Performance Simulation with the Coconut Multicore Framework for the Cell/B.E.

Performance Simulation with the Coconut Multicore Framework for the Cell/B.E.

By

Kevin Browne B.Sc. (Hons)
IBM Center for Advanced Studies Fellow

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree
Master of Science

McMaster University
© Copyright by Kevin Browne, September 21, 2009

MASTER OF SCIENCE(2009)
COMPUTING AND SOFTWARE

McMaster University
Hamilton, Ontario

TITLE: Performance Simulation with the Coconut Multicore Framework for the Cell/B.E.

AUTHOR: Kevin Browne B.Sc. (Hons)(McMaster University)

SUPERVISOR: Dr. Christopher Anand

NUMBER OF PAGES: xvii, 136

LEGAL DISCLAIMER: This is an academic research report. I, my supervisor, defence committee, and university, make no claim as to the fitness for any purpose, and accept no direct or indirect liability for the use of algorithms, findings, or recommendations in this thesis.

Abstract

The multicore revolution in chip design has fundamentally altered the demands placed on developers. Thread-level parallelism is critical to optimizing software performance on multicore chips. However thread-level parallelism presents challenges with respect to optimization, safety and program representation. Program models and compiler technologies must act as a bridge from applications to efficient hardware usage.

Coconut (COde CONstructing User Tool) is an ongoing project at McMaster to develop a platform for experimenting with novel ideas in reliable and high performance code generation, currently targeting the Cell/B.E.. The Coconut Multicore Framework uses a virtual machine abstraction layer to model multicore layer parallelism on the Cell/B.E.. The abstraction creates a correspondence between ILP and multicore layers of parallelism. The abstraction also allows us to perform efficient static analysis of virtual machine programs; with this ability we have developed a tool to automatically check for parallel bugs in linear time with respect to the atomic virtual machine instructions.

In this thesis we will discuss the creation of a performance simulation tool developed to simulate the execution of our virtual machine instructions on a Cell/B.E.. The tool has scalability to future many-core architectures, due to its linearly bounded runtime complexity. The tool allows for Coconut developers to contrast the performance of different scheduling algorithms. It provides meaningful feedback as to optimization opportunities by identifying data transfer latencies which cause execution to stall. The design and performance testing results of the performance simulation tool are presented.

Acknowledgments

I would like to thank my supervisors Dr. Christopher Anand and Dr. Wolfram Kahl for their wonderful support and guidance throughout my degree. Thanks to Dr. Anand in particular for encouraging me to discuss my research at many conferences and workshops. Thanks to Dr. Kahl in particular for introducing me to the wonderful Monad.

I would like to thank the IBM Toronto Lab Center for Advanced Studies, and Robert Enenkel, for their support of my research work thus far.

I would also like to thank my fellow students in the Coconut project. In particular Shiqi Cao for teaching me how to Haskell properly, and Gabriel Grant for coding the wonderful Runtime System.

I would also like to thank my parents, family and friends for their support and encouragement.

Contents

Abstract iii

Acknowledgments v

List of Figures xi

List of Tables xiii

Acronym Index xv

1 Introduction 1

1.1 Document Structure . . . 3

2 Parallel Computing 5

2.1 Motivations . . . 5

2.1.1 Performance Increases . . . 6

2.1.2 Frequency, Memory and Power Walls . . . 7

2.2 Flynn's Taxonomy . . . 9

2.2.1 Single Instruction Single Data (SISD) . . . 10

2.2.2 Single Instruction Multiple Data (SIMD) . . . 10

2.2.3 Multiple Instruction Single Data (MISD) . . . 11

2.2.4 Multiple Instruction Multiple Data (MIMD) . . . 11

2.3 Parallelism Levels . . . 13

2.3.1 Bit-level Parallelism . . . 13

2.3.2 Instruction-level Parallelism . . . 14

2.3.3 Thread-level Parallelism . . . 17

2.3.4 Process-level Parallelism . . . 18

2.4 Task Parallelism vs. Data Parallelism . . . 20

2.4.1 Task Parallelism . . . 20

2.4.2 Data Parallelism . . . 20

2.5 Memory Classifications . . . 20

2.5.1 Shared Memory 21
2.5.2 Distributed Memory 21
2.5.3 Distributed Shared Memory 22
2.5.4 Uniform Memory Access (UMA) 22
2.5.5 Non-Uniform Memory Access (NUMA) 22
2.6 Processing Element Communication 22
2.6.1 Computation Bound vs. Communication Bound 23
2.6.2 Processing Element Coupling 24
2.6.3 Parallelism Granularity 24
2.6.4 Performance Measurement 25
2.6.5 Network Topologies 26
2.7 Performance Measurement 26
2.7.1 Floating Point Operations Per Second (FLOPS) 26
2.7.2 Benchmarks 27
2.8 Multiprocessor Parallelism Software Challenges 27
2.8.1 Program Models 28
2.8.2 Program Correctness 29
2.8.3 Optimal Scheduling 31

3 Cell Broadband Engine **37**
3.1 Origins 37
3.2 Hardware Overview 38
3.3 Processor Element Design 39
3.3.1 PowerPC Processor Element (PPE) 40
3.3.2 Synergistic Processor Elements (SPE) 41
3.4 Communication Architecture Design 42
3.4.1 Design Overview 42
3.4.2 EIB DMA Transfer Behaviour 45
3.5 Communication Architecture Performance Analysis 49
3.5.1 Main Memory Bottleneck 49
3.5.2 Communication Pattern Bottleneck 52
3.5.3 Performance Tests 54
3.6 Overcoming the Frequency, Memory and Power Walls 57
3.6.1 Frequency Wall 58
3.6.2 Memory Wall 58
3.6.3 Power Wall 58
3.7 Current and Future Variations 58
3.7.1 Cell/B.E. 59
3.7.2 PowerXCell 8i 59
3.7.3 SpursEngine 59

3.7.4 Cell/B.E. 32 SPE Concept Design 60
3.7.5 Dual Processor Systems 60
3.8 Applications 60
3.8.1 Video Game Consoles 61
3.8.2 Cell/B.E. Blade Servers 61
3.8.3 Supercomputing 62
3.8.4 Cluster Computing 63
3.8.5 Grid Computing 63

4 Cell/B.E. Program Models, Frameworks and Solutions 65
4.1 Accelerated Library Framework (ALF) 65
4.2 Cell/B.E. Software Development Kit 66
4.3 Cell Superscalar (CellSs) 66
4.4 CorePy 67
4.5 Mercury MultiCore Framework 68
4.6 MPI Microtask 68
4.7 Open Multi-Processing (OpenMP) 69
4.7.1 Cellgen 70
4.7.2 IBM T.J. Watson 70
4.8 RapidMind 70
4.9 Sequoia 72
4.10 SysCellC 72
4.11 Possible Future Models, Frameworks and Solutions 73
4.11.1 Manticore 73
4.11.2 Open Computing Language (OpenCL) 73
4.11.3 Message Passing Interface (MPI) 74

5 Coconut Multicore Framework 75
5.1 Coconut Project History 75
5.2 Design Overview 76
5.2.1 Objectives 76
5.2.2 Description 77
5.2.3 Analysis 79
5.3 Components 83
5.3.1 Atomic Virtual Operations 83
5.3.2 Runtime System 86
5.3.3 AVOp Stream Generation 88
5.3.4 Computation Kernels 89
5.3.5 Verification Tool 89
5.3.6 Performance Simulator 91

5.4 Comparison with Other Frameworks 92
5.5 Current Status 94

6 Performance Simulation **95**
6.1 Motivation 95
6.2 Simulator Design Concepts 98
6.2.1 Simulator Types 98
6.2.2 Simulator Accuracy vs. Speed 99
6.2.3 Parallel Computer Simulation 99
6.3 Similar Tools 100
6.3.1 Multiprocessor Simulators 100
6.3.2 Cell/B.E. Simulators 100
6.3.3 Network Simulators 101
6.3.4 Alternative Performance Analysis Solutions 102
6.4 Performance Simulator Tool 102
6.4.1 Envisioned Usage 102
6.4.2 Objectives 106
6.4.3 Design Overview 107
6.4.4 Design Analysis 122
6.4.5 Implementation and Unit Testing 124
6.4.6 Performance Testing 124

7 Conclusion and Future Work **135**

List of Figures

2.1 Data dependency example 6
2.2 Single Instruction Single Data 10
2.3 Single Instruction Multiple Data 11
2.4 Multiple Instruction Single Data 12
2.5 Multiple Instruction Multiple Data 12
2.6 Register renaming 17
2.7 Problem category classification 32
2.8 Dynamic vs. Static Scheduling Algorithms 33
2.9 Task interaction graph example 34
2.10 Directed acyclic task graph example 35
2.11 Cannon's algorithm computes block $C(i, j)$ at each processor $P(i, j)$. 36

3.1 Cell Broadband Engine Overview 39
3.2 SPE DMA Transfer Internals 46
3.3 EIB Command Bus 47
3.4 EIB Data Arbiter 48
3.5 Communication Latency Effect 51
3.6 Inefficient Cycle Communication Pattern 53
3.7 Inefficient Heavy Inter-SPE Communication Pattern 54

5.1 Coconut Multicore Framework Overview 79
5.2 Coconut Multicore Framework Runtime View 80
5.3 Exposed Communication Latency 81
5.4 AVOp partial execution order induction 90

6.1 Automated Schedule Selection 104
6.2 Simulation Enhanced DAG Scheduling 105
6.3 Iterative Simulation Result Driven Scheduling 106
6.4 Performance Simulation Design Overview 109
6.5 All SPEs Executing 112

6.6 Some SPEs Executing, Some SPEs Waiting 113
6.7 All SPEs Waiting 113
6.8 Data Sharing Communication Pattern 126
6.9 Simulated Execution Time 130
6.10 AVOps Simulated per Second 131

List of Tables

2.1 Flynn's Taxonomy 9

3.1 Operations/Value Effect on Transfer vs. Computation Time . 50
3.2 RAM Capacity Effect on Square Matrix Mult. Problem Size . 62

5.1 Parallelism Correspondence 80
5.2 AVOp Instruction Set 85

6.1 Matrix Multiplication Performance Simulation Results 128
6.2 Network Bandwidth Simulation Test 134

Acronym Index

AC Address Concentrator

ALF Accelerated Library Framework

AVOp Atomic Virtual Operation

BIU Bus Interface Unit

Cell/B.E. Cell Broadband Engine

Cell/B.E. SDK Cell/B.E. Software Development Kit

CellSs Cell Superscalar

CMF Coconut Multicore Framework

Coconut COde COnstructing User Tool

DMA Direct Memory Access

DMAC DMA Controller

DSL Domain Specific Language

EIB Element Interconnect Bus

FLOPS Floating Point Operations Per Second

IBM International Business Machines Corporation

IDE Integrated Development Environment

ILP Instruction-Level Parallelism

IU Instruction Unit

LAN Local Area Network

LQCD Lattice Quantum Chromodynamics

LS Local Store

MASS Mathematical Acceleration Subsystem

MFC Memory Flow Controller

MIC Memory Interface Controller

MIMD Multiple Instruction Multiple Data

MISD Multiple Instruction Single Data

MPI Message Passing Interface

MPMD Multiple Program Multiple Data

ns Nanoseconds

NUMA Non-Uniform Memory Access

OpenCL Open Computing Language

OpenMP Open Multi-Processing

PDT Performance Debugging Tool

PE Processing Element

PPE PowerPC Processor Element

PPSS Power Processor Storage Subsystem

PPU Power Processor Unit

RAW Read-After-Write

SCEI Sony Computer Entertainment Incorporated

SIMD Single Instruction Multiple Data

SISD Single Instruction Single Data

SOI Silicon on Insulator

SPE Synergistic Processor Element

SPMD Single Program Multiple Data

SPU Synergistic Processor Unit

STI SCEI-Toshiba-IBM

TLB Translation Lookaside Buffer

UMA Uniform Memory Access

VSU Vector Scalar Unit

WAN Wide Area Network

WAR Write-After-Read

WAW Write-After-Write

XU Fixed Point Execution Unit

Chapter 1

Introduction

Since about 2004, chip manufacturers have increasingly turned to multicore architectures to increase performance. This paradigm shift has been dubbed the "multicore revolution" [HL08; GNS07]. Diminishing returns from frequency scaling, a critical technique used to increase processor performance in decades previous, were a major factor behind this technology shift.

One such architecture is the Cell Broadband Engine (Cell/B.E.), developed by Sony, IBM and Toshiba[KDH+05]. The Cell/B.E. is currently being used in the Playstation 3[BLK+07], and as such optimal usage of the processor is of high interest due to the userbase of over 20 million[Son09]. The Cell/B.E. is particularly interesting to computer scientists, some have referred to it as revolutionary[Hof06; Gsc07], due to its network-on-a-chip interprocessor communication and processor heterogeneity. While Cell/B.E. is not the first chip to have either of these properties, it is perhaps the first chip with these features to be so widely adopted.

These new architectures present novel challenges to software developers wishing to use them safely, optimally and easily. Safety is a difficult challenge due to race conditions that emerge when multiple processors use shared resources. These parallel bugs are difficult to diagnose using dynamic runtime tools, due to their occurrence being dependent upon small timing variations[RD00]. Static detection of these bugs at compile-time is an NP-hard problem, assuming the synchronization method is as powerful as semaphores[CMS01]. Optimization of parallel code is unfortunately no easier, as the general multiprocessor scheduling problem is NP-Complete[KA99]. There is also a lack of consensus as to which parallel programming model is best to represent a parallel program[ABC+06], with not much research done to analyze them empirically[HB06].

As a result of the daunting challenges facing developers wishing to take

advantage of these new architectures, new programming models and tools are being developed. For the Cell/B.E. in particular many tools such as RapidMind[McC06] and Cell Superscalar[BPBL06], amongst others[CHKW08; SYR+08; MML07; OIS+06], have been developed to overcome these challenges.

Coconut (COde CONstructing User Tool) is a compiler-technology project at McMaster, which is currently targeting the Cell/B.E. [AK07c]. Coconut was inspired largely by a desire to produce software that is both fast and safe, for applications such as medical imaging. Though strictly speaking it is more of a research platform than a fully developed commercial solution, Coconut can be thought of as another solution competing to overcome the challenges of developing for the Cell/B.E.. Coconut has had great success at delivering safe and optimal code at the ILP level on the Cell/B.E.; the Cell/B.E. SDK 3.0 SPU-MASS library currently includes code using Coconut ILP optimization techniques that is 4x faster than the alternative SIMDMath library created in C.

The Coconut Multicore Framework (CMF) is the current focus of Coconut research efforts; it targets multicore level parallelism on the Cell/B.E.. The framework design is made up of a virtual machine abstraction. Atomic virtual machine instructions execute on each processing element. These instructions are very high level, with only that information necessary to express multicore parallelism exposed. The instructions control all processor-level synchronization, and execute pre-loaded computations on data to produce results. The virtual machine and instruction abstraction purposely corresponds with ILP, so that Coconut ILP optimization techniques already developed may be re-used at the multicore level. A key feature of the CMF is the ability to perform static analysis with linearly bounded complexity, to analyze code safety and efficiency at compile-time. This has allowed us to design and implement a verification tool[AK08] capable of checking for parallel bugs; as a result we have not yet experienced a parallel bug at runtime in our virtual machine programs thus far.

Performance simulation is a potentially useful tool for the CMF for several reasons. Primarily because it allows developers to quickly contrast the performance of different scheduling algorithms, on a workstation instead of less easily accessible Cell/B.E. hardware, and with deeper information with respect to specific execution stalls and transfer latencies that would not be as easily available with other methods. Other reasons include that we can simulate theoretical architectures, both to analyze what architectures we may think *should* be produced and to be able to meaningfully experiment with scheduling on future or concept architectures before they become generally available. Finally, due to the efficient runtime of the performance simulator,

it may actually become a useful tool in scheduling algorithms themselves, as others in the literature have either done themselves or proposed[JSK+06; CHB07].

The primary goal of this thesis is to present the Performance Simulator tool created entirely by the author as part of the CMF. To do this, we will present the design of the simulator itself, as well as performance testing of the tool.

As part of developing the Performance Simulator, the author played a role in developing the CMF itself. The author implemented the Verification Tool after acting as an internal reviewer of its design, as well as helping to design the virtual machine abstraction and debug its implementation. This required a great deal of background knowledge and research with respect to parallel computing, the Cell/B.E. architecture, as well as similar multicore parallelism solutions targeting the Cell/B.E.. As a result, this thesis will have a secondary purpose of documenting the research done into these areas as well, so it may be leveraged by future Coconut project team members.

1.1 Document Structure

In Chapter 2, we discuss parallel computing as a field, and attempt to define the most important concepts and elaborate on the most important issues. Those issues most relevant to multiprocessor computing are emphasized.

In Chapter 3, we introduce the Cell/B.E. and overview its design, architectural variants and current usage. Particular attention is paid to the design of the on-chip communication network, due to its importance in performance simulation.

In Chapter 4, we survey other multiprocessor program models and solutions targeting the Cell/B.E.. Both commercial and research focused solutions are discussed.

In Chapter 5 we discuss the design of the CMF, going over its specifics, features and advantages. We also contrast the CMF with other Cell/B.E. program models.

In Chapter 6 we explain the Performance Simulator design and present performance test results. We also further discuss the motivation to build the simulator, envisioned usage and some background information regarding performance simulation specifically.

Finally in Chapter 7 we overview the results of the thesis and suggest future avenues for Coconut project research based on the thesis results.

Chapter 2

Parallel Computing

Parallel computing is defined as computation which involves multiple simultaneous computations taking place. Parallel computing is becoming increasingly relevant to the undergraduate computer science curriculum due to the increasing prevalence of multicore and multiprocessor systems that demand more skill in concurrency-related programming[Fek09]. As a result, the need to modify the standard computer science curriculum to prepare students to reason about parallel computing problems has been recognized[Has03]. One problem however, is that there has not yet emerged a clear consensus as to how to properly classify these new multiprocessor systems that have driven this new interest in parallel computing[Mar07].

As a result of the great confusion regarding parallel computing, that is, how to classify it, what we mean by it, how to teach it and whether it is being taught properly or taught enough at the undergraduate level, this chapter will attempt to provide an overview of some of the most important concepts of parallel computing. Any terminology used by subsequent chapters will then be clearly defined in this chapter. The chapter will focus on the motivations for parallelism, the different ways of classifying different aspects of parallel computing, measuring the performance of a parallel program, and the unique challenges the parallelism presents.

2.1 Motivations

There are several motivations for parallel computing, mostly relating to the fact that we would like to be able to compute tasks quicker. It is intuitive that if one can split a task into several subtasks that can be processed concurrently, the task could theoretically be completed quicker than it could sequentially.

Amdahl's law, presented in Section 2.1.1, helps to quantify this potential speed up of parallelization for a given program. The other obvious way to speed up a computational task would be to process it quicker sequentially. This was what was done by chip manufacturer's until about 2004, when a series of frequency, memory and power walls finally forced chip manufacturers to turn to parallel multicore architectures to increase performance as outlined in Section 2.1.2.

2.1.1 Performance Increases

Only certain portions of a sequential algorithm or program may be parallelized due to **data dependencies**. Data dependencies occur when an algorithm or program portion refers to the data of a preceding program portion. For instance, in the simple program in Figure 2.1, lines 1 and 2 do not have a data dependency and could be executed in parallel. In contrast line 3 depends on the results of lines 1 and 2, and so could not begin execution until both lines 1 and 2 have been processed without the potential for an erroneous result. If for instance line 3 was executed before line 2, unless Y happened to equal 10, an incorrect Z value would be the result. These *hazards* arise from *race conditions* and are discussed further in Section 2.8.2.

```
1: X = 4;
2: Y = 10;
3: Z = X + Y;
```

Figure 2.1: Data dependency example

Amdahl's Law

With the understanding that only certain portions of a program may be parallelized, it would be useful to be able to quantify the potential gain from concurrently executing program fragments that are parallelizable. **Amdahl's Law**[Amd67] expresses this concept very simply in Theorem 2.1.

Theorem 2.1 (Amdahl's Law)

$$\textit{speed up} = \frac{1}{r_s + \frac{r_p}{n}}$$

where $r_p + r_s = 1$, n is the number of processors, r_s represents the sequential portion of a program and r_p represents the parallel portion of a program.

Notice that as $n \to \infty$ we have that the speed up is $\frac{1}{r_s} = \frac{1}{(1-r_p)}$. This is interesting because it tells us that there exists a hard limit on how much parallelism can help speed up the runtime of a program. At a certain point, adding more processors is not going to make the program run any faster according to Amdahl's Law.

Gustafson's Law

The problem with Amdahl's Law is that it implies there is a theoretical point where adding parallelism will not increase performance, calling the sustainability of parallel architectures as a means to increase performance into question. **Gustafson's Law** was formed in response to Amdahl's Law by noting that it is virtually never the case that the portion of the code that is sequential is independent of the number of processors[Gus88]. What this means is that according to Gustafson's Law, as we scale the number of processors up, the sequential portion of the program will decrease in size — breaking through the limitation inherent in Amdahl's Law.

Gustafson's Law is important to parallel computing as a field, because for a long time pessimism surrounding parallel computing was at least partially due to Amdahl's Law, and Gustafson's Law helped to end this pessimism[Kri01]. It is presented in Theorem 2.2.

Theorem 2.2 (Gustafson's Law)

$$\textit{speed up} = n - r_s \times (n - 1)$$

where n is the number of processors, and r_s represents the sequential portion of the program.

2.1.2 Frequency, Memory and Power Walls

The other motivation for parallel computing is that as hardware based on sequential computation has reached hard physical limits impeding further performance gains, parallel computation has effectively become *necessary* for reaping further performance gains. These hard physical limits are typically referred to as the frequency, memory and power walls. How the Cell/B.E. in particular overcomes these walls is discussed Section 3.6.

Frequency Wall

In order to increase the CPU frequency in modern processors, designers have been required to make instruction pipelines deeper (see Section 2.3.2 for a brief

description of instruction pipelining) . The problem is that we have reached a point of diminishing returns for increasing pipeline depth[SBG+02]. When the number of pipeline stages is increased for the benefit of having the ability to process more instructions concurrently, it necessarily increases instruction latencies. When data dependencies on previous instructions combine with hard to predict branching behaviour in a program, a performance penalty is incurred which increases with the depth of the pipeline. At a certain pipeline depth, the negative effect of these penalties on program performance outweigh the positive effect of a longer pipeline (especially when power consumption is also taken into account).

A parallel architecture can overcome this problem by creating heterogeneous processing elements optimized for situations which involve a heavy amount of branching and for situations which involve a heavy amount of floating point computations. For instance, an architecture with different processing elements with different pipeline lengths to optimally handle each situation.

Memory Wall

The memory wall refers to the fact that processor speeds have been increasing faster than memory bandwidth, such that in modern processors DRAM latency can be measured in the hundreds of CPU cycles[KDH+05]. In his 1997 ACM Turing Award lecture, John Backus used the term "Von Neumann bottleneck" to refer to the fact that in the Von Neumann architecture the CPU is separated from memory, and that properly managing the bandwidth of this separation was key to programming[Bac78]. This essentially identified the potential for a memory wall, however the issue that caused this potential to be fulfilled was the divergence in performance between the CPU and memory bandwidth in unicore architectures. This problem was recognized and popularly identified by William Wulf and Sally McKee in 1994 when they noticed that the exponential growth in processor speed was far greater than the exponential growth in memory bandwidth[WM95]. If the trend continued, they realized that eventually a key impediment to program performance increases would be memory access latency. This wall would be hit at different times for different algorithms due to differing memory access characteristics[McK04], making a precise definition of when the memory wall was hit, or will be hit, difficult to specify in a general sense. Much effort has been exerted by hardware and algorithm designers on optimally using sophisticated hardware, such as multi-level caches, to lessen the effects of this wall[LRW91].

However parallel architectures can effectively alleviate this memory bottleneck by using a layered memory structure with asynchronous data trans-

fers between processing units — allowing for simultaneous data transfer and computation[Mud06].

Power Wall

Power density in processors has doubled roughly every 3 years since the early 1970s[SSS+04]. This trend was economically and technically infeasible to continue. It is economically infeasible because at a certain point consumers wouldn't be able to afford to run the processors if power consumption increased exponentially. It is technically infeasible because energy used in processors is often converted into heat which would at a certain point cause chips to fracture; without new technologies or expensive cooling systems (which would also increase power usage). Indeed, in 2004 Intel cancelled its Tejas and Jayhawk chips that were set to replace the Pentium 4 for these reasons[BKP07]. This event, along with Apple switching to Intel chips after its failure to develop a G5-based laptop, could be considered the microprocessor industry officially hitting the power wall.

Parallel architecture is capable of breaking through the power wall. When one considers that power usage scales exponentially relative to increases in clock frequency, it is correct to conclude that *with the same power usage* multiple processors running at a slower frequency could achieve greater performance than a single processor running at a higher frequency[Naf06]. Heterogeneous architectures that specialize processors for certain tasks[KDH+05], can provide further increases in power efficiency.

2.2 Flynn's Taxonomy

An important point about parallel computing is that it can take place in several different conceptual forms. In this section we seek to overview a common terminology for making distinctions between the different forms of parallelism found in hardware.

Table 2.1: Flynn's Taxonomy

	Single Data	**Multiple Data**
Single Instruction	SISD	SIMD
Multiple Instruction	MISD	MIMD

Flynn's taxonomy is a simple but widely used parallel computing classification proposed by Michael Flynn in 1966[M.J72]. The taxonomy classifies computer architectures according to the amount of data and instruction streams that are processed concurrently by processing elements (PEs). The acronyms for Flynn's taxonomy are more typically used in describing a classification and are found in Table 2.1.

2.2.1 Single Instruction Single Data (SISD)

Single Instruction Single Data (SISD) computing occurs when a single stream of instructions works with a single stream of data. The von Neumann architecture is an example of SISD, as well as early personal computer microprocessors such as the Intel 8088. The inclusion of a concurrent feature such as pipelining in an architecture does not preclude a processor from being categorized as SISD; what matters is that there is one stream of instructions operating on one stream of data. An example of SISD is shown in Figure 2.2.

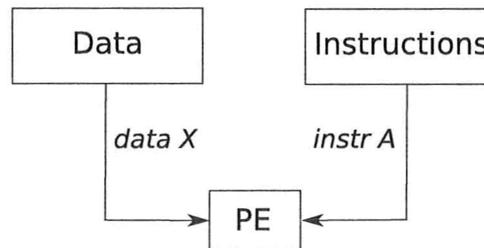

Figure 2.2: Single Instruction Single Data

2.2.2 Single Instruction Multiple Data (SIMD)

A single stream of instructions operating on multiple streams of data is called **Single Instruction Multiple Data (SIMD)** parallelism; also referred to as "Vector Processing" in the literature. There are abundant examples of SIMD parallelism in hardware, dating back to the TI-ASC and CDC Star 100 in the 1970s and the MMX instruction set made popular in the Pentium chipsets of the 1990s[EVS98]. The Cell/B.E. in particular derives much of its raw floating point computing power from its SPEs optimized for SIMD performance[KDH+05]. SIMD is very useful with computations that can be conceptualized as vectors, such as problems in linear algebra and multimedia applications. Scheduling code for SIMD processors is a current area of exper-

tise within the Coconut project[AK07a; Tha06; AK07c]. An example of SIMD is shown in Figure 2.3.

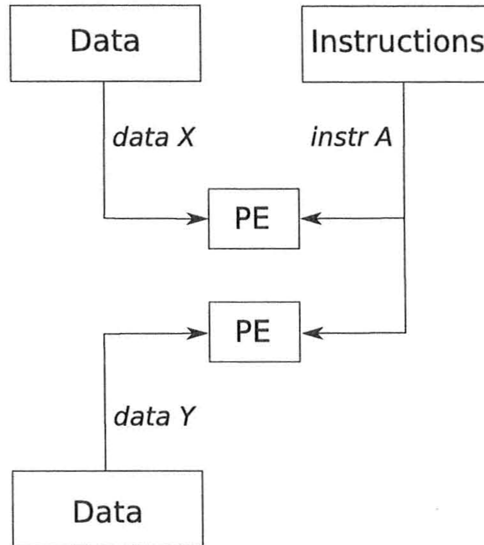

Figure 2.3: Single Instruction Multiple Data

2.2.3 Multiple Instruction Single Data (MISD)

In **Multiple Instruction Single Data (MISD)** parallelism, multiple streams of instructions operate on a single stream of data. This type of parallelism is virtually non-existent in existing hardware[FR96]. It could be useful as a fault-tolerance measure in hardware design, and it has been proposed that MISD architectures would be ideal for pattern matching on data streams[HSN+04]. An example of MISD is shown in Figure 2.4.

2.2.4 Multiple Instruction Multiple Data (MIMD)

Multiple processing elements asynchronously executing different streams of data is referred to as **Multiple Instruction Multiple Data (MIMD)** parallelism. Networks of workstations under different topologies are an example of MIMD parallelism[FR96]. Multicore level parallelism is another example of MIMD parallelism in hardware[AL07]. Scheduling and verifying the correctness of MIMD-level parallelism is the newest focus of the Coconut project, and the focus of this thesis. An example of MIMD is shown in Figure 2.5.

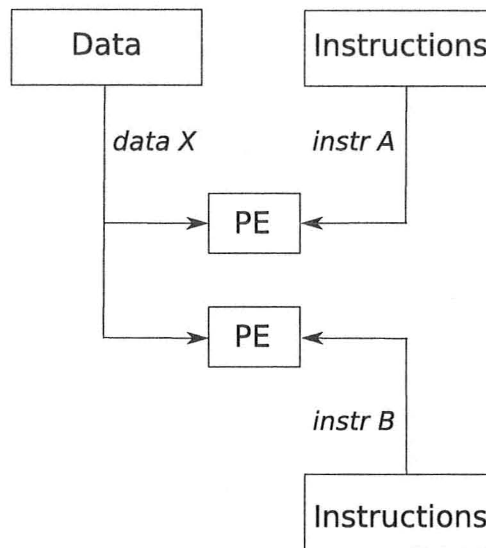

Figure 2.4: Multiple Instruction Single Data

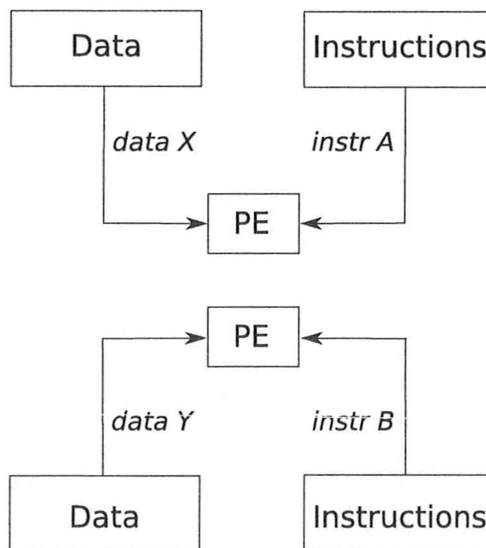

Figure 2.5: Multiple Instruction Multiple Data

Single Program Multiple Data (SPMD)

A subdivision of MIMD parallelism where multiple processing elements execute different parts of the same program on multiple data streams is defined as **Single Program Multiple Data (SPMD)** parallelism[CDG+93].

Multiple Program Multiple Data (MPMD)

Another subdivision of MIMD parallelism is referred to as **Multiple Program Multiple Data (MPMD)** parallelism, it occurs when multiple programs operate on multiple data streams[RSB94].

