Copyright
by
Apollo Isaac Orion Ellis
2011

The Thesis committee for Apollo Isaac Orion Ellis
Certifies that this is the approved version of the following thesis:

Jack Rabbit
An Effective Cell BE Programming System for High Performance Parallelism

APPROVED BY

SUPERVISING COMMITTEE:

____
Donald S. Fussell, Supervisor

____
Calvin Lin, Supervisor

Jack Rabbit
An Effective Cell BE Programming System for High Performance Parallelism

by

Apollo Isaac Orion Ellis, B.A.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Computer Sciences

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2011

Dedicated to Achieving Efficiency Within the Bounds of Humanity.

Acknowledgments

I wish to thank: Dr. Fussell, Dr. Lin, Sean Keely, Sarah Abraham, Peter Djeu, and My Family and Friends . . .

Jack Rabbit
An Effective Cell BE Programming System for High Performance Parallelism

Apollo Isaac Orion Ellis, M.S.C.S.
The University of Texas at Austin, 2011

Supervisors: Donald S. Fussell
Calvin Lin

The Cell processor is an example of the trade-offs made when designing a mass market power efficient multi-core machine, but the machine-exposing architecture and raw communication mechanisms of Cell are hard to manage for a programmer. Cell's design is simple and causes software complexity to go up in the areas of achieving low threading overhead, good bandwidth efficiency, and load balance. Several attempts have been made to produce efficient and effective programming systems for Cell, but the attempts have been too specialized and thus fall short. We present Jack Rabbit, an efficient thread pool work queue implementation, with load balancing mechanisms and double buffering. Our system incurs low threading overhead, gets good load balance, and achieves bandwidth efficiency. Our system represents a step towards an effective way to program Cell and any similar current or future processors.

Table of Contents

Acknowledgments **v**

Abstract **vi**

List of Tables **ix**

List of Figures **x**

Chapter 1. Introduction **1**

Chapter 2. Related Work **6**

2.1 MARS 6
2.2 Sequoia 9
2.3 MGPS and EDLTP 11

Chapter 3. Background **14**

3.1 Cell Architecture 14
3.1.1 Cell Programmability 16
3.1.2 Cell Programming Models 16
3.2 Thread Pooling 17
3.3 Sol MT our API 20

Chapter 4. Design and Implementation **25**

4.1 Work 25
4.2 Work Queue 27
4.3 Locking Discipline 28
4.4 Locks 30
4.5 Global Barrier 32
4.6 Thread Pool 33

4.7 Load Balance with Dice 37
4.8 Double Buffering 39

Chapter 5. Evaluation **40**
5.1 Lu Factorization in Jack Rabbit 41
5.2 Barnes Hut Simulation in Jack Rabbit 46
5.3 Mandelbrot Set Rendering: MARS vs Jack Rabbit 50

Chapter 6. Future Work **55**
6.1 Locality Control 55
6.2 A New High Level Language 56

Chapter 7. Conclusion **58**

Index **60**

Bibliography **61**

Vita **63**

List of Tables

5.1 LU Factorization running with Double Buffering enabled and disabled on 1 - 6 working cores for 4k, 3k, 2k, and 1k sized Matrices
Top six rows: Original run times in seconds
Bottom six rows: Double buffered run times in seconds
%red refers to percent reduction of the original run time . . . 42

5.2 Barnes Hut running with Double Buffering enabled and disabled on 1 - 6 working cores for 10000, 100000, and 1000000 bodies
Top six rows: Original run times in seconds
Bottom six rows: Double buffered run times in seconds
Speedup refers to the parallel speedup 48

List of Figures

1.1 Cell Processor Block Diagram 2

4.1 Work type class: C++ code snippet 26
4.2 Work Queue base class Buffer: C++ code snippet 29
4.3 Spin Lock Try and Acquire Methods: C++ code snippet . . . 31
4.4 Checkout procedure excuted by the SPE thread pool kernels: C++ code snippet 35
4.5 getWorkDMA procedure executed by the SPE thread pool kernels: C++ code snippet 36
4.6 Dicing sub-system run by the PPE scheduler thread: C++ code snippet 38

5.1 Coarse-Grain 2-D Parallel Algorithm for LU Factorization [5] . 41
5.2 LU Factorization running with Double Buffering enabled and disabled on 1 - 6 working cores for a 4k sized Matrix
X-Axis : Cores Active as Worker Threads
Y-Axis : Run time in seconds 44
5.3 LU Factorization running with Double Buffering enabled and disabled on 1 - 6 working cores for a 3k sized Matrix
X-Axis : Cores Active as Worker Threads
Y-Axis : Run time in seconds 44
5.4 LU Factorization running with Double Buffering enabled and disabled on 1 - 6 working cores for a 2k sized Matrix
X-Axis : Cores Active as Worker Threads
Y-Axis : Run time in seconds 45
5.5 LU Factorization running with Double Buffering enabled and disabled on 1 - 6 working cores for a 1k sized Matrix
X-Axis : Cores Active as Worker Threads
Y-Axis : Run time in seconds 45
5.6 Quad Tree example image. The outlined part of the tree might be passed into one task etc. 46

5.7 Mandelbrot Set Rendering System Results
X-Axis : Tasks Distributed to the system
Y-Axis : Run Time in seconds for 1000 Frames with 1000 per pixel Iterations
RED :: MARS
BLUE :: Jack Rabbit 51

Chapter 1

Introduction

In any successful processor design, including multi-processor design, there are constraints and requirements that need to be met. Some of the most important requirements for mass market multi-processors are high performance, cost effectiveness and power efficiency.

A recent example of a successful multi-core processor is the Cell processor. Cell was targeted to be power efficient, to be cost efficient, and to accommodate a large consumer market. Under these constraints Cell was designed as a simple hardware platform without shared memory and with minimal caches and cache coherency. Figure 1 shows the basic Cell layout, which illustrates its master-slave design, in which the main PPE core must actively keep the attached SPE cores fed and busy with data and computation. Additionally, Cell has no hardware scheduler, so literally everything regarding the management of an application is left up to the software.

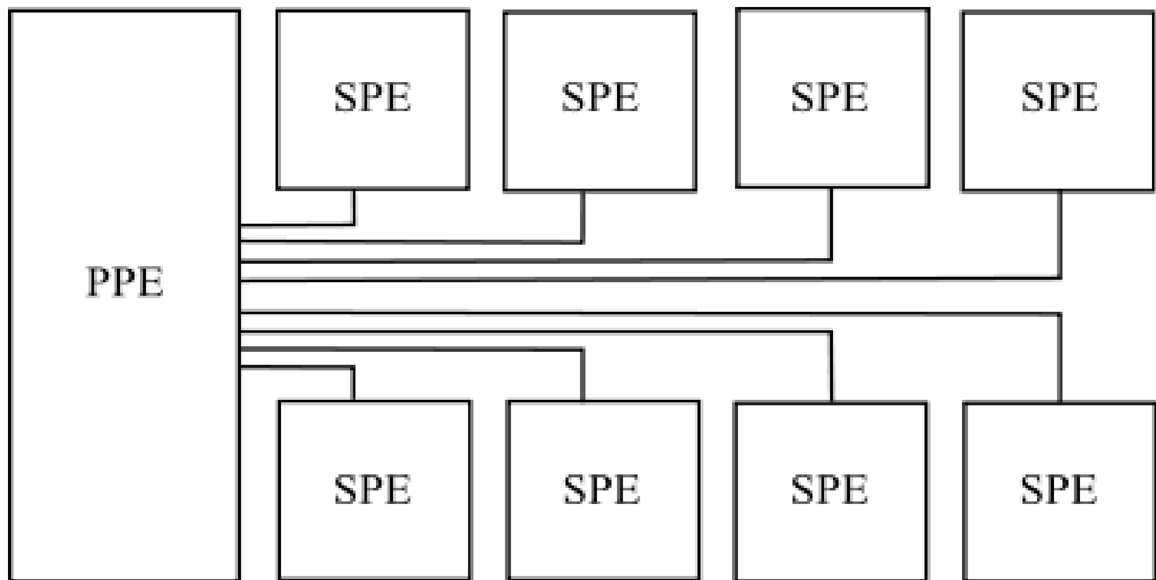

Figure 1.1: Cell Processor Block Diagram

This minimal hardware support puts great responsibility on the programmer and makes Cell difficult to program when trying to achieve good performance. The programmer is responsible for initiating the movement of code and data between main memory and the SPE cores via DMA, and the programmer is also responsible for relaying explicit information about the data and its location to the SPEs. DMA overhead is high, so to achieve good performance programmers need to make a small number of DMAs of large sizes, these DMAs should be latency masked, so their transfer times should overlap with computation. Moreover, to maximize the amount of computation overlapped with DMA latency, latency masking needs to be timed correctly and be at the right granularity. Scheduling of computation is completely left

to the programmer, and the mechanisms for launching computations onto the SPE cores are heavy duty and expensive. Programmers orchestrate scheduling and control code and data movement (including its granularity). Thus load balance becomes an issue with programming Cell, raising the complexity of software design for programmers. Furthermore there is no OS support for any of this, and all software layers on Cell must be built up from scratch.

As a response to the challenges of programming Cell a few notable systems have implemented software layers or runtimes. Each tries to achieve efficient parallelism by attacking a different one of the problems we have mentioned: DMA programming, thread management or scheduling, and load balance. Sequoia from Stanford addresses efficient parallel DMA programming and helps programmers to write bandwidth efficient programs [3]. MARS, by Sony, aims to avoid the overhead incurred by the basic Cell programming model with regard to scheduling thread execution on the SPE cores [7]. MGPS by Filip Blagojevic et al. improves Cell's load balancing with hybrid scheduling approaches [1]. These three systems each attempt to solve one problem with Cell programming, but none of them solves all three problems together, which is essential to achieving efficient parallel Cell programming.

