

Cell-MT

An efficient runtime for fine-grained parallelism

Apollo Ellis

Real-Time Graphics and Parallel Systems Lab, ICES
apichit@cs.utexas.edu

Bharadwaj Subramanian

Computational Visualization Center, ICES
bharadwajs@ices.utexas.edu

Abstract

Improvements in the performance of general-purpose processor systems have lately focussed on using powerful, yet simple cores in order to provide speedups which were previously attained in single processor systems using higher processor clock frequencies, super scalar architectures, or deeper pipelines. However, in order to take advantage of the computational power afforded by these multicore systems, specialized runtime systems are required which can utilize these resources efficiently, while providing a simple abstraction of the underlying architecture to the developer. In this report, the development of such a runtime system, Cell-MT, an API designed to provide fine-grained parallelism with work-stealing features and user specified job scheduling on the Cell Broadband Engine architecture is discussed.

Categories and Subject Descriptors D [1]: 3

General Terms Synergistic Processing Unit, Kernels, Reorder buffers, FILTERPROC, Filters, Work Queues

Keywords Fine-grained parallelism, work stealing, dynamic scheduling, heterogenous multi-core architecture

1. Introduction

Multi-core processors find wide applications in the fields of high performance computing, stream processing and other computationally intensive activities such as gaming and real-time graphics. In order to enable data parallel computation on these platforms, specialized runtime systems are required which can make use of these resources in the most efficient manner. Many run-time systems such as MPI and OpenMP have been developed in the past for such multi-core platforms; each have their own advantages and disadvantages.

The Cell Broadband Engine is a single-chip heterogeneous multi-core engine with 7 - 9 processors operating on a shared, coherent memory[CBEA20K]. The architecture is the result of a collaboration between Sony, Toshiba and IBM, and is targeted towards gaming consoles, HD stream processing applications and high performance computing. The Cell architecture is designed to provide a wide bandwidth to compensate for latency, and computational throughput is emphasized at the expense of simplicity of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]...\$10.00

program code. Hence, due to the unique architecture of the Cell Broadband Engine, a specialized runtime system is required to take advantage of the resources afforded by the architecture, and to provide a simplified interface to the developer.

In this report, we describe Cell-MT, a proposed fine grained run time system for data parallelism on the Cell Broadband Engine. In Section 2, we review the Cell Broadband Engine architecture in detail, followed by an overview of existing runtime systems for the architecture. In Section 3, we go over the design goals and implemented features of the Cell-MT runtime system. This is followed by a discussion of the implementation of the run-time and a comparison with a baseline in Section 5. Finally, avenues of future research and improvements to the runtime system are discussed in Section 6.

2. Prior Work

2.1 The Cell Architecture

The Cell Broadband Engine architecture (or the Cell Architecture, in short) consists of a single cache-coherent Power Processing Element or PPE, and 6-8 SPEs or Synergistic Processing Elements, meant for performing data parallel computations. All the processors execute instructions in-order; no branch prediction or out-of-order processing is implemented. The PPE is a 64-bit Power Architecture core with VMX Multimedia Extensions, and can run 32- and 64-bit applications and operating systems. The SPEs are a 128-bit, dual issue, SIMD-RISC architecture geared towards computationally intensive vector processing. The Cell Architecture is designed so that peak performance can be achieved by the coordinated execution of tasks between the SPEs and the PPE.

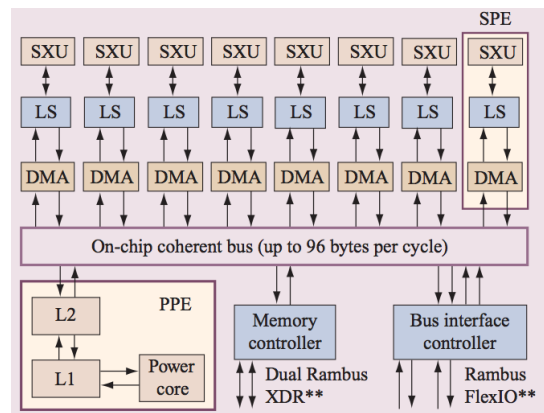


Figure 1: The Cell Architecture[KAH2005]