2.3 Parallelism Levels

The different levels of parallelism that may exist within hardware will be outlined in this section. Interestingly, the history of the modern microprocessor can be roughly divided into eras according to which level of parallelism was being exploited to increase chip performance at the time[MSVV00], with increases in bit-level parallelism giving way to increases in instruction-level parallelism followed by the task-level parallelism increases in modern multicore processors.

2.3.1 Bit-level Parallelism

Bit-level parallelism refers to the parallelism associated with the width in bits of registers in a processor. By going from 16-bit to 32-bit registers, one has in some cases doubled the computing power by operating concurrently on 32 instead of 16 bits. For instance if one was adding two 32-bit unsigned integers, in the case of the 16-bit architecture, one would have to add the lowest bits first, and then the highest afterwards in a separate operation (plus any carry over from the addition of the lowest bits). If one had the ability to operate on 32 bits in parallel however, this could be done in a single operation. Up until about 1985, increasing bit-level parallelism was an important way in which processing power was increased[MSVV00]. Since this time, the market has been dominated largely by 32-bit architectures and some 64-bit architectures. Exploiting other forms of parallelism started to make more sense at this time, as for most applications 32-bit or 64-bit precision is sufficient.

2.3.2 Instruction-level Parallelism

Instruction-level parallelism occurs when we execute the instructions of a sequential binary program concurrently. This form of parallelism takes many forms itself, with optimizations to exploit it taking place at the hardware level at runtime, but also by compilers and potentially by crafty programmers. The period of the late 1980s to the early 2000s is strongly associated with increasing levels of instruction-level parallelism as a means to increase performance in hardware design[MSVV00]. The different ways in which instructions can execute in parallel will be briefly defined in the remainder of this section, though it should be noted these are expansive fields of study with much published research.

Out-of-order Execution

Out-of-order execution occurs when instructions are executed in a different order than is presented in the compiled binary file of the program. Normally if an instruction is set to be executed, but its input operands are unavailable, the instruction will stall and computation will cease until the operands are available. With out-of-order execution, instructions are first collected in a queue, and execute when input operands are available. The key point being that an instruction in the queue that occurs *after* another instruction in the queue may be dispatched ahead of it to a processing element for execution; and so the execution is out-of-order. The results of the instructions are themselves queued, and are then put into the correct program order as given in the binary file as more instructions complete. As a result the program appears to have executed in the correct order, though this may not have been the case. The speed-up of out-of-order execution derives from hiding the latency of fetching data for instructions to operate on by executing other instructions where a stall in execution would otherwise occur.

Data dependencies may prevent out-of-order execution from being possible in all cases. Also, it should be noted that maintaining the behaviour of hardware exceptions was a key hurdle to overcome in enabling out-of-order execution to become a more common hardware ability[SP88]. Out-of-order execution became very common in processors starting in the 1990s, for instance in the Pentium Pro[KPH+98] and AMD-K5[Chr96].

Instruction Pipeline

An **instruction pipeline** allows for multiple instructions to execute concurrently. The key idea is that each instruction executing concurrently is execut-

ing a different stage in the set of stages required to execute an instruction. An instruction pipeline is somewhat analogous to an assembly line in automotive manufacturing. Instead of producing one automobile at a time, the steps are broken down, for instance as: obtain all parts, assemble frame, attach wheels and place vehicle in truck for shipment. By doing all of these steps concurrently, a manufacturer could have four cars being built at once instead of one.

Instruction pipelines in processors divide the processing of an instruction into independent steps. The steps may include: fetch instruction from memory, decode instruction and read registers, execute the instruction, access memory, and write the result into a register. In the same way that all the steps of an assembly line can be run concurrently, so can the stages of an instruction pipeline. If a CPU were to process one instruction at a time, its clock frequency would be limited to the time it takes to do all of these steps. However if a CPU were to use pipelining, its clock frequency could be increased to the time it takes to do the longest step in the pipeline.

As discussed in Section 2.1.2, hard to predict branching behaviour limits the performance of an instruction pipeline. An example of an architecture which used instruction pipelining is the Pentium 4[ZR04], which had a 20-stage instruction pipeline.

Superscalar Execution

Superscalar execution works by simultaneously dispatching instructions for execution. This can be done because the instructions are dispatched to different computational units within the processor, for instance an ALU and an FPU. In order for this to work, the processor hardware has to check for data dependencies that would create hazards before executing instructions concurrently. One can see that data dependencies will place an upper bound on the effectiveness of superscalar execution, with studies showing that for instance 8-way superscalar is likely to provide little benefit[MSVV00]. It is important to note that pipelining and superscalar execution could be occurring at once within the same processor, the instructions could be concurrently sent to pipelined execution units which also execute instructions concurrently in different stages.

Early processor designs contributing to superscalar concepts included the CDC-6600 and IBM-360/91[FB92]. In the 1990s pipeline execution was found in more mainstream computers, such as the AMD K5 and the MIPS R10000[SS95b].

Other Important Techniques

Other techniques may be used in conjunction with out-of-order execution, instruction pipelining and superscalar execution to better exploit instruction level parallelism. These techniques do not create a new conceptual form of instruction level parallelism, but instead allow the existing techniques to be used more effectively.

Register renaming is a technique which removes unnecessary data dependencies from serial code in order to remove any hazard of running them in parallel. This allows for instructions which would previously be run in serial to be run in parallel[SP88]. In Figure 2.6 an example of register renaming is shown. In this example we see that if instructions 4-6 could be performed using a different register than register 8, it would eliminate the dependency these instructions have on instructions 0-3, *without effecting the correctness of the results*. Eliminating these dependencies allows us to operate on these groups of instructions in parallel, which reduces the total execution time. Register renaming is thus a technique which allows us to exploit superscalar execution (and potentially out-of-order execution) more effectively by removing unnecessary data dependencies. It turns out that the only dependencies which register renaming cannot eliminate are *read-after-write* dependencies[SP88]. Register renaming can be done by a compiler or by hardware (the first instance of which being the floating point unit of the IBM 360/91[MPV93]).

Speculative execution and branch prediction are also important techniques for helping to exploit instruction level parallelism[GG97]. Speculative execution refers to execution for which the results may not actually be used. For example, load instructions may have somewhat predictable behaviour and the memory locations which they reference may be computed ahead of time. If the speculative execution turns out to be incorrect, say for instance the load memory instruction actually references a different address, a stall will be required while the proper computation is executed. Branch prediction refers to speculative execution where the instructions following a potential path of the branch are executed. If the speculation is correct, then instructions which have already entered the instruction pipeline can proceed, otherwise a stall will be required while the correct instructions begin to execute. Speculative execution and branch prediction thus help to exploit instruction pipelines in particular. This is done by speculatively allowing instructions for which we are not certain we will need the results for to execute, at the penalty of a stall if the speculation is incorrect. These techniques are common place in modern processors, for instance in the Intel Pentium Pro and SUN Ultrasparc[MSVV00].

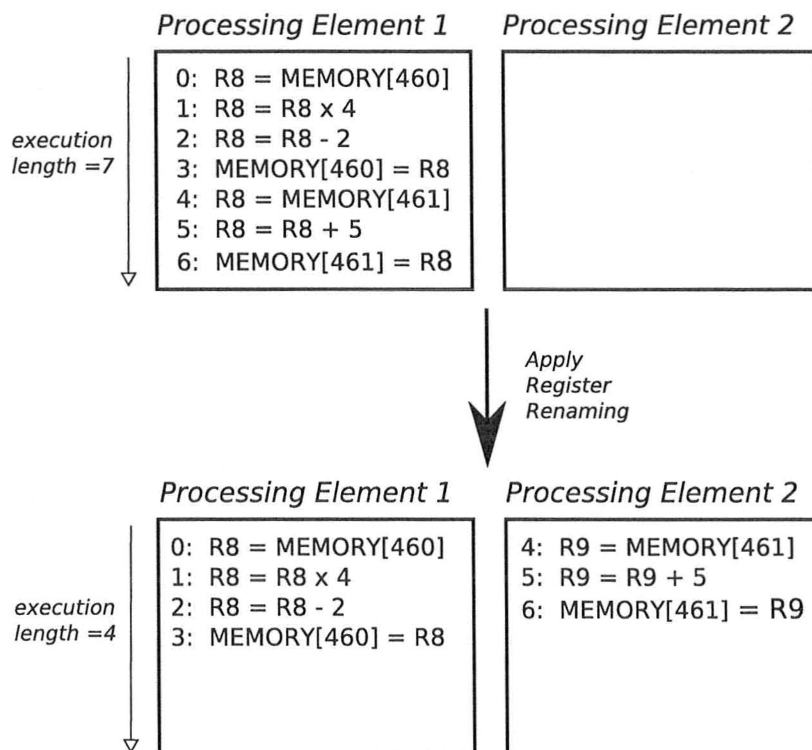

Figure 2.6: Register renaming

2.3.3 Thread-level Parallelism

In contrast to instruction-level parallelism, where instructions from a single stream of instructions are executed in parallel, **thread-level parallelism** occurs when more than one stream of instructions *from the same program (or process)* are executed concurrently[ACVP06]. The definition of what constitutes thread-level parallelism beyond this is less clear. Partially because what differentiates a thread, process and program is dependent upon the hardware and operating system of the machine. On a single core machine, a multithreaded program does not necessarily have threads executing concurrently, but instead multiple threads forked from the same program may take turns executing individually (time-sliced). However on a multicore or multiprocessor machine, a multithreaded program may have each thread assigned to a different core or a different processor, in which cases threads could be executing concurrently. It should be noted that even though in the case of a single

core the threads do not execute concurrently, they share access to common resources, and from the point of view of the programmer who may not have control over how hardware and the operating system is switching between threads, they might as well be concurrently executing. We say this because the programmer will have to worry about the same issues as they would if the threads were executing concurrently.

Thread-level parallelism is also the first level of parallelism whose efficiency and correctness is largely dependent upon software and the programmer, and not by hardware. With instruction-level parallelism, optimizations are often made by a compiler or programmer with knowledge of the underlying hardware, but concurrency based hazards are protected against at the hardware level. With thread-level parallelism however, not only is optimization highly dependent upon skilled program implementation, often at the level of program source code and not at the automated level of the compiler, but hazards due to concurrent execution must be handled by the programmer as well. Competition between threads for resources, thread synchronization overhead, context-switching overhead and sometimes conflicting performance vs. correctness goals all must be considered by a programmer[JFL98]. This makes optimizing and verifying the correctness of thread-level programming a particularly difficult problem for programmers[ACVP06].

Advantages of multithreaded programs include increased performance on multicore or multiprocessor systems[Eng00]. As well, multithreaded programs allow for computation heavy tasks to run on a thread separate from another thread which handles user I/O, which prevents the entire program from freezing from the vantage point of the user and improves performance by overlapping I/O and computation[JFL98].

Examples of architectures and operating systems capable of thread-level parallelism are common in modern computer systems. For example, Intel Core Duo processors running Windows XP are capable of thread-level parallelism[ACVP06].

2.3.4 Process-level Parallelism

Process-level parallelism, which occurs in multiprocessing systems, is the parallelism that occurs when we have two processes or programs executing concurrently[HNO97]. It is different from thread-level parallelism in the sense that in thread-level parallelism the concurrent streams of instructions are forked from the same process[ONH+96]. Process-level and thread-level parallelism can both occur in multicore systems, but only process-level parallelism can occur when the multiple CPUs of a machine architecture do not share re-

sources that may be required for thread-level parallelism, such as a cache. This can happen when the multiple CPUs are not on the same chip, but instead we have multiple processes interacting across an Ethernet network. One way to distinguish multiprocessor architectures is by the symmetry of the processors. Different classifications of multiprocessor hardware can also be made based on the type of communication infrastructure between the processors.

Processor Symmetry

The processing elements in a process-level parallelism capable multiprocessor system do not necessarily have to be the same, or have access to the same system resources.

A **symmetric multiprocessing** system is one in which two or more (identical) processors have equal access to the same memory[CG08], sometimes referred to as homogenous multiprocessing. It is characterized by the ability of a computing system with multiple processors to treat them like a single processor[IY00]. The Intel Core 2 Duo is an example of a symmettric multiprocessor[CG08].

In contrast an **asymmetric multiprocessing** system, sometimes referred to as heterogeneous multiprocessing, is one in which the multiple processors have different characteristics such as resource access, performance, power consumption and instruction sets[Mar07]. There is no such requirement that an asymmetric multiprocessor is able to treat its multiple processors as a single processor. An example of an asymmetric multiprocessor is the Tensilica Xtensa LX, which allows individual cores to be configured differently before processing different applications[Mar07].

Multiprocessor Architectures

Multiprocessor architectures themselves are roughly classified into different categories based on the way in which the CPUs communicate. **Multicore** systems, though containing multiple processors, are referred to as such because the multiple processors are contained on the same physical chip[Mar07]. In contrast systems referred to as **multiprocessor** systems are generally those with levels of communication between processing elements that are higher up than chip-level. These different levels of communication can themselves be more abstractly described to classify different types of multiprocessor systems, and this is done so in Section 2.6.2.

2.4 Task Parallelism vs. Data Parallelism

Parallel computing can also be divided conceptually into two very broad categories: task parallelism and data parallelism. This section will outline what is meant by this distinction.

2.4.1 Task Parallelism

Task parallelism, also referred to as functional parallelism[GP95], focuses on concurrent execution of tasks with multiple processes or threads on multiple computing nodes, operating on the same or different data, with explicit communication between the computing nodes[SSOG93]. An example of a task could be a C language function for matrix multiplication. Task parallelism can be thought of as a way of looking at parallelism, where we look at a computation as a series of interacting, perhaps order dependent tasks. This can take place at different layers in hardware, with the process/thread level of parallelism in hardware being a potential layer for task parallelism. One could also consider task parallelism at the level of a disturbed system, with each processing element executing a task. The important idea in considering task parallelism is that the task, and dependencies and interactions between tasks, is the emphasis of reasoning about concurrency.

2.4.2 Data Parallelism

In contrast to task parallelism, **data parallelism** is parallelism where there is concurrent execution of the *same* task on data, and with identical tasks the focus is then on the distributed nature of the data[HS86]. Data parallelism thus roughly equates to SIMD parallelism. As with task parallelism, data parallelism can occur at different levels of hardware. Data parallelism can be said to occur if a single thread or process, or multiple threads or processes, operate on the same or different data, from the same or different source. The key point concerning data parallelism is that we reason about the distribution of data, as the task is known to be the same across all processing elements.

2.5 Memory Classifications

Memory in parallel computing systems can be classified in several ways, perhaps most importantly as shared or distributed memory. The way in which the

memory of a parallel computer works defines which types of parallel computation are possible. A hybrid between shared and distributed memory exists and is called distributed shared memory. The type of memory model may effect which algorithms are possible, and whether processing elements access data in uniform time or not.

2.5.1 Shared Memory

In a **shared memory system**, all of the processing elements have access to the same shared memory address space[PZOL01]. An example of such a system would be a dual core microprocessor where each core has access to the same large block of random access memory. Scalability is a problem with shared memory systems, as it increases the effects of the memory wall discussed in Section 2.1.2. Adding additional processing elements to access the same shared memory won't increase performance if the pipeline bandwidth is already completely consumed fulfilling the data requirements of existing processing elements. The fact that memory is shared between two cores or computational units, does not necessarily imply that some lower-level of memory is shared, such as cache. Examples of shared memory processors include the Intel Pentium D, where two cores don't share a cache, or the Intel Core Duo, where the two cores do share a cache[FVP06].

2.5.2 Distributed Memory

In a **distributed memory system**, each of the processing elements can access only a portion of the total system memory. For instance each processing element could have its own local memory. If a processing element wishes to operate on data located in another processor's private memory, the data is exchanged through some communication mechanism. As large shared memory systems are expensive to build, an advantage of distributed memory systems is that they have a significant cost-performance advantage over shared memory systems[CK88]. The performance of distributed memory systems also scales better when increasing the number of processing elements as discussed in Section 2.1.2. However the systems are considered much harder to program for than shared memory systems, as for instance one cannot have shared memory data structures[KMVR90]. A distributed memory system's performance can also be highly dependent upon effective usage of whatever communication mechanism exists between processing elements[CK88]. The BlueGene/L supercomputer is an example of a system with distributed memory[AAA+02].

2.5.3 Distributed Shared Memory

Distributed shared memory systems have processing elements with private memory that can also access memory shared between processors. This shared memory access can be implemented through hardware, or it can be done through some software abstraction to give the illusion of physically shared memory[NL91]. Distributed memory systems can offer the advantage of shared data structures (and thus portability with code programmed for shared memory architectures), while allowing for scalability by alleviating the memory wall hardware bottleneck. An example of a software level abstraction to provide for shared memory access in a distributed system would be the OpenSSI single system image clustering system which allows a cluster of computers to share resources such as memory[LGV+05].

2.5.4 Uniform Memory Access (UMA)

Uniform memory access (UMA) parallel computing systems have the property that the time to access data is not affected by its location in shared memory, or by the processing element requesting the data[ZSLW92].

2.5.5 Non-Uniform Memory Access (NUMA)

In contrast, **non-uniform memory access (NUMA)** parallel computing systems have the property that the location of the data accessed relative to the processing element can have an order of magnitude effect on the time required to access the data[BTK06].

2.6 Processing Element Communication

An understanding of communication between processing elements is very important to the effective usage of a parallel computing system[KPP06; BCG+95; JGMR07; CK88]. The communication time requirements, and not the computation time requirements, may actually be the constraint on the performance of an algorithm, this distinction is discussed in Section 2.6.1. In Section 2.6.2 we discuss classifications of the distance between different processing elements on a network, which can have an order of magnitude effect on communication times between processors, determining which algorithms may be practical. The terminology commonly used to describe different algorithm types according to the communication required for their functioning is discussed in Section 2.6.3. Performance measurement of communication networks is described in terms

of latency or throughput, which are defined in Section 2.6.4. The different network topologies found within parallel computing systems are discussed in Section 2.6.5.

2.6.1 Computation Bound vs. Communication Bound

As outlined in Section 2.1.2, algorithms executing on a von Neumann architecture, or a parallel computing variant of it, must necessarily transfer data from somewhere in memory to the processing element(s) in order to operate on it. The importance or unimportance of this transfer of data to the algorithm's execution time is referred to as a bound. These concepts are important in the sense that they convey where the greatest speed up of the algorithm may potentially come from - optimizations in data transfer or optimizations in the computation.

Computation Bound

An algorithm is said to be **computation bound** (also referred to as computationally bound, or CPU bound) if the execution time is primarily determined by the computation required, and not the communication of data[RPK00]. Whether an algorithm is computationally bound or not for a given implementation may depend greatly on the architecture for which it is implemented. For instance, an algorithm could have a small memory bandwidth requirement, and a large computational throughput requirement, which would tend us to classify it as a computationly bound algorithm. But if we were to run this algorithm on a distributed system spread across the world, the primary constraint to execution time may become the communication cost due to increased transfer latencies.

Communication Bound

Conversely an algorithm is said to be **communication bound** (also referred to as I/O bound or memory bound) if the execution time is primarily determined by the communication time of data to processing elements, and not the computation of data[RPK00]. An algorithm is a likely candidate to be considered communication bound when the input data requirements of the algorithm cannot be transfered to processing elements as quickly as they can be processed by those processing elements.

2.6.2 Processing Element Coupling

Process-level parallelism in particular is often referred to as either tightly coupled or loosely coupled according to the communication distance and resources shared between processors, though it could perhaps be more accurately described as a spectrum from tight to loose.

Tightly Coupled

A **tightly coupled** multiprocessor system is one in which the CPUs communicate at the level of the system bus or some other on-chip network such as a token ring or crossbar switch[BL89; Gen02]. The ways in which these processors communicate with each other, and the resources that they share access to (for instance different I/O devices), varies from system to system[Wol04]. The processors should have access to a global memory space, even if it is through an abstraction in a memory hierarchy[BL89]. Multicore processors (sometimes called chip-level multiprocessors) are thus considered tightly-coupled multiprocessor systems, an example of which would be the AMD Opteron architecture[CH07].

Loosely Coupled

A **loosely coupled** multiprocessor system is one in which the multiple CPUs are contained in standalone computers (standalone in the sense that they do not have access to a global memory space in RAM or other shared resources) and these computers are connected by some network above the bus-level, such as high speed fiber networks[SS95a]. These networks are an order of magnitude or two slower than those of tightly coupled systems[Gen02]. Loosely-coupled multiprocessor systems are also referred to as clusters. An example of a loosely-coupled multiprocessor system would be a Linux Beowulf cluster[Gen02].

2.6.3 Parallelism Granularity

Parallel algorithms have different levels of 'granularity' with regard to the communication requirements (i.e. data dependencies) between the processing elements and the size of the data involved. As with the notion of processor coupledness, the notion of granularity is likely best perceived as a spectrum over algorithms and not fixed categories.

Fine Grained Parallelism

Parallel algorithms with high communication overhead per processing element and that operate on smaller quantities of data are an instance of **fine grained parallelism**[And92; CG08]. Algorithms with fine grained parallelism thus map well to more tightly coupled processors such as vector processors[AJ88].

Coarse Grained Parallelism

In contrast, parallel algorithms which operate on larger quantities of data and with lower communication overhead per processing element are an instance of **coarse grained parallelism**[And92; CG08]. As expected, algorithms with this property naturally map well to more loosely coupled parallel architectures such as workstation clusters[AJ88].

Embarrassingly Parallel

Embarrassingly parallel algorithms are those for which the tasks to be run in parallel are essentially completely independent tasks, with little if any data dependencies (and thus communication) between them[MMSC99]. Algorithms with this property map well to the most extreme loosely coupled parallel architectures, such as the SETI@home project[GcSS+05].

2.6.4 Performance Measurement

The performance of a communication network for a parallel system can be described in terms of latency and throughput, depending on which aspect of performance one wishes to describe.

Latency

Latency is the time that it takes for a single action within the network to occur[AP07b]. An example of latency would be the time it takes for a transfer of data from one processing element's local storage to another processing element's local storage to complete.

Throughput

Throughput, also known as bandwidth, is the quantity of data transferred per unit of time[AP07b]. An example of throughput in an architecture would be having a network component capable of 100 GB/s of inbound data transfers.

2.6.5 Network Topologies

Communication between processing elements in a parallel computing system can occur over many different network topologies. For instance buses and crossbar switches have been used by initial multicore systems for communication between cores and cache banks[ABC+06]. In contrast the Internet and its heterogeneous network of networks structure is effectively the topology of the SETI@home project[GcSS+05]. Other network topologies may include ring, star, tree or mesh.

Within a parallel computing system different topologies may be used for different purposes. For instance in the IBM BlueGene/L a tree network is used for global communication of messages to all other processing elements, and a higher bandwidth ring network is used for point-to-point messages[ABC+06].

2.7 Performance Measurement

Measuring the performance of a parallel computing system is possible in several ways. FLOPS is a measurement of the theoretical performance of a parallel computing system and is described in Section 2.7.1. There exist benchmarks for empirically measuring performance of parallel machines, and these are mentioned in Section 2.7.2.

2.7.1 Floating Point Operations Per Second (FLOPS)

FLOPS stands for Floating point Operations Per Second, it is used to describe how many floating point operations a computing system can perform in a second. For example, if each processing element in a 3 processor system was capable of 20 billion floating point operations per second, we would say the system is capable of a *theoretical peak performance* of 60 gigaFLOPS (or GFLOPS). It is a metric often used in scientific computing in particular[Smi88]. FLOPS is the measurement used to rank theoretical peak performance by `top500.org`, which ranks the top 500 supercomputers in the world.

The performance of algorithms on systems relative to one another is of obvious interest, but one cannot simply divide the floating point operations required by an algorithm by the FLOPS of each computer system to obtain the relative performance of an algorithm. Computing system characteristics such as communication bandwidth and the number of processing elements, as well as characteristics of the algorithm such as data dependencies which necessitate sequential execution, prevent this from being the case. For example, an algorithm which allows for a maximum of two parallel threads of execution will

not benefit from operating on a machine with three 20 GFLOP processing elements relative to an otherwise equal machine with two 20 GFLOP processing elements. This shortcoming leads to alternative models of comparing machine performance such as analytical model ling or benchmarks.

2.7.2 Benchmarks

Benchmarks capable of evaluating parallel performance are numerous and include SPLASH[SWG92], SPEComp[ADE+01], Perfect Benchmarks[CKPK90], Parkbench[HL00], LINPACK[DLP03] and EuroBen[vdS93]. Benchmarks have been created because it is desirable to have a single performance metric to compare machines[CKPK90]. Empirical benchmark results are reliable and feasible relative to attempting to analytically compare performance, which is considered infeasible due to machine complexity[CKPK90]. Different benchmarks target different types of parallel computing systems. For instance SPEComp targets mid-sized parallel servers[ADE+01], where as SPLASH targets shared memory multiprocessor systems[SWG92]. It is noteworthy that the LINPACK benchmark is used to rank supercomputers on `top500.org`[DLP03].

Existing benchmarks have been described as inadequate[ADE+01]. For example, the creators of the SPLASH benchmark have described it as inappropriate to use SPLASH as a definitive and quantitative benchmark to demonstrate one machine is superior to another[SWG92]. The founders of EuroBen state that they believe it is not possible for a single measurement to characterize the performance of high-performance parallel architectures, describing the goal itself as evasive and ambiguous due to the variability of both potential problem spaces and different target architectures[vdS93].

2.8 Multiprocessor Parallelism Software Challenges

Multiprocessor computing presents several challenges from the perspective of software. Important challenges include how 'best' to represent a parallel program as is discussed in Section 2.8.1, how to ensure correctness of parallel programs as is discussed in Section 2.8.2 and how to optimally schedule software on a parallel computing system as discussed in Section 2.8.3. Overcoming these challenges is not exclusively a matter of problem solving, but also identifying and exploiting useful trade-offs between these challenges. For instance by increasing the ease of program expressability in a given model, we may

lose potential optimization opportunities. Similarly, by allowing for more optimization we may lose opportunities to ensure safer software. These tensions will also be discussed.

2.8.1 Program Models

The issue of how to model a parallel program is a contentious one among researchers, some hold strong beliefs about which model is easiest despite a lack of empirical analysis in the literature[HB06]. The issue of which parallel programming model is 'best' is also not as simple as which model is 'easiest' to program in; a tension exists between programmer productivity and implementation efficiency[ABC+06]. One parallel program model may sacrifice implementation efficiency in exchange for higher safety and ease of programmability, for instance.

The most popular parallel program model at present is message-passing using the MPI specification[HB06]. The MPI specification is language-independent, it can be implemented as a set of functions in an API. It allows for both collective and point-to-point communication between processing elements, and is particularly dominant in distributed computing environments [SKP06]. It is low-level in nature, in the sense that the programmer manually implements communication between processing elements. The creators of MPI themselves specifically do not claim that it is uniformly superior to other models, citing advantages of universality, expressivity, ease of debugging and performance[GLS94]. The dominance of MPI is primarily attributed to its performance, which comes at the cost of an increased burden on the programmer [ABC+06].

An important distinction by which parallel program models may be distinguished is whether they allow for implicit parallelism, explicit parallelism or some combination. **Implicit parallelism** occurs by having a compiler or compile-time tool automatically parallelize a computation expressed in some language[UK88]. In contrast **explicit parallelism** occurs when a program model provides constructs specifically for allowing the programmer to create and manage concurrency[Fre96]. For a program model to be considered purely implicit, one would expect that no additional parallel language features would be required to help a compile-time tool exploit parallelism. Hybrid program models exist that may incorporate aspects of implicit and explicit parallelism. RapidMind is somewhat of a hybrid model, in the sense that a compiler handles portability to different architectures but that aspects of parallelism are made explicit to the developer[MWHL06].

Different models may be more suitable to different architectures or ex-

clusive to different architectures. For instance OpenMP[DM98] assumes a uniform shared memory in its execution model, which raises issues of scalability to larger distributed memory systems[CCZ04]. Researchers have speculated that the heavy demand that more communication-explicit models such as MPI places on programmers will cause the model to breakdown as future architectures begin to demand synchronization over thousands of processing elements[ABC+06]. This could be problematic, as models which abstract away hardware features, such as the number of processors or communication between them, are widely considered to be insufficient with respect to performance[Dei05]. Though less-specific and increased abstraction of the problem should allow for optimization techniques to find efficient mappings to hardware, fulfilling this potential is considered an open research problem[ABC+06].

2.8.2 Program Correctness

Multiprocessor programs have the burden of various *synchronization errors* in addition to those bugs possible in a sequential program. These synchronization errors may come in the form of *race conditions* or *deadlocks*. Race conditions generally occur due to a lack of synchronization, while deadlocks generally occur due to too much synchronization[RD00].

Race conditions occur when multiple simultaneously executing processes attempt to access a shared state without synchronization[FA99]. Race conditions are widely considered to be very common and difficult to diagnose errors in multiprocessor programming[FA99; CMS01]. The reason it is sometimes very important for multiple simultaneously executing processes to synchronize when accessing a shared state is due to the *hazards* created by data dependencies.

A **hazard** is what occurs when a parallel computing system attempts to simultaneously execute multiple instructions containing a read-after-write (RAW), write-after-read (WAR) or write-after-write (WAW) data dependency [LR97]. A RAW data dependency occurs when a variable in a sequential program is first read from and then written to. If when simultaneously executing, the write instruction executes before the read instruction, the resulting program state may be incorrect. Similar thinking leads one to see that WAR and WAW data dependencies may also lead to incorrect program state if simultaneous execution of the instructions were to occur, making these situations hazards as well. Sections of an algorithm with these hazard-causing data dependencies must be executed sequentially and atomically (i.e. without depending on operations executed by another processor), and are referred to as

critical sections[NM92].

Race conditions with a hazard can be referred to as **critical race conditions**, requiring synchronization to avoid indeterminate program state. Race conditions without a hazard, those where the multiple simultaneously executing instructions only read the shared state, may be called **non-critical race conditions** because even without synchronization indeterminate state will not arise.

The reason race conditions are considered so difficult for a programmer to debug is because their occurrence depends on small timing variations[RD00]. As a result, it may be difficult to reproduce the conditions in which a race condition bug arose in order to diagnose and fix the exact problem. The hazards that race conditions introduce may be mitigated by detecting race conditions either statically at compile-time or with development tools dynamically at runtime.

Semaphores are a relatively powerful form of synchronization intended to prevent race conditions, amongst other synchronization methods. An efficient solution to statically detect race conditions is improbable for a program using semaphores, as static detection of race conditions is NP-hard for programs that contain multiple semaphores[CMS01]. Efficient heuristics have been developed for the case of a program with multiple semaphores, and efficient exact algorithms for race condition detection have been developed for the case that the method of synchronization is weaker than semaphores[LKN96]. RacerX is one example of a static tool used to detect race conditions[EA03]. Unfortunately dynamic methods of detecting race conditions would require the ability to monitor every memory access, making them impractical[CMS01].

Many different synchronization methods exist to eliminate race conditions, including semaphores, barriers, monitors and non-blocking synchronization methods[HFM88; Din89]. An implemented synchronization object that eliminates simultaneous access to shared data is referred to as a **mutex**, for providing *mutual exclusion*[NH00]. **Semaphores** essentially work by having two functions which are called before and after a program accesses data shared between processes - the first function waits until the semaphore is made free and the other function frees the semaphore[Din89]. **Barriers** work by having a point in source code, the barrier, which a group of threads must all reach before any can go past it to force synchronization at that point[HFM88]. **Monitors** are objects consisting of variables only accessible to methods of the object; these methods are accessed with the condition that only one process may access them at any given time[Din89].