We have implemented Jack Rabbit, which is a thread pool work queue programming abstraction for the Cell architecture that addresses all three of the above problems: DMA efficiency, thread management, and load balancing. The abstraction of a thread pool work queue allows programmers to map algorithms to our system more naturally. We have implemented support for

automatic multi-buffering, making latency masking easier and more efficient. The thread pool architecture of Jack Rabbit helps alleviate the overhead associated with scheduling threads onto SPEs. To handle load imbalance on Cell, we allow tasks to change their granularity at runtime. Our system combines solutions to several programming difficulties and significantly improves the programmer's ability to get good performance on Cell.

We evaluate the performance of our system stressing the three axes of DMA efficiency, threading overhead, and load balance. We evaluate our use of automatic multi-buffering in an LU factorization application, which is typically bandwidth bound. Our results show up to a 36 percent increase in performance when multi-buffering is turned on. To evaluate a more challenging application, we implemented a Barnes Hut simulation. In this implementation we ran against our own system using a single worker thread, and observe a nearly linear 5.96× speedup when using six cores as worker threads. This result shows that our system is highly efficient even for irregular codes. It also shows that our system can execute over 4 million tasks with minimal overhead. Lastly, we implemented and tested the effects of our load balancing on a compute dense application, in particular a Mandelbrot Set renderer. We compare our performance against that in the MARS system, and we find that run times are the same or better in our system compared to MARS for typical load balanced cases. Things become more interesting as the experimental loads become unbalanced. Under unbalanced loads, performance remains stable in our system but scales sharply down in MARS.

The remainder of this thesis is organized as follows. First in Chapter 2 we provide a detailed review of the literature on the related work. This review includes an in depth look at Sequoia, MARS, and the MGPS system. In Chapter 3 we provide background material, including a review of the Cell architecture, its programmability, and the models typically employed to program it. Thread pools are also reviewed in Chapter 3, which leads into a look at modern implementations of the thread pool work queue pattern, and in particular the system Sol MT, from which this thesis work is derived. In Chapter 4 we discuss the design and implementation of our system, and in Chapter 5 we provide an evaluation of it. Chapter 6 contains the directions of our future work, and Chapter 7 provides our conclusions.

Chapter 2

Related Work

2.1 MARS

MARS, the Multi-core Acceleration Run-time System, designed for the Cell processor, is an implementation of the thread pool work queue pattern, which is very useful on the Cell platform. The design goal was to keep the PPE burden low and simplify maximum utilization of Cell's SPE cores, which includes keeping the threading overhead low. This is both important and non-trivial for programmers. MARS was created by a research group within the Sony Corporation of America Developer Support division, and we believe it to be one of the best implementations available [7]. We also discuss a few areas of possible optimization in the system.

MARS provides a well known programming model with a familiar set of tools, and Cell programmers only need to map their application to this thread pool work queue pattern to get them running on Cell. The thread pool is implemented as SPE kernels, which run persistently until the system is torn down. Additionally the SPE kernel occupies a minute amount of the SPE local store, leaving room for the full SPE task and data. It handles the fetching of the SPE task from the PPE queue, and the SPE API provides several sup-

porting data structures useful to a task parallel environment. This includes barrier support, semaphores, and even a parallel FIFO queue. Using MARS, a programmer can focus on optimizing the SPE task program and synchronization, while ignoring the mechanics of Cell's low level communication and threading.

To keep the PPE usage low, the MARS designers chose to center the implementation around the SPE. The most powerful processor in the Cell is the PPE, and it is the easiest to target, so most developers use the PPE extensively in their applications. Keeping the PPE free is thus very important to developers, and MARS accommodates this with its SPE-centric design. The PPE runs a simple host program coded against the host API, while the available SPE cores run the actual application programs coded against the SPE API. The host program initializes the work queue and provides a way for the user to submit application programs to run on the SPE cores. SPE cores, accessing the work queue asynchronously, then retrieve and execute SPE programs without intervention of the PPE.

Maximizing utilization of the SPE is simple in MARS, because programmers keep the SPE cores busy without explicit thread management or threading overhead. In general the overhead for managing MARS tasks is much lower than that of managing individual threads. Good load balancing helps to keep the SPE cores busy, so the programmer must insure tasks are provided in sufficiently high parallel granularity. If this condition is met, the MARS scheduler works very efficiently, and SPE cores achieve high system

utilization. Threading overhead is also kept low for homogeneous tasks, because task code and data sections are cached by the MARS kernel. Thus, if the task granularity provides good load balance, the only bottlenecks in the system are heterogeneous task loading, or possibly the task itself.

MARS seems to meet the design goal of keeping the PPE burden low, but there are more ways to simplify and maximize the utilization of Cell's SPE cores. MARS achieves load balance if there is sufficiently high task granularity, because the tasks are so small that the load imbalance can only last a short time, which is usually negligible. However, a system can additionally take explicit measures to load balance unbalanced programs. We have implemented such mechanisms in the work presented in this thesis, and we show improvements over MARS's performance. Another significant source of inefficiency is DMA usage in Cell, so good performance requires computation latency masking. MARS leaves this completely to the programmer, but we do not. Finally, the load time for tasks not stored in the kernel cache is another source of inefficiency, but there are ways around this. For example, task's binary code sections can be double buffered on the SPE by setting aside space for two binaries on each SPE and double buffering the code sections in.

MARS is a good implementation of the thread pool work queue pattern, and it is a useful programming system. It achieves good efficiency with respect to maintaining PPE availability, simplifying SPE utilization, and doing away with some of the Cell multi-threading overhead. Yet as mentioned, we have examined a few areas of optimization in the system.

2.2 Sequoia

The Sequoia programming system by Fatahalian et al. of Stanford University targets programmability and performance on exposed-communication architectures [3]. On these architectures, programmers must explicitly manage data locality throughout the lifetime of an execution. These management techniques are important for the performance, and correctness of computations. Sequoia aims to facilitate bandwidth-efficient programming with a task based system and abstractions in place for describing a machine and the flow of data through its memory hierarchy.

There are two important abstractions made in the Sequoia system. The first abstraction is that Sequoia models a machine's memory hierarchy as a tree of memory modules. A memory module is simply a type of machine memory such as volatile RAM, on chip cache, or in the case of Cell, even local store. Program data in Sequoia flows through the memory hierarchy, or tree, of a processor. In Cell this hierarchy consists of main memory at the top level and the SPE local store at the bottom level. The second important abstraction is that a unit of computation in Sequoia is a task. Tasks are mapped onto the machine's memory hierarchy by the programmer. Together these abstractions allow for Sequoia to achieve portability of applications across different platforms.

Sequoia computation is based on tasks and arrays. Tasks in Sequoia are side-effect free with call-by-value-result (CBVR) parameters, and are invoked like functions. With the CBVR semantics variables are passed into the

task where they remain unchanged from the view of the invoking task until computation returns. Tasks are of two variants: inner and leaf. Inner tasks break down computation and data into leaf tasks which then execute. The main data type in Sequoia is the array, which decomposes into array blocks through the Sequoia API. This decomposition is useful as leaves can only run on a processor when their data fits in the lowest level of memory in the hierarchy. The Sequoia language provides a mapping function, which acts on an array by chopping it into a set of array blocks. These blocks are then typically distributed to the tasks.

Arguments for tasks are transferred on the SPE via DMA and transferred back out, also via DMA, after the task runs. Arguments have three types, IN, OUT, and INOUT. This is useful because it informs the system as to which variables need to be sent via DMA to the SPE (IN, INOUT), and which need to be sent back (OUT, INOUT). Since DMA is expensive, it is important to save time on data transfer, and knowing which arguments need to be transferred helps with this.

Blocked matrix multiply maps well to the Sequoia system. In the inner task, Sequoia's SGEMM divides up 2D matrix arrays into blocks. A parallel loop launches a series of map reduces onto the 2D arrays. This computation eventually results in leaf tasks being executed, which contain the regular three loop version of matrix multiply code. The map-parallel construct sets up parallel iteration over a loop, and the map-reduce construct maps a task onto an array decomposition, which reduces on the SGEMM product.

Sequoia includes the compiler and run-time system for the language. Both have been implemented for Cell BE and distributed memory clusters. The Cell version outputs two types of C files from a Sequoia program. For inner tasks, it produces a PPE source and for leaf tasks a SPE source. The run-time is event driven, and a single persistent thread runs on each core loading leaf tasks from the PPE and executing each.

The evaluation for Sequoia is promising, and they have achieved almost linear speedup from 1 to 16 SPE cores for Matrix Multiply or SGEMM. Their gravity simulation achieved over 3 billion interactions per second, and the HMMER implementation was faster than existing results published on modern graphics hardware. However, some of there evaluation applications, such as SGEMV, are bandwidth bound and scale poorly on Cell.

2.3 MGPS and EDLTP

Blagojevic et. al. published work on the Multi-Grain Parallel Scheduler (MGPS), the Event Driven Task Level Parallel (EDTLP) scheduler [1]. The main contribution of the work is that the schedulers deal with different types and levels of parallelism, and this achieves good load balance. This is important because Cell and other heterogeneous platforms indeed offer many types and levels of parallelism, and load balance is an issue. MGPS is adaptive and changes its parallel scheduling strategy based on the system state, while EDTLP is concerned mainly with efficient task level parallelism. An MPI based run-time system, uses these schedulers to map processes onto the

Cell processor's cores. The group evaluates the performance of the system on RAxML (Randomized Axelerated Maximum Likelihood), which is a computational biology method that uses Maximum Likelihood to generate large phylogenetic trees. Their strategy is very useful, and can even be generalized.

Each scheduler works together with the underlying run-time to offload computation to the SPE. Offload-able functions are designated with source annotations. The PPE is over-subscribed with 8 MPI threads, and a custom scheduler employing voluntary context switching replaces the MPI scheduler to reduce overhead. In the case of EDTLP the scheduler may select to either offload a given annotated function or to throttle offloading for that function. That decision is based on whether the function passes a granularity test, which is the evaluation of a cost function comparing the cost of running a task on an SPE with that of running the task on the PPE. To support offloading, the system maintains a PPE binary and an SPE binary for each function. The SPE binaries are distributed to the SPE cores ahead of time to reduce overhead.