The PPE is designed to be the main processor running on the Cell Engine, and is responsible for running the operating system

threads, and also for coordinating the SPEs. Towards this end, the PPE is designed to support two threads concurrently. The PPE also has a 32 kb L1 cache, and a 512 kb L2 cache, allowing it to take advantage of MM facilities in the host OS. For all practical purposes, the PPE acts as a standard issue processor.

The SPU, or the Synergistic Processing Unit, is a collection of all the components which are involved in the operation of a single SPE. The SPE is a component geared specifically towards data-parallel execution of tasks. Each SPE has a 256kb Local Store which it can operate upon, and access to main memory is restricted to DMA transfers. All code and data that the SPE operates on must reside in the local store. Any additional data or instructions that cannot be stored in the local store need to be transferred explicitly from main memory via DMA before execution. The SPEs also provide 128 128-bit general purpose registers, used for vector operations; no support is provided for scalar operations, and must be handled by the compiler. Furthermore, the SPUs also have a Memory Flow Controller, or the MFC, which contains a DMA controller and an associated MMU for all host memory accesses. The MFC is capable of doing DMA transfers and translating addresses independent of the SPE. Finally, the SPU contains an Atomic Unit, used to handle synchronization operations with other SPEs and the PPE. The SPUs can perform operations on four 32-bit integers, or four single-precision floating-point numbers per cycle. Due to the small size of the SPU local store and the unique characteristics of the architecture, writing an application targeted to utilize the SPU is a challenging problem.

In order to facilitate communication between the SPEs, the PPE, the external memory and the external I/O, a high speed internal bus, known as the Element Interconnect Bus or EIB is provided. Each unit on the EIB can send or receive 16 bytes of data every clock cycle. The bus provides one address bus and four 16-byte wide data rings, and has a theoretical maximum bandwidth of around 200 GBps, providing each functional element 25.6 GBps each way.[CBEAFI]

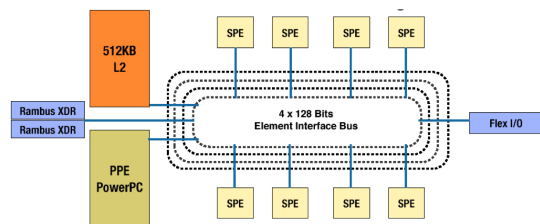


Figure 2: The Element Interconnect Bus

The Memory Interface Controller, or MIC, is connected to the external Rambus XDR memory through two XIO channels operating at 3.2 GHz, providing a peak bandwidth of 25.6 GBps. Each XIO channel has separate read and write queues; an arbiter alternates dispatch between each of the queues for a minimum of 8 dispatches in either queue, or until the queue is empty, whichever is shorter. Finally, the Cell Architecture also implements resource allocation to provide controlled access to the memory controller, to allow time-critical applications to meet timing targets.[CBEAFI]

Finally, the Cell architecture provides two flexible I/O interfaces with seven transmit and five receive Rambus FlexIO links each, providing a total peak bandwidth of 76.8 GBps. Data and commands are sent through these I/O interfaces as packets; processing these packets adds overhead which reduce the effective throughput of the I/O interface.

2.2 Runtimes for the Cell Architecture

2.2.1 MARS

MARS stands for Multicore Application Runtime System, provided by Sony as a generic runtime system for multicore architectures. The MARS defines a few generic terms with respect to the target multicore system in order to be as broadly applicable as possible. It provides for a Host processor, which handles the scheduling and primary OS thread handling, and one or more MPUs or Micro-Processing Units which perform the actual computation required. It also defines a Host storage, which refers to a common memory area, and an MPU storage local to the MPU. Finally, a Workload is defined as a generic unit of work scheduled to be executed on an MPU. In terms of the Cell architecture, the MARS terminology equivalents are as follows: the PPE acts as the Host; the SPUs act as the MPUs; the host memory acts as the Host storage, and the local store on each SPU acts as the MPU storage. The MARS provides for lightweight context switching between workloads, while minimizing the runtime load of the Host processor. The MARS API provides an MPU-centric runtime environment for multicore architectures and handles scheduling operations from the MPU side.