Unfortunately, these attempts to eliminate race conditions may lead to a state referred to as a **deadlock**, where no thread can advance because every

thread is waiting for a lock to be released by another thread[FA99]. For instance, if a programmer forgets to have a thread unlock a shared resource, one thread may wait forever for that resource, which may have a cascading effect across all threads. Many non-blocking synchronization methods have been developed to get around the possibility of deadlocks, however they depend on various hardware-level atomic operations which may or may not be supported from architecture to architecture[Moi97]. Deadlocks themselves may be detected by various tools, including the RacerX tool capable of detecting race conditions[EA03].

2.8.3 Optimal Scheduling

In the *general case multiprocessor scheduling problem* we don't consider properties specific to the algorithm that is being scheduled when scheduling it. In contrast one can consider *problem specific* scheduling algorithms that are designed to schedule for specific problem domains such as matrix multiplication. One can imagine hybrid scheduling algorithms that are capable of general scheduling but may also incorporate algorithm-specific information or properties in making scheduling decisions, however these will not be discussed. Assumptions about the architecture of the multiprocessor system can also vary across different scheduling algorithms.

General Multiprocessor Scheduling Problem

The general multiprocessor scheduling problem is an NP-Complete problem of great interest to computer scientists. It should be noted that there are known simplifications of the multiprocessor scheduling problem that absolve it of the NP-Complete property and allow for polynomial time solutions, but that these simplifications are largely impractical in useful real world scheduling scenarios[KA99].

The problem in its most basic categorization can be stated as, "Given a number of processors p and a set of tasks T where each task t_i has a given length l_i, what is the minimum possible time required to schedule all tasks in T on p processors without tasks overlapping?". This version of the problem does not take into account constraints such as temporal dependencies between tasks, data requirements of tasks, processor heterogeneity and data storage limitations of processors. Outside of some special cases these constraints can be expected to increase the difficulty of the problem.

The general multiprocessor scheduling problems tends to categorization based on additional problem property assumptions that usefully narrow

the focus of problems to be considered for algorithm creators. The scheduling problem can be subdivided into different categories, and it is prudent to do so. Even though ideas for solving one category of the problem may be relevant in another, algorithms targeting a specific variant of the problem can potentially work or be designed better under more limited assumptions. The categorization we present here is similar to that presented by Kwok and Ahmad[KA99]. Figure 2.7 outlines the relative classification of the different problem categories we will discuss.

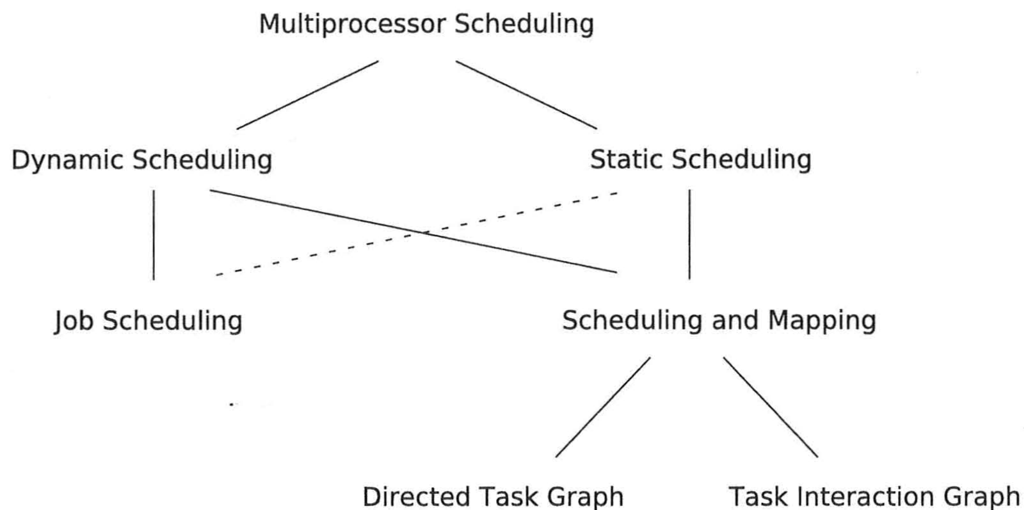

Figure 2.7: Problem category classification

The problem of multiprocessor scheduling can be subdivided broadly into **static** and **dynamic** scheduling[KA99]. In static scheduling tasks are scheduled to processors at compile time. The scheduling characteristics, such as task processing time, task dependencies and communication times are often known before the scheduling algorithm executes. A disadvantage of static scheduling is that one may have to 'guess' these numbers[GP85]. Often times a scheduling algorithm designer or user will have to test out tasks and measure execution times to have the algorithm produce good schedules.

In dynamic scheduling tasks are scheduled at runtime, and the scheduling characteristics may be unknown before and during program execution[AG91]. We should note that dynamic scheduling can in some sense be thought of as a series of static scheduling decisions. As a result, in particularly 'bursty' problem spaces where periods of scheduling are followed by relatively large periods of computation, static scheduling heuristics may still be relevant. There is also a tension between overhead in scheduling tasks and processing the tasks

themselves. A dynamic scheduler that produces efficient schedules may itself be inefficient - potentially removing the gains of the efficient schedules it produces.

Whether static or dynamic scheduling is optimal in particular circumstances depends on the trade-off between having better information as to the state of the program when making a scheduling decision, as in dynamic scheduling, or having more time, and presumably processing power, to make the decision, as in static scheduling. In static scheduling, this increased processing power could be used to analyze more of the task subgraph to be scheduled, for instance. This trade-off example is illustrated in Figure 2.8.

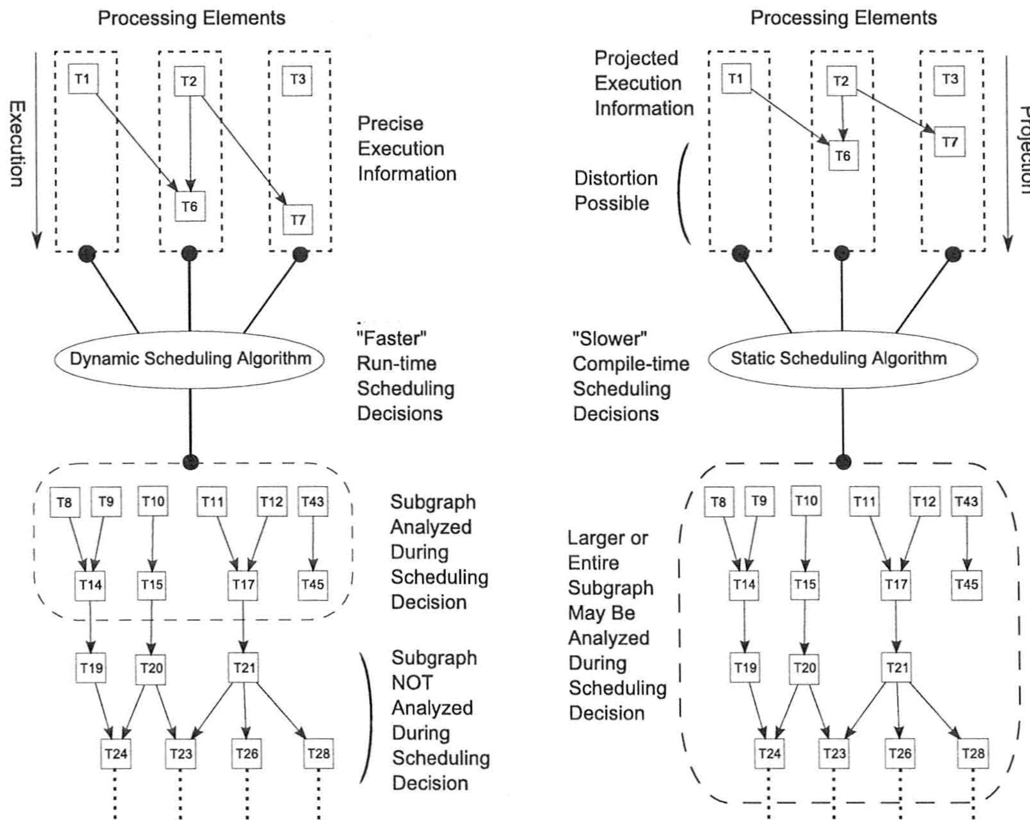

Figure 2.8: Dynamic vs. Static Scheduling Algorithms

Within both static and dynamic scheduling one can further subdivide the multiprocessor scheduling problem. One can look at what has been referred to as the **scheduling and mapping problem**[KA99] where tasks interact and/or depend on one another. In contrast one can look at the case where tasks do not interact or depend on one another (referred to as job scheduling). Job

scheduling is widely thought of as an exclusively dynamic scheduling problem, though it could be considered a static scheduling problem as well, as the dashed line in Figure 2.7 indicates.

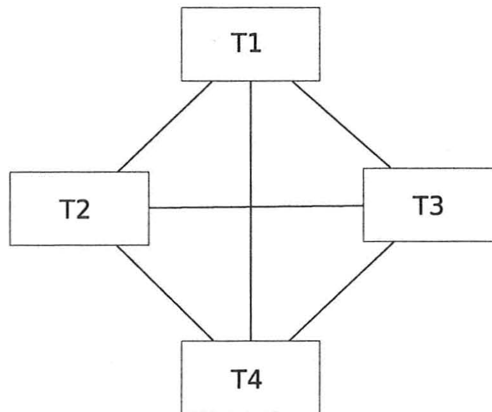

Figure 2.9: Task interaction graph example

Again within the scheduling and mapping problem, one can usefully further subdivide the problem into **task interaction graph** scheduling and **task precedence graph** scheduling. Task interaction graph scheduling involves scheduling a graph where nodes represent tasks and vertices represent required interprocess interaction[Bok81]. All tasks are considered to be executing simultaneously, there is no order between different tasks, as is shown in Figure 2.9. The aim of the scheduling algorithms is to minimize program completion time through a proper mapping of tasks to processors. For instance, by taking advantage of processor specific efficiencies for given tasks, and by minimizing communication costs[Sto77].

Task precedence scheduling dealing with directed acyclic graphs, or DAG scheduling, stretches back almost 50 years to Hu's scheduling algorithm [Hu61]. In DAG scheduling algorithms, tasks are represented by nodes in a graph, and edges represent dependencies between tasks. Tasks do not start until all the tasks that they depend on complete. Tasks and edges can have weights representing computation and communication costs. An example of a DAG to be scheduled and its node and edge weights is given in Figure 2.10.

Problem Specific Scheduling Algorithms

Multiprocessor scheduling algorithms may be built to schedule a particular algorithm. By focusing a scheduling algorithm on a particular problem domain, one can utilize properties unique to the problem domain to further

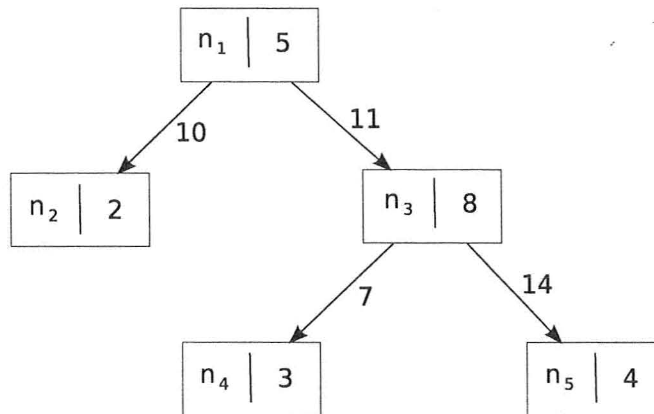

Figure 2.10: Directed acyclic task graph example

optimize performance. A simple example of this is Cannon's algorithm for matrix multiplication[LRF97].

Cannon's algorithm uses a 2D grid of processors. It is assumed they are connected in such a way that the processors directly above, beneath and beside one another communicate more efficient than non-adjacent processors. Algorithm 1 is a pseudocode version of Cannon's Algorithm.

Algorithm 1 (Cannon's Algorithm) *Assume $n = height(A) = height(B) = width(B) = width(A)$, and that we have an $n \times n$ grid of processors P. Each processor $P(i,j)$ will be left with the result submatrix $C(i,j)$ at the algorithm's completion.*

1. *Each $P(i,j)$ processor beings with block submatrix $A(i,j)$ and $B(i,j)$.*
2. *Circular shift submatrices: i-th row of A i positions left, j-th column of B j positions up.*
3. *Multiply submatrices at each process $P_{i,j}$.*
4. *Circular shift $A(i,j)$ submatrices left.*
5. *Circular shift $B(i,j)$ submatrices upwards.*
6. *Repeat steps 3-5 for the remaining submatrices.*

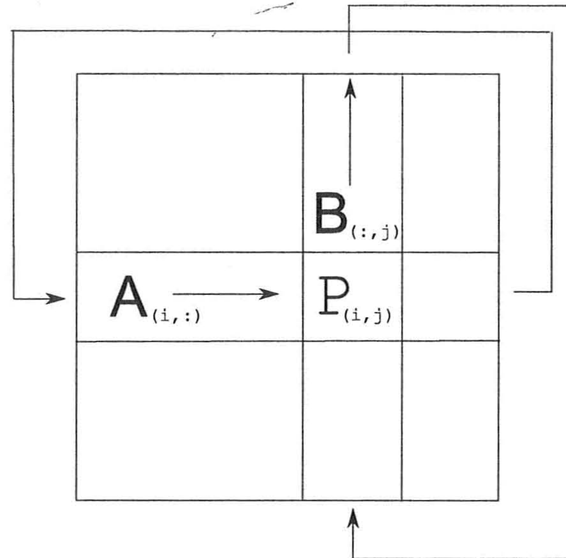

Figure 2.11: Cannon's algorithm computes block $C(i,j)$ at each processor $P(i,j)$.

An illustration of the algorithm is given in Figure 2.11. This algorithm uses the structure of the matrix multiplication operation and its input and output data, as evident in the way it continually redistributes data amongst the processors for matrix multiplication. In a system where communication costs are expensive enough to bottleneck performance when sending and receiving data from a central source, this efficient circular shifting of the data could optimize performance.

Many other multiprocessor matrix multiplication algorithms exist, such as Broadcast-Multiply-Roll[HlJT93], SUMMA[GW97], PUMMA[CDW94], as well as many multiprocessor scheduling algorithms specific to certain problem domains. The important point is that problem specific multiprocessor scheduling algorithms can leverage problem domain knowledge to optimize performance.

Chapter 3

Cell Broadband Engine

The Cell/B.E. is the current target platform for Coconut and the ideas discussed in this thesis. This chapter will give a brief background of the Cell/B.E.'s origins, an overview of the hardware itself followed by an analysis of each major component, current applications of the Cell/B.E., current and expected future versions of the architecture, as well as an explanation as to how the Cell/B.E. overcomes the frequency, memory and power walls discussed in Section 2.1.2. It is important to note that the hardware specifics discussed in this Chapter are for the original Cell/B.E. architecture, with variants discussed specifically in Section 3.7.

3.1 Origins

The Cell/B.E.'s existence originated from discussions between Sony, Toshiba and IBM in the summer of 2000 which occurred as a result of the interest of Sony Computer Entertainment Incorporated (SCEI) in an architecture capable of delivering 1,000 times the computational power of the Playstation 2[KDH+05]. It was decided then that traditional architectures would not meet this need and as a result a more holistic design approach was used, incorporating ideas from wider fields of study than typical in processor design phases, such as software programming models. The eventual performance objective for the Cell/B.E. turned out to be 100 times the computational power of the Playstation 2, and in March 2001 the SCEI-Toshiba-IBM (STI) Design Center was opened in Austin, Texas to develop the technology at the cost of roughly $400,000,000 USD.

3.2 Hardware Overview

As a result of the more holistic design process and ambitious goals, the Cell/B.E. has a unique design that is regarded as revolutionary[Hof06; Gsc07]. This hardware overview is based on the IBM produced and publically available Cell Broadband Engine Programming Tutorial[IBM08c], as well as the paper "Introduction to the Cell multiprocessor"[KDH+05].

The Cell/B.E. is a heterogeneous network-on-a-chip multicore architecture, containing nine processors on a single chip connected by a high-bandwidth memory coherent bus. The heterogeneity is due to the two different types of processing elements: the **PowerPC Processor Element (PPE)** and the eight **Synergistic Processor Elements (SPE)**. The processing elements have been designed with optimal performance of different tasks in mind. The PPE is primarily targeted towards control processing of the SPEs and relatively branch heavy code, and as such is responsible for managing the SPEs and is intended to run the operating system. The SPEs are primarily targeted at data-heavy high performance SIMD floating point computations. Though technically capable of running an OS, SPEs are not intended to do so. The SPEs each contain a **Local Store (LS)** for data and code, as well as their own program counters, and as such even though the PPE must spawn SPE threads for execution they can be conceptualized as independent processors. The processors execute asynchronously from one another; as such much desired synchronization between concurrently executing SPE and PPE threads must be handled explicitly in software.

The **Element Interconnect Bus (EIB)** is the mechanism through which the processing elements communicate, providing data transfer as well as signal and mailbox mechanisms to facilitate interprocessor communication. The EIB uses a ring structure between processing elements for transferring data, and a tree structure for issuing commands. Each of the SPEs contains a **Memory Flow Controller (MFC)** capable of requesting **Direct Memory Access (DMA)** transfers from the EIB, which in turn carries out the transfer request by communicating with the transfer target. In the case of inter-SPE transfers, this is done with the other SPE's MFC. In the case of data transfers between the SPE and off chip main memory, a **Memory Interface Controller (MIC)** provides the interface between the EIB and main memory. The **I/O Controller** acts as the interface between the EIB and I/O devices, including possibly another Cell/B.E. processor. It is split into two separate configurable elements, **IO1F0** and **IOIF1**.

At a clock speed of 3.2 GHz the Cell/B.E. has a total theoretical peak performance of 14.6 GFLOPS double precision or 204.8 GLFOPS single preci-

sion. An overview diagram of the Cell/B.E. architecture is presented in Figure 3.1.

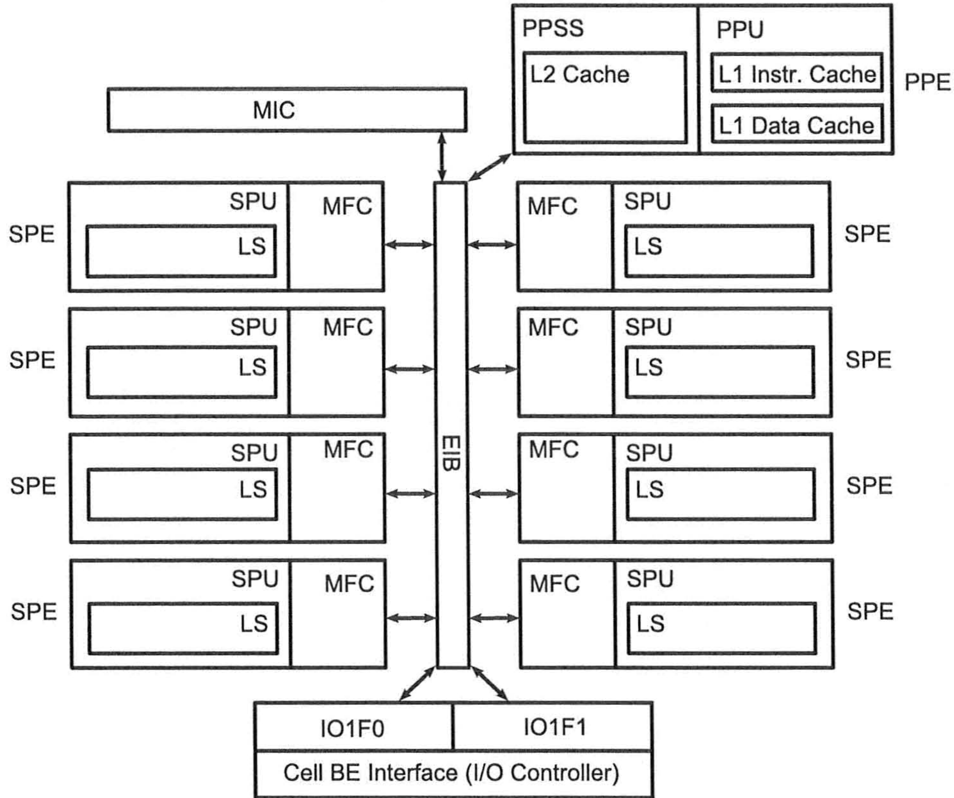

Figure 3.1: Cell Broadband Engine Overview

In Section 3.3 the different processing elements of the Cell/B.E. will be described in more detail. As communication between processing elements is vital to performance in multiprocessor settings[KPP06; BCG+95; JGMR07; CK88], we undertook a thorough literature search of information pertaining to the EIB and the related interface unit's design and performance. The results of this search (presented in Section 3.4) were used in the design of the Performance Simulator tool (presented in Chapter 6) developed for the CMF.

3.3 Processor Element Design

The heterogeneity of the Cell/B.E. allows for processors to specialize performance towards certain tasks, increasing total system performance. The PPE is specialized towards branch-heavy code and is described in Section 3.3.1, the

SPEs are specialized towards raw floating point operation performance and are described in Section 3.3.2.

3.3.1 PowerPC Processor Element (PPE)

The PPE is a dual-threaded, general purpose 64-bit PowerPC-Architecture compliant RISC processor.

The PPE is actually made up of two units: the **Power Processor Unit (PPU)** and the **Power Processor Storage Subsystem (PPSS)**. The PPU handles execution of instructions, and the PPSS handles internal memory requests from the PPE as well as external communication with processors or I/O devices. A conventional cache hierarchy is supported, with 32KB first-level instruction and 32KB first-level data caches included in the PPU and a 512KB second-level cache included in the PPSS.

Power Processor Unit (PPU)

The PPU can be thought of from the standpoint of a programmer as a dual core processor with two independent processing units. Two simultaneous threads of execution are supported, with most non-system-level resources such as caches and queries shared by the state of both execution threads. Two simultaneous threads of execution are part of the reason the PPE is suited for OS-like functions such as behaving as a server to SPE clients. Another reason is the inclusion of branch prediction, which is absent from the SPEs. The PPU contains the complete set of 64-bit PowerPC registers, allowing for programs written for PowerPC architectures to generally execute without modification. The PPU contains an Instruction Unit (IU) for instruction fetch, decode, branch, issue and completion, a Fixed Point Execution Unit (XU) for load and store instructions and a Vector Scalar Unit (VSU) for vector and floating-point instructions.

The PPU pipeline depth is 23 stages, bucking a trend of increasing pipeline depths, such as the 31-stage Pentium 4 pipeline[ZHNB06].

Power Processor Storage Subsystem (PPSS)

The PPSS contains the 512KB second-level instruction and data cache, various queues and the **Bus Interface Unit (BIU)** to handle requests to the EIB. The second-level cache allows for software control over cache resources via replacement-management tables.

3.3.2 Synergistic Processor Elements (SPE)

The SPE is a 128-bit RISC processor built with a new instruction-set designed for data-rich compute-intensive media applications demanding high floating point operation performance. The SPE itself is split into two components: the **Synergistic Processor Unit (SPU)** contains the LS and is responsible for computation, and the aforementioned MFC is responsible for communication with the EIB.

Synergistic Processor Unit (SPU)

The SPU includes 128 registers, all of them 128-bits wide, as well as the LS responsible for storing both instructions and data. This large register file allows instructions to be ordered to hide instruction execution latencies, part of the reason the SPE is particularly efficient with regard to floating point performance. The instruction set implemented by the SPU is new, and is referred to as the SPU Instruction Set Architecture - it is purpose built for the Cell/B.E. architecture.

The SIMD instructions allow for different element widths, importantly 2 x 64-bit double precision and 4 x 32-bit single precision. Two instructions per cycle may be issued, with one slot for floating-point operations, and the other for loads and stores, branches and byte permutation operations. The single precision SIMD instructions take six cycles and are fully pipelined, whereas the double precision SIMD instructions have a maximum issue rate of one instruction per seven cycles. This is the reason for the disparity in single precision and double precision floating point operation performance. The figure of 25.6 GLFOPS of single precision performance comes from the fact that single precision multiply-add instructions count as two floating point operations, and as a result we have $8 \times 3.2 \times 10^9 = 25.6 \times 10^9$ GFLOPS.

The LS is un-translated and unprotected storage when accessed by its SPU. The SPU issues DMA instructions to transfer data to and from the LS from main memory. The LS has a 256KB capacity for both data and code. This means that proper utilization of LS space through small computational kernels operating on appropriate sizes of data is key to deriving optimal performance from the Cell/B.E..

Memory Flow Controller (MFC)

The MFC is responsible for SPE communication with the rest of the chip as it physically connects to the EIB. DMA transfers, signaling and mailbox mechanisms are provided to facilitate communication. The MFC allows for DMA

transfers to be concurrent with execution on the SPU, which gives software the ability to hide data transfer latency with computation. This process and the internals of the MFC will be discussed in more detail in Section 3.4.

3.4 Communication Architecture Design

The communication architecture is important to the overall performance of the Cell/B.E., and as a result both its design and performance has been extensively studied in the literature[KPP06; CS06; JGMR07; CRDI07; VKJ+07; AP07b; AP07a]. In this section we will outline the design of the EIB in Section 3.4.1, as well as present an examination of the behaviour of the EIB during DMA transfers in Section 3.4.2.

3.4.1 Design Overview

The description and understanding of the Cell/B.E. communication architecture's design and behaviour presented in this section is derived particularly from the work of Thomas Ainsworth and Timothy Mark Pinkston in this area[AP07b; AP07a], and that of Michael Kistler, Michael Perrone and Fabrizio Petrini[KPP06].

As the communication architecture is closely coupled to the notion of memory architecture in the Cell/B.E., one point should be made about the Cell/B.E. and its memory structure: main storage is the effective address space that includes main memory, each SPE's LS, and memory mapped registers. So when we speak of transfers between main memory and SPEs, it is helpful to understand that these transfers are occurring in the same effective-address space. This also means that the Cell/B.E. can be described as a distributed shared memory architecture, as discussed in Section 2.5.3.

Communication takes place over the EIB, with each of the PPE, eight SPEs, MIC, IOIF0 and IOIF1, containing a BIU as an interface to the EIB. Each of the components connected to the EIB is capable of transferring 25.6 GB/s, both inbound and outbound, for a total of 51.2 GB/s cumulative inbound and outbound bandwidth. The exception to this is the I/O controller, whose two components IOIF0 and IO1F1 together have a peak bandwidth of 25 GB/s outbound and 35 GB/s inbound. This bandwidth is actually configurable between the two units, such that one unit could have 15 GB/s inbound and 15 GB/s outbound and the other 20 GB/s inbound and 10 GB/s outbound. The cumulative theoretical bandwidth of the EIB is not found simply by multiplying the bandwidth of these units however, as will be explained.

Three main types of communication are supported over the EIB:

- Signal communication
- Mailbox communication
- DMA transfers

Signal and mailbox communication are meant to be low latency signaling mechanisms for synchronization between the processing elements. DMA transfers are meant for transfer of blocks of data and are the only method supported of moving data from an SPE's LS to and from main memory. Execution at each processing element, SPE or PPE, continues concurrently after communication requests have been made. In the case of the SPE, requests are handled by the SPE's MFC, in the case of the PPE, requests are handled by the PPE's PPSS. This allows for code execution to hide data transfer and communication latency.

Signal communication is supported by two 32-bit signaling channels in each SPE, Sig_Notify_1 and Sig_Notify_2. The SPE can read its signals with the read-blocking SPU channels SPU_RdSigNotify1 and SPU_RdSigNotify2. These channels are written to by the PPE or another SPE using memory-mapped addresses. The communication of the signal data itself occurs in the same way as a DMA transfer does over the EIB. However the signal channels allow for easier group communication between processors due to an ability to treat write operations as logical OR operations, allowing for message accumulation.

As with signal communication, mailbox communication occurs over the EIB in the same way as a DMA transfer. Each SPE has two write-blocking outbound single-entry mailboxes, and a read-blocking inbound four-entry mailbox. Mailboxes have a 32-bit length. The PPE performs writes to and reads of memory-mapped addresses to use an SPE's mailbox. Mailbox communication is more suited towards point-to-point server-client style communication than group communication.

DMA transfers can be up to 16KB length, by a multiple of 16 bytes, as well as 1,2,4 or 8 byte lengths. All packets of data sent over the EIB are 128-byte length, and are sent pipelined as 8 16-byte fragments. Thus if a transfer is greater than 128-bytes in length, it must be sent in multiple packets. Transfer latency is improved if the size of the transfer is a multiple of 128-bytes, and if the effective and local storage addresses are 128-byte aligned. When DMA commands are issued, a 5-bit Tag Group ID is assigned by software, allowing for the completion status of a specific DMA transfer to be checked on, or

waited on, later in execution. Several categories of DMA transfer commands are worth mentioning.

There exists in the form of a single DMA command, a DMA list command which can request a list of DMA transfers, up to 2048, for a maximum of 32MB of data transfer to or from the LS specified by a single command. The list itself must be stored in the LS.

There also exist fence and barrier commands which allow software to affect the order in which packet transfers for DMA transfers are carried out. In the absence of these mechanisms, when multiple DMA transfer commands have been issued, the individual packets of data required to fulfill the commands will be transfered out of order with respect to the order the commands were issued in, when possible and beneficial. A fence DMA transfer command will have the effect of ordering the command against all preceding commands in a tag group, such that it will not begin its packet transfers until all preceding commands have completed. A barrier command will order the command against all preceding and succeeding commands in the tag group. There also exists a barrier command that will order the command against all previous and succeeding commands, regardless of the tag group.

Finally also worth mentioning are the DMA transfer synchronization commands which are designed to support more complex synchronization between elements. The *getllar* command when issued sets a reservation on a specified area of main storage by modifying a synchronization variable. If another SPE or PPE then modifies this synchronization variable, the reservation is lost. A *putllc* command that attempts to write to the memory location will only work if the SPE issuing it has the reservation. If the SPE has lost its reservation, then it must issue another getllar command for the location before it can write to it with a putllc command.

The EIB operates at 1.6 GHz, half the clock frequency of the processing elements. Data transfer over the EIB is lossless and pipelined, and takes place over four 16-byte width rings, 2 running clockwise and 2 running counter-clockwise. In addition to the four rings, the EIB also contains a shared pipelined command bus which has a tree network structure connecting all elements, as well as a pipelined central data arbiter which has a star network structure connecting all elements. Each of the rings is capable of 3 concurrent data transactions as long as they do not overlap. This may lead one to believe that the EIB bandwidth would be $1.6 \times 10^9\text{GHz} \times (3 \times 4) \times 16\text{B} = 307.2\text{GB/s}$. It turns out that this is not the case due to snooping capability limitations of the command bus, which allows for only 8 concurrent data transactions across the four rings.

3.4.2 EIB DMA Transfer Behaviour

The way in which the three networks internal to the EIB work together to perform data transfers can be explained best by looking at how a DMA transfer occurs on an SPE, as shown in Figure 3.2. DMA transfers are referenced from the perspective of the SPE, as either *puts* in the case of an outbound transfer or *gets* in the case of an inbound transfer. When an SPE issues a DMA get or put operation, it first uses its channel interfaces to put the command into the MFC's **DMA Controller (DMAC)**. Specifically, the command is put into the DMAC's *MFC SPU command queue*, which is for commands issued by the SPE. The *MFC proxy command queue* is for commands issued by other elements such as the PPE or MIC. The MFC SPU command queue has 16 slots, and the proxy command queue has 8 slots, though IBM documentation recommends this specific size not be assumed by software. These queue sizes are important however, as an attempt to insert a command into a full queue will result in performance degradation.

