWorkRequest(FUNC_ADD, // Function Index
       a, chunkByteCount,      // Read/Write Buffer (read-only)
```

```
      b, chunkByteCount,          // Read-Only Buffer
      c, chunkByteCount,          // Write-Only Buffer
      NULL,                       // User data
      WORK_REQUEST_FLAGS_RW_IS_RO, // Flags
      NULL,                       // Callback function
      wrGroup                     // WRGroupHandle
    );
  }

  // Complete the Work Request Group (no more
  //   Work Requests will be added to it)
  completeWRGroup(wrGroup);

  // Wait for the Work Request Group to complete
  waitForWRGroupHandle(wrGroup);

  // Display the results
  printf("Arrays (post-calculation)...\n");
  for (int i = 0; i < N; i++)
    printf("%d: %f + %f = %f\n", i, A[i], B[i], C[i]);

  // Close the Offload API
  CloseOffloadAPI();

  // All Good
  return EXIT_SUCCESS;
}
```

A.2 File: arrayAdd_shared.cpp *(SPE)*

```
#include <stdio.h>
#include "spert.h"
```

```
#include "arrayAdd_shared.h"

inline void add(float* A, float* B, float* C) {
  register int i;
  for (i = 0; i < L; i++)
    C[i] = A[i] + B[i];
}

#ifdef __cplusplus
extern "C"
#endif
void funcLookup(int funcIndex,
                void* readWritePtr, int readWriteLen,
                void* readOnlyPtr, int readOnlyLen,
                void* writeOnlyPtr, int writeOnlyLen,
                DMAListEntry* dmaList
               ) {

  switch (funcIndex) {

    case SPE_FUNC_INDEX_INIT:  break;
    case SPE_FUNC_INDEX_CLOSE: break;

    case FUNC_ADD:
      add(readWritePtr, readOnlyPtr, writeOnlyPtr);
      break;

    default:
      printf("ERROR :: Invalid funcIndex (%d)\n", funcIndex);
      break;
  }
}
```

A.3 File: arrayAdd_shared.h *(PPE + SPE)*

```
#ifndef __HELLO_SHARED_H__
#define __HELLO_SHARED_H__

#define N   1024    // NOTE : N should be a multiple of L
#define L   64      // NOTE : Should be a multiple of 4

#define FUNC_ADD      1

#endif //__HELLO_SHARED_H__
```

A.4 Output

Please Note: Some of the output was ommitted for brevity.

```
Arrays (pre-calculation)...
0:   0.000000 + 3.000000 = 0.000000
1:   1.000000 + 3.000000 = 0.000000
2:   2.000000 + 3.000000 = 0.000000
...
1021:   1021.000000 + 3.000000 = 0.000000
1022:   1022.000000 + 3.000000 = 0.000000
1023:   1023.000000 + 3.000000 = 0.000000
Arrays (post-calculation)...
0:   0.000000 + 3.000000 = 3.000000
1:   1.000000 + 3.000000 = 4.000000
2:   2.000000 + 3.000000 = 5.000000
...
1021:   1021.000000 + 3.000000 = 1024.000000
1022:   1022.000000 + 3.000000 = 1025.000000
1023:   1023.000000 + 3.000000 = 1026.000000
```

Appendix B

Charm++ Code Example

The following code presents a simple example Charm++ program. All Charm++ programs have a *main* chare (similar to how C/C++ applications all have a *main* function). Execution of a Charm++ application begins in the constructor of the *main* chare object.

The program first creates an array of chare objects. Once the array of chares has been created, a message containing a random number is broadcast to all members of the chare array. Upon receiving the message, the array members print the value contained in the message and then send a message back to the main chare indicating that they have completed their *work*. Once the main chare has received replies from all of the array chares, the program exits.

This simple example Charm++ program by no means encompasses all of the features available to Charm++ (either via the Charm++ programming model or the rich set of libraries and frameworks that have been developed for Charm++). The reader is encouraged to further explore Charm++, libraries/frameworks for Charm++, and Charm++ applications at *http://charm.cs.uiuc.edu/*.

Please note: While the hello.C file includes both the declaration of the chare classes along with the bodies of the member functions, this is not a requirement. The source files of a Charm++ application can follow the more familiar C++ style where the class is declared in a header file and the bodies of the member functions are located in an associated source file.

B.1 File: hello.C

```
#include <stdio.h>
#include "hello.decl.h"   // Generated by Charm++ tools

/*readonly*/ CProxy_Main mainProxy;
/*readonly*/ int nElements;

// The "Main" Chare
class Main : public CBase_Main {
  public:

    Main(CkArgMsg* m) {
      // Process command-line arguments
      nElements = 10;  // Default to 10 "Hello"s
      if ((m->argc) > 1) nElements = atoi(m->argv[1]);
      delete m;

      // Print info on the run for the user
      CkPrintf("Running Hello on %d processors for %d elements\n",
                      CkNumPes(), nElements
                     );

      // Create a global reference to the Main chare's proxy
      mainProxy = thisProxy;

      // Create the array of "Hello" chares and then broadcast
      //   a message to them instructing them to say hi
      CProxy_Hello arr = CProxy_Hello::ckNew(nElements);
      srand(0);
      arr.sayHello(rand() % 10);
      // NOTE: Alternatively, a single member of a chare array
      //   can be singled out via the [] operator.  For example,
      //   to only send the above message to the first chare in
      //   the chare array, use: "arr[0].sayHello(rand() % 10);"
    };

    void done(void) {
      static int helloFinishedCount = 0;

      // Increment the finished counter and check to see
      //   if all elements of the "Hello" array have completed
      helloFinishedCount++;
      if (helloFinishedCount >= nElements)
        CkExit();
    };
};

// A 1D Array of Chares
class Hello : public CBase_Hello {
  public:

    Hello() { }                          // General Constructor
    Hello(CkMigrateMessage *m) { } // Migration Constructor
    void sayHello(int num) {

      // Say "Hello"
      CkPrintf("Hello from element %d on processor %d (num:%d)\n",
                         thisIndex, CkMyPe(), num
                        );

      // Send a message to the main chare indicating that
      //   this element has finished its "computation"
      mainProxy.done();
  }
};

#include "hello.def.h"    // Generated by Charm++ tools
```

B.2 File: hello.ci

```
mainmodule hello {

  readonly CProxy_Main mainProxy;
  readonly int nElements;

  mainchare Main {
    entry Main(CkArgMsg *m);
    entry void done(void);
  };

  array [1D] Hello {
    entry Hello(void);
    entry void sayHello(int num);
  };
};
```

B.3 Output

```
$ charmrun +p4 ./hello
Running Hello on 4 processors for 10 elements
Hello from element 0 on processor 0 (num:3)
Hello from element 4 on processor 0 (num:3)
Hello from element 8 on processor 0 (num:3)
Hello from element 2 on processor 2 (num:3)
Hello from element 6 on processor 2 (num:3)
Hello from element 3 on processor 3 (num:3)
Hello from element 7 on processor 3 (num:3)
Hello from element 1 on processor 1 (num:3)
Hello from element 5 on processor 1 (num:3)
Hello from element 9 on processor 1 (num:3)
```

References

[1] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. CellSs: A Programming Model for the Cell BE Architecture. In *Proceedings of the ACM/IEEE SC 2006 Conference*, November 2006.

[2] Brian Bouzas, Robert Cooper, Jon Greene, Michael Pepe, and Myra Jean Prelle. MultiCore Framework: An API for Programming Heterogeneous Multicore Processors. Paper in Mercury Computer System's Literature Library (http://www.mc.com/cell/lib/index.cfm).

[3] A. C. Chow, G. C. Fossum, and D. A. Brokenshire. A Programming Example: Large FFT on the Cell Broadband Engine. White Paper in IBM Technology Group Library (http://www-306.ibm.com/chips/techlib/techlib.nsf), October 2005.

[4] Department of Computer Science,University of Illinois at Urbana-Champaign, Urbana, IL. *The CHARM (4.5) programming language manual*, 1997.

[5] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[6] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Processor. *IBM Journal of Research and Development: POWER5 and Packaging*, 49(4/5):589, 2005.

[7] L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.

[8] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[9] David Kunzman, Gengbin Zheng, Eric Bohm, and Laxmikant V. Kalé. Charm++, Offload API, and the Cell Processor. In *Proceedings of the Workshop on Programming Models for Ubiquitous Parallelism*, Seattle, WA, USA, September 2006.

[10] Parallel Programming Laboratory. *Converse Programming Manual*. Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL. http://charm.cs.uiuc.edu/manuals/html/converse/manual.html.

[11] B. Minor, G. Fossum, and V. To. Terrain Rendering Engine (TRE): Cell Broadband Engine Optimized Real-time Ray-caster. White Paper in IBM Technology Group Library (http://www-306.ibm.com/chips/techlib/techlib.nsf), October 2005.

[12] M. Sakamoto, H. Nishiyama, H. Satoh, S. Shimizu, T. Sanuki, K. Kamijoh, A. Watanabe, and A. Asahara. An Implementation of the Feldkamp Algorithm for Medical Imaging on Cell. White Paper in IBM Technology Group Library (http://www-306.ibm.com/chips/techlib/techlib.nsf), October 2005.