The MARS works by defining a MARS context which is a secondary thread on the Host handling the scheduling and management of workloads on the MPUs. The MARS Host API provides functions for managing the MARS context, to initialize and schedule workloads for execution, and methods for synchronization between the Host and the MPUs. The MPU-side API, on the other hand, enables the code to manage the program state, synchronize with the Host and other MPUs, and also to provide for scheduling initialized workloads. Finally, the MARS also provides an MPU kernel, which stays resident in the MPU once initialized throughout the life of the MARS context; it is used to demand-load workloads from the main memory to the MPU memory. The MPU kernel provides for priority based co-operative scheduling, and is responsible for fetching tasks from the workload queue present in the host storage. The overall flow of control within a MARS context is given in Figure 3. [MARS]

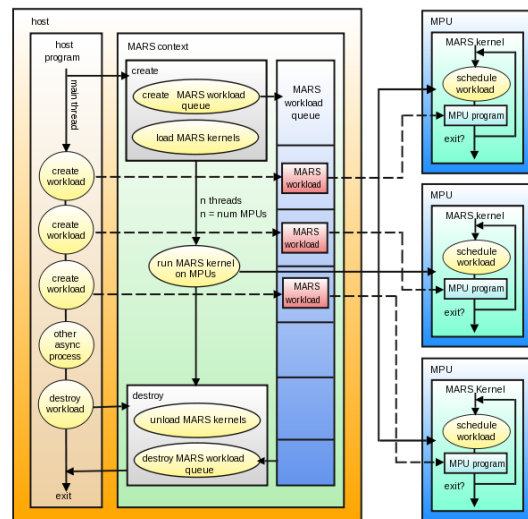


Figure 3. The MARS context and workflow[MARS]

The general usage flow for MARS is as follows: the Host program initializes a MARS context, and prepares the MARS kernel for each MPU. Then the workloads are initialized and are loaded into a shared workload queue. Workload scheduling can be done from both the Host and the MPU; waiting is done, if required,

for synchronization between different workloads. Finally, after all workloads have been executed, the MPU kernel is unloaded and the MARS context is finalized. The Host program is responsible for breaking up the computation into MARS tasks which the MPUs can then execute independently. Each MARS task (or workload) contains both the code and the data section, and is considered as a complete unit. The task contents are accessed from the Host storage via DMA and are executed on the MPU. The MARS provides for task switches with both full context saves or none at all; partial context saves are not supported. The state diagram for MARS tasks are as in Figure 4.

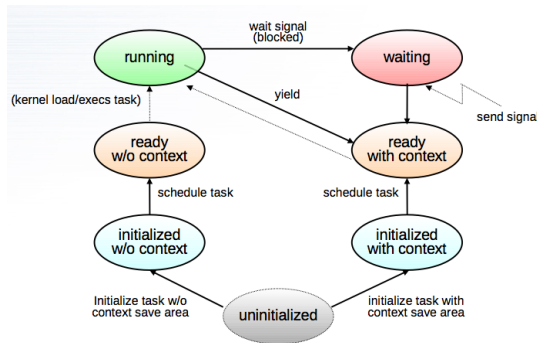


Figure 4. The MARS task state diagram[MARS]

Although the MARS provides a level of abstraction away from the details of the Cell architecture, it does not implement fine-grained control of processes. Furthermore, load balancing strategies such as are not implemented in the MARS implementation for the Cell architecture, thereby resulting in poor resource utilization. Finally, MARS does not implement parent-child task waiting; as a result, implementing programs which involve parent-child synchronization is difficult in the MARS.