Once inside the SPU command queue, the DMAC will eventually select the command for processing. The protocol the DMAC uses for selecting commands could not be found in public documents, but we are told that commands in the SPU command queue will take priority over those in the proxy command queue and that the DMAC alternates between get and put commands. When a command is selected for processing, a command bus request is put onto the BIU queue. Before the command is put onto the BIU queue, two things could happen to it. If the command is a DMA list command, the DMAC will request a list element from the local-store interface, and when this list element is returned the command entry is updated. The command must then be reselected in order to send a bus request to the BIU. The other possibility is that the command requires address translation, in which case the DMAC queues the command to the MMU for translation from a Translation Lookaside Buffer (TLB). Once again after the command is processed, it must be reselected by the DMAC for processing. The command will remain in the DMAC command queue until all of the necessary packet transfers (i.e. command bus requests) have been completed, at which point the command is removed from the queue.

Commands for data transfer issued by a component's BIU are accepted by the command bus, shown in Figure 3.3, which is responsible for distributing commands, setting up transactions and handling coherency. Each unit has a limited number of commands that may be issued to the EIB at a single time, and this is controlled via a token mechanism. When a BIU issues a command to the command bus, the command bus holds the token. The BIU can only issue

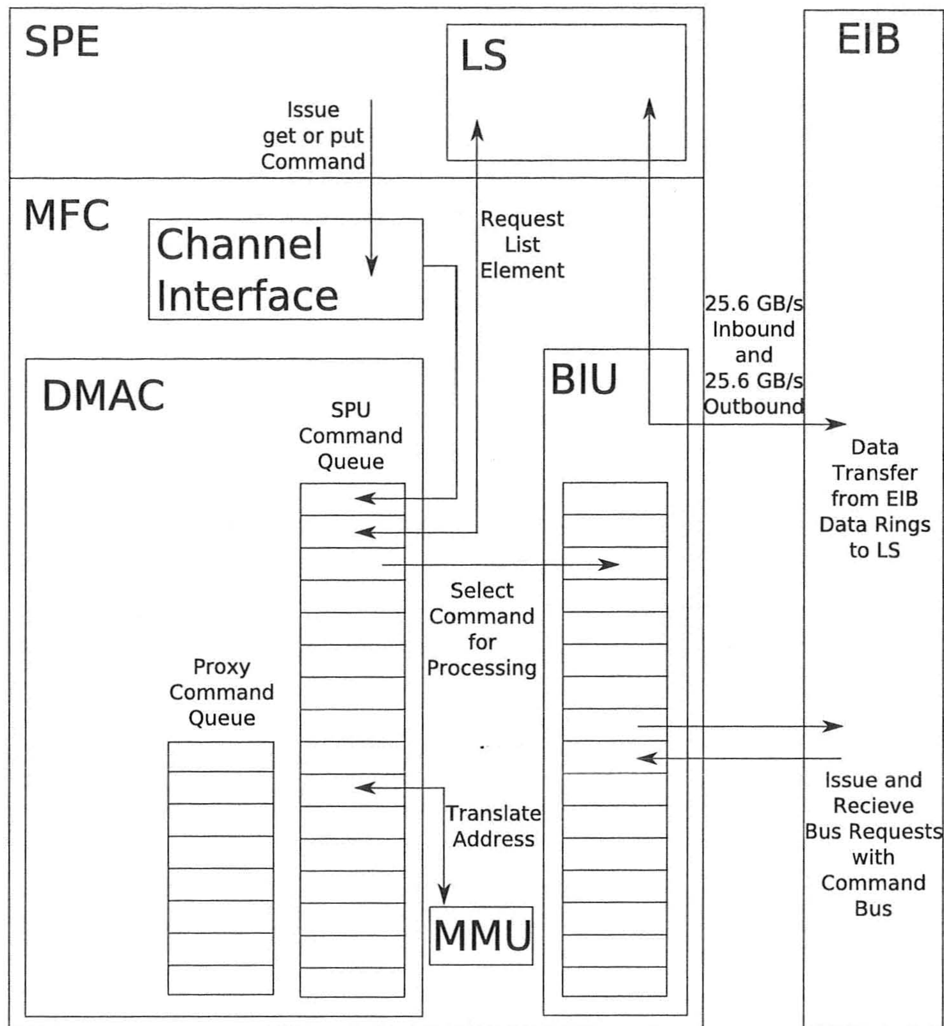

Figure 3.2: SPE DMA Transfer Internals

(or accept) commands if it has available tokens. While the EIB can support up to 64 requests from each element, only the MIC supports this maximum, as all other elements (such as an SPE's MFC) support only 16 outstanding requests.

The command bus connects to each BIU with an Address Concentrator (AC), which form a tree structure with root node AC0. Together the ACs provide round robin access to the command bus, with two levels of priority.

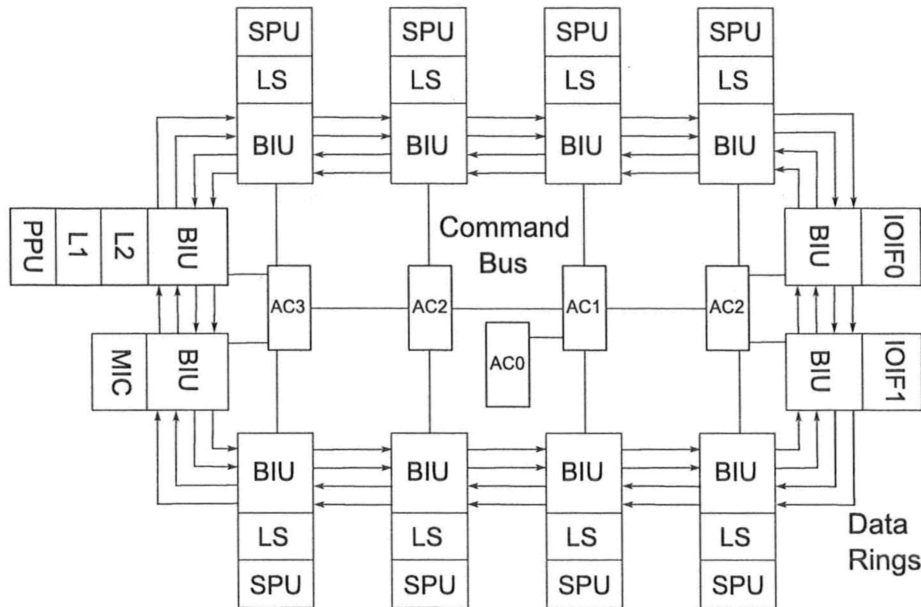

Figure 3.3: EIB Command Bus

The MFC has highest priority, and all other bus elements are of lower priority, but of equal priority to each other. We could not find in the literature the exact level of priority the MIC has over other bus elements, but we suspect it to be enough priority to fully utilize the MIC bandwidth given the well understood potential of main memory access to act as a bottleneck to performance.

The command bus can initiate one coherent transaction (with respect to main memory) every two bus cycles, and a non-coherent transaction every bus cycle. This design feature, combined with the pipelined nature of the EIB data transfer rings is what limits the maximum number of in-flight transfers to eight. This also gives the EIB an overall bandwidth limitation of 204.8 GB/s. The coherent-only transfer bandwidth is 102.4 GB/s. SPE-SPE transfers are non-coherent, whereas transfers involving the PPE, MIC or I/O Controller (in the case of multiprocessor Cell/B.E. configurations discussed in Section 3.7.5) are coherent. Non-coherent transactions may be interleaved with coherent transactions, to ensure that at each bus cycle a ready transaction is initiated.

After the command bus initiates a transaction, the next step is for the data arbiter, shown in Figure 3.4 to select one of the 4 data rings to carry out the packet transfer. The data arbiter has the same two-priority-level round

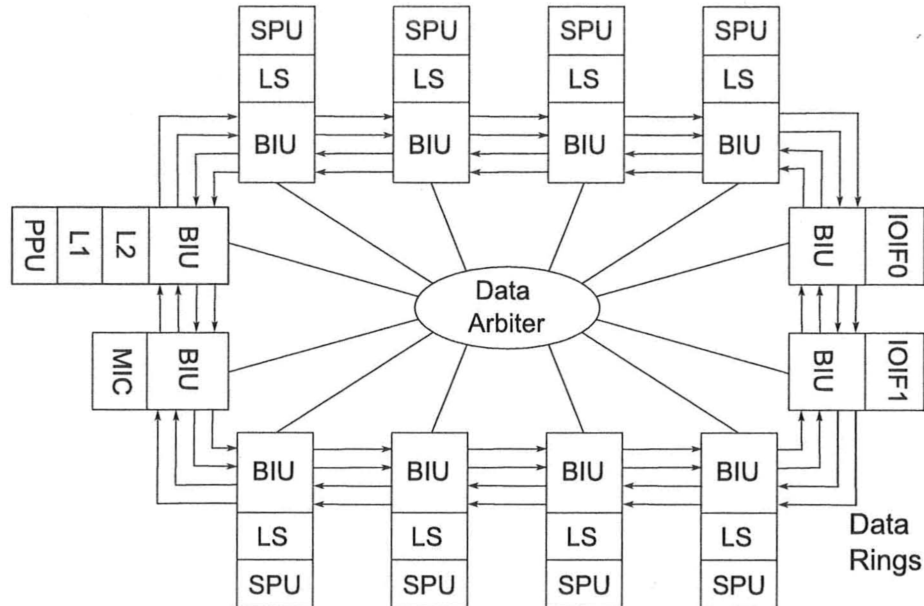

Figure 3.4: EIB Data Arbiter

robin access scheme as the command bus. The data arbiter will always select one of the two rings that allows for the shortest path for the data transfer. In other words, packets will always transfer along the shortest path, no more than 6 *hops* along BIU connections to the EIB. Hops will be used to refer to packet transfer path length. This matters because one can imagine scenarios where the longest transfer path may have lower latency due to a high level of traffic over the shortest path, but the EIB will still not allow for the longer transfer path. Each data ring can support a new transaction every 3 bus cycles so long as the paths do not overlap. This combined with the 8 hop packet length is what allows us to say that an EIB data ring may support 3 simultaneous transfers.

Once the data arbiter selects a ring and the packet transfer is made, the final step in a DMA transfer is the receiving of the packet by the BIU. In the case of an SPE, this is done by having the BIU transfer the packet to the MFC in one cycle, and in the next cycle the MFC transfers the data into the SPE's LS.

3.5 Communication Architecture Performance Analysis

The performance of the EIB is critical to the performance of the Cell/B.E. in many problem spaces. This is due to the limited on-chip communication bandwidth relative to computational power, particularly with respect to main memory bandwidth. The purpose of this section is to analyze the performance of the EIB as a communication architecture, in order to illuminate the importance of the EIB to overall chip performance.

3.5.1 Main Memory Bottleneck

The main memory bandwidth provided by the MIC is a performance bottleneck for the Cell/B.E., referred to as *the* bottleneck to the performance of most algorithms[VKJ+07]. This bottleneck will be illustrated by examining the amount of data re-use necessary to prevent the bottleneck from impacting performance, to give the reader an appreciation of the severity of the problem.

Dongarra et al. describe the problem of "main memory access rate" in the following clear and concise way[BLK+07]. The main memory of the Cell/B.E. is accessed through the MIC, with a peak theoretical outbound transfer rate of 25.6 GB/s. The performance of a single SPE is 25.6 GFLOPS for single precision floating point operations. As single precision floating point values require 4 bytes to represent, this means that in order to hide transfer latency with computation in the case of a single SPE streaming data from main memory, one would have to perform four floating point operations on each value. Now considering that in reality we have 8 SPEs executing at 25.6 GFLOPS, this would require $8 \times 4 = 32$ floating point operations per floating point value transferred out from main memory to hide the latency. This metric is referred to as *computational intensity* by others in the literature[WSO+07]. This is pointed out by the team as being a particular problem for sparse scientific computing operations, limiting the chip to 12% of its theoretical peak. This also means that as we use more SPEs in a server-client communication style, there is a definite potential for decreasing rates of return on performance. Which in turn raises concerns about the scalability of the Cell/B.E. architecture, if main memory access bandwidth cannot scale with the number of SPEs.

To take another look at the problem, we consider matrix multiplication. IBM and Sony researchers were able to show that they were able to get close to the theoretical peak performance (204.8 GFLOPS) of the Cell/B.E. for

single precision matrix multiplication (512×512 and 1024×1024 size), 201 GFLOPS of performance were achieved[CRDI07]. Through double buffering to hide latency, the MIC was not a significant bottleneck to performance.

This should not be too surprising, given an analysis of the problem. Assuming we use 64×64 block sizes for the matrix, then supplying all 8 SPEs with 2 (A and B) single precision floating point (4 bytes) input matrix blocks would require 8 SPEs $\times$ 2 input matrices $\times$ 4 bytes $\times$ (64×64) block size $= 64^3$ bytes of data transfer from the MIC. Given that the MIC is capable of 16 bytes of outbound data transfer per bus cycle, and that the bus executes at 1.6 GHz, we have a main memory transfer rate of 16 bytes $\times (1.6 \times 10^9)$ cycles per second $= 25.6 \times 10^9$ bytes per second. One computes that it takes 64^3 bytes / (25.6×10^9) bytes per second $= 10,240$ nanoseconds to transfer the required data out to the SPEs. As $n \times n$ square matrix multiplication requires n^3 operations, and SPEs each have 25.6 GLOPS performance in single precision, we get that it takes 64^3 operations / (25.6×10^9) operations per second $= 10,240$ nanoseconds for each SPE to perform the matrix multiplication. In other words, it takes exactly as much time to send the data for processing as it does to process it, which means through a buffering mechanism we should be able to easily hide the latency of data transfers. This explains why matrix multiplication should indeed not cause the MIC to be a bottleneck to performance.

Table 3.1: Operations/Value Effect on Transfer vs. Computation Time

Operations per Value	**Computation Time (ns)**	**(Transfer / Computation) Time**	**Potential Latency Hiding**
32	10,240	1	100%
16	5,120	2	50%
4	1,280	8	12.5%
1	320	32	3.125%

This is consistent with the idea that it takes 32 operations per single precision floating point value to hide the latency in the case of 8 SPEs, as 64^3 operations / $(2 \times (64 \times 64))$ values $= 32$ operations per value. But if we were to hold the data requirement of 2 64×64 blocks constant, but decrease the amount of computation required, we find that communication rapidly becomes a constraint to performance. This is not unreasonable, as matrix multiplication

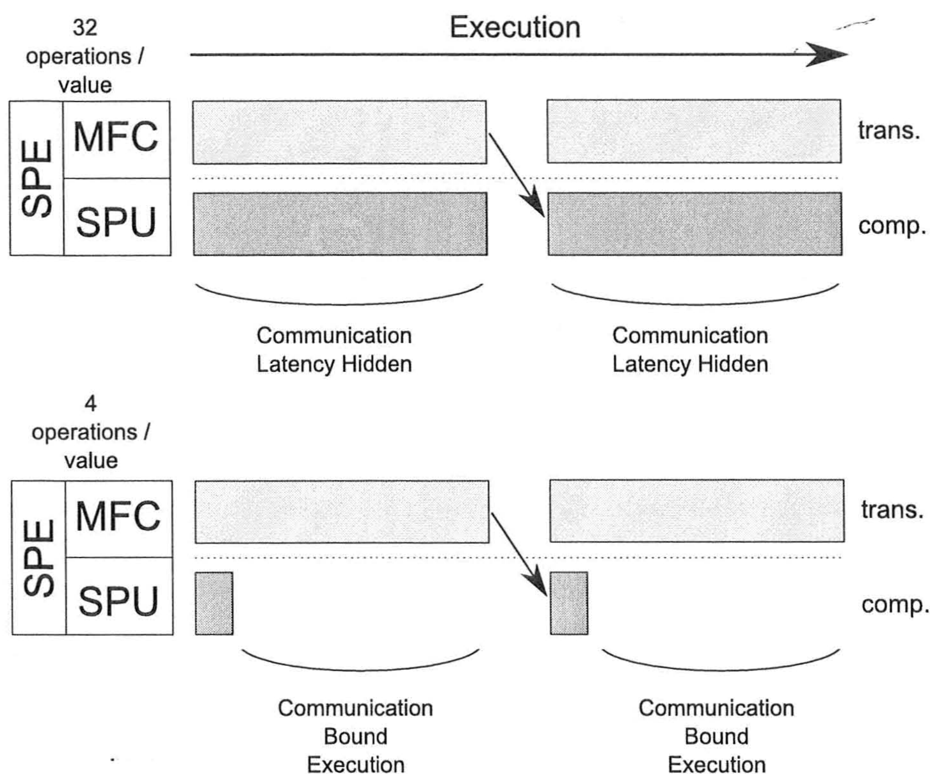

Figure 3.5: Communication Latency Effect

has cubic time complexity (implying heavy re-use of at least one input value), and many useful algorithm kernels (such as search or sort kernels) will not have such high computational complexity. If we reduce the amount of operations per value to 4, we find that computation will only take $((64 \times 64) \times 2) \times 4$ operations / (25.6×10^9) operations per second = 1,280 nanoseconds. Now it takes eight times as long to transmit the data as it does to process it, and communication can no longer be hidden to such a degree through buffering. As a result, presumably only 12.5% of communication cost could be hidden through buffering. This results in a communication bound execution time due to main memory access latency, and the effects of this are shown in Figure 3.5. The effect on the ratio between transfer and computation time for several operations per value scenarios is shown in Table 3.1.

It should also be stated that main memory access may be less than the 25.6 GB/s provided by the MIC due to the need for the MIC to ac-

cess a memory bank to get the actual data, exasperating the problem further. When describing the performance of the MIC, the same IBM and Sony researchers[CRDI07] tell us that memory operations such as refresh and scrubbing typically reduce bandwidth to about 24.6 GB/s. The authors also tell us that if streaming read and write requests to the MIC are intermingled, the effective bandwidth may drop to roughly 21 GB/s.

As one can see through this example, main memory bandwidth through the MIC is a significant potential bottleneck to performance, unless perhaps data is somehow re-used sufficiently.

3.5.2 Communication Pattern Bottleneck

In order to cut down the amount of data that must be transfered from main memory, one may be tempted to share data amongst the SPEs in some form. Perhaps by sharing input values, or perhaps by passing intermediate values from SPE to SPE in a sort of pipeline style computation. These ideas may indeed be effective at alleviating the critical main memory bottleneck. However without consideration of the communication pattern between processing elements, inefficient transfer times could theoretically result.

As an illustrative example of an inefficient communication pattern, consider the case of all 8 SPEs simultaneously sending 2 or more 16KB blocks of data to their counter-clockwise neighbour. This would cause both of the 2 counter-clockwise EIB data rings to become saturated with data transfers, where as the clockwise EIB data rings would have completely unused bandwidth. As each ring is only capable of 3 simultaneous transfers, and each SPE would be attempting 2 or more transfers simultaneously, only 6 transfers of the 16 or more attempted transfers could execute simultaneously, capping potential bandwidth at 153.6 GB/s. This despite the fact that the EIB is capable of 8 simultaneous transfers and 204.8 GB/s of bandwidth. So bandwidth is already negatively impacted by this arrangement without considering other factors.

Now if one considers additional transfers and communication from the MIC to each of the SPEs, one can reason that 4 of the SPEs will receive data over the saturated data rings, and 4 of the SPEs will receive data over the unsaturated data rings, given the 6 hop shortest path rule. The MIC transfers will take priority over SPE transfers due to the design of the data arbiter, and so the SPE-SPE transfers occurring over the already saturated rings will be further impacted. For example, if the MIC attempts to send its maximum amount of 25.6 GB/s outbound data to an SPE over an already saturated ring direction, this will limit bandwidth available to the SPE-SPE transfers to 128

GB/s assuming absolute priority of MIC transfers over SPE-SPE transfers. The described situation is illustrated in Figure 3.6, with 'cycle transfer rates' referring to those transfers occurring as a cycle of SPE-SPE transfers.

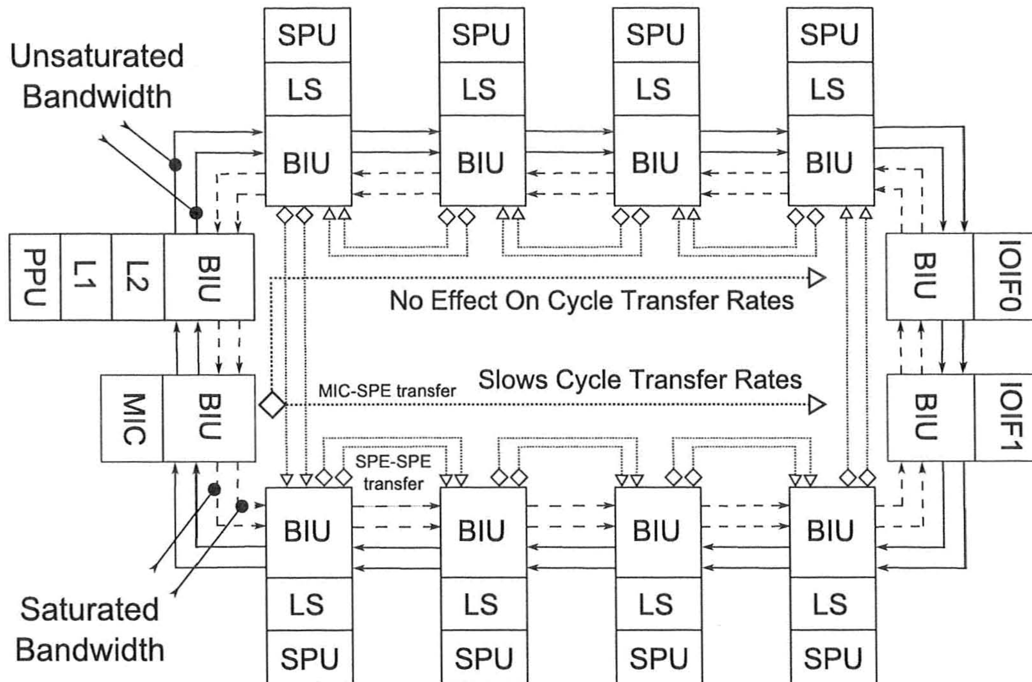

Figure 3.6: Inefficient Cycle Communication Pattern

Another possibility is that 2 SPEs are communicating heavily, with 2 or more transfers occurring simultaneously and constantly in *both* directions. It is expected that the inbound and outbound bandwidth limits of each SPE's BIU would become saturated. These SPEs would also consume half of the available bandwidth on the 4 rings connecting them. Assuming fair allocation of the rings to data transfers, which according to EIB documentation can be expected, this *could* theoretically negatively impact the transfer rates between units not related to these 2 SPEs. This is because the shortest path transfer design feature of the EIB which prevents transfer paths greater than 6 hops will send the DMA transfer packets along the shortest path and not necessarily the path with the most available bandwidth.

This situation is illustrated in Figure 3.7, with the shortest communication path between 2 SPEs containing the already heavily loaded EIB rings. Though technically the bandwidth of the data rings could support precisely the maximum amount of transfers occurring over the 4 data rings between the 4 SPEs (102.4 GB/s), if *any* other data transfers from the MIC or any other

computational unit were attempting to use these rings increased communication latencies would be expected. Locating heavily communicating SPEs out of each other's shortest-distance packet transfer paths is thus a good practice for optimizing communication bandwidth usage; overlapping paths may be expected to cause performance degradation.

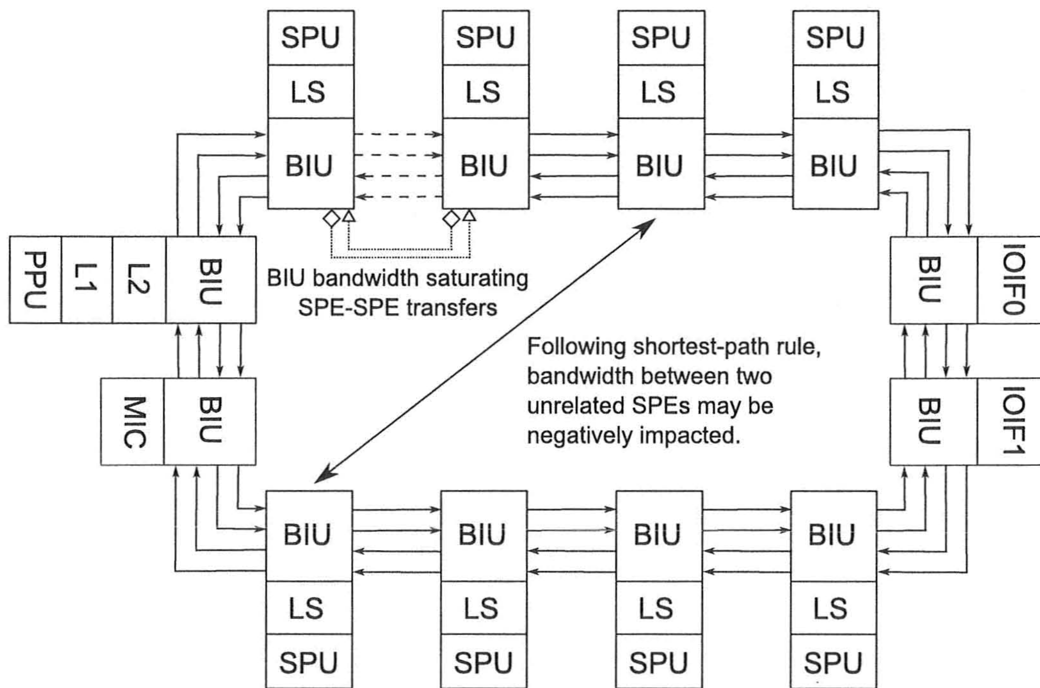

Figure 3.7: Inefficient Heavy Inter-SPE Communication Pattern

An awareness of the EIB's topology with respect to SPE and other element's relative positions thus seems important to preventing communication transfer latency from being a larger cause of performance degradation than necessary. It is disappointing then to learn that the current API does not allow the programmer to know the physical location of the SPEs and to control the layout[JGMR07].

3.5.3 Performance Tests

Much actual performance testing of the EIB has been done in the literature [KPP06; JGMR07; CRDI07; VKJ+07]. A survey of some of these results is useful to appreciate the non-theoretical real-world capabilities of the EIB.

Cell Broadband Engine Architecture and its first implementation - A performance view

In the paper "Cell Broadband Engine Architecture and its first implementation - A performance view", IBM and Sony researchers performed tests on the SPE to SPE DMA transfer abilities of the Cell/B.E., performed on an actual Cell/B.E. running in a hypervisor mode without an OS[CRDI07; Dal09]. Each SPE of the 8 available SPEs was paired with another SPE, and one of the SPEs was made to perform streaming writes to the other SPE which performed streaming reads, and vice versa. The test was performed 7 times, pairing different SPEs with each other to evaluate the effectiveness of different communication patterns. In 5 of the 7 tests, near peak bandwidth was achieved with an aggregate EIB bandwidth between 186 to 197 GB/s (91% to 96% of peak bandwidth). Only 1 of these 5 tests achieved 186 GB/s, with the rest achieving 197 GB/s. It is curious that this test is the only test where none of the communication paths overlap, and yet it has lower bandwidth than others where paths do overlap.

The remaining 2 tests had significantly less bandwidth. In one case where the SPEs are furthest apart at 6 hops, the bandwidth achieved was only 78 GB/s (38% of the theoretical peak). Because only one transfer may happen on each ring (due to the overlapping of transfer paths), we would have a peak bandwidth expectation of 102.4 GB/s. The authors suggest that the bandwidth was limited beyond this due to limitations in the arbiter design. In the other badly performing case, 95 GB/s of bandwidth (46% of theoretical peak) is achieved when the SPEs are five hops away from each other, and because of path overlap transfers on the same ring are still prevented from taking place.

These tests tell us a few useful things. Firstly, that even under ideal conditions where the maximum bandwidth should be achievable, the EIB falls short of its theoretical peak, but still gets 90% or more of the theoretical peak. Secondly, that if paths overlap in such a manner as to prevent simultaneous data transfers, significant performance degradation may result. Note that these experimental results are presented in Table 6.2, we compare our Performance Simulator tool against them to verify its accuracy.

Cell Multiprocessor Communication Network: Built For Speed

In the paper "Cell Multiprocessor Communication Network: Built For Speed", several tests involving SPE to SPE and SPE to main memory communication performance provide useful information about the EIB[KPP06]. The tests were

performed on an experimental evaluation board, and were correlated with results from an *internal* version of the IBM Full-System Simulator which includes performance models for the MFC, EIB and memory subsystems. They say that the correlation between the simulator and hardware results was "good" and that the simulator allowed them to observe system behaviours that would otherwise be impossible to observe. Though not a performance test, this information about how the tests were conducted tells us that it is possible to performance model the EIB and related components in software and to gather useful information from doing so.

Firstly, the latency and bandwidth for get and put DMAs to another SPE from an SPE, and to main memory from an SPE, were tested and presented in graphs (preventing the precise numerical results from being presented here). Two things stick out, one that for transfer sizes below the packet length of 128-bytes, we have exactly the same performance. And though noteworthy, it is expected, because we know that packet sizes are 128-bytes regardless of the data size actually being transferred. Also we notice that the latency of gets from main memory is about 1,100 nanoseconds compared with about 750 nanoseconds for puts to main memory and gets and puts to another SPE. This seems to confirm that memory bank access itself may bottleneck the potential outbound performance of the MIC.

Another series of tests compares get and put bandwidth to and from another SPE's LS, and to and from main memory, but for batches of DMA commands ranging from a single DMA to 32 DMAs before waiting for DMA completion. In these tests again it is the DMA get from main memory that lags in performance, with between roughly 15 GB/s to 18 GB/s of bandwidth being achieved, which scales upwards with larger batch sizes. The other gets and puts achieve from about 22 GB/s to about 25 GB/s of bandwidth, again scaling with the larger batch size.

The next test was a "hot spot" test, examining what would happen if some or all SPEs targeted another SPE's LS or targeted main memory, with either streaming gets or puts. In the case of gets and puts to another SPE's LS, near peak performance of about 25.6 GB/s is achieved. Interestingly, in the case of gets from the MIC, in the case of a single SPE we again see much less than the desired 25.6 GB/s outbound MIC performance, with performance of about 17.5 GB/s. However in the case of 2 or more SPEs issuing gets from main memory, the performance jumps to the expected 25.6 GB/s! And strangely it is puts to main memory which achieve worse than the theoretical peak in this situation, with only about 24.5 GB/s of performance in the case of 8 SPEs. These results are referred to as counterintuitive by the authors.

Another series of tests were performed to examine how different pat-

terns of communication would perform. In the complement pattern, each SPE executes a sequence of transfers to a fixed target SPE which is determined by complementing the bit string identifying the source SPE. This pattern achieved virtually optimal EIB bandwidth at 200 GB/s (98% of theoretical peak). This shows that under the right conditions the EIB is capable of performance extremely close to its theoretical peak. The pairwise pattern was also tested, where each SPE i communicates with its logical neighbour $i + 1$. The authors note that the SPEs are not necessarily adjacent physically in hardware. In the case of a put operation to facilitate the DMA transfer, the performance was 40 GB/s worse than in the case of a get operation to facilitate the DMA transfer which was itself very near peak performance at about 195 GB/s. The authors tell us that this is explained by the fact that gets have better contention resolution properties than puts under heavy loads. Another case referred to as uniform communication was tested, where each SPE randomly targeted another SPE's LS for DMA transfer. This resulted in very poor EIB bandwidth of 80 GB/s.