There are two strategies for parallel distribution of work used in the system. Task level parallelism is handled by EDTLP, but actually both schedulers can handle this type of parallelism. Loop level parallelism is handled by MGPS only. Loop level parallelism is a way to offload iterations of annotated loops from SPE to SPE when SPE cores become idle. MGPS, supporting both task level and loop level parallelism, can decide to use either at run time. MGPS uses task level parallelism first, and this case is similar to EDTLP. Yet

if an SPE becomes idle during the computation, the MGPS system detects this and triggers loop level parallelism.

Loop level parallelism is not handled in EDTLP, and before MGPS was created, a statically hybrid scheduler was implemented called EDTLP-LLP. By observing the results they obtained on several computations of RAxML, the authors of the system found that making a static decision about when to use EDTLP or LLP yielded different results from one computation to the next. The EDTLP-LLP scheduler had to be re-tuned for different RAxML computations. MGPS was created to fit this need for an adaptive hybrid scheduler.

The results on MGPS and EDTLP influenced the work in this thesis, though we try to be more general. Their results on RAxML show that MGPS scheduling is always a win, but also that on large problem sizes the MGPS scheduler converges to EDTLP. This means that when the computation is large enough the granularity control doesn't have a big effect. On small computations where the tasks in the system are not so numerous, it can become necessary to trigger a change in the granularity of parallelism. Support for this type of multi-grain parallelism is key to load balancing and achieving high performance. We agree that granularity control should be adaptive, and we try to support a wider variety parallelization strategies. Nevertheless the results on EDTLP, EDTLP-LLP, and MGPS provide an important contribution.

Chapter 3

Background

3.1 Cell Architecture

The Cell Broadband Engine processor was developed by Sony, Toshiba, and IBM, and the architecture was designed to achieve better power and performance to area ratios, and better memory bandwidth. The Cell contains of a Power PC core called the PPE (Power Processing Element). Connected to the PPE via a high data bandwidth bus are 8 SPE (Synergistic Processing Element) cores. The cores in Cell - the PPE and SPE - are general purpose. The PPE is the more powerful core, with two hardware threads, and better branch prediction, while the 8 SPE cores are still powerful, but simpler in design. Both the PPE and SPE are in-order processors.

The PPE is made up of a 64 bit PPU (Power Processing Unit), and a 32KB L1 cache, supported by a 512KB L2 cache. There is support for two hardware threads on the PPU, and the chip features a dual issue pipeline. Instructions from both threads are interleaved to maximize usage of the dual issue support. The PPU is also a SIMD unit with a 128 bit vector vector scalar unit (VSU). The VSU supports four wide integer and floating point operations, 8 wide short ops, and 16 wide byte operations. It is also responsible for all

floating point operations vector or not. Four instructions are fetched on the PPU per cycle by the instruction unit (IU), and the execution unit (XU) is responsible for fixed point and load/store instructions. The PPE typically runs the operating system and handles the various IO operations.

The SPE consists of a SPU (Synergistic Processing Unit), a 256 KB local store , and contains module called the Memory Flow Controller (MFC). The SPE 256 KB local store is basically a large register file. The actual register file on the SPE is 128 entries of 128 bit registers. All instructions on the SPE are 128 SIMD instructions, and scalar instructions are packed for execution. The MFC gets and puts data to and from main memory, via direct memory access (DMA) commands. The SPE each support 16 outstanding DMA requests at a time. DMA commands on the SPE use the same address translation and page tables as the PPE. Because of this, data addresses can be send back and forth between the PPE and SPE for coherent transfer of data. The DMA commands are independent of the SPU and can move data and code during execution on the SPU. DMA commands can be issued by the SPE, and can also be triggered by the PPE. DMA commands for data sized at 16 bytes or less are atomic and all SPE cores view the transfer as such. Transfers of more than 16 bytes must be aligned on a 16 byte boundary, and be a multiple of 16 bytes in size [6].

3.1.1 Cell Programmability

The aspect of Cell that reportedly presents programmers with the most challenge is the local store and management of that memory [6]. The second challenge in programming Cell is the SIMD nature of the SPE. This can be avoided altogether by not programming with SIMD instructions, but this leaves out a lot of potential performance for applications. Branches also can lead to an expensive pipeline flush and eat away at performance, because Cell does not support branch prediction. Lastly only a single programming context is supported at one time on the SPE, and this can cause task switching overhead. Programmers who use good practices for Cell and have an understanding of the hardware and mechanisms of the processor, can deal with all these programming challenges, but in the common case these challenges are difficult ones to overcome for a programmer.

3.1.2 Cell Programming Models

In this section we look at popular models for programming Cell starting with the function offload model. That model views the SPE as an accelerator for use on sections of code where performance is important. Code to be offloaded is compiled with a separate SPE compiler and linked into the original PPE binary. At run time the PPE code will make a library call launching the SPE code. This programming model is supported by the IBM libraries libspe and libspe2. In the second library iteration libspe2, the call to launch the SPE code is blocking. This is typically avoided with the use the pthreads library

on the Linux operating system.

Another model, known as the computational acceleration model, is a more SPE centric way of programming Cell. In this model the most computationally intensive portions of a program are run on the SPE cores. The PPE then acts as control processor for the computation. Additionally computations can be partitioned to run in parallel on the SPE cores. However this requires careful tuning of the DMA commands involved in the computation for both performance and correctness. Shared memory and message passing algorithms can both be mapped to this model.

Finally, a streaming model can be used on Cell. For streaming, the SPE are set up as a serial or parallel pipeline. Each SPE would be one stage and would process all the data for that stage at each step. The data would then be passed down the pipeline for further computation. Each SPE could also contain each piece of the pipeline and get a parallel distribution of the data at each step. The PPE would then act as a stream controller orchestrating data in and out of the pipeline. There are many other programming models available for programming Cell, such as the Device extension model, the Shared memory multi-processor model, and the asymmetric thread run-time model. For a discussion see the work by Kahle et al. [6].

3.2 Thread Pooling

The idea of using a group of persistent threads which have the ability to execute different computational tasks at different times is referred to as

Thread Pooling, or the Thread Pool Pattern. Brian Goetz, author of "Java Concurrency in Practice," argues the usefulness of thread pools for server software architectures [4]. In the server environment many tasks arrive at the server for processing. The server may need to manage multiple tasks at one time. A natural way to think about doing this would be to launch a thread to handle each new task. However, thread spawning incurs overhead as does thread cleanup. Additionally, oversubscribing a processor or system will incur further overhead because of the context switching that will undoubtedly occur. These phenomena are not unique to a server environment. In parallel programming there is often also a need to divide work into parallel tasks. These tasks then need to run in some context or thread. The same problems thus arise when distributing tasks to a processor or system. Thread pools eliminate the need for constant thread set up and tear down. They also discard the need for heavy duty thread context switching.

Work Queues are often employed in league with thread pool programming. A work queue is simply a set of tasks. Tasks can be added to the set and removed, thus the term "queue." Generally task queues or work queues are implemented with a linked list and an associated monitor or locking discipline. Thus they are simply a specialization of a concurrent queue implementation. When a thread in a thread pool becomes idle it should go to the work queue and try to acquire some task to execute. This process is repeated as long as work is available.

Tasks and work queues can be represented any number of ways. In

Java, a task could simply be a class object implementing Runnable. In this scenario the work queue would be a class maintaining a synchronized list of task objects. Threads in the thread pool would then have a reference to this work queue. They would poll the queue for tasks and call the run method on any task polled. In the Boost library in C++ tasks are simply wrappers on functions invoked with the () operator. In MARS the Cell run-time, and tasks in our own system, are actually code and data sections parsed out of a binary ELF. Task descriptors that are also function objects are passed around until the time a task is executed. At this time a task's binary elf is loaded onto a processor by the pool thread and the entry point to the binary is invoked with the () operator.

Thread pools and work queues are normally associated with task parallelism. A task parallel computation is sometimes depicted as a directed acyclic graph or DAG. The edges in the graph represent dependencies in the computations, so tasks can be dependent on other tasks in the system. Dependencies introduce the possibility of deadlock. If a task is running and is dependent on a task still the queue to run, deadlock can occur. Supporting dependencies explicitly in the system, by allowing tasks to declare what they are dependent on, can help alleviate the risk of deadlock. However, this can complicate the implementation of the thread pool and the work queue. Another complication arises because there is sometimes need for the thread pool to support context switching. If a thread is allowed to submit tasks to the work queue, those tasks could be dependencies of the submitting task. It is common for run-

times to employ context switching mechanisms to allow a submitter to give up the thread while waiting for its submitted task to finish. This mechanism is commonly referred to as parent child wait. We will see this and other considerations being made in modern thread pool systems in the next section.

3.3 Sol MT our API

In the high performance graphics and parallel systems lab at The University of Texas at Austin, Sean Keely designed Sol MT, a software infrastructure for multi-core environments. The extension of the system to Cell processor is the topic of this thesis. This section now summarizes the original Sol MT system.

Sol MT, or Sol Multi-Threaded is a programming system including a programming model, API, and runtime system. The system was originally intended to support a high level of parallel efficiency on new architectures from Intel such as Larrabee. The API is based on a sophisticated thread pool and work queue implementation. The API includes objects, external work queue functions, internal thread pool functions, and scheduling functions.

The task object in Sol MT adheres to an interface specified by the `Work` class. When a user writes a task for the system to execute, the task needs to be coded as a class inheriting from the `Work` class. Pointers to the task can then be passed around the system as references to `Work` objects. The `Work` class is composed of four elements. First, the `execute` function is virtual and should be implemented by the user task. This function is called when the task

is dequeued from the work queue. Second, the `typeID` field of the `Work` class is an unsigned integer referring to the task's type. The user must set this so it's globally available and distinguishable for scheduling purposes. Third, the `Work` class contains data fields used to track dependencies. For example, if a work object has another object dependent on it, waiting for it to finish, this is signified by a `hasWaiter` field. Last, a child class of the `Work` class may optionally implement the virtual `Dice` function and return true from function named `Diceable`.