2.2.2 SPURS

SPURS stands for SPU Runtime System, and is provided by Sony for its PS3 line of gaming consoles. It is used to handle SPU workload balancing on a higher level - between different libraries which use the SPU for their computational requirements. SPURS targets three basic processing models: non-parallel processing, where each workload/task has to be processed by a single SPU and no sharing can be done; parallel processing, where the workload is split into two or more jobs, which can then be assigned to run on SPUs more or less independently; and finally stream processing, where each SPU performs a part of the processing on the input, and hands off the result to the next SPU in the stream. The SPUR system provides for non-preemptive, cooperative, SPU-driven scheduling so that high priority workloads that go in and out of the ready state get serviced in a reasonable time. The SPU kernel as provided by the SPUR system is 2kb in size, and provides implementations of the job streaming and multi-tasking models.

The SPUR System uses the following terminology: The SPURS kernel is a small binary, resident on all SPURS SPU threads which schedules workloads and policy modules on to the SPU local store. The Policy Module is an SPU binary which processes Workloads; SPURS provides policies for multi-tasking and job streaming. A Workload is interpreted as a group of tasks or jobs that can be processed by multiple SPUs. Finally, SPURS also defines an atomic notifier which is an 128 bit data structure used to hold workload scheduling parameters. The advantage of the SPUR system is that it provides program independent task parallelism for code written for the Cell architecture.

In the SPUR System, each program is broken down into multiple tasks that can be executed on the SPUs. Inter-task communi-

cation is done through libraries provided by the system; the SPU task modules schedule tasks in parallel. In order to save context while switching between tasks, a designated RAM context area in the host storage is provided; context saving and restoration is done via DMA transfers. Each SPU task is scheduled by the SPU; it is designed to be lock-free and follows a fixed memory layout. The SPUR system provides useful libraries for communication between SPU tasks like signals, barriers and semaphores. The task state diagram for SPURS is similar to MARS.

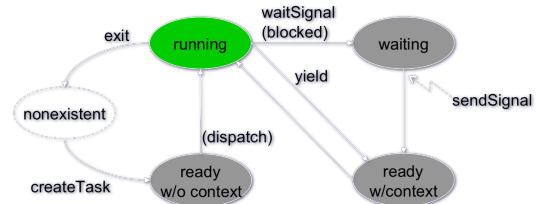


Figure 5. The SPURS task state diagram

Apart from the traditional task based processing, SPURS also supports stream processing. Each stream job may execute thousands of times per frame; furthermore, the data being processed and the complexity of the operations vary from application to application. Stream jobs are processed in a similar way to task based processing; jobs are scheduled such that jobs in the stream are executed sequentially. DMA transfers are interleaved with job execution stages so as to prefetch data required for jobs.

The SPUR System, like MARS, does not provide fine-grained parallelism nor does it possess load balancing features. As a result, although these run-time systems provide an easier base for harnessing the power of the Cell architecture, maximum efficiency may not be reached due to improper load balancing. Cell-MT is a proposed alternate runtime system, which aims to build upon the base provided by MARS and SPURS, while providing load balancing, fine-grained parallelism control and user specified dynamic scheduling features to the client program.

2.3 ALF

ALF, or the Accelerated Library Framework, provides a programming environment for data- and task-parallel programs on heterogeneous multicore systems, like the Cell. It follows a MPMD (Multiple Program, Multiple Data) programming model, where multiple programs can be scheduled to run on multiple processing elements, known as accelerators, at the same time. The framework includes functionality for memory management, parallel task management, double buffering and dynamic load balancing for parallel tasks. It also provides API support for defining execution order for tasks by defining task dependencies; the scheduler provides an optimal parallel scheduling scheme for the tasks based on the given dependencies.[IBMALF]

ALF provides a host-level, and an accelerator level runtime library. The tasks themselves are divided into three types: Application, which runs on the host, Accelerated Library, which uses the ALF accelerator APIs, and the Computational Kernel. A Computational Kernel is defined as an accelerator routine which takes in a given set of input data, and returns output. The data is separated into separate portions, known as work blocks; for a single task, each of these work blocks can be executed in parallel. Care must be taken to ensure that the work block size is suitable for the local storage limits of the accelerators; in the Cell architecture, the work block size is limited by the size of the local store.