Returning to the "hot spot" test, in the case of puts to main memory and gets from main memory, the author's show us a graph of the distribution of latency times for each individual DMA transfer. The distribution shows an average of about 5.6 microseconds, with a worst case latency of about 13 microseconds. This tells us that the transfer latency may vary for similar transfers with the same source and target destination, with perhaps a 200% or greater transfer time difference. It is also tells us that the network is reasonably fair even under these extreme conditions, in the sense that no single DMA transfer was held up in transit to an extreme degree (orders of magnitude greater) relative to other transfers. In other words, even under strained conditions, we shouldn't expect vastly divergent transfer times for similar transfers.

3.6 Overcoming the Frequency, Memory and Power Walls

The Cell/B.E. overcomes the frequency, memory and power walls through its unconventional design. The fact that the Cell/B.E. does overcome these major walls to performance is part of the reason Coconut has decided to target the Cell/B.E. and so will be briefly reviewed. The explanation here is derived from the Cell/B.E. Programming Tutorial[IBM08c].

3.6.1 Frequency Wall

The frequency wall is overcome by heterogeneity between the SPEs and the PPE. The SPEs each have a large registry file, which allows for more in-flight instructions, and eliminates the overhead involved with out-of-order instruction execution. The SPEs also have local storage, which allows for memory transfers to occur concurrently with computation, and eliminates the overhead of speculation. The PPE in contrast is built to execute two threads simultaneously, eliminating the overhead of trying to optimize for single-thread performance. By specializing the PPE and SPE processing elements, the overhead of certain features normally desired in a more general processing element can be eliminated, which allows for higher frequency.

3.6.2 Memory Wall

The Cell/B.E. combats the memory wall through its 3-level memory hierarchy consisting of main storage, local stores on each SPE and the large registry file of each SPE. The multi-layered memory hierarchy allows for programmers to keep data close to where it is being used with creative algorithms exploiting data locality, similar to cleverly written sequential programs targeting shared-memory multicore systems with a cache hierarchy. However an added benefit of the Cell/B.E. memory hierarchy is that its network-on-a-chip design allows for more in-flight data transfers than conventional processors. More simultaneous transfers with more memory bandwidth allows for more hiding of memory latency. The programmer-controlled asynchronous DMA transfers the SPEs are capable of allows for the programmer to handle this latency-hiding.

3.6.3 Power Wall

The power wall is also overcome by the heterogeneity of the processing elements. By allowing processor elements to focus on specific tasks, the overhead of including optimizations for other tasks is eliminated, increasing the power efficiency of the chip.

3.7 Current and Future Variations

A look at what variations of the Cell/B.E. architecture currently exist, as well as potential future iterations is prudent. Given the investment in the architecture by the parties involved, it would not be unreasonable to imagine that the architecture will continue to be improved and enhanced into the

future. A Cell/B.E. Technology Roadmap featured in a past Roadrunner related presentation[Koc07] notably makes reference to a concept Cell/B.E. variant with 32 SPEs, though the architectures on the roadmap should be taken as "estimations only and subject to change without notice" according to the slides.

3.7.1 Cell/B.E.

The original Cell/B.E. architecture has already been outlined in this chapter. The Cell/B.E. contained in the Playstation 3 is notable for having only 6 accessible SPEs. This is because one SPE is disabled to increase wafer yields (even if that SPE is not defective), and another SPE is set aside for OS functionality[BLK+07]. The Cell/B.E. was also reduced to 65nm SOI (Silicon on Insulator) in 2008, from the original 90nm, with a further decrease to 45nm expected in 2009.

3.7.2 PowerXCell 8i

The PowerXCell 8i variant of the Cell/B.E. architecture addresses the deficiency in double precision floating point performance of the first Cell/B.E.. The PowerXCell 8i has an increased peak performance of 102 GFLOPS for double precession[SKGF08], over 7x the performance of the Cell/B.E.. The PowerXCell 8i also supports industry-standard DDR2 SDRAM memory, which allows for cost efficiency and memory capacities of up to 32GB in dual processor configurations[CHKW08]. The consequences of increasing RAM capacities for Cell/B.E. based systems are illustrated in Section 3.8.2.

3.7.3 SpursEngine

While not considered a direct decedent of the Cell/B.E.[Shi07], it is worth noting that Toshiba has developed the SpursEngine as a media oriented coprocessor targeted towards consumer electronics. The processor uses 4 SPEs and operates them at a lower clock frequency 1.5 GHz, producing 48 GFLOPS performance. As the processor does not contain a PPE, which is vital to the Coconut Mulitcore Framework design, the SpursEngine will not be a target of the CMF in the future.

3.7.4 Cell/B.E. 32 SPE Concept Design

This potential future architecture is not confirmed, but it has appeared in IBM Cell/B.E. Technology Roadmaps since 2007 as a "concept" design expected in 2010-2011[Koc07]. This concept architecture would have 32 SPEs, 2 PPEs and have 1 TFLOPS of single precision floating point performance. No information is given as to the EIB memory bandwidth.

This information is useful, as it confirms that IBM is at least thinking about Cell/B.E. architectures with 4x the cores of current chips. This suggests that developers and programming models should plan for an exponentially increasing number of cores into the future. This dramatic increasing of cores may exasperate the problem of limited bandwidth from main memory. The situation of increased computational power not being matched by increases in memory bandwidth has already played out in single core systems[WM95; BLK+07], and so we are very mindful that it may continue to do so in network-on-a-chip architectures.

3.7.5 Dual Processor Systems

Though not a separate variant of the Cell/B.E. per se, dual processor Cell/B.E. systems can be thought of as a variant of the Cell/B.E. processor on the level of software. By connecting two Cell/B.E. systems as discussed in Section 3.4, developers effectively have 16 accessible SPEs[BWSF06]. This however comes at the cost of a new communication constraint to deal with in the sense that the BIC connecting the two chips has a limited bandwidth, making chip-to-chip communication a potential performance bottleneck. As dual processor systems are standard in Cell/B.E. Blade Servers (see Section 3.8.2), optimal scheduling in dual processor systems is an important problem.

3.8 Applications

The applications that the Cell/B.E. is intended for and being used in should in part guide the development of any programming model. With applications in video game consoles, blade servers, supercomputing, cluster computing and distributed computing it is clear that Cell/B.E. has caught on in the areas it was intended for - those involving parallel high performance floating point calculations. We find that on the more tightly coupled level of video game consoles and supercomputers, linear algebra operations, game engine related algorithms and molecular dynamics simulation algorithms are of particular

interest. As we go into cluster and grid computing environments more suitable for loosely coupled and embarrassingly parallel problem spaces, we find applications in astrophysics and bioinformatics to be of particular interest.

3.8.1 Video Game Consoles

As discussed, the Cell/B.E. is currently being used in the Playstation 3 video game console[BLK+07]. Playstation 3 sales were at over 20 million units by December 31st, 2008[Son09]. The Playstation 3 has relatively limited RAM at 256 Megabytes, and this limitation is a constraint on potential scheduling algorithms (discussed in Section 3.8.2).

Game engine related algorithms and linear algebra operations for graphics related processing are thus of particular value in considering Cell/B.E. scheduling on the Playstation 3.

3.8.2 Cell/B.E. Blade Servers

Another application for the Cell/B.E. has been in IBM BladeCenter blade systems targeted at clients with high performance computing needs. The BladeCenter systems use two Cell/B.E. processors in a dual processor configuration. The first BladeCenter systems, QS20 and QS21 used the original Cell/B.E. architecture with relatively poorer double precision floating point performance and maximum RAM capacities[IBM06; IBM08a]. The QS21 BladeCenter system had a maximum capacity of 2 Gigabytes of RAM. The QS22 BladeCenter however uses the PowerXCell 8i variant discussed in Section 3.7.5, also in a dual processor configuration, and thus has the expected increase in double precision floating point performance as well as a vastly expanded RAM capacity - up to 32 Gigabytes[IBM08b].

Effect of Increased Memory Capacity

The PowerXCell 8i variant and related QS22 BladeCenter greatly increase the problem sizes that can be efficiently calculated with the Cell/B.E. architecture. With the Playstation 3 and earlier BladeCenters, if the problem size grew beyond the capacity of the relatively limited RAM, expensive operations to hard disk or other storage would be required. However with the new RAM capacity increases vastly greater problem spaces can be efficiently computed without expensive disk access.

For instance looking at the simple but illustrative problem of dense square matrix multiplication, the RAM capacities of the systems give limita-

tions to the size of the matrices A, B and C that could be kept in main memory at once without some sort of swapping between disk and RAM taking place. Assuming all of the RAM was available for data, which would not be the case in practice as space would be needed for code, a simple division of the floating point memory capacity by 3 followed by a square root of the result gives the largest matrix size that could be computed without disk access being necessary. The result of this calculation is shown in Table 3.2 for several systems and memory capacities.

Table 3.2: RAM Capacity Effect on Square Matrix Mult. Problem Size

	RAM	**32-bit Float Capacity**	**Max N $\approx$**
Playstation 3	256 Megabytes	6.710×10^7	4730
QS20	1 Gigabyte	2.684×10^8	9459
QS21	2 Gigabytes	5.368×10^8	13377
QS22	32 Gigabytes	4.294×10^9	53509

The problem size capabilities of a computing system are of interest to developers of a programming model such as the CMF, as an indication of just what problems may be computed on the computing system. Observations about main memory as a limiting factor to computing problems of a given size on the Playstation 3 have been made by others[BLK+07].

3.8.3 Supercomputing

The Cell/B.E. is used in the IBM Roadrunner supercomputer at the Los Alamos National Laboratory, which was the first supercomputer to sustain 1 PFLOPS of performance[Top08]. Compute nodes in the system consist of four PowerXCell 8i variant Cell/B.E. processors and two AMD Opteron dual-core microprocessors. The Cell/B.E. processors act as accelerators, which contribute over 96% of the theoretical 1.3 peak PFLOPS[SKGF08].

The Roadrunner has been built for the U.S. Department of Energy for the purpose of simulating the aging of nuclear materials to evaluate the reliability and safety of the American nuclear arsenal. As a result, molecular dynamics simulation algorithms targeted to the Cell/B.E. are of particular value.

3.8.4 Cluster Computing

The Cell/B.E. is being used in cluster computing, for example Astrophysicist Gaurav Khanna created a cluster of 8 Playstation 3 consoles for supermassive black hole simulation[Gar07]. Terrasoft Solutions also sells clusters of 6 or 32 Playstation 3s, with Yellow Dog Linux pre-installed[BLK+07]. Terrasoft Solutions has also built a facility for bioinformatics research related to gene-finding and sequence alignment, containing roughly 2500 Playstation 3 consoles[SKST08]

Cluster computing is as of yet a less prominent usage of the Cell/B.E. architecture relative to the Playstation 3, Roadrunner and Folding@Home (talked about in Section 3.8.5). This may have to do with problems of using the Playstation 3 as a cluster node[BLK+07]. These problems being the main memory access rate of the Cell/B.E., network interconnect speed, main memory size, programming difficulties and the relative lack of double precision floating point performance on the Playstation 3.

3.8.5 Grid Computing

Grid computing with the Cell/B.E. is perhaps most famously exemplified by the Folding@Home distributed computing project for protein folding[SKST08]. This along with the Terrasoft Solutions work in bioinformatics discussed in Section 3.8.4 suggests that optimal bioinformatics related algorithms, particularly in embarrassingly parallel problem sets, are potential problems a programming model may be designed around as possible use cases.

Chapter 4

Cell/B.E. Program Models, Frameworks and Solutions

The purpose of this chapter is to review the existing program models, frameworks and solutions that target the Cell/B.E.. Though likely not a completely exhaustive list of the available solutions, a review is useful to put the Coconut Multicore Framework discussed in Chapter 5 into context of similar tools, as we do in section 5.4.

4.1 Accelerated Library Framework (ALF)

The Accelerated Library Framework (ALF), developed by IBM and Los Alamos National Laboratory researchers, consists of an API which is built to assist in the development of parallel applications for Cell/B.E. systems and systems like Cell/B.E.[CHKW08]. ALF is actually included as part of the Cell/B.E. SDK[IBM07b]. ALF works by dividing a program into portions taking place on a host processor (the PPE in the case of the Cell/B.E.) and those taking place on an accelerator processor (the SPEs in the case of the Cell/B.E.).

A host runtime library and accelerator runtime library are provided to the developer. Work is divided into a control process which executes on the host, and computational kernels which execute on the accelerators. The ALF API is used to write the computational kernels, and these computational kernels are responsible for performing the actual computing work (e.g. floating point calculations). By creating tasks and defining execution orders and task dependencies, an ALF runtime can then manage these tasks, do scheduling optimization, take care of data movement by using double buffering, and can handle errors. This split between computational kernels and data movement

allows developers to focus at the low-level ILP level of computational kernels or the level of tasks at the multicore MPMD level[IBM07a].

The performance of ALF appears to be more suited to some problems rather than others, as poor performance was reported for a Jacobi algorithm to solve a 2D Poisson equation, inefficient communication was to blame, though the same author notes that it is well suited for other problems[Hil07]. For example, a Lattice Quantum Chromodynamics (LQCD) library was ported to the Cell/B.E. with success using ALF as one programming model amongst others[Spr07].

4.2 Cell/B.E. Software Development Kit

The Cell/B.E. SDK is the standard programming model for development on the Cell/B.E. provided by IBM[IBM07b], which includes different compilers, libraries, code examples and other tools. The Cell/B.E. SDK is the most low-level programming model, essentially made up of an API for imperative languages and a standard supportive tool chain.

A GNU tool chain is provided which contains compilers targeting the PPU and SPU for the C, C++ and Fortran languages. The IBM XL C/C++ compiler is included, it is a high-performance cross-compiler that has been optimized for the Cell/B.E.. Libraries are included to provide a standard low-level API for developers to use hardware specific features, such as the SPEs. Notable libraries include the SPE Runtime Management Library for access to SPEs, the SIMD Math library for short vector math functions, and the Mathematical Acceleration Subsystem (MASS) libraries which contain mathematical intrinsic functions optimized for the Cell/B.E. Tools such as the IBM Full-System Simulator and Performance Debugging Tool (PDT) are included.

IBM Eclipse serves as the Integrated Development Environment (IDE) for the SDK; it integrates tools for ease of use and greater productivity, such as the GNU tool chain, debugging tools and Full-System Simulator. Eclipse also provides the expected syntax highlighting and GUI overview of program constructs that are found in source code.

4.3 Cell Superscalar (CellSs)

The Cell Superscalar (CellSs) programming model was created by Barcelona Supercomputing Center researchers[BPBL06]. The model aims to allow developers to modify existing C code with simple CellSs annotations, and then

a source-to-source compiler compiles that code to target the Cell/B.E. with separate PPE and SPE C files that are then themselves compiled to binary code. The C code with CellSs annotations is essentially sequential code, with parallelism handled automatically - ease of use in taking advantage of the Cell/B.E. is the key objective.

Functions in C code are annotated such that they are specified as tasks to be executed on SPEs. A task dependency graph is then constructed by a runtime library, with task scheduling and data dependencies handled by the runtime system itself. Data locality is exploited by the scheduling algorithm, to try to reduce the amount of transfers required between SPEs and main memory, and amongst SPEs. A subgraph of the task dependency graph is considered at each step, with the ready nodes (those ready to execute) and a subgraph of each of them analyzed to take advantage of data locality.

A performance example involving matrix multiplication was shown to scale very well with the number of SPEs, but a Cholesky factorization example did not scale as well due to highly connected dependency graphs and the usage of six different tasks at different levels of granularity. Other researchers looking at using CellSs for QR factorization have said that due to the handling of task scheduling on the PPE, it was not competitive performance-wise with their own solutions[KD09].

It seems that the key advantage of CellSs at present is in its ease of use relative to hand optimizing for performance. Though not initially targeting other architectures, the creators believe the model generic enough to target other multicore architectures, and this is very likely true and another advantage of this approach.

4.4 CorePy

CorePy is a Python package that allows one to create and execute SPE programs for the Cell/B.E. from Python (as well as PowerPC and VMX programs). It is an open source project, developed by Chris Mueller, Andrew Friedley and Ben Martin, and is released under the BSD license[MML07].

The motivation to provide a Python package that allows for Cell/B.E. development is that a scripting language like Python increases developer productivity. Productivity is increased by the large standard libraries of Python, by the more concise syntax relative to lower-level languages, and by being able to rapidly edit and execute code. In comparison to a C code tool chain, with a much less expansive standard library and lower-level language constructs, this argument seems particularly valid.

The package allows one to run existing SPE programs from Python and to create new SPE programs. Synthetic programming, which involves generating instruction sequences from a higher level language, is one approach behind CorePy, with Python acting as a meta-language. The authors also show that it is possible to implement a lightweight interactive SPU debugger completely in Python; this may allow the productivity increases of a scripting language to be beneficial in the often tedious debugging process.

4.5 Mercury MultiCore Framework

The MultiCore Framework (MCF) created by Mercury Computer Systems researchers is an API to assist in programming heterogeneous multicore software, targeting the Cell/B.E.[BCG+06]. A manager program executes on the PPE and distributes work to worker threads executing on SPEs. The important abstractions in the Mercury MCF are distribution objects and channels. Distribution objects define the dimensions of the data for the manager program in main memory and the data for workers in local stores. Channels connect worker memory to manager memory, and require a distribution object to be created. Through these abstractions, the Mercury MCF is able to organize multicore parallelism without having a developer have to deal with hardware-specific details.

4.6 MPI Microtask

Message Passing Interface (MPI) Microtask is a programming model proposed by IBM researchers at the Tokyo Research Laboratory[OIS+06]. In this programming model, the developer partitions the application into a series of microtasks, with communication handled by the popular MPI explicit communication model[HB06]. These microtasks are expected to execute on the SPEs and perform computationally intensive work, and a special supportive microtask executes on the PPE to handle aspects such as control-intensive functions and I/O processing. A preprocessor then divides the microtasks into basic tasks (blocks of computation with no communication except for the beginning and end). This allows a streaming model to be formed, and the preprocessor can optimize scheduling by clustering basic tasks with strong dependencies together.

The benefits of the MPI Microtask model include that it frees a developer from having to worry about managing the local store. The MPI communication model is well known; this may shorten the learning curve for a

new developer. The model also hides hardware details from the programmer, which could be beneficial for portability. Finally, the explicit exposed communication allows scheduling algorithms to analyze dependency information for optimization purposes.

The researchers were able to use an initial prototype of the model to compute LU decomposition, 1D FFT and matrix multiplication problems. The results are described as promising, but that further work on the clustering algorithm would be desirable. Though this work is well referenced, at this time it does not appear to have been taken beyond this initial prototype implementation.

4.7 Open Multi-Processing (OpenMP)

Open Multi-Processing (OpenMP) is an industry standard API for supporting shared memory multiprocessing on multiple platforms[DM98]. Many different architectures from workstations to supercomputers are supported, along with Windows or Unix operating systems. Languages the API supports include C, C++ and Fortran. OpenMP is well supported by industry, and is notable for its high portability.

OpenMP is at its core a standardized API for expressing shared-memory parallelism, much like MPI is a (de facto) standard API for expressing message passing parallelism. OpenMP is made up of the following parts: control structure, data environment, synchronization and a runtime library. The control structures govern the flow of control of a program; the set available with OpenMP was designed to be minimalist. Data environment refers to the scope of variables, which can be made private or shared, amongst other types. Synchronization refers to the synchronization methods made available, both implicit and explicit. The runtime library is made up of functions to assist with parallelism, such as functions for setting the mode that a program should run in, as well as standard environment variables which assist in applications which need a portable runtime environment.

Even though OpenMP supports shared memory multiprocessing, and the Cell/B.E. is a disturbed memory architecture, developers have attempted to support OpenMP for Cell/B.E. for the obvious reasons of its wide use and support. Support for OpenMP on Cell/B.E. means existing code can be re-used, and developers can use a programming model they may already know very well[CCZ04].

4.7.1 Cellgen

Cellgen is a partial implementation of OpenMP for the Cell/B.E., created by Virginia Tech researchers [SYR+08]. Cellgen is available as open source under the GPL[Sch09].

Parallel sections of code are identified by developers, who then must annotate data as either private or shared. One difference between OpenMP and Cellgen is that in Cellgen shared variables are classified as either in, out or inout variables. These classifications denote whether the data is either input data, output data or input and output data with respect to the local stores. Cellgen then uses this information to automate data transfers to ensure proper data locality. Cellgen also notably unrolls loops in an effort to hide data transfer latency.

Cellgen parallelizations were compared against hand-coded parallelizations of the same problem. Cellgen parallelization was competitive with hand-coded parallelization in the case of a memory bandwidth benchmark developed by the researchers, and in the case of a Bayesian phylogenetic interference method which uses a Markov chain Monte Carlo sampling method to construct phylogenetic trees from DNA and AA sequences.

4.7.2 IBM T.J. Watson

An implementation of OpenMP for the Cell/B.E. was done by researchers at IBM T.J. Watson Research[OOS+08]. The implementation works by having a compiler transform OpenMP constructs in source code into intermediate code which calls functions in an associated runtime library. The runtime library provides the synchronization and thread management functions necessary to support OpenMP for the Cell/B.E.. This compiler is itself built on top of the IBM XL compiler; existing code in the XL compiler for supporting OpenMP on AIX multiprocessor machines with Power processors was leveraged in their implementation. Performance testing of the implementation was done using several standard benchmarks. The performance testing showed satisfactory speed-up for many applications, but some limitations in the current implementation (such as static buffer optimization) caused poor performance in some cases.

4.8 RapidMind

RapidMind is a Waterloo, Ontario based company, founded by Michael McCool and Stefanus Du Toit, that provides a commercial multicore solution that

targets the Cell/B.E. in addition to GPUs: The RapidMind Development Platform[McC06]. The company has its origins in the Sh project[MT04] at the University of Waterloo, which targeted programmable GPUs. The company was notably acquired by Intel in August of 2009[Sha].

The platform interface is exposed to the user via C++ header files and having the developer link to a library. A user specifies parallel computations using three C++ types: Values, Arrays and Programs. Values and Arrays are data containers, Programs execute instructions, instructions that may include control flow. Though primarily a data-parallel SIMD model of parallelism, the inclusion of control flow in Program types allows for task-parallel SPMD parallelism as well. RapidMind is made such that developers do not have to throw away existing sequential single core code, but instead rewrite C++ applications to RapidMind types and automatically gain multicore performance on a variety of platforms.

RapidMind becomes embedded in the compiled application, and manages program execution at runtime, automating and optimizing tasks such as load balancing and synchronization. The platform is portable to different target architectures, such as Cell/B.E., NVDIA GPUs or x86 processors. The platform also provides runtime performance monitoring diagnostics. The platform takes care of parallel safety - race conditions and deadlocks cannot occur.

The advantages of RapidMind are very clear from a developer's point of view. Rather than having to completely rewrite existing code for every new multicore platform, or rewriting in a new and different parallel language, it allows for very quick performance gains on multicore architectures with relatively simple modifications to existing code. This is obviously a major gain, as the problems of synchronization safety, optimization for different architectures and ease of program expression are all significantly alleviated if not eliminated. RapidMind is reported as having performance results that were twice as good for a quaternion Julia set renderer compared to the best known existing version done with the Cell/B.E. SDK code[Mon08]. The main advantage of RapidMind however seems to be purely economical, in that it requires much less development effort to achieve roughly the same or better performance than one can with hand tuned code.

A disadvantage of RapidMind is that there is potential for vendor lock-in, in that once a developer has converted their programs to work with RapidMind, they could become very dependent on RapidMind for future performance gains, which is an obvious concern. While a developer could in theory switch from RapidMind to another platform in a reasonable amount of time, this would still impose an obvious cost.

4.9 Sequoia

Sequoia is a programming language developed by Stanford researchers which targets the Cell/B.E., amongst other architectures[FHK+06]. The main idea behind Sequoia is to allow the developer to explicitly control the movement of data throughout the different levels of a machine memory hierarchy. Architecture memory hierarchies are abstracted as trees of memory, which also expose how data is transferred in the hierarchy.

Tasks are the main construct of Sequoia, they operate within a specific memory space on data located only in the memory space, and perform side-effect free computation with call-by-value parameters. Passing arguments to tasks is how a developer expresses data movement in the system - it is the only means to do so. Tasks can only communicate with other tasks by calling child subtasks or returning a result to a parent task.

To perform matrix multiplication, a task (referred to as a leaf task) may take in subblocks of a matrix and perform multiplication. Meanwhile another parent task (referred to as an inner task) will call the leaf task repeatedly and through its call-by-value parameters pass the appropriate matrix multiplication blocks. Task instances are then mapped to different levels of the memory hierarchy. When mapping this matrix multiplication program to the Cell/B.E., the leaf tasks would execute on the SPE and the parent task would execute on the PPE. The Sequoia compiler is a source-to-source compiler that given a Sequoia program and mapping specification for a target machine would output C code. In the case of the Cell/B.E., sets of C files for the PPE and SPE would be output.

Sequoia has been performance tested against existing implementations of algorithms such as SGEMM and FFT3D - the resulting performance was competitive. As of August 2009, Sequoia had been implemented (compiler and a runtime system) for the Cell/B.E. and for distributed memory clusters[Lab09]. An alpha version is expected to be made public soon.

4.10 SysCellC

SysCellC is a Cell/B.E. software solution by INPG researchers, it is a proposed design flow to automate the transformation of programs expressed in SystemC to implementations targeting the Cell/B.E.. SystemC is an open source system design language[MRR03], and it effectively becomes the high level language in which programs under this model are expressed in by the developer. SystemC allows one to express machine architectures as hierarchies of computa-

tion modules and communication channels. Multiple programming models are supported by SystemC, including streaming models. Using a streaming model, a kernel would correspond to a computation module and a stream would correspond to a communication channel. The SystemC programs created by a developer are eventually converted to C code files for the PPE and SPE, after some intermediate scheduling using profiling results of the SystemC code. An interesting point is that these C code files are actually architecture independent, as MPI primitives used for communication and synchronization hide all of the architecture dependent code.

4.11 Possible Future Models, Frameworks and Solutions

Some other models, frameworks and solutions are worth mentioning. These include those programming models which are not yet fully implemented or not yet targeting the Cell/B.E., but which may in the future.

4.11.1 Manticore

Manticore is a research project being conducted by researchers at the University of Chicago and Rochester Institute of Technology, the aim of which is to create a functional language for parallel programming[FRR+07]. What sets Manticore apart from other parallel languages is that it attempts to support parallelism at multiple levels. Implicit mechanisms are used to exploit fine-gain parallelism, and explicit mechanisms are used to exploit coarse-grain parallelism. While Manticore is not targeting Cell/B.E. yet to our knowledge, it would very likely be suitable for Cell/B.E. and similar architectures.

4.11.2 Open Computing Language (OpenCL)

Open Computing Language (OpenCL) is a framework and open standard that allows for developers to execute their code across heterogeneous parallel-computing platforms[Mun08]. Kernels are the basic unit of execution and written in a language resembling C99 with extensions. An API is also provided to manage the parallel execution. An OpenCL implementation for Cell/B.E. is reportedly described to be "in the works" but with no specified date of availability; others have taken to implementing a minimal subset of OpenCL[McM09].

4.11.3 Message Passing Interface (MPI)

MPI is an obvious candidate as a programming model for the Cell/B.E., as along with OpenMP it is a de facto industry standard for expressing parallelism. A problem with porting an MPI application to the Cell/B.E. is the limited local store size of each SPE[KJS+07]. If the application size exceeded the size of the local store, some sort of mechanism to bring code into the local store as it is needed would be required. A partial implementation of core features of MPI 1 has been done by a group of IBM, Florida State University and Sri Sathya Sai University researchers[KSK+07]. MPI was also used as part of the MPI Microtask[OIS+06] and SysCellC[MRR03] solutions.

Chapter 5

Coconut Multicore Framework

In this chapter the present state of the Coconut Multicore Framework will be outlined, including the design objectives, an overview of the framework, and descriptions of individual components. A comparison is also made with other available frameworks and solutions discussed in Chapter 4.

5.1 Coconut Project History

The Coconut project is an on-going compiler technology research project at McMaster University[KAC06; ACK+04; AK07c; AK07b; AK08]. Its inspiration is derived from the current state of high-performance signal processing software development for medical imaging applications. At present programmers will often have to make low-level optimizations to account for higher-level model characteristics which themselves may have a high rate of change. This is an undesirable situation with respect to software development economics, as well as code optimization and safety.

The primary goal of Coconut is to remedy this situation, and others like it, by providing a coherent path from mathematical specification to verified and highly optimized machine code[KAC06]. This goal is achieved by Coconut at present through the usage of Domain Specific Languages (DSLs) tuned to different levels of abstraction, which are then compiled into optimal and parallel code to various target instruction sets and architectures[ACK+04]. The parallelism leveraged thus far by Coconut has been ILP[AK07b]. The DSLs themselves are embedded within Haskell code[AK08; PCK07], the language in which the vast majority of Coconut technologies have been implemented. The instruction set initially targeted was PowerPC+Altivec due to its popularity in signal processing applications. In recent years with the shift of the project

towards the Cell/B.E., the SPU Instruction Set has been targeted[AK07c].

This technique of compilation from higher-level DSL abstractions to lower-level target languages has been successful in real-world software - the Cell/B.E. SDK 3.0 SPU-MASS library currently includes code using Coconut ILP optimization techniques. For comparison, the library is 4x faster than the alternative SIMDMath library created in C.

The high abstraction of Haskell in which DSLs are implemented allows for quick prototyping of new ideas within Coconut that would not be possible otherwise. Though the DSLs themselves require some domain-expertise, and to work with them requires Haskell programming skills, this has not stopped undergraduate mathematics students from being able to make meaningful contributions to Coconut.

With the ILP layer of Coconut having reached a mature enough state to produce industrial-quality code, the focus of the project has shifted towards the multicore layer of parallelism, initially targeting the Cell/B.E.[AK08; AK07b]. The framework developed specifically for the multicore layer of parallelism to be discussed in this chapter is tentatively being referred to as the Coconut Multicore Framework (CMF).

5.2 Design Overview

In this section the objectives of the CMF will be reviewed in Section 5.2.1, followed by a high-level description of the design in Section 5.2.2 and a justification for the design in Section 5.2.3. More detailed and technical descriptions of the individual components of the CMF are found in Section 5.3.

5.2.1 Objectives

The design objectives for the CMF are purposely similar to the ILP layer of Coconut. The two main objectives of optimal and safe code remain the most critical, as well as the goal of providing a clear path from mathematical specification to code. Part of providing this clear path involves being able to generate code based on mathematical specifications to different target architectures, even though the Cell/B.E. is the only target architecture for now. As Coconut has a wealth of optimization capability at the ILP layer, an auxiliary goal is to create an abstraction that would allow us to leverage these capabilities at the SPMD multicore layer. The target applications remain signal processing software and similar problem domains.

5.2.2 Description

The design chosen to fulfill these goals is a virtual machine model. This means that each processing element, in the case of the Cell/B.E. each SPE, runs a program which executes virtual instructions. These virtual instructions are referred to as **Atomic Virtual Operations** (AVOps). The PPE has the job of starting up each of these virtual machine threads on the SPEs. The program made up of the virtual machines executing on each SPE and the PPE thread that manages their startup is known collectively as the **Runtime System**.