Tasks are introduced into the system via the `External Work Queue API`. The Sol MT work queue is a three level system. The top level buffer, the `Prefilter Buffer`, is posted to via the API. A pass is made over the `Prefilter` importing tasks into the second level buffer. This second level buffer is called the `Reorder Buffer (ROB)`. As its name implies, the contents of the `Reorder Buffer` may be reordered at run time by the scheduler. The `Ready Buffer` is the bottom level of the queue. It is filled with a subset of the contents of the `ROB`. The motivation behind this three level system is the reduction of contention and overhead. While tasks are posted at the top level, the middle and bottom layers remain available to the thread pool. Similarly, while the thread pool accesses the bottom level, the top and middle layers are available for scheduling.

A thread in the thread pool is referred to as a worker thread or a `worker`. A `worker` has two small thread local storage spaces for tasks called the `inbox` and the `outbox`. It is the job of a `worker` to move tasks from

the ROB to the Ready Buffer. It is also responsibility of a worker to fill its own inbox from the Ready Buffer. Note that if a task cannot be obtained through the Ready Buffer, a worker may employ work stealing to fetch jobs from other workers' inboxes. We return to this feature in the discussion of dependencies later. A worker can also create and post tasks into its outbox, and these tasks are flushed periodically to the Prefilter. A worker may optionally post a task locally to its own inbox, but of course a worker's main duty is to execute its tasks. This posting and flushing functionality is available through the Internal Thread Pool API.

Sol MT is also equipped with a scheduling sub-system as mentioned above. The schedulers in Sol MT are known as filters. They act on the top level buffer (Prefilter Buffer) and ROB to filter in new tasks to the existing work queue as set by the programmer. Filter procedures are passed into the system as function pointers through the API and are formed into the Filter Chain. The chain of procedures called by the scheduler may be reordered and filters may be removed by the user programmer. A Filter procedure in the Filter Chain must take the ROB and the Prefilter as arguments and return a new ROB, after emptying the Prefilter.

Dice is function called by the scheduler in the system. Dice is used to change the granularity of a task. For example, consider a task implemented as loop with independent iterations and a lot of computation per iteration. The iterations could actually be parallelized. This is known as loop level parallelism, this parallelism strategy may not be desired if there are multiple

such tasks in the system. Breaking each task up by breaking up the loops could flood the system with work and incur overhead for the numerous task invocations. With the original strategy, as the computation nears its end, the tasks will eventually run out. Yet it could be that some threads in the pool are not working while a handful of threads finish the last few tasks , and at this time it is useful to utilize loop level parallelism. `Dice` enables the system to make this decision when the queue is getting low to spread the large tasks out to each processor. The interface for `Dice` is in the `Work` class as mentioned above, and it takes a task, possibly returning more tasks of smaller granularity to represent the original task.

Along with the above mechanisms, programmers using Sol MT may wish to express mutual exclusion across tasks. Sol MT provides mutually exclusive tasks known as `Locked Jobs`. `Locked Jobs` are declared with mutex objects, and these mutex are acquired before the execution of these jobs. The system tries to acquire the mutex, and if it cannot, the task is put on a deferment list. In some computations this method of mutual exclusion can lead to performance gains. Instead of letting a task block trying to acquire a lock, another task runs in its place. However this could also lead to starvation. As always with mutex use, the computation could essentially be serialized. This occurs if the majority of the tasks require the same mutex.

Sol MT handles dependencies by allowing a worker thread to post a task locally to its inbox and then wait on the execution of that task to complete before resuming computation. This requires a software context switch and is

essentially a parent child wait pattern. Yet the implementation of parent child wait in Sol MT had to take into account work stealing, and that significantly complicates the issue. When a `worker` finds the work queue empty, it iterates over its fellow `workers' inboxes` looking for jobs to steal and import into its own inbox. Notice that in parent child wait a task is posted to a worker's inbox. With work stealing, this task may be stolen by another worker thread and this requires common knowledge throughout the system. Any worker completing a task must make sure to signal any other dependents in the system.

The overall design of Sol MT was adapted for the work presented in this thesis. We re-target to Cell, but keep many of the mechanisms and features of the Sol MT programming system. However, Sol MT was not designed for Cell. In the next section, we discuss the re-design of the run-time for the Cell BE. We discuss the design decisions made, and implications thereof, as well as how such a system was implemented for Cell.

Chapter 4

Design and Implementation

The basic design of Jack Rabbit was borrowed from the Sol MT API, but we will visit Jack Rabbit's major components here, including the Work type, the three level work queue and its locking discipline, the thread pool architecture, global barrier, the load balancing scheme called dicing, and double buffering. We also look at the design and implementation of the prerequisites for these features of the system - in particular a lock library, dynamic task loading and a parallel FIFO queue.

4.1 Work

Both the Jack Rabbit system and the Sol MT system revolve around the concept of `Work`, which is a class used to represent the executable SPE tasks in the system. The work object pointers are passed around the system via the work queue, and thus any new user defined type of SPE task must inherit from the `Work` class. The `Work` class definition is displayed in Figure 4.1. The SPE uses `workTypeId` to identify task types and locate binaries, and we discuss that later in this section. The member args is used to pass data into the tasks, and addresses of data structures located in main memory. `Work` also includes

```
class Work
{
private:
        void* args;
        unsigned long long workTypeId;
        bool doubleBuffer;
        short numBuffers;
public:
        uint64_t bufferAddrs[16];
        short bufferSizes[16];

        virtual inline unsigned int GetId() { return workTypeId; }
        virtual inline void* GetArgs(){ return args; }
        virtual inline bool DoubleBuffer(){ return doubleBuffer; }
        virtual inline short GetNumBuffers(){ return numBuffers; }

        virtual bool Diceable() = 0;
        virtual void Dice(unsigned int asking,Buffer<Work*>&) = 0;
        virtual ~Work(){};
}__attribute__((aligned(16)));
```

Figure 4.1: Work type class: C++ code snippet

data for double buffering, and `Dice` is the load balancing function used to split a task from one into many.

4.2 Work Queue

The work queue underwent three iterations of design and implementation. The first iteration formed the queue as an unbounded structure based on the STL list API, another iteration reformed the queue as a lockless bounded structure, and finally it became bounded and lock based. The first iteration came from the desire for a straightforward port of the Sol MT API. In that API, STL list is used to implement each level of the work queue. Push back and pop front were simple to design into the queue, but push front and pop back complicated things, as did the bracket access operator ([]). Designing these features into the queue meant having to store a mapping of logical indices to their actual values' locations. Regardless of the portability issues, we decided on a new design that would be simpler and more efficient, as it removed the need for two buffers for indices and values. We considered a design with a bounded queue using lockless push back and pop front methods. We dropped the requirement for the pop back and push front because they were not required for the core feature set we were borrowing from Sol MT. Thus we simplified the design of the queue class and reduced it's size. Dicing discussed in the last section on Sol MT, helps achieve load balance in Jack Rabbit but it was not easy to implement together with the original lockless design. In the dicing subsystem we have to pull tasks out the queue at arbi-

trary locations and replace them with multiple tasks logically representing the original task. Implementation of these operations is not straightforward with a lockless scheme, so we switched to a course grain locking scheme in order to accommodate this.

The basic work queue type `Buffer`, in Figure 4.2, is a template class, and takes a template parameter Type, typically of type `Work` pointer, and a `Size` parameter which defaults to 1k. Because the work queue is bounded, measures are taken from inside the system to ensure no overflow occurs. The class implements `push_back` and `pop_front`. Both the SPE cores and the PPE use `push_back` and `pop_front`. The implementation can be seen in the `Buffer` class diagram. `Push_back` and `pop_front` are used by the PPE when pushing task elements from the user program onto the back of the `PreFilter`, and also pushing them to the back of the `ROB` while popping them from the front of the `PreFilter`. Additionally `push_back` and `pop_front` are used by the SPE when the `Ready Buffer` is filled by the `ROB`, and when the SPE `inbox` is filled by the `Ready Buffer`.

4.3 Locking Discipline

The locking discipline for the work queue is nested, and in order to avoid deadlock all lock acquisitions follow the same order and all releases happen in the opposite ordering. When SPE acquire locks for the bottom two layers of the queue (the `ROB` and the `Ready Buffer`) they must acquire the `Ready` lock first followed by the `ROB` lock. To avoid deadlock they must first release

```
template <class C, unsigned int Size = 1024>
struct Buffer
{
        Buffer():head(0),tail(0),length(0){}
        void push_back(C c)
        {
                buffer[tail] = c;
                tail++;
                length++;
                if(tail==Size)
                {
                        tail = 0;
                }
        }
        C pop_front()
        {
                int retHead = head;
                head++;
                length--;
                if(head == Size)
                {
                        head=0;
                }
                return buffer[retHead];
        }
        int size()
        {
                return length;
        }
        C buffer[Size] __attribute__((aligned(128)));
}__attribute__((aligned(128)));
```

Figure 4.2: Work Queue base class Buffer: C++ code snippet

the `ROB` lock because another SPE cannot acquire the `ROB` lock without first acquiring the `Ready` lock. Thus the `Ready` lock is released last. The scheduler thread that does task insertion and dicing, acquires the `Prefilter` lock first and then the `ROB` lock, and releases in the opposite order after it acts on the queue.

4.4 Locks

The lock library contains a spin lock, and a primitive built on top of the spin lock, referred to as a manual split gate. The spin lock provides the methods `try`, `acquire`, and `release`. `Try` tries to acquire the lock once, succeeding and locking if the lock is free and failing otherwise. `Acquire` spins in a loop until try is successful, and `release` unconditionally sets the lock to open. The gate implements `enter`, `wait`, `open` and `close`. `Enter` increments the count of threads which have arrived at the gate. A call to `enter` is normally followed by a call to `wait`, which if the gate is closed, blocks until the gate opens. `Open` opens the gate manually, allowing those waiting to move on, and `close` manually closes the gate, causing all further calls to wait to block until `open` is invoked. Because the waiting on the gate is split between the methods `enter` and `wait` and the opening and closing are manual, we title this object a manual split gate.