The framework is well suited for computationally intensive, data parallel problems where operations are performed on a large data set, and there are no inherent dependencies within the data. It

is also suited for task-parallel problems, where the user can specify the task dependencies; the ALF runtime then schedules the tasks in such a way so as to obtain the maximum parallelism. A schematic of the ALF programming model is included in Figure 6.

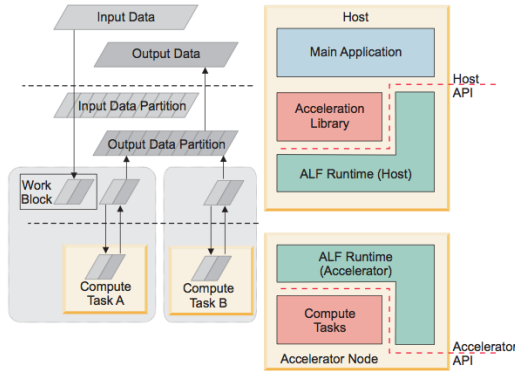


Figure 6. The ALF Programming model

3. The Cell-MT runtime

3.1 Design Goals

Cell-MT is designed to be an efficient runtime for fine-grained parallelism in the Cell architecture. The goal of the system is to provide an C++ based programming model and API for writing code which can take advantage of the Cell architecture, without having to deal with the complexity of scheduling and managing code instances running on the SPU. It follows a task parallel model where the runtime system handles delegating task units to different SPUs, and manages the scheduling of these tasks either automatically, or based upon a user provided dynamic scheduling. It also provides for automatic granularity control, where a job may specify how it can be divided into multiple jobs, and the dicing is performed on a need-only basis by the runtime system. Work-stealing features are also provided in order to maximize resource utilization and for load balancing. The runtime also is capable of simulating data parallelism by allowing for user scheduling. Parent-child waits and mutex mechanisms are also provided for synchronization. Finally, an API similar to Closure is provided for multi-tasking using the Cell-MT system.

3.2 Programming Model

Cell-MT provides a PPE-side library, an SPU-side library, and an SPU kernel. The PPE-side library is used to manage the scheduling of jobs on the SPU and the higher level thread management operations. The SPU-side library is used for fetching and posting jobs from the job queue managed by the PPE, and for managing locks, barriers and other synchronization devices. The SPU library also implements work stealing features for load balancing, as a part of the job fetching process. The SPU kernel is responsible for loading and executing the user code.

The internal organization of the Cell-MT system is as follows: The PPU thread runs a scheduler, known as FILTERPROC, which allows for user provided dynamic scheduling. The PPU also maintains a Re-order Buffer and a Ready buffer; each scheduled work unit goes into the re-order buffer, and the FILTERPROC re-orders the buffer in the correct priority. Each SPU contains an inbox from which a resident SPU kernel takes a work unit for processing. The SPU kernel is responsible for loading the corresponding SPU program representing the computation done in the work unit from main memory using DMA, and running it; the details of this operation is given in later sections. If the SPU kernel finds that the inbox is empty, it polls the Ready buffer for work units and tries to fill its

inbox with work units from the Ready buffer. If the Ready buffer is found to be empty, the kernel then calls an internal function to fill the Ready buffer from the Re-order buffer. If both the Reorder buffer and the Ready buffer are empty, the kernel looks for a work unit to steal from a random victim SPU.

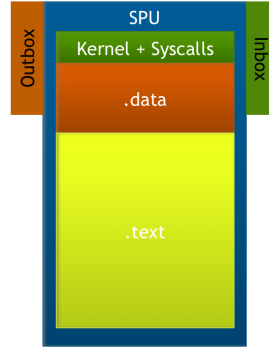


Figure 7. A single SPU.