The Runtime System loads computational kernels onto each SPE's local store at the start-up of each virtual machine thread, as well as the initial batch of AVOps. As AVOps are being executed sequentially and in-order by the virtual machine, they are double buffered in from main memory to the SPE's local store. The AVOps include operations for loading data from main memory, sending data to main memory, sending and receiving data or signals from other SPEs, and importantly, executing the pre-loaded computational kernels against data located on the SPE's local store. The purpose of allowing signals between SPEs, aside from regular data transfers, is to allow one SPE to signal another using a lightweight mechanism that it is ready to receive a data transfer.

So for example, if using the CMF one wanted to perform a matrix multiplication against two blocks of data small enough to fit onto a single SPE's local store, the computation would proceed as follows. First one would need a computational kernel capable of performing the matrix multiplication given two blocks of data, the kernel would be expected to produce a third output result block of data. Then a simple stream of AVOps would be needed, the stream would need to include operations to load the blocks of data from main memory, wait operations to ensure the completion of the input block loads, an operation to execute the pre-loaded matrix multiplication kernel against the input blocks, and an operation to send the result data back to main memory. Giving the runtime system the kernel, the stream of AVOps, as well as some basic configuration information including information such as input data sources, would then allow the computation to occur. Computations involving multiple SPEs and communication between them would obviously require streams of AVOps and relevant data and kernels for these SPEs as well.

The computational kernels themselves that carry out operations locally on each SPE given input data blocks, such as linear algebra computations, are generated at compile-time by whichever means the developer desires. They can be created using C, or they can be created using the ILP layer of Coconut,

it is of no consequence to the Runtime System's operation.

The generation of AVOp streams for each of the SPEs can similarly be done however the developer desires, the Runtime System will operate correctly as long as the streams of AVOps match the proper specification. A longer term goal of the CMF is to create DSLs for specific problem domains from which streams of AVOps may be generated, given a problem specification. This is only one envisioned method of generating AVOp streams. Another possible method considered for future implementation is a framework for scheduling loops represented using imperative sequential language constructs. Yet another possibility is to implement directed acyclic graph scheduling algorithms[KA99], and to provide functions for generating the graphs that are to be scheduled for different problem domains. One of the most critical envisioned ways to generate AVOps is to apply existing ILP optimization techniques within Coconut at the multicore level of parallelism, as the CMF can be considered analogous to ILP parallelism. At the time of writing, AVOps are being generated by Haskell programs written for specific problem domains.

Though a key design feature of the CMF is the separation of ILP and multicore layers of parallelism, a longer term goal of the Coconut project is to increasingly merge code generation for the two layers of parallelism. Such that given a single high level mathematical problem specification, a safe and optimized executable can be created with both ILP and multicore layers generated separately by the respective layers of Coconut.

Another key design feature of the CMF is that at present the AVOp streams are generated in their entirety at compile-time. There are also no AVOps for branching (and as a consequence, looping). The execution order itself, due to the asynchronous execution of SPEs (and thus asynchronous execution of each virtual machine), is necessarily a partial execution order. This also means that the CMF in its current form is targeting specifically those problems which can be scheduled in their entirety at compile-time. Though it should be noted that it is still possible to have dynamic decision making embedded within a CMF program, through branching within the computational kernels.

One side-effect of generating the entire AVOp stream at compile-time is that we can then perform linear-time static analysis of the program. A Verification Tool[AK08] has been built that can check for possible deadlocks or race conditions to ensure that neither exists, and this is documented in more detail in Section 5.3.5. A performance simulation tool has also been built that can simulate the execution of AVOp streams on the Cell/B.E., to provide compile-time feedback that can be used to evaluate the suitability of different schedules quickly. This tool is documented extensively in Chapter 6,

and discussed briefly in Section 5.3 along with the other system components, as it is the most substantial individual contribution to the project by the author.

The way in which the various components of the CMF fit together to ultimately produce output data is shown in Figure 5.1. An example of what the system looks like at runtime, performing an operation like matrix multiplication, from the perspective of the PPE, an SPE and main memory, is shown in Figure 5.2

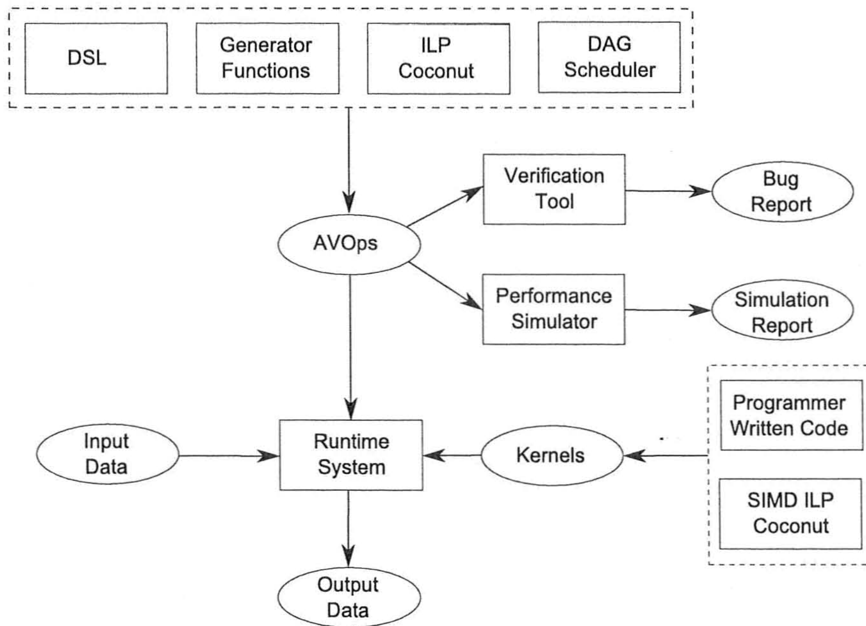

Figure 5.1: Coconut Multicore Framework Overview

5.2.3 Analysis

The CMF virtual machine abstraction design composed primarily of the Runtime System and AVOp streams was designed as such for several reasons which are worth outlining.

ILP Correspondence

One critical gain experienced with the CMF abstraction is that due to its similarity to ILP, it is possible that existing Coconut ILP optimization techniques could also be used at the multicore level. In fact, one can see in Table 5.1 that the *entire system itself is an analogy to ILP*. Techniques developed for the ILP

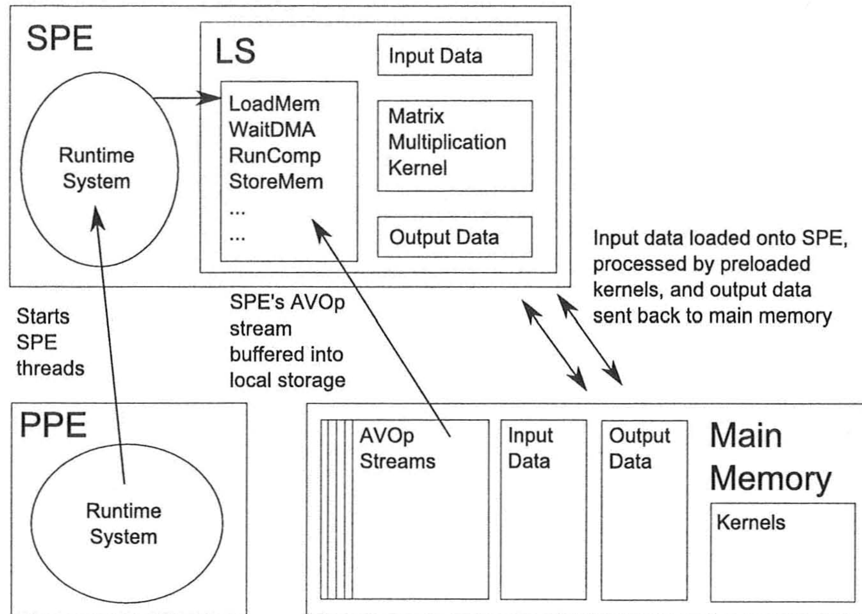

Figure 5.2: Coconut Multicore Framework Runtime View

layer, in particular those which are designed for software pipelining, such as the MultiLoop[AK07c], can then be applied to blocked-data algorithms at the multicore level with little to no modification necessary. The key difference is that at the ILP level, hardware hides execution order of the instructions but maintains order independence, whereas at the multicore level the soundness of the parallelism and synchronization will be left up to software. This is not foreseen as a major problem however, due to the static analysis abilities of the CMF which allow us to verify the soundness of synchronization.

Table 5.1: Parallelism Correspondence

Instruction Level Parallelism	Multicore Level Parallelism
CPU	Chip
Execution Unit	Core
Register	Buffer/Signal
Load/Store Instruction	DMA
Arithmetic Instruction	Computational Kernel

Exposed Parallelism

The AVOps *expose* important aspects of parallelism by abstracting away all but the necessary information. This exposure is illustrated in Figure 5.3, where we first notice that a WaitSignal AVOp will not complete execution at least until the associated SendSignal has been executed (not including the transmission time of the signal itself). We then know that in order to schedule optimally, we wish to have the SendSignal execute as far temporally ahead of when the WaitSignal is expected to execute as possible, as otherwise the WaitSignal will stall, impeding execution. The same thinking applies to the SendData and WaitData pair in Figure 5.3. This pattern of one SPE signaling another to indicate that it is ready to receive a data transfer is an important pattern in our virtual machine abstraction. It captures how SPEs work asynchronously in between the communication occurring to synchronize transfers, rather than stalling further computation completely to co-ordinate communication.

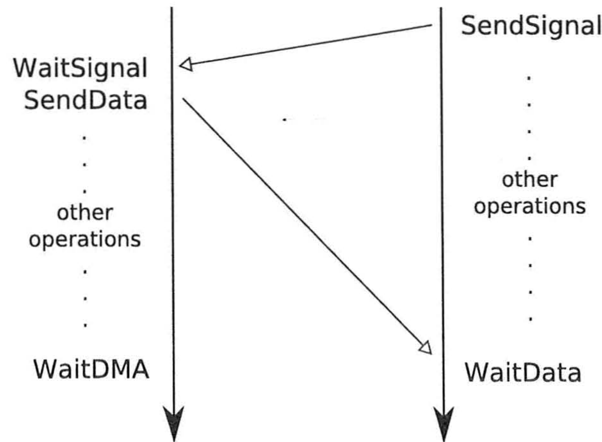

Figure 5.3: Exposed Communication Latency

Portability

The Runtime System itself can be implemented on different architectures. At present the only implementation is for the Cell/B.E.. However for future iterations of the Cell/B.E., as well as similar network-on-a-chip multicore architectures, existing infrastructure for the specification and generation of AVOp streams can still be leveraged despite having to re-implement the Runtime System on a new target architecture. This is in some sense similar to how the ILP layer of Coconut, while initially targeting PowerPC instruction

sets, was able to quickly be targeted towards the SPU Instruction Set of the Cell/B.E.'s SPEs. The virtual machine abstraction gives us the ability to target new architectures quicker than otherwise possible by re-using relevant problem specification and scheduling optimization techniques.

Strong Parallelism-Level Distinction

The separation of layers of parallelism, that is, ILP parallelism within the pure function computational kernels (i.e. no communication or synchronization), and multicore level parallelism embedded within the AVOp streams, allows for the developer to focus on the problems independently. This is beneficial as it allows developers to specialize at creating optimization techniques for a specific layer of parallelism. Indeed, in the work that has been done thus far on Coconut towards MRI and linear algebra problems, we have been able to productively have one developer work exclusively on the multicore layer of parallelism while another developer works simultaneously on generating the ILP optimized computational kernels.

Static Compile-Time Analysis (Safety and Simulation)

The ability to perform static analysis allows us to automatically check for parallel bugs, and to do performance simulation, at compile-time. As a result of the Verification Tool, the Coconut team has thus far not encountered a parallel bug at runtime. When constructing the initial Runtime System we encountered parallel bugs, and we encounter parallel bugs at compile-time when the Verification Tool alerts us of them. But we have not encountered, as was discussed in Section 2.8.2, those time consuming and expensive race condition bugs that occur based on small variations in timing. This obviously creates a large increase in productivity, as no parallel debugging is required. This also has the effect of freeing the developer to focus on optimization, rather than creating more conservative algorithms with respect to safety, at the expensive of runtime efficiency. For safety-critical applications that could benefit from network-on-a-chip parallelism, such as MRI and other signal processing problem domains, the CMF provides a competitive advantage over other systems that do not offer such high safety.

The Performance Simulator allows one to quickly compare the simulated runtime of different algorithms, saving the developer from having to do full system tests, and thus allowing for increased productivity. The ability to identify particular bottlenecks flagged during the simulation is also beneficial when trying to optimize scheduling.

A trend of increasing usage of static compile-time analysis tools in industry has been noticed[WMZ+08], with popular tools such as Coverity, FindBugs and JLint being used to find defect patterns in code that are likely errors. The CMF is therefore able to offer static compile-time analysis features are not only useful, but also increasingly used and expected of development environments.

5.3 Components

In this section, we overview the various CMF components in more detail. In Section 5.3.1 in particular, we go over all the different AVOps themselves in more detail.

5.3.1 Atomic Virtual Operations

The AVOps are executed asynchronously on each SPE, and are double buffered in to each SPE's local store by the Runtime System. Each AVOp and its arguments are 128-bits (16 bytes) in length. The first byte identifies the AVOp itself, with remaining bytes defining any AVOp arguments. Each AVOp is decoded by the SPE's Runtime System thread, and its prescribed functionality is then carried out, similar in concept to Java Bytecodes[VRCG+99]. The AVOp streams at present are produced as plain text line delimited output by Haskell AVOp generator functions for readability purposes, and the Runtime System can then convert them to the internal 128-bit format. The AVOps themselves are detailed in Table 5.2.

It should be noted that there exists an "Exit Program" AVOp that is simply appended to each AVOp stream. Its only function is to let the SPE Runtime System thread know that it can stop interpreting AVOps.

It is also expected that in future iterations of the Runtime System, AVOps and kernel functions will be loaded into the SPE explicitly by AVOps defined by the user (LoadMemory AVOps). This functionality has for now been handled by SPE Runtime System threads for the sake of simplicity, but as the CMF targets more difficult problems perhaps requiring multiple or many computational kernels, it is desirable to give the AVOps direct control over which computational kernels are present as well. Giving the AVOps the responsibility for double buffering in the AVOps is also desirable, as it will make all communication explicit in the AVOps.

AVOp	**LoadMemory**
Arguments:	*LocalStoreAddress DataLength GlobalAddress DMATag*
Description	The SPE issues a DMA get command, to transfer the data of length DataLength located in main memory at address GlobalAddress in the local store address LocalStoreAddress. The DMATag can be checked by a WaitDMA instruction to ensure transfer completion, which for instance makes it safe to use the transferred data in a computation.

AVOp:	**StoreMemory**
Arguments:	*LocalStoreAddress DataLength GlobalAddress DMATag*
Description:	The SPE issues a DMA put command, to transfer the data of length DataLength located on the local store of the SPE at address LocalStoreAddress to main memory at address GlobalAddress. The DMATag can be checked by a WaitDMA instruction to ensure transfer completion, which for instance makes it safe to use the local storage address for the result of a new computation (knowing that the result that was once there is now safely in main memory).

AVOp:	**WaitDMA**
Arguments:	*DMATag*
Description:	Executed on the SPE from which the transfer was initiated, this AVOp will stall execution until the DMA transfer with the tag DMATag has completed. This allows one to ensure transfer completion, and thus safety to use resources associated with the transfer, such as the data transferred itself or the memory involved in the data transfer.

AVOp:	**SendData**
Arguments:	*LocalStoreAddressL DataLength SPEID LocalStoreAddressE DataTag DMATag*
Description:	The SPE issues a DMA put command, to transfer the data of length DataLength located on the local store of the SPE at address LocalStoreAddressL to the local store of the SPE identified by SPEID at address LocalStoreAddressE. The tag DMATag is used by the SPE executing the SendData instruction to ensure that the transfer has completed. The DataTag is used by the remote SPE to check that the transfer has completed, using a WaitData AVOp.

AVOp:	**WaitData**

Arguments:	*LocalStoreAddress DataTag*
Description:	Executed on the target of a SendData AVOp executed on another SPE, the AVOp will stall execution until the DMA transfer with tag DataTag has completed. The DataTag is made sure to be transferred after the block of data has been transferred using fencing commands. Ensuring completion of the transfer allows for safe usage of the data.

AVOp:	**SendSignal**
Arguments:	*SPEID SignalID*
Description:	The SPE issues a command to modify the signal register of another SPE with ID SPEID, using its memory mapped address. The SignalID itself is a single bit set within a 32-bit register, which is then merged with the existing signals through an OR operation. If the SignalID bit is already set in the signal register, then a potential parallel bug may occur, as the SignalID should have been consumed by a WaitSignal AVOp before this new signal arrived.

AVOp:	**WaitSignal**
Arguments:	*SignalID*
Description:	Stalls AVOp execution and repeatedly reads the signal register on the SPE the AVOp is executed on until the signal bit SignalID is set by an associated SendSignal AVOp.

AVOp:	**RunComputation**
Arguments:	*ComputationID [LocalStoreAddressI] [LocalStoreAddressO] [Parameter]*
Description:	Executes a computational kernel, pre-loaded to the SPE's local store at the Runtime System start up, identified by ComputationID. The computation is run with input blocks identified by a list of local store addresses LocalStoreAddressI, and produces output identified by a list of local store addresses LocalStoreAddressO. The Parameter argument is a list of any necessary parameters, such as data sizes, to assist the kernel in executing. Importantly, the computational kernel is a pure function operating on data local to the SPE (i.e. all communication is explicit within the AVOp streams).

Table 5.2: AVOp Instruction Set

5.3.2 Runtime System

The Runtime System is written in assembly language (on the SPU) and Python[Lut96] on the PPU, and was implemented by McMaster undergraduate student and Coconut research group member Gabriel Grant.

The Runtime System is executed from a command prompt, and expects as input a specifically formatted text file which gives the following information:

- Binary file location(s) which contain raw input data
- Binary file location in which to store output data
- Size of input, output data files
- Pre-compiled computational kernel file locations
- AVOp stream file locations

The AVOp stream files, one for each SPE involved in the computation, are converted into the binary 128-bit per AVOp structure expected by the Runtime System. The binary input files are loaded into main memory, sequentially using the sizes of the files as offsets into a global memory space, that the Runtime System then corresponds to the global memory addresses referenced by the AVOps. So if one was doing a single precision matrix multiplication, involving 64x64 matrices, then the first matrix A would be stored between memory addresses 0 to 16383, and the second stored between memory addresses 16384 to 32767, assuming the files containing the input data were specified to the Runtime System in that order. The output data files sequentially make up the rest of the global memory space. In the case of the 64x64 matrix multiplication, during execution the memory space occupied by C would be from 32768 to 49152, and this memory would be dumped to the binary output file by the Runtime System upon execution completion.

Once the Runtime System begins execution, the following will take place in sequence:

1. Configuration file is interpreted

2. AVOps are converted to 128-bit Runtime System internal representation

3. Input data is loaded into main memory from binary files

4. SPE Runtime System threads are launched

5. Kernels and first block of AVOp streams are pre-loaded to SPEs

6. SPEs begin executing and continually buffering in respective AVOp streams until complete

7. Output data is written to binary output files

It is up to the user to then interpret the binary output files however they desire, just as it was up to the user to create the input data however they desire. In practice the Coconut research team has used the NumPy[MSL+07] extension in Python to create input data and interpret output data with relative ease.

The AVOp streams themselves can be generated however the user desires, as is discussed in Section 5.3.3, and the same is true of the computational kernels (SPU Instruction Set assembly code). The configuration file required by the Runtime System to specify all these objects is for now manually created by Coconut developers, however automatically generating this file is of course possible as part of the AVOp stream and computational kernel generation process.

The AVOp streams are buffered into the SPEs using two 16KB buffers. As AVOps are 16B length, this means a 16KB DMA transfer from main memory to the SPE is initiated after every 1024 AVOp instructions to buffer in the next set of AVOp instructions. The set of AVOps immediately next should have already been loaded at this point, unless those 1024 instructions were able to execute before the load could complete. While theoretically this is possible, as it only takes about 100 cycles for an AVOp to be interpreted and executed (not including time waiting for transfer completion or executing computational kernels), in practice this should never occur as executing kernels and data transfer latencies will take up non-trivial amounts of time.

As was discussed in Table 5.2, the signaling mechanism of the CMF is implemented using each SPE's signal register. When a signal is sent using a SendSignal, what is happening is that the SPE sending the signal is performing an OR operation on the memory mapped signal register of the receiving SPE. Each bit of the signal register identifies a different signal, from 0-31. If two

SPEs were to send the same signal, and they were both to be OR'd into the signal register, than there would be no way for the accompanying WaitSignal AVOps to realize that two signals had arrived and a deadlock would result. Luckily the Verification Tool discussed in Section 5.3.5 checks that exactly this situation will not occur.

The Runtime System uses DataTags and a WaitData instruction to check on a receiving SPE that a data transfer from a remote SPE has completed. When the SendData AVOp is executed on the sending SPE, it takes a DataTag, a number from 0-31, as an argument. A fence instruction is used to ensure that the DataTag is sent only after all of the packets required to send the actual data have completed. This ensures that the DataTag will arrive at the receiving SPE only after the data block transfer has been completed. As such, WaitData instructions with a DataTag argument will stall until that particular DataTag has been written.

DataTags are an abstraction to deal with the issue that an SPE which did not initiate a transfer is unable to check a DMATag to ensure that transfer's completion. In the case of an SPE executing a LoadMemory, StoreMemory or SendData AVOp, the SPE is initiating the transfer, and checking for transfer completion merely involves executing an SPU instruction to check that the DMA transfer with the given DMATag has completed. DMATags come numbered 0-29, as DMATags 30-31 are reserved by the Runtime System to ensure safety in loading in the AVOps into the buffers and the computational kernels.

5.3.3 AVOp Stream Generation

The generation of AVOp streams themselves is an ongoing topic of research for the Coconut project. At present AVOp streams are created using generator functions in Haskell, which given specific problem parameters, such as input data dimensions, generate streams of AVOps for that problem domain. Other AVOp stream generation methods are being considered for future implementation, including loop scheduling, DAG scheduling and ILP Coconut layer techniques.

One thing that is common to all techniques however, due to the correspondence purposely made between ILP and the CMF, is a need to treat resources such as local storage memory space, signals, DataTags and DMATags much like one does registers in ILP. It is up to the programmer to assign these resources so that they don't overlap, just as a compiler does registers in ILP.

5.3.4 Computation Kernels

Computational kernels are pure functions, with no communication with other SPEs embedded within them. All of the real computational 'work' takes place within these kernels. While computational kernels may be developed using any means, be it by implementing C code functions or by hand coding them in assembly, the Coconut project has focused specifically on developing kernels using the ILP layer of Coconut. This work began in earnest in the summer of 2009, with several undergraduate Coconut researchers developing kernels for signal processing applications.

5.3.5 Verification Tool

The Verification Tool was implemented by the author as a Haskell module, based on the algorithm defined in a chapter of the book "Process Algebra for Parallel and Distributed Processing" [AK08], written by Coconut research team founders Anand and Kahl. A full explanation of the algorithm behind the Verification Tool is beyond the scope of this thesis, but an overview of the key ideas is instructive.

Firstly, our claim that the Verification Tool is able to check for situations where parallel bugs may occur (race conditions or deadlocks) in linear time is theoretically sound. Our virtual machine abstraction does not contain instructions which allow for access to non-local memory (the data must be transferred to local memory explicitly by an instruction), and this means that we do not use mutexes as a synchronization method. While it may be an NP-hard problem to check for race conditions in a program using multiple semaphores[CMS01], our synchronization method is weaker than semaphores, and thus it is theoretically sound that an algorithm to detect parallel bugs is possible[LKN96]. The Verification Tool is able to return the location of the error within the AVOp streams, and the type of error.

The Verification Tool requires several assumptions about the AVOp streams in order to function correctly. Firstly, the AVOp streams themselves must be presented to the algorithm as a single list of pairs *(SPEID, AVOp)*, referred to as the *presentation order*. A program is considered *locally sequential* if every *(SPEID1, SendSignal SPEID2 SignalID)* is followed by a corresponding *(SPE2, WaitSignal SignalID)*, and every *(SPEID1, SendData SPE2 LocalStore-Address DataLength LocalStoreAddress DMATagID DataTagID)* is followed by corresponding *(SPEID2, WaitData DataTagID)* and *(SPEID1, WaitDMA DMATagID)* instructions. The Verification Tool requires its input to be locally sequential, and it verifies that a program is *order independent*. A program

is order independent if given the same input (in main memory) all possible execution orders produce the same output (in main memory).

The reason we discuss all possible execution orders, is because the AVOps themselves define a partial execution order, with the locally sequential presentation order the Verification Tool uses being one possible total execution order. The execution order of AVOp streams is a partial order because only the AVOp streams for each individual SPE are total orders, we know for certain that each AVOp in an SPE's stream of AVOps will be executed sequentially. However the only guarantee that one instruction on an SPE will execute before or after another comes from pairs of Send and Wait instructions, be they signals or data transfers. We know that if one SPE signals another, and the other SPE waits for that signal, that all of the instructions on the signaling SPE before the signal was sent will have executed *before* all of the instructions on the receiving SPE after the signal was received. This situation is visualized in Figure 5.4. In this way, AVOps induce a partial execution order across all SPEs.

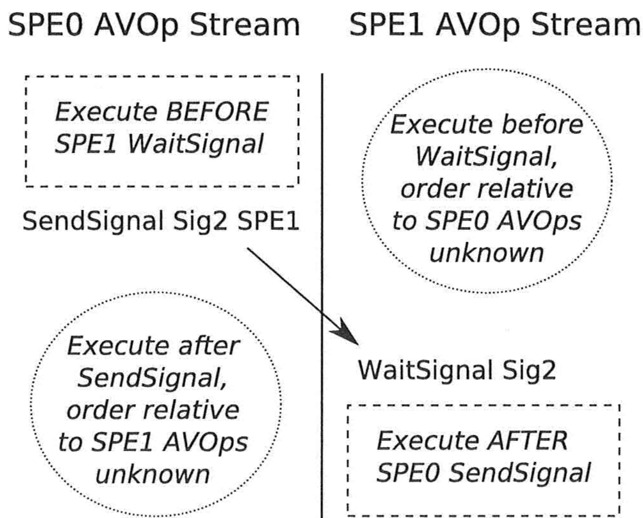

Figure 5.4: AVOp partial execution order induction

A quick example of how the Verification Tool works is given as follows. If given a locally sequential program, two identical signals (e.g. 4) are being sent by SendSignal operations from two different SPEs, to the same SPE, we may have a synchronization issue. If both signals arrive before the first WaitSignal on the receiving SPE, that WaitSignal will consume the signal, and with no incoming signal to set the appropriate signal bit of the signal register, the second WaitSignal will stall indefinitely and deadlock the system. If the

signals arrive in the wrong order, the wrong information may be conveyed. For example the SPE receiving the signal may send data to an SPE believing that it is safe due to the signal being received, when it may not be safe, as it was the wrong signal, or rather the signal from the wrong SPE, that was received.

The Verification Tool filters for such a situation by checking the SendSignal operations targeting the same SPE with the same signal. It does so by checking, when encountering a SendSignal, that the last WaitSignal to have consumed the particular signal on the receiving SPE, is guaranteed to have executed *before* this SendSignal will execute. If this were not the case, it is possible that two SendSignal instructions sending the same signal will be targeting the SPE in a window in which they could conflict. It is possible to check for a guarantee that the SendSignal will execute *after* the last time a WaitSignal consumed a signal, because the Verification Tool keeps track of the partial execution order induced by the AVOps as it sequentially checks the presentation order. If this property cannot be guaranteed, then an error is reported by the Verification Tool and the SendSignal will be flagged.

In this way, a single linear pass through the AVOps in the presentation order, constructing a partial execution order and keeping track of possible state, the Verification Tool is able to filter out the possibility of such synchronization errors.

The Verification Tool can also check for other properties to ensure a lack of parallel bugs. For instance, when checking a RunComputation instruction, the Verification Tool can check the memory locations of the inputs to the computational kernel. If a WaitDMA or WaitData instruction has not confirmed the arrival of the data, which is part of the state kept track of by the Verification Tool, then an error can be flagged at the RunComputation, as it is possible that data has not yet arrived.

The Verification Tool is able to do all of this with linear-time complexity, firstly because it requires only a single pass through the presentation order of AVOps. But also because there is a limited amount of possible state that can exist, and thus needs to be checked or updated, at any step of the algorithm checking a single AVOp. This important property allows for scalability. As processing elements in future network-on-a-chip systems scale into the hundreds or thousands, if the algorithm were exponential in complexity runtime would quickly become infeasible.

5.3.6 Performance Simulator

The Performance Simulator, which just as the Verification Tool executes with linear time complexity, provides a compile-time metric with which to com-

pare the performance of different AVOp streams. The main benefit of this is increased programmer productivity, as the more time consuming full system testing to compare algorithms becomes much less necessary. The data from performance simulation includes information on where specific latencies were problematic (i.e. caused a Wait AVOp to stall), which is more illuminating information than is available with a full system test. The full details of the Performance Simulator are presented in Chapter 6.

5.4 Comparison with Other Frameworks

There are many similarities and differences between the CMF and the other frameworks targeting Cell/B.E. that were discussed in Chapter 4. As the CMF at present is not a completely developed end-to-end solution, in the sense that AVOp generation is still an active area of work, it is difficult to make a full comparison. In the sense that at the level of the virtual machine abstraction, communication and tasks are done explicitly through AVOps and computational kernels. However at future higher-levels of the CMF for AVOp generation, tasks and communication may not be explicit, they may possibly be generated implicitly from a higher-level specification. An effective and popular way of comparing programming models is to contrast which aspects of the system, such as synchronization, task creation, task mapping, data distribution and communication, are either explicit or implicit[ABC+06; SYR+08]. At the layer of the virtual machine abstraction, basically everything is explicit, however as we move to higher-level AVOp generation techniques an obvious goal will be to make some of these properties implicit to make programming with the CMF easier. With this said, we can still make some meaningful comparisons between the CMF and existing models.

Though it is theoretically possible to create a method of translating programs written in a traditional imperative language to an AVOp representation, this is not planned. Such a system could work by translating the parallelism of the imperative language into an abstract intermediate format, and then further translation to our AVOp streams. As it is however, the CMF is somewhat unique in that it does not attempt to plug-in to existing languages. Many solutions, and many of the more popular solutions, such as RapidMind and OpenMP, attempt to augment existing imperative languages with APIs or new abstractions to wedge easier multicore parallelism into existing systems. Due to the commercial focus of these frameworks, this should not be surprising given the issue of software development economics. The CMF joins other frameworks spawned from more academic environments, such as Sequoia and

CorePy, which perhaps are less inclined to consider the cost of abandoning legacy code, in moving away from imperative code and towards higher-level abstractions.

One property common to the CMF and many other models, such as ALF and the Mercury MLF, is a clear separation of the levels of parallelism. In the CMF all ILP parallelism is contained in the computational kernels, and all multicore level parallelism is expressed by the AVOp abstraction. The ALF developers note, as we have, that this has the benefit of allowing developers to specialize[IBM07a]. The usage of the PPE as a controlling thread that is responsible for farming out work to the SPEs is also a common feature to virtually all programming models, but this is likely due to the design of the Cell/B.E..

A design that takes into account portability concerns is common to many of the other models, for example RapidMind and API type solutions such as OpenMP, OpenCL and MPI. The authors of those models created for Cell/B.E. specifically, such as CellSs, make a point to suggest that their model should be portable to other architectures. The virtual machine abstraction is how the CMF intends to achieve portability. That this is a common concern factored into model designs is not surprising, given processor lifecycles and the desire to leverage existing parallelization technologies rather than build them from scratch for each architecture.