As stated, the lock library revolves around the spin lock, so it is worth briefly examining the implementation. The Cell SPE has a coherent cache with four lines at 128 bytes each. Intrinsics are provided to use these caches

```
bool SpinLock::Try()
{
        uint32_t status=1;
        do
        {
                mfc_getllar((void*)localLockArea,globalLockEA,0,0);
                mfc_read_atomic_status();
                if(localLockArea[0]==1)
                        localLockArea[0]=0;
                else
                        return false;
                mfc_putllc((void*)localLockArea,globalLockEA,0,0);
                status=mfc_read_atomic_status() & MFC_PUTLLC_STATUS;
        }while(status);
        return true;
}
void SpinLock::Acquire()
{
        uint32_t status=1;
        do
        {
                mfc_getllar((void*)localLockArea,globalLockEA,0,0);
                mfc_read_atomic_status();
                if(localLockArea[0]==1)
                {
                        localLockArea[0]=0;
                }
                else
                {
                        while(localLockArea[0]==0)
                        {
                                DmaGet((void*)localLockArea,globalLockEA,8);
                        }
                        continue;
                }
                mfc_putllc((void*)localLockArea,globalLockEA,0,0);
                status=mfc_read_atomic_status() & MFC_PUTLLC_STATUS;
        }while(__builtin_expect(status,true));
}
```


Figure 4.3: Spin Lock Try and Acquire Methods: C++ code snippet

for atomic operations. The intrinsic `getllar` maps to a cache coherent DMA instruction known as get lock line and reserve. The SPE locks the bus and creates a reservation on a 128 byte data value in main memory, reading in the value to its local store. After modifying the value, the SPE can use `putllc` to put the value back with a locked line, conditional upon whether a reservation still exists.

The spin lock makes use of getllar and putllc in its try and acquire methods, see Figure 4.3. First an `getllar` command is issued on the 128 byte main memory address stored in `globalLockEA`. The value of the memory location is returned in `localLockArea`. Only the first integer element of `localLockArea` is used but it is a 128 bytes value. After checking the first element to see if the lock is free, the code either sees that the lock is free and changes its status to taken, or spins waiting for the lock to become free. A normal DMA command checks the value of the lock while spinning. This is faster than locking the line each time, and is similar to a test and test and set, or TATAS spin lock. Finally an `putllc` command is issued attempting to write the change back to the main memory location to acquire the lock. This command will fail if anyone else has changed the value in the meantime, so no writes are lost and atomicity is maintained.

4.5 Global Barrier

Jack Rabbit, like Sol MT, has a global barrier construct, which is implemented with the manual split gate, and a few API calls. Before the PPE

submits jobs, it must make the API call `BeginSubmit`. This closes the the `Completion Gate` which is of type manual split gate, and sets a boolean value `UserFinished` to false. When the user finishes submitting jobs, the API call `EndSubmit` sets the `UserFinished` boolean to true. It is then legal in the system to wait at the global barrier by calling `WaitForCompletion`, which causes the PPE to enter and wait at the `Completion Gate`. We meanwhile track the total number of jobs in the system with an atomic counter which is incremented on job submission and decremented on job completion. When the counter reaches zero after an SPE completes the last job, the `UserFinished` value is checked to see if the `Finished Gate`, also of type manual split gate, should be closed. When the `Finished Gate` closes the SPE workers all enter and wait on that gate, but the last SPE to enter, before waiting, will open the `Completion Gate` before waiting to signal the PPE that all jobs are done. The `WaitForCompletion` call will then return control to the calling user's code, having effectively been used as a global barrier.

4.6 Thread Pool

The job of the persistent thread pool is to load tasks from the concurrent work queue and execute them. In order to execute a task, the task's code and data sections are loaded onto the SPE. Using an elf parsing scheme observed from the MARS implementation, we acquire the data necessary to move the task over to the SPE. `Parse` stores the effective address of the code and data sections, and also the entry point of the task, in a structure. After

acquiring the task descriptor from the queue, the SPE can use a DMA to load the task from the queue into its local store. The SPE must also set aside this space in its local store for the task by explicitly moving down the stack pointer via a linker script. After loading the task, the entry point is cast to a function pointer variable, and the SPE can jump to the entry point of the task via the function call operator.

The SPE thread pool kernel functionality is encapsulated by the checkout procedure in Figure 4.4 and the `WORKJOB` macro. The checkout procedure maintains the lock discipline discussed earlier. Since the `Ready Buffer` is the most readily available location to obtain work from, SPE workers check the `Ready Buffer` first. The `Ready Buffer` may be empty but if not the SPE worker will fill its `inbox` with contents from the `Ready`. Otherwise the SPE worker will lock the `ROB`, and try to fill the `Ready Buffer` while holding both locks. Getting work from the `inbox` is simple as no locks are required. The `WORKJOB` macro's main function is call the `getWorkDMA` procedure, but before we examine `getWorkDMA`, we take a closer look at the use of the elf parser.

The elf parser runs on the PPE during the registration of jobs, and its core function is to get the entry point for the code, and the DMA addresses of the code and data sections in main memory. The header file `elf.h` provides all the structures needed to parse an elf image. The image is a large chunk of bytes accessible via an extern SPE program handle type variable named after a SPE binary file. These variables are typically located globally in the user's PPE program. Parse casts the image to a header type, and iterates through

```
bool Checkout(Buffer<Work*,MAILBOX_SIZE> &FillMe)
{
        FetchLock.Acquire();
        DmaGet(&Ready,args.Ready,sizeof(Ready));
        if(Ready.size() == 0)
        {
                ROB_Lock.Acquire();
                DmaGet(&ROB,args.ROB,sizeof(ROB));
                if(!FillReady())
                {
                        ROB_Lock.Release();
                        FetchLock.Release();
                        return false;
                }
                DmaPut(&ROB,args.ROB,sizeof(ROB));
                ROB_Lock.Release();
        }
        {
                int len=MAILBOX_SIZE - FillMe.size();
                if(Ready.size()<len)
                {
                        int size = Ready.size() / WorkerThreadCount;
                        if(size >= 1)
                        {
                                len = size;
                        }
                        else
                        {
                                len = 1;
                        }
                }
                for(int j=0; j<len; j++)
                {
                         FillMe.push_back(Ready.pop_front());
                         i++;
                }
        }
        DmaPut(&Ready,args.Ready,sizeof(Ready));
        FetchLock.Release();
        return true;
}
```


Figure 4.4: Checkout procedure excuted by the SPE thread pool kernels: C++ code snippet

```
uint64_t getWorkDMA(Work* Job)
{
        Work jobTemp __attribute__((aligned(128)));
        DmaGet(&jobTemp,(uint64_t)Job,sizeof(Work));
        int id = jobTemp.GetId();
        if(id == CachedId)
                return infos[id].entry;
        else
                CachedId = id;

        DmaGet((void*)infos[id].textVAddr, \
        infos[id].textEA,infos[id].textSize);
        if(infos[id].dataFound)
        {
                DmaGet((void*)infos[id].dataVAddr, \
                infos[id].dataEA,infos[id].dataSize);
        }
        return infos[id].entry;
}
```

Figure 4.5: getWorkDMA procedure executed by the SPE thread pool kernels: C++ code snippet

the program header segments, storing the entry point and effective addresses of the code and data in an info struct. For each job type the info structure is sent to the SPE workers.

Coming back to `getWorkDMA` displayed in Figure 4.5 we see that SPE first uses a DMA call to get the object associated with a task's `Work` pointers. It checks the job type id in the object, and prepares to use the info struct data to do a DMA to read in the binary of the job. If the task was just executed, the DMA of the binary is not necessary. After `getWorkDMA` is called, the entry

point for the job is cast to a function pointer and after a call to `spu_sync`, our SPE kernel jumps to the program code.

4.7 Load Balance with Dice

Load balancing occurs on the PPE during the insertion of tasks into the queue by calling the `Dice` function on tasks in the `ROB`. As in Sol MT the virtual `Dice` function of a task must be implemented in order for the dicing load balancer to work. Typically the dice function should split a task into numerous smaller tasks, equivalent to the original task, and the function should return these new tasks. The system inserts the new tasks in the location of the original task in the queue. These tasks are inserted in the order that they existed in the small buffer used to return the new tasks from the function `Dice`.

The dicer code displayed in Figure 4.6 operates by creating a new `Reorder Buffer` and then using system state to decide how to dice the `Reorder Buffer`. A new `Reorder Buffer` is created to hold the new diced jobs and the old jobs in their logical order. `Threshold` is the number of jobs per SPE that we determine will keep the system busy, and is meant to be an adjustable parameter, right now it is set to about 16. Along with the worker thread count we use `Threshold` to decide if we will dice at all. If we do need to dice we use `Threshold` and worker thread count again to decide how many jobs we need back in the system, and, assuming each job will be diceable, we calculate how many tasks to ask each of the original tasks for. Finally we

```
int size = ROB.size();
if(size<(THRESH_HOLD*WorkerThreadCount) && size>0)
{
        unsigned int need = (THRESH_HOLD*WorkerThreadCount) - size;

        unsigned int asking = need/size;

        int len = size;

        if(asking<=1)
                return;

        Buffer<Work*,1000> newROB;
        for(int i = 0 ; i < len ; i++)
        {
                if(ROB.front()->Diceable() && need > 0)
                {
                        if(need < asking)
                                asking = need;
                        if(asking==1)
                                asking = 2;
                        int oldSize = newROB.size();
                        ROB.front()->Dice(asking, newROB);
                        need-=newROB.size()-oldSize;
                        ROB.pop_front();
                }
                else
                {
                        newROB.push_back(ROB.pop_front());
                }
        }
        ROB.clear();
        ROB.splice(newROB);
}
```

Figure 4.6: Dicing sub-system run by the PPE scheduler thread: C++ code snippet

begin iteration over the `Reorder Buffer` dicing tasks and placing them along with non-dicable tasks into the new `Reorder Buffer`, which is spliced onto the original `ROB` at the end. The new `Reorder Buffer` is only needed because our simple `Buffer` class does not support insertion, and it would be slower anyways.

4.8 Double Buffering

Double buffering is a simple and useful mechanism that allows for automatic latency masking across independent tasks. Before running a task, we use asynchronous DMA to begin data transfers of a second task's data buffers, specified in the task descriptor object. When the first task finishes, we wait for completion of the second task's DMA transfers, and then the second task runs. The API describing a job has three members for double buffering in its interface. There is a count for the number of buffers to set aside, the size of each buffer, and pointers to the buffers themselves. The SPE threads can load two jobs at a time and start double buffering right away. The buffering can be turned on or off, as specified by the task descriptor. Up to 16 buffers can exist on a task, because of the number of DMA tags available for independent transfers. The buffer sizes can vary, but must adhere to the Cell DMA alignment and 16-byte-multiple transfer size limitations.