The bootstrap process sets up the Cell-MT system for processing work units. First, the PPU thread initializes the Reorder buffer to accept work units and also starts the kernel thread running on each of the SPUs. This is followed by the user creating an initial set of work units which are put into the Reorder buffer. The PPU also sets up a barrier for all threads to be idle in order to indicate that the processing is complete; once the barrier is reached, the PPU thread terminates the SPU kernels, cleans up the buffers and finishes.

In order to handle work units that produce more work units for further processing, each SPU is provided with an outbox into which newly generated work units are inserted. Once the outbox is full, the work units in the outbox are flushed to a Prefilter buffer; the FILTERPROC takes the work units in the Prefilter buffer and schedules the jobs into the Reorder buffer, thereby completing the cycle. The Ready buffer is necessary due to contention issues that arise during SPU access of the Reorder buffer due to constant polling by FILTERPROC.

Each work unit also may optionally implement a “Dice” function which specifies how a particular unit may be broken down to be processed in parallel; this function must be implemented by the developer. The system makes use of the Dice function to break down a large work unit based upon certain criteria pertaining to the runtime state of the system, for instance, the system load or the idleness of SPUs.

3.3 The Job System

The most important object in the Cell-MT system is the Work object, which represents the computation to be performed on an SPU. The Work object contains information about the job being processed, and contains a pointer to the binary of the SPU program in the main memory. Any program that needs to be executed under the Cell-MT system needs to be broken down into SPU programs which can fit into the local store. The Work unit may also optionally implement the Dice function. Every time a job is posted to a buffer, a lock is obtained on the buffer so that write operations are atomic. In particular, the Reorder buffer is necessary since there is always the possibility that the FILTERPROC would change the contents of the Reorder buffer; hence direct accesses from the SPU to the Reorder buffer are disallowed. Instead, a “Fill” call is exposed at the API level which waits on the Reorder buffer until a lock can be obtained, locks the Ready buffer, and tries to move as many work units as possible from the Reorder buffer to the Ready buffer. During this call, neither the FILTERPROC nor the SPUs can access any of the buffers.

The FILTERPROC is a generic filtering procedure which is implemented as a list of Filter objects. The FILTERPROC first obtains a lock on the Reorder buffer, and moves all the work units posted in the Prefilter buffer into the Reorder buffer. Then, each of the Filter objects are applied in turn to the Reorder buffer to schedule the jobs in some order. By extending the Filter class, and reimplementing the filtering function, the user can control the way work units are scheduled to be pushed into the Ready buffer, thereby providing user controlled dynamic scheduling.

3.4 The SPU Kernel

The SPU Kernel consists mainly of 3 basic features: an underlying lock library, a structure representing a list of work units known as the the WorkQueue and system call interfaces. The WorkQueue contains the IN and OUT boxes for the kernel into which work units are put for processing. It also contains the locks necessary to protect these boxes. The functions available on the queue are the backend functions to the system call library exposed to the user. For instance, the system call "Post" is used to call Post on the Queue to send new jobs to the outbox and to check for values which indicate a Flush should be performed to post the jobs to Reorder buffer, via the Prefilter buffer. The WorkQueue also implements Get for getting work form the in box, and "PostLocal" for posting to the SPU's own inbox. The "PostLocal" method is used for implementing parent-child behavior where the parent needs to get pushed back into the queue in order to be processed after the child.