What makes the CMF unique is the virtual machine abstraction, and particularly the correspondence with ILP parallelism that goes along with this abstraction. The CMF should allow for easier translation of ILP optimization techniques to the multicore level than competing frameworks. The virtual machine abstraction, and the exposure of parallelism, allows for verification and simulation to take place. The ability to formally verify correctness and to simulate execution is not unique to the CMF, but we suspect they can be done in a more computationally efficient manner than other frameworks.

The fact that the CMF does not yet have control-flow mechanisms at the multicore level, and the fact that AVOp streams are generated in their entirety at compile-time, limits what problems the CMF can handle to those which can be statically scheduled at compile-time (that is without some creative embedding of control-flow at the level of computational kernels). Most competing frameworks do not have this restriction, they either allow dynamic creation of new tasks at runtime or multicore level control-flow. This limits the problem domains appropriate for the CMF relative to most other frameworks.

5.5 Current Status

The current status of the CMF may be of interest to the reader. The Runtime System has been implemented and unit tested. AVOp generation has been achieved using generator functions in Haskell. Correctness tests for matrix multiplication parallelizations have been executed, demonstrating that the Runtime System itself is functioning correctly. An immediate future goal includes performance testing of the Runtime System on signal processing problem domains using computational kernels generated by the ILP layer of Coconut.

Chapter 6

Performance Simulation

The Performance Simulator has been designed, implemented and tested by the author as part of his contribution to the CMF. In this chapter we will go over the motivation to build the Performance Simulator, some general concepts behind the design of parallel system simulators, as well as some similar performance analysis tools. The Performance Simulator design is explained, and an overview of the implementation and testing is given. Performance testing of the Performance Simulator is also performed in order to demonstrate the potential effectiveness of the tool to compare the efficiency of different algorithms.

6.1 Motivation

There are several motivations to build a performance simulator for the CMF that are worth outlining. The most important reason to build a performance simulator is that it allows for quick, compile-time analysis of an AVOp program's runtime, allowing us to quickly compare different schedules. The motivations to simulate performance for parallel applications specifically have been noted in the literature by Eric A. Brewer and William E. Wheil[BW93], we go over these arguments first as they still hold today. These arguments include that workstations are cost effective and available relative to more scarce multiprocessor hardware, that simulation is repeatable, that simulation can measure everything and record all state, that simulation is non-intrusive and finally that simulation is versatile. Another argument in favour of simulation that we suggest is that it may also be used as a tool to actually help perform scheduling, and not just as a compile-time tool for programmers to evaluate existing schedules. Key arguments against simulation are that it is too

slow and too inaccurate, we will refute these arguments with recent literature suggesting otherwise.

Firstly simulation is advantageous because access to a physical multiprocessor system is not required during development. Though Coconut has relations with IBM through the Center for Advanced Studies program at IBM Toronto, which allows us access to QS22 Cell/B.E. Blade Servers, these machines must be booked in advance in order to ensure exclusive usage during testing. Performance Simulation allows us to check code performance anytime, without exclusively booking a limited resource. We can then better utilize the time that we do book on Cell/B.E. Blade Servers. This is obviously a more cost effective usage of resources, as Cell/B.E. Blade Servers are significantly more expensive in hardware cost and power consumption than client workstations. Though testing on a Playstation 3 is as cost effective as client workstations, the fact that it only has 6 SPEs, limited main memory and a lack of double precision performance makes it relatively less appealing for testing performance of certain problems[BLK+07].

Another advantage of simulation is that it is exactly repeatable, unlike runtime execution. The entire state of the system during simulation can be known, and it can be repeated or done piecemeal, to track down desired information about execution. The utility of this is similar to step-through debugging tools available for sequential code; indeed simulation can be used to debug code[GG74]. The problem with monitoring dynamic execution is that the safety of parallel programs depends on small timing differences[RD00], tracking state may have side effects which effect these small timing differences, skewing the meaning and accuracy of results. Note that because our Verification Tool filters out the possibility of parallel bugs, the usage of step-through simulation as a debugging tool is not an advantage for the CMF specifically.

Simulators also have the advantage of being able to measure virtually every aspect of state. Aspects such as network contention, which are difficult or impossible to measure at runtime, may be simulated and observed. Indeed, when using the IBM Full-System Simulator, researchers have noted that it has allowed them to view aspects of execution otherwise impossible to observe[KPP06]. The results of simulators can be logged, and visualized in a human-friendly format, such as concurrent state graphs, where problems are more easily exposed[BW93].

As was alluded to, simulation also has the benefit of being non-intrusive. Performance profiling at runtime, by for instance keeping track of the state of program execution while it executes, introduces intrusion effects. Though these effects can be mitigated by compensating for measurement overhead, on some level runtime performance profiling can never be sure if it is measuring

the objective truth of program execution (as it would be without profiling), as profiling itself cannot entirely prevent itself from obfuscating what "true execution" would be[MS04]. Simulation completely eliminates the problem of intrusion effects, though the accuracy of the results is still a concern.

Simulation is also versatile, in the sense that code may be simulated on multiple target architectures, or even theoretical architectures that do not yet exist. Simulation of execution on theoretical architecture has yielded useful results in the literature, for instance to expose algorithm problem areas[CHV04]. Simulation was used in the design of the NUMAchine multiprocessor by University of Toronto researchers in the mid-1990s, to explore different architectural trade-offs[ea95]. The ability to simulate theoretical architectures is of particular interest to Coconut as a research group, as concept architectures, though not yet confirmed to be put into production, have appeared on a IBM Cell/B.E. Technology Roadmap[Koc07]. The ability to fine-tune algorithms through simulation before these architectures are available to the public could give Coconut and the CMF a competitive advantage over competing solutions. It could also shorten the time gap between the availability of new hardware and the efficient usage of that hardware.

While all simulation advantages thus far have already been cited by Brewer and Weihl, one additional advantage we believe simulation provides is the ability to use it *as part of* a scheduling algorithm to optimize multiprocessor performance. An example of using performance simulation, or at least static performance prediction, as part of a scheduling algorithm, is the Titan scheduler[JSK+06]. This scheduler uses the PACE toolkit, along with a genetic algorithm, to optimize scheduling. The fitness function relies on PACE runtime predictions to evaluate the desirability of schedules, allowing a genetic algorithm iteratively applied to an initial set of schedules to optimize performance.

One final argument in favour of simulation is that just as static analysis tools are becoming increasingly prevalent[WMZ+08], and thus perhaps in the future *expected* as part of any development solution, one can foresee a trend towards standardized static performance prediction. For instance it has been proposed that performance model ling based on simulation be specified in UML[BM03], due to the ability of simulation to increase development efficiency. As a result, we are motivated to build a simulator to keep the CMF competitive over the long term. The existence of such a tool in development solutions may eventually be expected by developers.

Though potential disadvantages of simulation are cited as being that it is too slow and too inaccurate[BW93], we do not foresee these being an issue for our Performance Simulator. We do not believe that the accuracy of

simulated performance relative to real execution performance to be an issue, because there is much evidence of reasonably accurate simulation. The performance predictions made by the PACE toolkit are noted as having a 5% average error[JSK+06]. Specifically with respect to simulating the Cell/B.E., the IBM Full-System Simulator was noted to produce results that were "good" [KPP06]. A performance model for the Cell/B.E. has also been created and successfully validated against results executed on real hardware[WSO+07]. Due to the high-level of abstraction the CMF operates on, we are not very concerned about the slow speed of simulation. Though low level instruction set simulation is slow, when higher-level abstractions are used simulation that is both fast (150x faster than an instruction set simulation) and accurate (less than 6% error) have been reported[CHB07]. As our AVOps are a relatively high-level abstraction with only the most important features of parallelism exposed, we believe that fast simulation should be possible for us as well.

6.2 Simulator Design Concepts

It is useful to briefly review some issues in simulator design, such as the different types of simulators, the contention between accuracy and speed, and the issues specific to parallel computer simulation, to give our Performance Simulator's design some context.

6.2.1 Simulator Types

The following classification of the different types of simulators is derived from "The Efficient Simulation of Parallel Computer Systems" [CDJ+91]. Note that the computer executing the simulation is referred to as the **host computer**, while the computer being simulated is referred to as the **target computer**.

Instruction-level simulation is when a host computer simulates at a high level of detail the effect of executing instructions on the target computer. This is a highly accurate form of simulation; the level of detail to which target computer instructions are simulated could be taken down to the physical hardware if desired. The problem with instruction-level simulation is that hundreds of host computer instructions could be required to simulate target computer instructions, which may become too inefficient to be practical for large programs.

Distribution-driven simulation is when probabilistic models are used to model the behaviour of a program, and simulation occurs through statistical approximation. It is typically used for high-level comparisons of

different parallel architectures. Its appeal is derived from the fact that it eliminates the need to simulate both the large amount of instructions on each processing elements of a parallel computer, and the need to simulate the communication occurring between processing elements.

Trace-driven simulation occurs when a trace of program execution on a computer with the same instruction set as the target computer is used to simulate execution on a target architecture. The drawback of this method is that it requires a computer with the same or similar instruction set as the target computer. The problematic and hard to reproduce small timing variations that make parallel programs hard to debug, also make trace-driven simulation ineffective for simulating parallel computer systems as these small timing differences can effect when and what events occur[DG90].

Execution-driven simulation is when a host computer executes the instructions to be simulated directly on the host. Precise timing information of target instruction execution is lost compared to instruction-level simulation, but the target architecture specific timing information can still be accounted for with a high degree of accuracy. The appeal of execution-driven simulation is the high performance gains that are achieved in exchange for relatively small losses in accuracy. The scalability of execution-driven simulation due to its relative efficiency makes it better suited than instruction-driven simulation for parallel architecture simulation.

6.2.2 Simulator Accuracy vs. Speed

The contention between simulation accuracy and simulation speed is a critical issue in simulator design; researchers often attempt to find fair trade-offs to achieve desired levels of accuracy and speed[CHB07; DPA99; DG90]. In general what can be observed is that simulating at higher levels of abstraction as opposed to simulating the precise effect of target machine instructions leads to improved simulator speed by orders of magnitude, at a relatively small cost to accuracy.

6.2.3 Parallel Computer Simulation

Parallel computer simulation introduces the issue of scalability as the number of processing elements may reach into the thousands, and a need to simulate or predict the performance of communication networks between processing elements[CDJ+91]. In order to be scalable, parallel computer simulators will simulate based on runtime assumptions about more coarse grained objects, such as blocks of instructions, as opposed to individual assembly

instructions[HMS+09]. Network simulation is itself a field of study[Bar93; Kes88; YKJ02], with similar trade-offs between accuracy and speed depending upon the level of abstraction. The reason the communication network's performance must also be simulated or predicted is that during parallel computer simulation **processor-interaction points** will occur, where a processing element's execution is dependent upon the action of another processing element, or communication to or form another processing element[CDJ+91].

6.3 Similar Tools

In this section we briefly overview some similar tools to our Performance Simulator, including multiprocessor simulators, Cell/B.E. simulators, network simulators as well as some alternative solutions for performance analysis.

6.3.1 Multiprocessor Simulators

Multiprocessor simulators target different types of parallel architectures (based on the communication network), and may themselves execute in serial or parallel. LAPSE is a simulator that executes in parallel, and targets message-passing networks[DHN94]. MaxPar is a simulator that targets shared memory systems without caching, and executes in serial[CSY90]. Simulators that target shared memory systems with cache-coherent shared memory include PROTEUS[BDCW92], TANGO[DG90] and WWT[MRF+00], with WWT executing in parallel.

Many multiprocessor simulators, such as PACE[JSK+06] and WARPP [HMS+09] amongst others[CHB07], feature the ability to configure network, hardware, operating system or application models. In more recent literature, simulators with high runtime efficiency are being used to assist in scheduling algorithms themselves[JSK+06], or are being considered by the authors for usage in *automatic* algorithm design space exploration[CHB07]. This can be seen as a natural extension of the role of simulation, instead of having a programmer use the results of simulation to find optimization opportunities or compare algorithms, this process itself can be made automatic.

6.3.2 Cell/B.E. Simulators

The Cell/B.E. has an official IBM Full-System Simulator that is available as part of the Cell/B.E. SDK[JB07]. The simulator allows one to simulate both single processor and dual processor Cell/B.E. systems. The simulator also has

different modes, one which places an emphasis on speed to provide increased interactivity to developers, and one which provides slower but cycle accurate simulation.

An internal version of the IBM Full-System Simulator appears to exist [KPP06], which includes performance models for the EIB, MFC and memory subsystems. It is not clear if this internal version of the simulator includes features not accessible in the version included as part of the SDK, but the results were reported as having a good correlation to real hardware.

Cell/B.E. simulation has been reported outside of IBM, by researchers at Lawrence Berkeley National Laboratory [WSO+07]. Using the Mambo [BPE+04] cycle accurate full system simulator and a Cell/B.E. performance model, Cell/B.E. performance was successfully predicted for applications such as stencil computations and sparse matrix-vector multiplication. The performance model is not specified precisely, but we are told that static timing analysis is performed on code snippets that execute only on the data of an SPE's local store (the same concept as computational kernels in the CMF). We are also told that the DMA transfer latencies are modelled by taking into account resource constraints and assuming a fixed cost to start-up DMA transfers (1000 cycles). The model is able to accurately predict Cell/B.E. performance compared to execution on physical hardware. The authors suggest that Cell/B.E. performance prediction is "far easier than traditional superscalar architecture". Interestingly, the researchers also simulated execution on a theoretical Cell/B.E. with improved double precision throughput.

6.3.3 Network Simulators

We are interested in network simulation because the Cell/B.E. is a parallel computer with a communication network, and that communication network's performance must be accurately modelled for simulation to be accurate. Some network simulators are purpose built for Local Area Network (LAN), Wide Area Network (WAN) or peer-to-peer network simulation, others allow the user to configure network properties. The issue of scalability, and the issue of contention between accuracy and efficiency, are as important as they are in multiprocessor simulation.

NetSim is one example of a network simulator, it was developed to simulate LAN (Ethernet) performance, and is a sequentially executed discrete event simulator[Bar93]. REAL is a network simulator which targets WAN networks for simulation, simulation of thousands of nodes is possible using a distributed version of the software[Kes88]. OverSim is an example of a peer-to-peer network simulator which uses discrete event simulation[BHK07].

Other network simulators, such as Ns-2, feature the ability to specify network topology, protocols and routing algorithms[YKJ02]. Network-on-a-chip architectures have been simulated using this software, in an attempt to evaluate network-on-a-chip design options. OPNET is another simulator that actually provides a complete development environment to specify network topologies, simulate and then to conduct performance analysis[Cha99].

6.3.4 Alternative Performance Analysis Solutions

Alternative solutions exist for performance analysis, notably static profile-based prediction tools that do not rely on simulation, and performance profiling tools that record the behaviour of software as it actually executes.

Results for runtime performance profiling tools can be reported in various ways, as a log of events or as a statistical summary of events. Particular concerns with runtime performance profiling is that they may introduce intrusion effects due to their need to sample execution state[MS04]. An example of a runtime performance profiling tool is XenMon, created to monitor and profile Xen-based virtual environments[GGC05].

Static profile-based prediction methods[CG94; DB00] use performance profile information about how frequently certain blocks of code are executed, combined with execution times of these blocks, to compute projected performance for a larger program.

6.4 Performance Simulator Tool

The design of the Performance Simulator is given in this section, including its envisioned usage and objectives, to give context to the design decisions that have been made.

6.4.1 Envisioned Usage

The critical reason to develop the Performance Simulator was to provide Coconut developers with feedback to help optimize and contrast generated AVOp schedules. However while developing the Performance Simulator, we realized it had potential to be integrated into our development environment as a tool to automatically select an AVOp schedule from amongst several generated by competing heuristics. Furthermore, the Performance Simulator could be used as a tool to enhance scheduling algorithms themselves. These latter two

potential usages fall out from the linear-time complexity efficiency of our Performance Simulator.

Schedule Optimization Feedback for Developers

The envisioned usage of the Performance Simulator is first and foremost as a tool to allow Coconut developers to quickly and *meaningfully* compare the performance of different AVOp schedules from a workstation, without having to access a Blade Server. One might reasonably ask why it is not sufficient to execute the Runtime System and AVOp streams on the IBM Full-System Simulator, which can be executed on a workstation. Though this is a possibility, it has several disadvantages. The IBM Full-System Simulator is less convenient to use than a tool that we can integrate directly into our software. The results are also less meaningful than what our Performance Simulator can provide, as we can log information about simulated execution that corresponds to each individual AVOp instruction abstraction. For instance by operating at the level of the AVOp abstraction, our Performance Simulator can easily log exactly how long a specific AVOp Wait instruction will stall, information which is particularly useful to developers looking for optimization opportunities. As memory access is considered the performance bottleneck for most algorithms on Cell/B.E.[VKJ+07], the ability to detect these stalls and our ability to reduce or eliminate them is of particular interest.

Automated Schedule Selection

More ambitious usage of the Performance Simulator includes the possibility of integrating it into the process of AVOp generation, such that it automatically selects the best of several competing schedules. As multiprocessor scheduling is an NP-complete problem[KA99], we cannot expect to create algorithms that produce provably good solutions in provably good runtime (where provably good runtime is defined as polynomial time). Instead heuristics and approximation algorithms must be used. These algorithms may have the property that solutions may not necessarily be optimal, they may only be "good". These algorithms may also not be able to produce good results for all inputs; for some particularly challenging inputs the algorithms may produce very slow schedules. As a result, it may be prudent to schedule AVOps using multiple heuristics that are applied to a common high-level mathematical problem description (in keeping with the goal of Coconut of providing a solution from high-level specification to optimized machine code). The resulting AVOp streams could then be applied to the Performance Simulator, and the optimal result could

be selected. This usage scenario is depicted in Figure 6.1

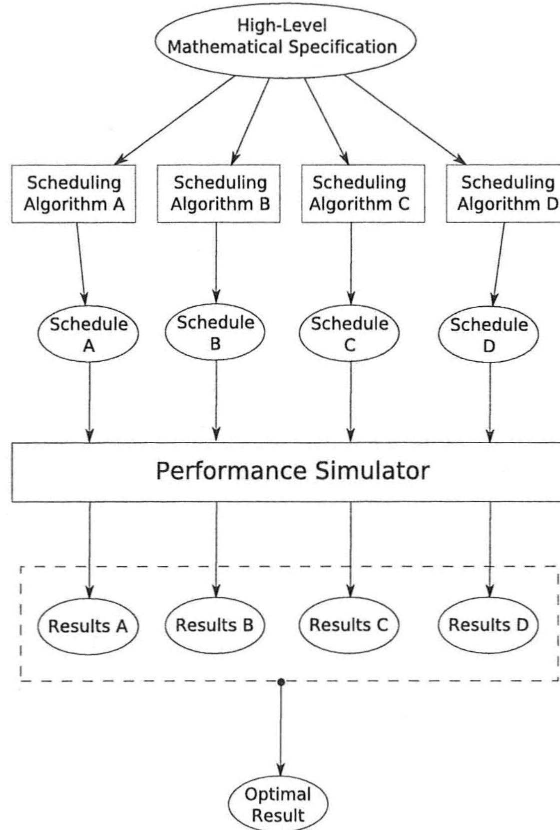

Figure 6.1: Automated Schedule Selection

Simulation Enhanced Scheduling Algorithms

An even more ambitious usage of the Performance Simulator would be to integrate it within scheduling algorithms themselves. This idea is only made possible by the efficient runtime of our Performance Simulator; indeed others who have integrated or propose integrating simulation into scheduling algorithms also use efficient simulators. The Titan scheduler discussed in Section 6.1 which uses simulation results to drive a fitness function in a genetic algorithm working on a population of schedules is one such example[JSK+06]. Other researchers with simulation technology orders of magnitude faster than instruction set simulation have expressed interest in creating automatic design exploration techniques to discover solutions in the design space, using simulation as a method of quickly evaluating potential solutions[CHB07].

We foresee potential uses for the Performance Simulator in scheduling algorithms as well. As proper cost information, such as data transfer latency costs and kernel execution costs, is critical to scheduling algorithms (for instance DAG scheduling algorithms[KA99]), there is an opportunity to use the Performance Simulator to accurately project the runtime cost of making different scheduling moves. In this way greedy algorithms could be formed, making optimal moves from the task graph to be scheduled to AVOp streams. This scenario is shown in Figure 6.2.

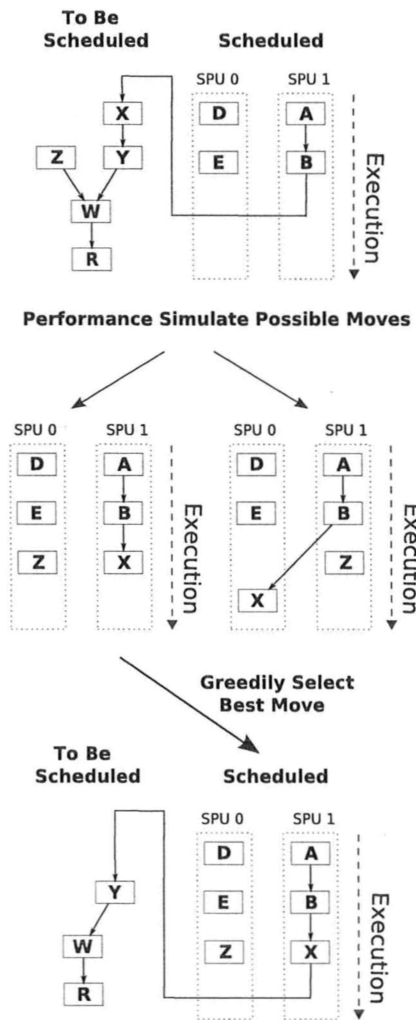

Figure 6.2: Simulation Enhanced DAG Scheduling

Another possibility closer in form to the Titan scheduler, is a scheduling algorithm that when given an initial schedule(s), applies simulation and uses

the result(s) along with the initial schedule(s) to create new and improved schedules. This iteration could be stopped once certain conditions are met, such as a desired level of efficiency achieved known to be within theoretical possibility, or when a certain number of iterations have occurred. Algorithms could identify stalls in the result data, and possibly apply techniques to re-arrange the schedule to try to reduce their occurrence and/or severity. This scheduling possibility is shown in Figure 6.3.

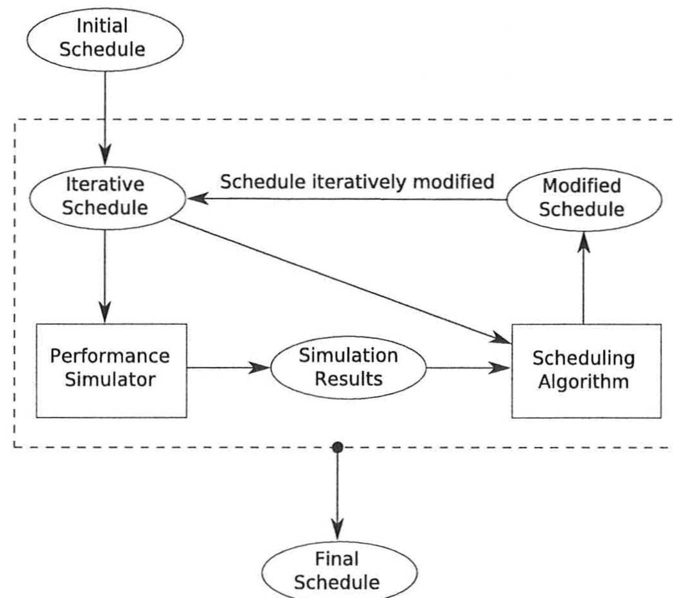

Figure 6.3: Iterative Simulation Result Driven Scheduling

6.4.2 Objectives

While formal software engineering style requirements are beyond the level of specification desired for an exploratory concept tool like our Performance Simulator, we did have certain objectives in mind when designing the tool. These objectives include high efficiency, versatility, scalability and reasonable accuracy.

One objective is that the tool have a linearly bounded complexity, as a single pass through the (albeit merged) AVOp streams was enough for the Verification Tool. This should be similar with the Performance Simulator, as in both cases there is a bounded amount of state that is possible while evaluating any given AVOp, as the Cell/B.E. itself is bounded by resources such as limited local stores, signals and DMATags. The fact that state is

bounded means that there is also a bounded amount of analysis that could realistically occur on the state, this bounded amount of analysis will not scale with the number of AVOps. As such, $\mathcal{O}(n)$ linearly bounded time complexity should be possible, even if each AVOp evaluation contains some polynomial time algorithms, they are still applied to a bounded amount of state.

Versatility is an objective in the sense that we want to be able to simulate theoretical Cell/B.E. architectures. A high degree of modularity is how we hope to achieve this - by separating simulation algorithms (AVOp execution and network simulation), computation execution time definitions and network bandwidth configuration, it should be possible to quickly alter components from one architecture's simulation to another. This should allow us to simulate future architectures more quickly, by only having to modify what is necessary and leveraging existing simulator code where possible. This should also allow us to explore different potential architectures for advantages and disadvantages.

Scalability is a concern closely coupled to high efficiency and versatility. We want the Performance Simulator to be efficient enough that it can be scaled to not only the concept 32 SPE Cell/B.E. architecture[Koc07], but also potentially architectures with thousands of processing elements. This objective is achieved not only through efficient runtimes, but a runtime complexity that does not grow exponentially with the number of processing elements or the number of AVOp instructions.

Accuracy is another objective of the Performance Simulator; however we note that the primary goal of the tool is to allow a Coconut developer to contrast the effectiveness of different algorithms. If a developer can recognize that an algorithm's schedule is superior to that produced by its alternatives, then we are content with this level of accuracy. We desire simulator results that are close to physical hardware execution performance; this goal is secondary however to being able to use the tool for comparing schedule efficiency.

6.4.3 Design Overview

In this section we will explain how the algorithms that make up the Performance Simulator work, including a discussion of key design decisions as well as a breakdown of the modules involved. We should note that this iteration of the simulator was implemented specifically for the single processor 8 SPE Cell/B.E. architecture, but due to the design of the simulator, it is easy to simulate 6 SPE Cell/B.E. architecture with very little modification, and possible to simulate dual processor 16 SPE Cell/B.E. Blade Servers with reasonable modifications. An overview of the Performance Simulator modules and data

flow is found in Figure 6.4, it will be referenced as we explain the design.

The nanosecond is the unit of time that is used to record runtime performance of individual SPEs, the entire AVOp program, as well as the latencies of individual data transfers. This decision was made because the SPEs and PPEs execute at 3.2 GHz, where as the EIB command bus executes at 1.6 GHz[AP07b; AP07a]. Nanoseconds thus have the advantage of not being tied to any one component's clock frequency, and of not having to be converted to a unit of time as clock cycles likely would be before being reported to the user as runtime performance.

Firstly, to model the execution of AVOp instructions themselves, it would be unnecessary for reasonable prediction accuracy, and computationally prohibitive, to simulate them on the level of the SPU Instruction Set. Instead after a review of the Runtime System implementation, we compute that each AVOp instruction will take approximately 100 SPU cycles (or 0.3125 nanoseconds) to be interpreted by the SPU and to execute. This does not include time spent waiting by the instruction for a DMA transfer to complete or signal to arrive, nor does it include the time to execute computational kernels in the case of the RunComputation AVOp. This AVOp configuration is implemented as a function, and represented by AVOp Execution Cost Function in Figure 6.4.

The model ling of the cost of executing computational kernels that are executed by the RunComputation AVOp is similarly done by a function, mapping from computational kernels to runtimes expressed in nanoseconds. This decision was made for several reasons. Firstly, because we are likely to be generating computational kernels with the ILP layer of Coconut, it is also likely that at this step we can output expected runtimes of these kernels on an SPE. Even if this were not the case, or if we were to create computational kernels without Coconut, this would not be a major impediment. Either static timing analysis could be performed to effectively determine kernel execution time[WSO+07], or the kernels could be executed on SPEs to determine the execution time. If the runtime of the computational kernel is not static, that is it conditionally depends on the input, this is more difficult for the simulator to take into account. At this point the computational kernel's runtime would have to be profiled and the function mapping from kernels to runtimes would become based on statistical estimates. This is not foreseen to be a major issue for the effectiveness of the Performance Simulator as a tool, considering the image processing and linear algebra applications that the CMF is initially targeting. The Kernel Cost Function found in Figure 6.4 represents the function that must be provided by the developer, along with the associated AVOp streams, to perform simulation.

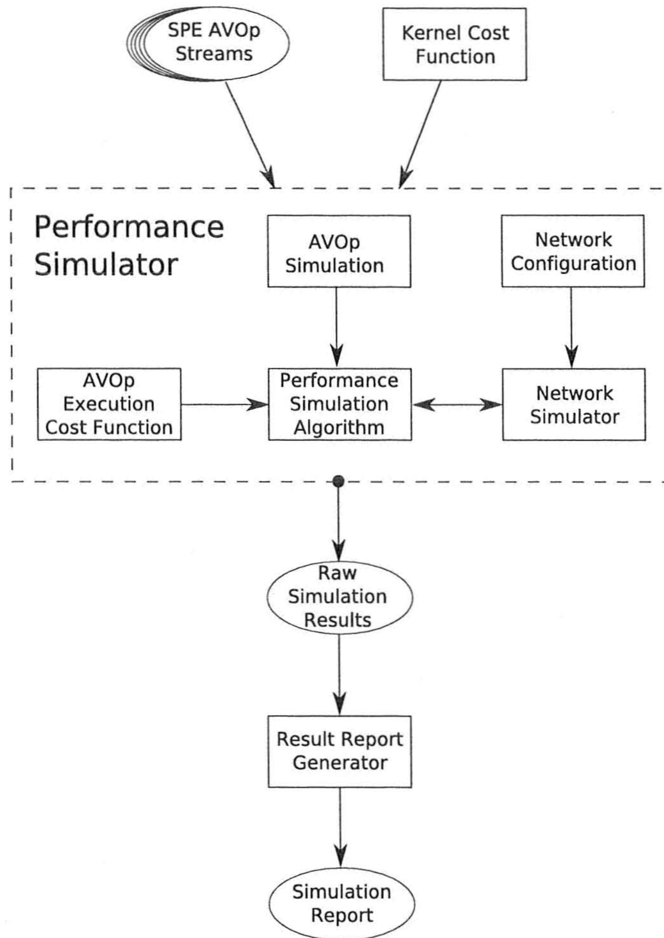

Figure 6.4: Performance Simulation Design Overview

While model ling computational kernels this way provides a tremendous performance gain, as no actual computation takes place, an important consequence of this design decision is that our simulator does not actually compute the result of the computation it is simulating. This would normally be a high price to pay, as one of the main advantages of parallel machine simulation is the ability to debug programs by stepping through execution[BW93]. Debugging capabilities are however noticeably absent from our Performance Simulator objectives listed in Section 6.4.2. This is because our Verification Tool already assures that program results will be independent of execution order[AK08], and as a result we have not yet had to deal with a parallel bug

in our AVOp streams.

The next design decision to be explained is perhaps the most crucial to the functioning of the Performance Simulator. Simulating AVOp instructions on a single SPE cannot be done in isolation of the simulation of AVOps on other SPEs, firstly because communication between SPEs such as signaling and data transfers will have an effect on SPE runtimes. But also because even in the case that SPEs do not interact with one another via signaling or data transfers, they will still have an effect on each other's runtime through their interaction with main memory. Packets for data transfers from the MIC to SPEs are distributed to SPEs in a round robin priority, so the amount of data other SPEs are transferring to and from main memory will directly effect the rate of transfer, and thus transfer latencies, of a given SPE.