Chapter 5

Evaluation

Here we present our evaluation of the Jack Rabbit run time system's performance in the areas of DMA latency masking, scalable threading overhead, and load balance. All benchmarks were run on a single Cell BE processor in a Sony Playstation 3 console running Fedora Core Linux 8, and all development was done on a Fedora 9 PC with the IBM SDK for Multi-core Acceleration version 3.1. To evaluate DMA latency masking we implemented an LU Factorization algorithm because it is typically bandwidth bound, and we wanted to see the increases in performance due to our double buffering mechanisms. To evaluate our system's implementation with regard to scalable threading overhead we chose a scientific application, Barnes Hut, with large workloads that would require a significant number of tasks to be executed. We ran the Barnes Hut program using Jack Rabbit with only one worker thread and compare against the version using six worker threads. Last, for the evaluation of our load balancing mechanisms we first show that we are competitive in general with another well implemented system, MARS. We then go on to show that MARS, with no load balancing mechanisms, does poorly on unbalanced loads, while our system remains steady in performance.

```
for k = 1 to n  1
        broadcast {akj : j  mycols, j  k} in process column
        if k  mycols then
                for i  myrows, i > k        { multipliers }
                        ik = aik /akk
                end
        end
        broadcast { ik : i  myrows, i > k} in process row
        for j  mycols, j > k
                for i  myrows, i > k,
                        aij = aij  ik akj   { update }
                end
        end
end
```

Figure 5.1: Coarse-Grain 2-D Parallel Algorithm for LU Factorization [5]

5.1 Lu Factorization in Jack Rabbit

Double buffering is important in a system like Jack Rabbit because it allows programmers to parallelize their application into tasks while keeping DMA overhead low across those tasks. We chose LU Factorization as an evaluation application for our double buffering because it is similar to matrix vector multiplication, which is bandwidth bound as mentioned in the work by Kayvon et al. [3]. We used an algorithm from Professor Michael T. Heath's online lecture slides shown in Figure 5.1, adapted to our task queue architecture [5]. Since we could parallelize without concern for double buffering, our algorithm is fairly simple. Overall we wanted to show that in an application where bandwidth need is high, our double buffering API could alleviate some

Table 5.1: LU Factorization running with Double Buffering enabled and disabled on 1 - 6 working cores for 4k, 3k, 2k, and 1k sized Matrices
Top six rows: Original run times in seconds
Bottom six rows: Double buffered run times in seconds
%red refers to percent reduction of the original run time

Threads	4k	%red	3k	%red	2k	%red	1k	%red
6	67.27	NA	36.53	NA	17.27	NA	5.29	NA
5	72.66	NA	39.14	NA	17.72	NA	4.98	NA
4	82.88	NA	43.58	NA	18.83	NA	5.33	NA
3	101.41	NA	51.89	NA	21.39	NA	5.43	NA
2	137.23	NA	68.27	NA	27.20	NA	6.29	NA
1	248.01	NA	119.77	NA	45.14	NA	9.38	NA
M-Buffer	4k	%red	3k	%red	2k	%red	1k	%red
6	58.89	12.47%	30.91	15.40%	13.90	19.50%	4.04	23.64%
5	61.37	15.53%	31.49	19.55%	13.70	22.67%	4.09	17.81%
4	67.95	18.01%	33.94	22.12%	14.35	23.79%	4.00	24.85%
3	82.07	19.07%	39.34	24.18%	15.73	26.50%	3.98	26.64%
2	114.42	16.63%	52.81	22.64%	19.81	27.17%	4.65	26.21%
1	207.92	16.16%	93.93	21.58%	33.48	25.82%	7.01	25.25%

efficiency issues involved with parallelization.

The algorithm in Figure 5.1 computes the LU factorization of non-singular square matrix, but that factorization may or may not be found as we do not use pivoting. We adapt the algorithm to our system as follows. The outer loop in our algorithm iterates across all columns of the original matrix A. We parallelize the computation of a single column of multipliers by dividing the column up and distributing it to a small number of tasks larger than the number of SPE worker threads. These tasks compute the multipliers from the part of the column they are responsible for. The computed multipliers for

a column are stored in the L matrix. Then we do n^2 iterations, where n is the size of the matrix less the number of columns already multiplied through. During update we calculate the new value for the elements of the matrix A using the multipliers, storing the updated elements back in A. The matrix U thus overwrites the matrix A. We push all the update jobs for one of the outer iterations into the system at once and execute the update step in parallel. We launch the n^2 tasks to update the matrix and send a column of L and a column of A into each task. At the end of the launching of these update tasks we wait on a barrier to insure the multipliers are calculated from a fully updated matrix. A barrier is also needed at each iteration before the update tasks are launched to insure the completion of the multiplier calculation.

In the regular version the update tasks need to do two DMA transfers sized at the dimension of the matrix, while in the double buffered version these DMAs are removed and the buffers containing the respective columns of A and L are available through the task parameters. In addition in the regular version a DMA is required to put the column of A back into main memory. In the double buffered version this DMA of the column of A to main memory is launched but is not waited on until the barrier is reached at the end of the update phase.

Table 5.1 shows the run times and speedups for our LU Factorization evaluation and Figures 5.1 through 5.1 give graphical displays of the data. The table reveals that the scaling in our LU Factorization implementation is fairly poor. We attribute this to the serialism of the required computation and

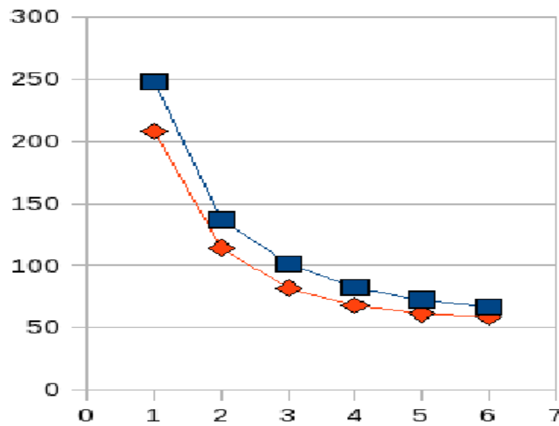

Figure 5.2: LU Factorization running with Double Buffering enabled and disabled on 1 - 6 working cores for a 4k sized Matrix
X-Axis : Cores Active as Worker Threads
Y-Axis : Run time in seconds

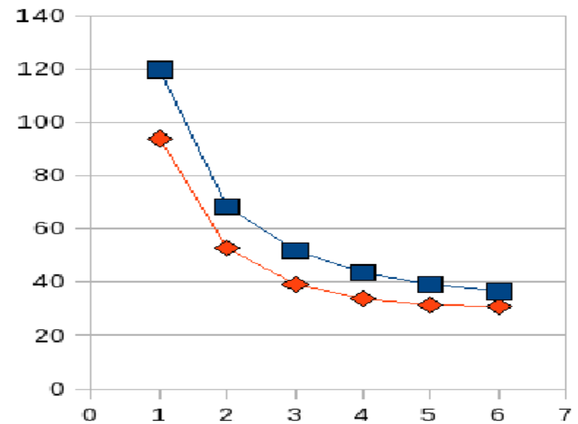

Figure 5.3: LU Factorization running with Double Buffering enabled and disabled on 1 - 6 working cores for a 3k sized Matrix
X-Axis : Cores Active as Worker Threads
Y-Axis : Run time in seconds

the data dependencies within LU Factorization, which is not an embarrassingly parallel algorithm, as discussed in Lewis' article [8]. Double buffering however, is shown to be effective. The speedups due to double buffering get larger as the size of the matrix gets smaller. The worst case speedup shows up in the six thread version with a 4k matrix. During that run double buffering incurs only a 12% reduction in the original run time. The best case occurs when the matrix is 1k and double buffering incurs a 26 percent reduction of the original run time. This is accounted for by the fact that the DMA transfer latency is not getting smaller, but the matrix is, and the compute time is going down with the matrix size. As the DMA time starts to dominate the task execution

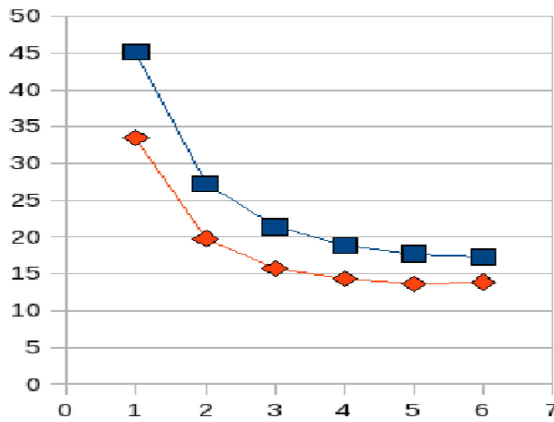

Figure 5.4: LU Factorization running with Double Buffering enabled and disabled on 1 - 6 working cores for a 2k sized Matrix
X-Axis : Cores Active as Worker Threads
Y-Axis : Run time in seconds

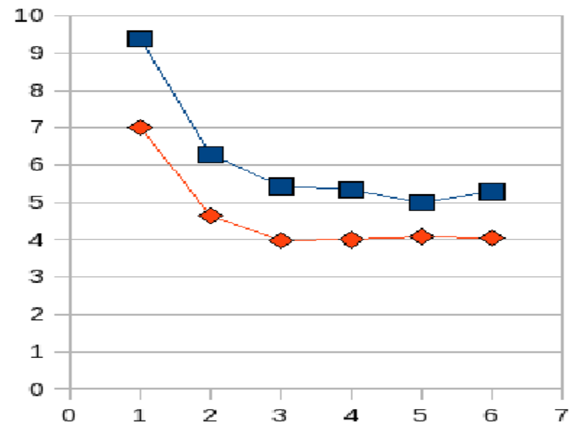

Figure 5.5: LU Factorization running with Double Buffering enabled and disabled on 1 - 6 working cores for a 1k sized Matrix
X-Axis : Cores Active as Worker Threads
Y-Axis : Run time in seconds

time, double buffering has a bigger effect. Double buffering is thus valuable for bandwidth bound applications where compute time is near DMA time. This effect is important because a small compute to bandwidth ratio is typical to a large number of applications including image processing and large matrix manipulations. However, the affects of double buffering begin to dwindle once compute starts to overwhelm DMA time. As an aside this effect is made more clear in the next sub section on Barnes Hut.