The lock library for Cell-MT is largely build around the Atomic operations provide on the platform. GETLLAR is one such instruction provided by the Cell architecture, and is used to get data by locking a line and obtaining a reservation. Another atomic operation, PUTLLC, then can put data back into main memory conditionally ie. only if the reservation still exists. A polling technique similar to those used in CAS based systems is used to obtain read modify writes, atomic adds, and atomic decrements. The lock library thus implemented consists of a Spin Lock and a Manual Split Gate. The SpinLock is left up the reader to infer functionality from; however the Split Gate Manual may not be as intuitive. The Gate is used to implement thread synchronization at the final stage where there are no more work units left to be processed and some of the SPUs have finished processing; in this case, the Gate is closed by those threads which have finished. Once this occurs the gate can be waited on by other threads incrementing the occupancy count. Once certain events occur in a given system the threads which have synchronized behind the gate can be let free by opening the gate and signaling their release.

The System call interface is fairly simple in the Cell-MT model. The user is provided with ways to Post work to the Reorder buffer, to post work to the SPU's own inbox, to post work that needs to be waited on to be finished before proceeding, and ways to begin the waiting process. The system calls are implemented with the jump instruction on function pointer which points to a location in the kernel. These locations are known since the kernel ELF is pre-parsed and this determines the way code is loaded onto an SPU. Thus, SPU programs written by the user can be run on the SPU directly without too many restrictions.

3.5 Work Stealing

Work stealing is implemented in an extremely simple fashion in Cell-MT. If there are no work units in the inbox, or in the Ready buffer and the Fill instruction to fill the Ready buffer fails, then the kernel simply tries to pick a processor close to itself (determined by an ordering of SPUs) and steals work from that core by fetching a work unit from the victim's inbox and doing a PostLocal on its queue. This indicates that the inbox lock of the cores need to be exposed; hence individual SPU WorkQueues are almost global.

Once the inbox lock is obtained by a stealing thread the work can be stolen; when an SPU is stealing work from an inbox, the victim SPU cannot access its inbox until the operation is completed. Thus atomicity of stealing operations is ensured, and load balancing is done among the SPUs.

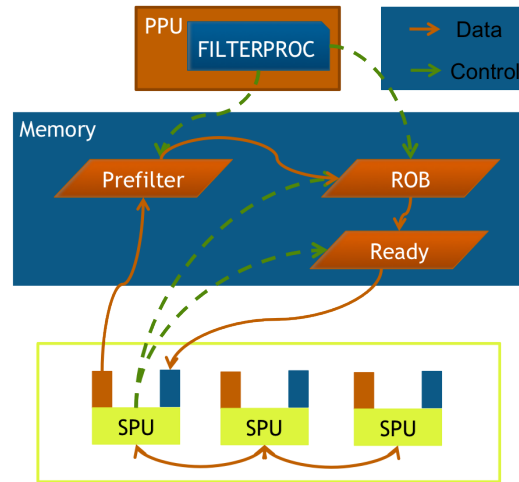


Figure 8. The Cell-MT programming model.

4. Implemented Features

The Cell-MT programming model has been implemented in C++ with a limited set of features. The features that have been implemented are as follows: A Reorder buffer has been implemented in main PPU memory which acts as the entry point of the programming model. The user code includes the Cell-MT library in its PPU module and then calls the function WorkQueueInit and registers its SPU programs. The registration mechanism is necessary since the kernel requires knowledge of the location of the SPU programs in memory so that unnecessary duplication of executable code is not done. The registration mechanism is fairly simple - the ELF binary is simply parsed and stored in a data structure called the Registry, along with an index uniquely identifying the executable. WorkQueueInit then spawns the kernel threads in each of the SPUs in order to fetch work from the Ready buffer. The Reorder buffer can then be posted to and the threads spawned in WorkQueueInit can begin working on the posted jobs. The kernel consists of the full lock library mentioned above, along with the syscall interface. However, only a limited subset of the syscalls have been implemented; support has been reduced to a subset of Post and PostLocal along with PostToROB and Flush.