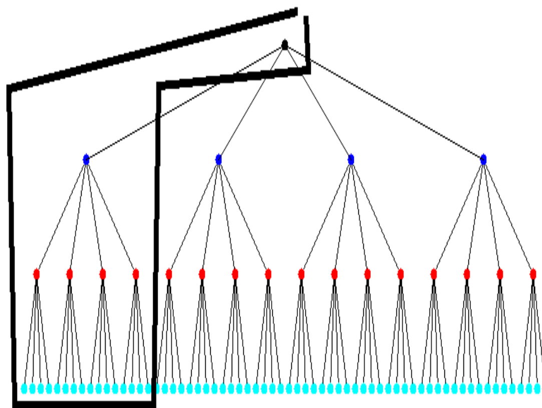

Figure 5.6: Quad Tree example image. The outlined part of the tree might be passed into one task etc.

5.2 Barnes Hut Simulation in Jack Rabbit

We implemented the Barnes Hut solution for a 2 dimensional n-body simulation because we wanted to show how a scalable scientific application would run within our framework. We use a quad tree to achieve the n log n speedup of Barnes Hut, with a serial builder, while our simulation step is parallel. We outline our algorithm, look briefly at the data structures used, and discuss results of our Barnes Hut computations and their implications.

Our algorithm works as follows. We split the tree up into sections based on its depth first traversal order. So the first split would contain the root node, its left child and possibly the sub tree of nodes beneath, see Figure 5.2. We then designate a set of bodies to accompany a split of the tree into the SPE task. Each SPE task will compute the update of the bodies it is responsible for against its split of the tree. Then the results will then be sent back to

main memory, and a barrier insures that computation finishes for all tasks. A rotation of the sets of bodies across the different tree splits then occurs on the PPE. This rotation is implicit and is done by simply sending the j-1th task's body set to the jth task, and relaunching all tasks. When all the update tasks finish, a series of move tasks are launched to complete the n body simulation computation for given time step.

The building of the quad tree is done on the PPE and it is a simple depth first recursive implementation. However, instead of using a pointer tree, we pack our nodes into an array, in depth first order, so that we can ship them easily to the SPE and the SPE can traverse down a portion of the tree depth first using a loop. The tree nodes each contain a left and right child index into the array and a next field which contains the index of the next sibling node. Next is used to determine where to index into the array in the case that a node need not be traversed further and rather should be approximated as per the multipole acceptance criteria calculation. In Barnes Hut the multipole acceptance criteria is often used to determine whether updates against a given subtree's bodies can be approximated as a single update on the subtree's bodies' center of mass. We make a static decision about leaf size and recurse until this is met. As we are testing scalability not robustness, we ignore the fact that floating point error may cause a tree to be unbuildable in some cases. This never occurred in the 3000 plus iterations and tree builds we employed for our evaluation.

Table 5.2: Barnes Hut running with Double Buffering enabled and disabled on 1 - 6 working cores for 10000, 100000, and 1000000 bodies
Top six rows: Original run times in seconds
Bottom six rows: Double buffered run times in seconds
Speedup refers to the parallel speedup

Threads	10K	Speedup	100K	Speedup	1M	Speedup
1	1.11	1	60.16	1	5878.82	1
2	0.59	1.88	30.19	1.99	2942.93	1.99
3	0.4	2.77	20.21	2.97	1964.41	2.99
4	0.31	3.58	15.3	3.93	1474.81	3.98
5	0.27	4.11	12.29	4.98	1182.49	4.97
6	0.24	4.62	10.33	5.82	985.79	5.96
M-Buffer	10K	Sp	100K	Sp	1M	Sp
1	1.11	1	59.63	1	5422.53	1
2	0.57	1.94	29.96	1.99	2914.19	1.86
3	0.4	2.77	20.05	2.97	1945.34	2.78
4	0.31	3.5	15.19	3.93	1460.59	3.71
5	0.26	4.26	12.2	4.89	1170.11	4.63
6	0.24	4.65	10.25	5.82	976.55	5.55

After the tree is built our algorithm launches numerous tasks, each responsible for a batch of tree nodes and a batch of bodies to update. The update phase will loop through the tree for each body, updating its properties until it gets to an index that is not in its local section of the tree. When this completes, the body set is sent back to main memory, the barrier is met, and the rotation occurs before the next set of tasks is launched. When all bodies have interacted with all other bodies, the move tasks are launched. Each of these tasks is responsible for doing a DMA of 512 bodies and moving them based on the leapfrog verlet integrator algorithm borrowed from the CS380P course on Parallel Systems taught by Calvin Lin at the University of Texas at Austin. The algorithm was borrowed originally from the Cuda SDK.

We ran our Barnes Hut Simulation on several working sets: 10000 Bodies for 100 iterations, 100000 bodies for 10 iterations, and 1000000 bodies for 1 iteration. We run each of these working sets in Jack Rabbit with 1, 2, 3, 4, 5, and 6 threads. We implemented a version that uses our double buffering API and one that does not. Table 5.2 shows the run times for each of the cases with and without double buffering along with the scaling at each point. The best case is the non-double buffered version of the 1000000 particle working set. We observe a 5.96 parallel speedup here. The worst case is the 10000 particle working set with double buffering.

We attribute the differences in scalability to the reduction of overhead for synchronization and the decrease in load imbalance. In the million body case more tasks are executed per synchronization cycle than in the 10K case.

This is what is meant by reduction of overhead for synchronization. Also load imbalance between synchronization points may be higher in the 10K case where there are only about 20 tasks are executed. Load imbalance matters less in the million body case, when about 2000 tasks are put into the system before synchronization between rotations of body sets.

As mentioned in the last section, double buffering has little effect on computations like Barnes Hut where the compute time is much larger than the DMA time. We see in table 5.2 that the over all performance is always increased with double buffering. However the performance gains are not substantial.

5.3 Mandelbrot Set Rendering: MARS vs Jack Rabbit

The Mandelbrot rendering application was chosen for evaluation of load balancing in Jack Rabbit because we could easily compare performance numbers with MARS, as a Mandelbrot rendering application was provided in their sample suite. We were able to easily port the program over to our system as the interfaces of Jack Rabbit and MARS are both of the task queue thread pool nature. The run times are about even in both systems, but when the load gets unbalanced our dicer kicks in and stabilizes our run times as opposed to MARS whose run times get progressively worse. Figure 5.3 shows the run times as function of the number of tasks used in the system.

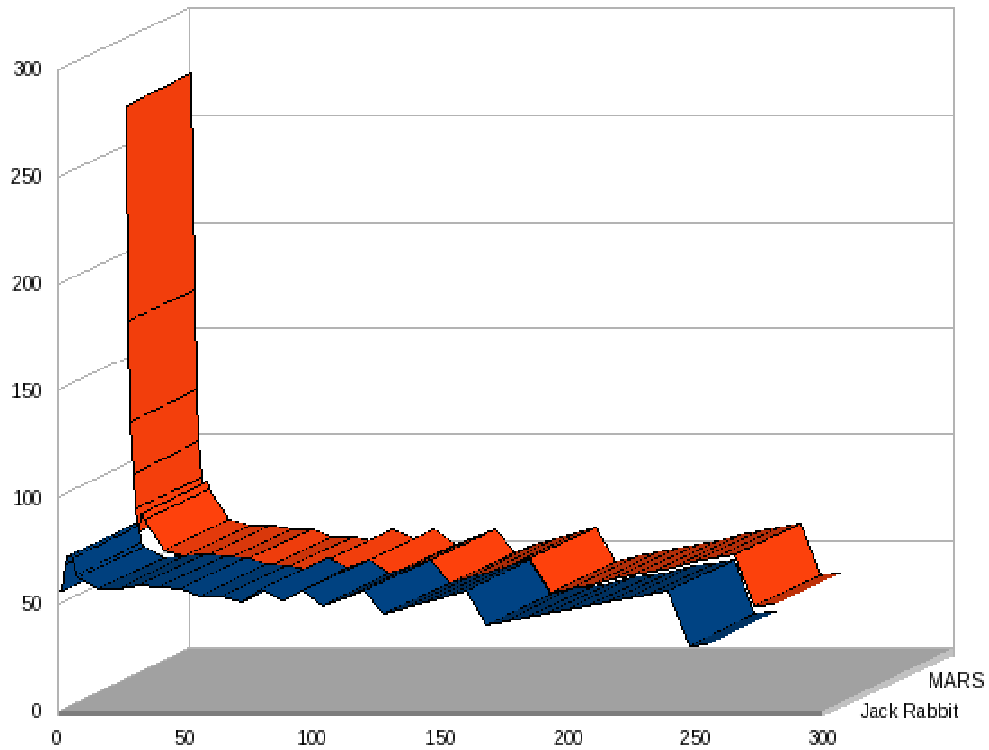

Figure 5.7: Mandelbrot Set Rendering System Results
X-Axis : Tasks Distributed to the system
Y-Axis : Run Time in seconds for 1000 Frames with 1000 per pixel Iterations
RED :: MARS
BLUE :: Jack Rabbit

The Mandelbrot set rendering parameters were set up by the MARS sample implementation and we leave these alone with the exception of making the task granularity parameters variable. The renderer does a max of 1000 per pixel iterations rendering the fractal into a 640 x 480 image. One thousand frames are rendered per run and a duplicate gray scale image is produced for each fractal image per frame. All computations are done by the SPE tasks and

the image is broken up into 128 tasks by default. Each task renders a number of rows of pixels equivalent to the height of the image divided by the task count. Figure 5.7 has a jigsaw pattern because after the division of the height by the task count the floating point result is rounded to determine how many lines will be rendered by each task. We edit the source for the Mandelbrot application in both MARS and Jack Rabbit versions and allow the number of tasks to change. We justify this with our belief that in a large dynamic system the number of tasks of a given type may be of variable number and compute density may vary. This could be the case in a task based rendering engine or physics simulation. It is important to show that our system is adaptive in these cases.