Along with the Reorder buffer, the Ready buffer has also been implemented to reduce contention of the Reorder buffer as explained earlier. However, this is not a problem in the current implementation since user specified Filters have not been implemented; hence the FILTERPROC passively copies work units posted to the Prefilter buffer into the Reorder buffer. The issues with contention will apply mostly to future work when user scheduling is implemented. In the kernel the loader performs a DMA copy of the SPU executable code from the memory location provided by the Registry entry for the SPU program. The kernel then jumps directly in to the user code which is loaded to a compiler specified location in the Local Store, along with its text and data. Once finished, the code can jump back into kernel executable code via the syscall interface also described above. The inboxes and outboxes of the model have also been implemented and these are filled from the Ready buffer by the kernel work fetching code in order to facilitate processing. However the inbox and outbox are a fixed size; variable inbox sizes would be ideal in the final model. A PPU/SPU linked list protocol

for dynamic lists such as the reorder buffer and ready buffer has also been implemented; since the memory accesses from SPU to main memory are tedious, a linked list class was not used. Instead, a global, pointer based manipulation of the linked list is performed. The “Fill” function, used to fill the Ready buffer is however is implemented with a PPU syscall, or an SPU stop and signal function since it needs to be accessed by both the PPU and the SPU. Thus, a basic skeleton of the entire framework is in place, and further features need to be added; however, at present, the system can load jobs and running SPU code on them with minimal effort.

5. Results

A simple program for adding two numbers together was implemented in the a doubly nested loop and print the value of the final sum as a test for checking the job scheduling system and to simulate unbalanced loads. Since the loops are bound by input and the input is generated at random, this workload should exhibit poor load balancing, and the Cell-MT implementation should be able to avoid it. The loops are bounded by 1000 each and we run the experiment over a number of test cases varying the number of cores and the number of jobs scheduled. The results are shown for a single core as a baseline, against all six cores running in tandem using the Cell-MT system; the plot shows the number of jobs versus the execution time for each of the cases.

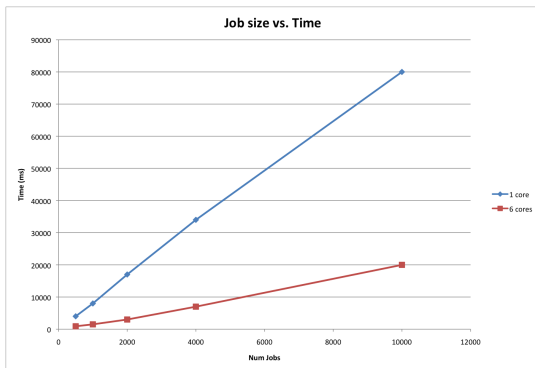


Figure 9. Preliminary runtime results for the Cell-MT system.

6. Future Directions

The API must be implemented in full with respect to the original SolMT Implementation, by Sean Keely et. al from UT Austin. Parent-Child wait mechanisms through a context switch code implemented in assembly also need to be provided. Spill registers will also be needed apart from the ability save both the stack and heap in order to perform context switches. Support for user scheduling through the implementation of the FILTERPROC running on the PPU whenever jobs are posted a new Prefilter Buffer also needs to be performed. Finally, the “Dice” functionality also needs to be implemented to provide for fine-grained parallelism.

Acknowledgments

Sean Keely, Okan Arikan, Donald Fussell, Calvin Lin , John Bates (Sony) and MARS team (Sony)

References

- [CBEA20K] Cell Broadband Engine Architecture from 20,000 feet, <http://www.ibm.com/developerworks/power/library/pa-cbea.html>
- [CBEAFI] Cell Broadband Engine Architecture and its first implementation, <https://www.ibm.com/developerworks/power/library/pa-cellperf/>
- [KAH2005] J. A. Kahle et al, Introduction to the Cell Multiprocessor, IBM J. Res. & Dev. Vol. 49 No. 4/5 July/September 2005

- [MARS] MARS Presentations, <ftp://ftp.infradead.org/pub/Sony-PS3/mars/presentations/>
- [IBMALF] Accelerated Library Framework for Cell Broadband Engine Programmers Guide and API Reference, http://publib.boulder.ibm.com/infocenter/systems/topic/eiccn/alf/ALF_Prog_Guide_API.v3.1.pdf