We draw attention to the fact that the run times in Jack Rabbit are faster than MARS in some cases and slightly slower than MARS in other, but the delta between the two is always fairly small. When Jack Rabbit is faster, over runs where tasks counts are above 144, it is up to 10% faster. However, when MARS is the fastest, over the runs with less than 136 and more than 24 tasks, it is only up to 3% faster. The fact that the run times are similar is important because it qualifies our system as a competitor when load balance is not an issue. Over the fine grain loads where Dicing is is not invoked by our system, we are faster than MARS on every run. Over the runs where Dicing is invoked but load balance is not an issue we see that Jack Rabbit is incurring some dicing overhead. The biggest delta in these cases is at 40 tasks where Jack Rabbit's run time for one thousand frames is 57.9 seconds or $\tilde{1}$7.3 FPS,

and MARS's is 55.9 seconds or ~17.8 FPS. So we have fallen behind by roughly half a frame at this point, however the largest over all delta is at 256 tasks when Jack Rabbit is running with little to no load balancing overhead. In this case we run at 31 FPS and MARS is at 28 FPS, this is a much larger delta.

Lastly, we take a look at the difference in the system run times that constitute the spike in performance decrease by MARS. It would be unfair to only pay attention to the last few cases where the task number is smaller than the SPE number. It is obvious that this will result in a win for our system as MARS performance would no doubt eventually reduce to that of the single threaded version in the 1 task case. However we do show this in the graph to emphasize the difference that is made by having a few large tasks enter the system with out any load balancing. This is the case in the 1 task version through the 24 task version. We also wanted to see how bad the performance degraded in our system due to load balancing overhead incurred by the dicer. This evaluation yielded good results, for example in the single task submission case, where the dicer does all the parallelization, our performance does not degrade at all, but rather increases by a fragment. This is because it is faster to push a single job into the system, acquire the Prefilter lock once, and let the dicer do all the parallelization directly into the ROB, than it is to acquire the Prefilter lock at every job post and spuriously inhibit the scheduler from moving newly posted jobs into the ROB due to the lock not being available.

Dicing is a powerful tool for controlling load balance, especially when load is varying dynamically. We have synthesized this dynamic variance in

our benchmark, but it could occur in a variety of applications. For instance in Barnes Hut we mentioned that the 10000 body case could incur more load imbalance between jobs. So if the number of bodies were to vary as in some particle simulations, load balance would be an issue. Also, if a given update job takes a long time because the tree is unbalanced, the tasks in the system would run low, but the dicer could handle this case. Finally, it is important for the dicer not to incur overhead if it is not needed, and the Mandelbrot runs where the task count is above 144 show that we do not incur dicing overhead.

Chapter 6

Future Work

6.1 Locality Control

The Jack Rabbit scheduler, like that in Sol MT from which it is derived, provides an opportunity for locality control. Since the user is allowed to influence the way the scheduler reorders tasks, he could use it to reorder them based on data location. This enables a locality control scheme as follows.

The first step is to require a task to declare its data structures locations up front. The run time would then receive a large group of jobs to do, and the scheduler would sort them into groups according to their data requirements. The worker threads would then check to see if they had a given task's data in their thread local memory, perhaps because a task that used the same data just executed, before fetching a task. This seems like a good idea, however a natural question to ask about this scheme is, why not just use double buffering. The simple answer is that on some architectures asynchronous DMA is not supported, but even on Cell DMA is limited by the number of separate transfers that can be double buffered at one time. Double buffering could be used in league with the proposed locality scheme and this would mean even more data could be local by the time a computation runs.

6.2 A New High Level Language

Future work will also try to address the short comings of our system, while building on the things our system does well. The thread pool works well with our line of irregular and scientific, and regular compute intensive applications. However the run time code is not as smart as it could be. By using a language we could make certain assumptions that would allow the run time to make smarter decisions about managing programs. As our results are very positive we feel that the programming model and mechanisms of our system would be good to base a language around. The language would provide more flexibility to the run time system, and also provide more optimization opportunities

As a more concrete example of what we are discussing, consider a language like ZPL, where there are essentially no pointers, and arrays are the main data type [2]. The fact that all data structures are based on arrays is a very useful assumption to make, and is not too restrictive in our case as, in our Cell application suite, almost all the data sent to the SPE is already in array form. Also our double buffering works on a buffer or array system. In a language with these restrictions task data could possibly be set up for double buffering, with no programmer effort. In an array language with some construct for independent loops it could also be possible to automatically dice a task based on its loop iterations, and the array bounds involved. Not only may it be possible to dice and double buffer tasks automatically, but within the constraints of the right language, code could be written sequentially, diced,

and effectively re-spliced into SIMD code for efficient lane programming.

Chapter 7

Conclusion

The work presented in this thesis provides evidence that our system, Jack Rabbit, is effective for programming processors like Cell where raw machine mechanisms such as DMA and thread management are exposed to the programmer. Our system provides a software layer for programming Cell using the thread pool work queue pattern, which is familiar to multi-core programmers. We implement a DMA double buffering API to help with latency masking, and our high performance and low overhead system is more efficient than programming the system's transient threads. We also supply the programmer with load balancing mechanisms which are useful and provide high performance.

Within our system programmers can focus on parallelizing applications without having to worry about managing the hardships of programming Cell. Our double buffering API, thread pool architecture, and load balancing mechanisms help to achieve this. With our double buffering API programmers can count on having low latency for loading data into tasks. This is important in a thread pool work queue model because it enables programmers to break up code and data without worrying about the implications on latency masking.

Of course double buffering needs to be tuned. In fact we observe that DMA latency should be near compute time to achieve good speedup as in the case with our LU Factorization application. Interestingly, making tasks smaller to get the compute time down to something around the DMA time means creating more tasks, and this leads into the second notable feature of our system. Our threading overhead is very low and allows for good scaling with lots of tasks. Threading overhead being low not only supports double buffering, but also supports compute intensive embarrassingly parallel applications like Barnes Hut, as shown in our Barnes Hut scalability results. Lastly on Cell load balance is a concern, and there should be mechanisms for achieving good load balance. To this end, we have provided dynamic load balancing controls and have demonstrated their effectiveness in our Mandelbrot set rendering application.

In summary we implement a system providing programmers with low threading overhead, an easier way to achieve DMA efficiency, and load balancing mechanisms. Whereas other systems only handle programming difficulties in some of these areas we show improvements in each area. Thus we believe consideration of a Jack Rabbit-like programming system may be desirable in future parallel system research efforts.

Index

A New High Level Language, 56
Abstract, vi
Acknowledgments, v

Background, 14
Barnes Hut Simulation in Jack Rabbit, 46
Bibliography, 61

Cell Architecture, 14
Cell Programmability, 16
Cell Programming Models, 16
Conclusion, 58

Dedication, iv
Design and Implementation, 25
Double Buffering, 39

Evaluation, 40

Future Work, 55

Global Barrier, 32

Introduction, 1

Load Balance with Dice, 37
Locality Control, 55
Locking Discipline, 28
Locks, 30
Lu Factorization in Jack Rabbit, 41

Mandelbrot Set Rendering: MARS vs Jack Rabbit, 50
MARS, 6
MGPS and EDLTP, 11

Related Work, 6

Sequoia, 9
Sol MT our API, 20

Thread Pool, 33
Thread Pooling, 17

Work, 25
Work Queue, 27

Bibliography

[1] Filip Blagojevic, Dimitrios S. Nikolopoulos, Alexandros Stamatakis, and Christos D. Antonopoulos. Dynamic multigrain parallelization on the cell broadband engine. Principles and Practice of Parallel Programming, 2007.

[2] Bradford L. Chamberlain, Sung-Eun Choi, E. Christopher Lewis, Calvin Lin, Lawrence Snyder, and Derrick Weathersby. Zpl: A machine independent programming language for parallel computers. IEEE Trans. Software Eng. 26(3): 197-211, 2000.

[3] 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. Supercomputing, 2006.

[4] Brian Goetz. Thread pools and work queues. `http://www.ibm.com/developerworks/java/library/j-jtp0730/index.html`, 2002. Retrieved on 4-25-2011.

[5] Professor Michael T. Heath. Parallel numerical algorithms: Chapter 6 lu factorization. Department of Computer Science, University of Illinois at Urbana-Champaign, Class slides for CSE 512 / CS 554.

[6] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. IBM Journal of Research and Development, 2005.

[7] Geoff Levand. Multi-core application runtime system for the cell broadband engine. ftp://ftp.infradead.org/pub/Sony-PS3/mars/presentations/MARS-SIGGRAPH-2008.pdf, 2008. Retrieved on 4-25-2011.

[8] Brad Lewis and Kris Richards. Lu factorization case study using fast: Dataflow parallelism with the forte application scalability tool. http://developers.sun.com/solaris/articles/FAST/lu_content.html, 2010. Retrieved on 4-25-2011.

Vita

Apollo Isaac Orion Ellis was born in Berkeley, California on 21 June 1981, the son of Gilbert Ellis and Linda Diane Reed. He received the Bachelor of Arts degree in Computer Science from the University of California at Berkeley in May 2008. He interned at Sony Playstation in Foster City, California, where he worked on the Cgc compiler for the Playstation 3 SDK. He applied to the University of Texas at Austin for enrollment in their Computer Science Master's Degree Program, was accepted, and started graduate studies in August 2008. During his academic graduate career at UT Austin he was a part of the lab for High Performance Graphics and Parallel Systems managed by Professors Donald S. Fussell and Calvin Lin. He returned to Sony Playstation as an intern in 2010. During this internship he worked with the research team on the Playstation Move motion controller. His interests lie in the areas of parallel systems, computer graphics rendering, and game engine development.

Permanent address: 423 Scottsdale Rd.
Pleasant Hill, California 94523

This thesis was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.