This interaction amongst processing elements is of course to be expected in parallel computer simulation, it is effectively the issue of processor-interaction points discussed in Section 6.2.3. What is unique about our AVOps is that they simulate such a high level, with all computation effectively tucked away in computational kernels, that the AVOps themselves become atomic communication operations. This makes the execution of virtually every AVOp a processor-interaction point, through the binding of SPE data transfer performance to EIB performance. This is true in the sense that an AVOp initiating a data transfer will very likely slow down the data transfers of other SPEs due to EIB bottlenecks. Even Wait AVOps that recognize the completion of transfers can be thought of as processor-interaction points, in the sense that recognizing the completion of a transfer frees up EIB transfer capabilities and speeds up other data transfers possibly involving other SPEs. It is the round robin switching between transfer packets, as opposed to some sort of transfer queue where all packets of a transfer on the EIB are delivered before another transfer proceeds, that creates this high level of processor interaction.

As a result a critical decision was made to simulate the AVOp streams with the following principle: we always simulate the next AVOp set to execute globally across all SPEs relative to where the EIB has already been simulated, and we always simulate the EIB up to the execution of this AVOp if it has not been already, and we do this until all AVOps have been executed. The current runtime of each SPE and the EIB is kept track of separately as the simulation progresses. However because simulation of the SPEs is so dependent upon EIB simulation, this becomes a simulation 'floor', in that AVOps are not simulated until the EIB has been simulated up the AVOp, and the EIB is not simulated past a point unless all AVOps that need to be simulated at that point or before it have been.

This helps to ensure accurate simulation, as at every point that an

AVOp is executed, all of the process-interaction that may effect this AVOp's execution will have already been accounted for. Now this next AVOp to be simulated may cause interactions with future AVOps on other SPEs. Determining which AVOp is next to execute globally is a process made trickier by the fact that an SPE could be stalling execution as part of a Wait AVOp, waiting for the completion of either a data transfer or signal. Note that because signals are implemented as DMA transfers at the hardware level, we will simply refer to them as data transfers as well from now on. In the case of a Wait AVOp, simulating the AVOp is not just a matter of accounting for the effect on SPE runtime (as it is for RunComputation AVOps), or starting up a new data transfer. Simulating the AVOp perhaps means stalling execution of the SPE until the associated transfer or signal has completed, if it has not completed already.

We cannot simply "jump ahead" the SPE's runtime to the point where the data transfer has been completed, because it is possible that before this time more process-interaction effects will occur that will have an effect on the transfer rate of the Wait instruction's associated transfer. We might project that at the current level of EIB network activity, the Wait transfer will complete in 10,000 nanoseconds. However if at 5,000 nanoseconds two other transfers were to complete, and at 8,000 nanoseconds another transfer were to complete, this would result in an inaccurate simulation as these transfer completions may speed up the transfer rate of the transfer originally in question. Furthermore, it may not even be *possible* to project the completion time of a Wait instruction, as it is possible that the associated transfer has not even been initiated by another AVOp yet. This is a possibility because SPEs execute asynchronously, there is no guarantee that a Wait instruction will execute after its associated transfer AVOp has been executed. Thus information about the transfer needed to attempt to project latency, such as transfer size, is unavailable at this point in the simulation.

We therefore maintain an SPE state as part of our simulator, and an SPE is either executing or waiting for the completion of a transfer (in the case that we have simulated a Wait AVOp's execution, but that its associated transfer has not yet completed). When we attempt to follow the principle of executing the next AVOp set to execute globally across all SPEs, we are faced with three situations:

1. All SPEs are executing.

2. Some SPEs are executing, some SPEs are waiting for transfer completion.

3. All SPEs are waiting for transfer completion.

In the situation that all SPEs are executing, illustrated in Figure 6.5, determining which SPE should be simulated next is easy. One simply simulates the AVOp on the SPE whose time to execution relative to the EIB's present runtime is shortest, and simulates the EIB up to the point where this AVOp is executed if it is not simulated to this point already. It is possible that the EIB has itself been simulated to this point, because if two or more AVOps are equally far behind in execution, we arbitrarily pick which AVOp to execute because we know the other(s) will be executed next anyway, and at this point we would simulate the EIB up to the point where that AVOp was executed. When the simulation first starts, this situation exists, as all SPEs are executing and are all at equal runtime with the EIB.

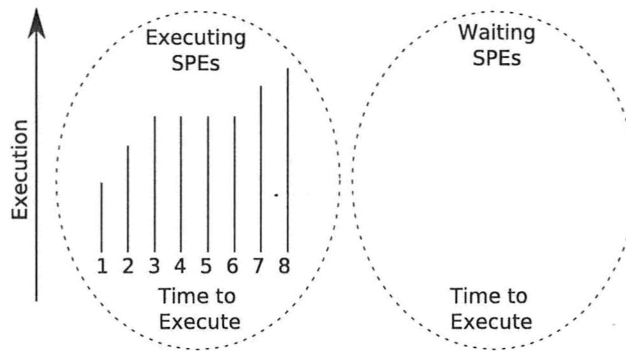

Figure 6.5: All SPEs Executing

The case in which some SPEs are executing and some SPEs are waiting for transfer completion is trickier. Because we always want to simulate the next AVOp set to execute globally, we have to determine if that AVOp will be one belonging to the executing SPEs or one belonging to the waiting SPEs. To do this, we project whether the first waiting SPE to have its transfer complete will have its transfer complete *before* the first executing SPE is set to execute its AVOp. So as a result we simulate the EIB up to the point that *either* one of the transfers for which an SPE is waiting completes, or until the time that the first executing SPE is set to execute. This way, if we do find that an SPE waiting for a transfer has had that transfer complete before the first executing SPE is set to execute, then we know to simulate the AVOp on that waiting SPE (and we conveniently have simulated the EIB up to that point, as desired). If we find that no transfer with an SPE waiting for its completion has completed before the time that the first executing SPE is set to execute, then

we simulate the AVOp on that executing SPE, and again we have conveniently simulated the EIB up to that point. This situation is depicted in Figure 6.6.

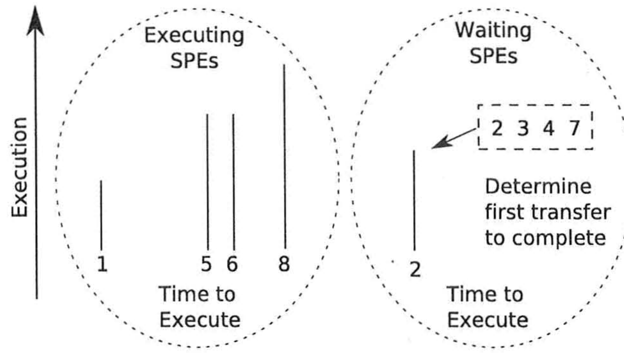

Figure 6.6: Some SPEs Executing, Some SPEs Waiting

The situation that all SPEs are waiting for a transfer to complete is shown in Figure 6.7. In this situation, we simply simulate the EIB until the first transfer for which an SPE is waiting completes. We then simulate the execution of the next AVOp on that SPE, knowing that we have simulated the EIB up to the point that the AVOp was simulated. A legitimate concern with this approach is that the AVOps may contain a deadlock parallel bug, and execution of simulation could itself deadlock. This is a fair point, but because the Verification Tool can be used to filter AVOps to ensure safety before simulation, it is not of particular concern. One small point to make is that it is possible that multiple SPEs could have their transfers complete at the same time, but the AVOp to simulate can just be selected arbitrarily in this situation (as the other SPE's AVOps will be selected for simulation immediately next anyway, with no effect on the end result of the simulation).

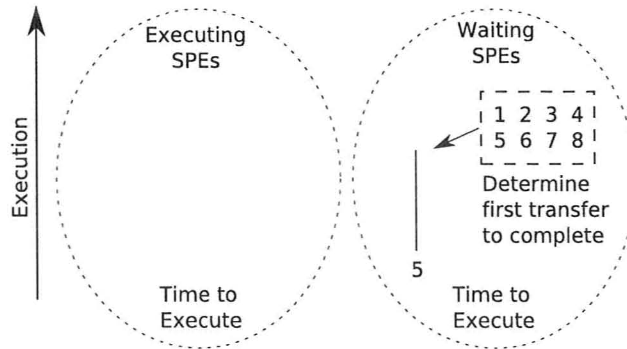

Figure 6.7: All SPEs Waiting

The key overarching algorithm of the Performance Simulator can now be formed by combining the logic that we have gone over for each of these three scenarios. Algorithm 2 presents this algorithm as pseudo code, with the details of simulating individual AVOps and simulating the EIB left for further explanation. Algorithm 2 is the Performance Simulation Algorithm component of Figure 6.4.

Algorithm 2 (Performance Simulation Algorithm) *Note that the simulation is complete when all AVOps have been simulated.*

```
repeat until simulation complete
    if (all SPEs are executing) then
        find an SPE with minimum runtime
        simulate the EIB up to this SPE's runtime
        simulate next AVOp on this SPE
    else if (all SPEs are waiting) then
        simulate EIB until first transfer with an SPE waiting is complete
        simulate AVOp on this SPE
    else if (some SPEs are waiting and some SPEs are executing) then
        find an SPE with minimum runtime X
        simulate EIB until first transfer with an SPE waiting is complete
            OR until the minimum runtime X
        if (EIB simulated to minimum runtime X) then
            simulate AVOp on SPE with minimum runtime X
        else
            simulate AVOp on SPE waiting for completed transfer
```

As mentioned, simulating an AVOp involves adding the runtime cost of executing the AVOp, derived from a function mapping AVOps to runtime costs, to an SPE's simulated runtime. The other aspects of simulating an AVOp's execution depend on which type of AVOp we are simulating. The AVOps are broken down into three different types:

1. Transfer initiation AVOps

2. Wait AVOps

3. Computation AVOps

The transfer initiation AVOps are LoadMemory, StoreMemory, SendData and SendSignal. The wait AVOps are WaitDMA, WaitSignal and WaitData. The only computation AVOp is RunComputation. We can then deal with what is required to simulate each type of AVOp separately.

The simulation of Computation AVOps, that is RunComputation instructions, is very simple. We use the function provided to the simulator by the user which maps computational kernels to runtimes, and we add the resultant runtime to the SPE's current runtime within the simulation. We identify which computational kernel the RunComputation is executing by the RunComputation instruction's Computation argument.

To simulate transfer initiation AVOps, we create the network transfer on the Network Simulator, which simulates the EIB. The Network Simulator provides a transfer creation function to facilitate this. The size of the transfer is determined by the size argument of the AVOp initiating the transfer, with adjustments for packet size (128-bytes) made if the size is not divisible by 128-bytes. Signals do not have a size argument, and are assumed to be a single packet DMA transfer. Transfers on the simulator need to be uniquely identified, so that we can recognize when transfers associated with specific wait AVOps have completed. We use the arguments of the transfer initiating AVOp to build a unique identifier for each transfer. For example in the case of a SendSignal AVOp, the unique identifier would become the signal and target SPE. Note that the same unique identifier may be used many times in a simulation, it is unique for the time it is being used. We know it is unique for the time it is being used, because if it were not, it would imply that the AVOps are attempting a transfer that would be indistinguishable from other transfers. For instance if two signals with the same signalID, are being sent to the same SPE. This type of situation could cause a parallel bug such as a race condition or deadlock, as the waiting SPE would not know which transfer or signal is which. The Verification Tool filters out the possibility of these situations occurring.

The simulation of Wait AVOps first requires us to check with the Network Simulator to see if the associated wait has completed. We use the arguments of the Wait AVOp we are simulating, along with the SPEID that it is being simulated on, to construct the unique identifier for the transfer. We then check a completion set that the Network Simulator maintains, to see if the transfer has completed. If it has, then the Wait AVOp will not stall, and we simply pull the transfer out of the Network Simulator's completion set, and the SPE remains in an executing simulation status. If it has not, then we set the SPE to a waiting for transfer completion status. The next AVOp on this SPE will be simulated when the transfer completes, as explained by Algorithm

2.

Now that we have reviewed how different types of AVOps are simulated, we summarize this logic in the pseudo code of Algorithm 3. Algorithm 3 corresponds to the AVOp Simulation component of Figure 6.4.

Algorithm 3 (AVOp Simulation) *This algorithm implements the simulation of AVOps mentioned in Algorithm 2.*

```
function simulateAVOp (speID, AVOp) :
    find AVOp execution cost using AVOp Execution Cost Function
    add AVOp execution cost to runtime of SPE speID
    if (type(AVOp) == computation) then
        find computation cost using Kernel Cost Function
        add computation cost to runtime of SPE speID
    else if (type(AVOp) == transfer initiation) then
        formulate unique transferID from AVOp arguments
        create network transfer on Network Simulator
    else if (type(AVOp) == wait) then
        formulate transferID from AVOp arguments
        check Network Simulator completion set for transferID
        if (transferID is not in completion set) then
            set SPE speID to waiting
```

The Performance Simulator keeps track of when all transfers complete relative to when the transfer's associated wait instruction is executed. If a wait instruction executes before the transfer completes, that is execution stalls to wait for the transfer to complete, we refer to the difference in time between the two events as *latency.* If a wait instruction executes after the transfer completes, the SPE does not stall, and we refer to the difference in time between the two events as *leadency.* The Performance Simulator can record all occurrences of latency and/or leadency, or it can be made to record one or the other, and at different levels of severity. We record latency because these stalls represent obvious opportunities to improve total execution time. The reason we record leadency occurrences is because if an SPE is receiving data before it needs it, from a shared resource such as main memory that other SPEs are transferring to and from, then it may be contributing to the severity of latency of transfers for other SPEs. By eliminating leadency of transfers associated with shared resources, we could reduce the latency of other transfers associated with the shared resource while the transfer with leadency is occurring. This

could speed up total runtime, at no cost to the SPE for which the leadency was eliminated.

The raw simulation results include the execution time of each SPE, the execution time of the system as a whole (defined as the maximum of all SPE execution times), and the latency and leadency experienced by wait instructions. These results are stored in a data structure, and a report generating function allows for output to a file for more user friendly consumption. These are the Report Result Generator and Simulation Report of Figure 6.4.

Network Simulator

The simulation of the EIB network is a simulator unto itself. What we ultimately wish to simulate is how long it will take transfers to occur. Several different approaches were considered:

- Assume best or worst case transfer times
- Discrete event simulation
- Continuous mathematical model

The problem with best or worst case transfer times, is that though they may be quick to compute, they are wildly inaccurate. For instance, if one outbound transfer from the MIC was occurring to an SPE's MFC, that transfer could be expected to occur at 25.6 GB/s. However due to round robin packet scheduling, if each SPE's MFC was requesting 10 transfers from the MIC (i.e. main memory), then an individual transfer could be expected to occur at 0.32 GB/s. By assuming best or worst case transfer times for every transfer, far too much accuracy would be lost.

Discrete event simulation is appealing due to its potential ability to model EIB performance very accurately. With discrete event simulation, we could model each individual component and protocol of the EIB, such as model ling each BIU and simulating each packet of an individual transfer. The biggest problem with discrete event simulation is that it would be far too expensive to compute. Another problem with discrete event simulation is that we simply don't know enough about the EIB functionality to simulate it at that level of detail, the information that would be required is not publicly available to our knowledge.

We do know enough about the EIB's design to build a continuous mathematical model, that is a model that would analyze network traffic to determine reasonably accurate transfer rates but would not simulate individual

packets or the functioning of individual EIB components. We believe that a continuous mathematical model can provide reasonably accurate simulation times. Knowing that transfer packets are distributed in round robin priority, with a layer of priority for the MIC above all other transfers, is the critical piece of information that allows us to build an accurate model. This is a middle ground that we feel is an appropriate mix of accuracy and speed

As has been discussed in Algorithm 2, the EIB simulator is expected to function in two different modes. It must be able to simulate until the first transfer on a waiting list has completed, and it must be able to simulate until either the first transfer on a waiting list has completed or until a certain time has been reached.

The EIB network simulator thus maintains lists of transfers, a completion list to denote transfers that have been completed, and a waiting list to denote transfers that are waiting for completion. This information is kept in a broader network status data structure, that also includes the current simulation time (relative to the start of simulation), the current simulation mode, and the status of all transfers on the network. This network status data structure can be edited by the performance simulation algorithm, by for instance adding a unique transferID to the waiting list or changing the transfer mode. Once the network status is set up, the network simulation is then 'executed' with an execute function.

The transfer status in the network status data structure refers to whether a transfer is in-flight, that is its packets are being transferred at a given transfer rate over the EIB, or whether it is in start-up, where this is defined to be the SPE-internal transfer start-up behaviour that was discussed in Section 3.4.2. This start-up behaviour is modelled as a fixed cost of 57.1875 nanoseconds, based on literature of such EIB transfer start-up costs[AP07b; AP07a]. We are not the first to model start-up transfer costs of the EIB as a fixed value and the packet transferring phase as a rate of transfer, others have done so to successfully optimize the use of static buffers for DMA transfers on the Cell/B.E.[CS06]. The fact that others have published successful results model ling transfer behaviour in this way suggests our approach is not unreasonable. These researchers did not however extend their model beyond a single SPE, which we must do for our simulation to be useful.

Transfers that are in-flight may effect the transfer rate of other transfers, due to the two-level priority round robin packet scheduling protocol that we assume will provide fair access to network bandwidth. This assumption appears to be sound, as even in tests of EIB performance in situations of heavy contention which may be expected to expose unfair/divergent transfer latencies, vastly divergent transfer times did not result[KPP06]. Transfers that are

not in-flight, that are still being processed internally by the SPE and have not yet been placed in the BIU command queue for a packet transfer as was discussed in Section 3.4.2, are assumed to have no effect on transfer rates because they are not yet requesting access to the command bus or data arbiter to perform a packet transfer. As a result, when simulating EIB transfer rates we must re-calculate transfer rates every time that a start-up transfer changes phase to an in-flight transfer, after the prescribed start-up time has elapsed.

Determining transfer rates of in-flight transfers is done using Algorithm 4. The reasoning behind this algorithm requires further explanation. We know that the EIB has a total bandwidth of 204.8 GB/s, but that we can't simply divide this bandwidth by the total number of transfers to establish transfer rates. This is because specific network resources, such as the MFC of each SPE, also have specific bottlenecks much lower than 204.8 GB/s. For instance, even though 204.8 GB/s of bandwidth may be available, if all transfers on the EIB are inbound or outbound from main memory, then total bandwidth available for these transfers will actually be 25.6 GB/s, the bandwidth of main memory. Similarly, we know that only six simultaneous transfers are possible on two of the four rings in a given direction. As such, if all transfers on the network were going counter-clockwise or all were going clockwise, the bandwidth available would only be 153.6 GB/s. Network resources with bandwidth limitations include inbound and outbound main memory (25.6 GB/s), inbound MIC and MFC (25.6 GB/s), outbound MIC and MFC (25.6 GB/s), ring directions clockwise and counterclockwise (153.6 GB/s), and the command bus (204.8 GB/s). Also, keeping in mind Figure 3.3, ring connections from each BIU to each BIU have a limited transfer rate in each direction of 51.2 GB/s (25.6 GB/s on each of two rings in a direction).

Putting aside higher priority MIC transfers for a moment, the way we distribute network bandwidth is to assume that network resources such as MFCs or data rings, following the fair distribution of a round robin protocol, will distribute bandwidth evenly to all transfers involved with that resource. So if we have ten transfers that are using a specific MFC's outbound bandwidth of 25.6 GB/s, we will assume that they would each receive 2.56 GB/s of bandwidth assuming no other transfers on the network. We determine transfer bandwidth by iteratively finding the most contentiously allocated network resource, that is the network resource that when we assume fair allocation of its remaining bandwidth would result in transfers with the slowest relative transfer rates. We then allocate this bandwidth to these transfers, and repeat this until the bandwidth of all transfers has been determined. When we allocate bandwidth to transfers, potential bandwidth of all components involved in the transfers is reduced by the transfer rate of the transfers involved for

each transfer. For instance, if two outbound MIC transfers, targeting two different MFCs, were being allocated 12.8 GB/s of bandwidth each, then the MIC's outbound bandwidth would be reduced from 25.6 GB/s to zero, and each MFC's inbound bandwidth would be reduced by 12.8 GB/s, the data rings involved would have their bandwidth reduced as well as the EIB itself (i.e. the command bus).

The reason we allocate bandwidth iteratively to the most heavily congested network components is that it allows us to maximize potential transfer bandwidth while respecting the bandwidth limits of all individual components involved. If one were to allocate bandwidth to transfers without taking into account specific network resource limitations, impossible transfer times could result. If one were to allocate transfers to anything but the transfers of the most congested component first, then the transfer rates that would result may not be the maximum possible transfer rates assuming fair allocation. For instance, if we had the situation of two outbound MIC transfers targeting two different SPE's MFCs, then it is clear that the MIC becomes the bottleneck. If we were to assume that either MFC were the bottleneck to its respective transfer, we would allocate 25.6 GB/s of bandwidth to that transfer, but the MIC only has 25.6 GB/s of outbound transfer. This approach results in either inaccurate or impossible transfer times. Allocating bandwidth to the most contentious components is necessary to establish proper transfer times.

Algorithm 4 (EIB Transfer Rate Determination Algorithm) *Two-level priority of round robin scheduling favours MIC transfers, so they are calculated first.*

1. *Distribute main memory bandwidth evenly to MIC transfers*

2. *Identify the network resource bottleneck with the most contention*

3. *Distribute available bandwidth evenly to transfers associated with the bottleneck*

4. *Repeat 2-3 until all transfers have bandwidth allocated*

Another key point about Algorithm 4 is that because the MIC transfers have priority over all others, before we go into the phase of iteratively detecting the most contentious network component and distributing bandwidth, we

distribute the MIC bandwidth evenly to its respective transfers. Main memory, what the MIC provides access to, has a bandwidth of 25.6 GB/s for both inbound and outbound transfers, compared to the MIC's bandwidth of 25.6 GB/s for inbound transfers and 25.6 GB/s for outbound transfers. So though EIB protocols dictate that MIC transfers have highest priority it is actually main memory that effectively becomes the bottleneck that is first accounted for.

Another idea that allows us to use Algorithm 4 is that we are able to determine which network components will be effected by a network transfer using the six-hop maximum transfer path rule discussed in Section 3.4.2. If a transfer between BIUs is less than six-hops away from one another, we can establish the exact path every packet will take, and thus the exact rings that will be used for the transfer. In the case of a transfer whose path is six hops in length, two transfers paths are possible. Though this assumption is admittedly without documentation of its correctness in the literature, we assume that transfer paths will be used according to how congested they are relative to the alternative path, as this seems reasonably intuitive. So for a six hop transfer whose clockwise path intersects that of ten other transfers and whose counter-clockwise path intersects that of five other transfers, we assume it will use its counter-clockwise path 2/3 of the time and its clockwise path 1/3 of the time.

Finally, we know from EIB performance testing that if transfer paths overlap with a heavy amount of transfers occurring, for instance each SPE streaming transfers to and from the SPE six hops away from it[CRDI07], that total EIB bandwidth can go down to roughly 80 GB/s[KPP06]. This has to do with the limitations of the data ring arbiter design[CRDI07], and the fact that transfers can only proceed simultaneously on a ring if their paths do not overlap. We do not have enough information to know how exactly EIB performance degrades, but multiple tests show a lower threshold of performance of about 80 GB/s, for situations that are unrealistic and unlikely to ever occur in real software. Though extraordinarily unlikely that all BIUs would stream data constantly in six hop maximum length paths that overlap, we do still try to provide a mechanism to account for this situation. Before applying Algorithm 4, we lower the total bandwidth of the command bus according to how much overlapping network activity is occurring. An interchangeable function maps network activity, such as the maximum amount of overlapping transfers or the maximum number of components that are connected via transfers with overlapping paths, to a penalty of reduced command bus bandwidth. In its current implementation, a simple linear function is used to degrade performance according to the maximum number of components that are connected

via transfers with overlapping paths.

Now when simulating the EIB in the mode to identify the first waiting list member to complete, we simulate the EIB incrementally. Every time a transfer's phase changes, either from start-up phase to in-flight phase, or from in-flight to complete (where it is then placed in the completion list), we stop and re-calculate transfer rates as these will be effected. A new in-flight transfer could slow down the transfer rate of other transfers, the absence of a now completed transfer could speed up the transfer rate of other transfers. So we simulate the EIB incrementally until a transfer phase change. We can calculate how much of each transfer has occurred in between each phase change by multiplying the transfer rate by the time it took to reach the phase change. We do this until a transfer that is a waiting list member is put into the completion list (i.e. when its phase is the one to change).

In the other mode, where we simulate the EIB to either the first waiting list member to complete, or to some specified time, we simply add a conditional to stop simulation if that given time has been reached before a transfer phase change.

The performance simulation algorithm can check the network status data structure, particularly the completion list and waiting list, to see what transfers have completed. Information with respect to total transfer time is recorded when transfers are sent to the completion list, and is used to establish the amount of stalling due to transfer latency.

The bandwidth of each network component, transfer start-up costs and the command bus performance degradation penalty function, are all very easily modifiable network simulator properties, and so we represent these as the Network Configuration module in Figure 6.4. The logic presented here to simulate EIB performance corresponds to the Network Simulator module of Figure 6.4.

6.4.4 Design Analysis

A few consequences of the Performance Simulator design described in Section 6.4.3 are worth going over. Firstly, the simulation is normally *deterministic*: the result of a simulation given the same AVOps and Kernel Cost Function will be the same every time. No random values are used in calculations, no user input is taken in and no control-flow exists in the AVOps. In Algorithm 2, there may be cases where two AVOps must be simulated at the same point, and though we do simulate one AVOp and then the other, the effect as far as simulation is concerned is no different. It is as if they executed simultaneously as far as the effect on simulation is concerned. This determinism is a result of

meticulously accounting for processor-interaction points, by always simulating the next AVOp set to execute globally across all SPEs relative to where the EIB has already been simulated. The only way that non-determinism could be introduced at present is if the Kernel Cost Function mapped kernels to random runtime values (likely with some statistical probability).

Another consequence of the Performance Simulator design is that we should in fact see a linearly bounded runtime complexity. This is because only one pass through the AVOps is required for simulation, with changes to simulation state being made to execute each AVOp. These changes may not always require the same work computationally, if many transfers are occurring on the EIB network, then more analysis will be required to determine transfer rates. But only so many signals, DMATags, DataTags and local storage space exist. As a result there is only so much state change for each AVOp, and only so much calculation to determine state change. We can expect that the Performance Simulator will have linearly bounded performance as the same communication pattern or scheduling algorithm scales upwards and increases the number of AVOps. However from scheduling algorithm to scheduling algorithm we can expect a variation in simulation runtime for the same number of AVOps due to the different levels of analysis required to perform changes in simulation state.

One fine point about the current Performance Simulator design is that it does not account for the effect of AVOp streams being buffered into each SPE's local store by the Runtime System. Recall from Section 5.3.2 that the Runtime System double buffers in 1024 AVOps (or 16KB worth of data) every 1024 AVOps. In the present Runtime System implementation, this is done implicitly by the Runtime System and not explicitly by AVOps. For now accounting for AVOp stream buffering has been left out of the Performance Simulator, because the future plan is for AVOps to explicitly handle loading in of AVOp buffers for computation using LoadMemory AVOps. If we decide not to do this, it is simply a matter of having the Performance Simulator simulate a 16KB transfer after every 1024 AVOps have executed on an SPE.

A few aspects about network performance are also not accounted for in the model. As was discussed in Section 3.4.2, the MFC's DMAC contains an SPU command queue and a proxy command queue. An attempt to insert a new transfer into a full queue will result in transfer performance degradation. As the SPU command queue is of size 16, and the proxy command queue is of size 8, if transfer requests by an SPE exceed 16 or transfers requested from an SPE exceed 8, the Performance Simulator will not account for this degradation. The reason we chose not to account for this is because model ling on this level would require more computation for something that we are unsure

how to model precisely, for what we perceive to be little gain. We aren't told in the EIB literature exactly how performance will degrade, so it is difficult to model with certainty. One possible model ling of the situation would be that transfers could not enter the in-flight stage until they have entered this queue. We consider it to be an unlikely situation in practice that so many transfers will be occurring simultaneously into or out of a given SPE's LS. Furthermore, the Verification Tool is actually capable of filtering for and flagging the possibility of this situation occurring in the partial execution order.

Another aspect of performance not taken into account by the Performance Simulator is main memory performance, in the sense that 25.6 GB/s of performance is not always possible. Researchers have noted that performance can degrade to 21 GB/s in the case of intermingled reads and writes[CRDI07]. Others have noted main memory performance below 25.6 GB/s in the case of a single SPE issuing batches of DMA gets from main memory[KPP06] We do not wish to model hardware at this level due to the computation this would impose on simulation, and we cannot do so accurately due to our lack of knowledge about the component's functionality, so we do not account for this main memory degradation with our Performance Simulator.

6.4.5 Implementation and Unit Testing

The Performance Simulator was implemented in Haskell[PCK07], as with the Verification Tool and AVOp stream generation functions created thus far. This allows for smooth integration of the various tools, as simulation and verification can be automatically done as part of the process of AVOp stream generation. The Performance Simulator components and Network Simulator components are implemented as separate modules, with monads used due to the high amount of state in each algorithm.

Unit testing is automated with a test harness implemented as a Haskell module. White box testing is done to ensure each AVOp is simulated correctly. Black box testing is done to ensure correct simulation of some situations that could be considered difficult to simulate correctly, for example every SPE signaling every other SPE.

6.4.6 Performance Testing

To demonstrate that the Performance Simulator meets its core objectives we perform several performance tests. The most important motivation to building a simulation tool as part of the CMF was to allow developers to compare the efficiency of different algorithms in a reasonably accurate manner. For this

reason we compare the simulation results of several different types of AVOp streams for performing matrix multiplication. We also perform a test focused on the Network Simulator aspect of the Performance Simulator, comparing our result to a very similar test conducted by other researchers on real hardware.

Matrix Multiplication Comparison Test

The purpose of this test is to demonstrate that the Performance Simulator can identify the difference in performance for different AVOp schedules and computational kernel implementations of the same basic problem. We choose matrix multiplication as it is a problem of much interest to researchers[KAD09; SHW+08; CRDI07; Ear08]; it forms the basis of many other linear algebra problems. We assume that we have two different computational kernels, capable of performing matrix multiplication on 64×64 blocks of data, one implemented with the standard algorithm with $\mathcal{O}(n^3)$ complexity, and an implementation of the Strassen algorithm with $\mathcal{O}(n^{2.807})$ complexity[Str69]. We aim to perform a parallelized block matrix multiplication on matrices of sizes 512×512, 1024×1024 and 2048×2048. Two different types of AVOp schedules are created to accomplish this, one which uses a simple double buffering mechanism to load input data onto the SPEs to perform the block multiplication, and one in which SPEs share input blocks to reduce the bottleneck on main memory.

In the case of both AVOp schedules, the double buffering schedule and the data sharing schedule, a block matrix multiplication is being performed. In both schedules, each 64×64 block $C_{i,j}$ is computed on a single SPE by loading in the appropriate blocks $A_{i,1..n}$ and $B_{1..n,j}$ and accumulating the resulting multiplication in $C_{i,j}$ by way of a matrix-multiplication-add computational kernel. In both schedules, all SPEs will compute the exact same number of $C_{i,j}$ blocks. The input matrix sizes have been fixed to ensure this is the case, for simplicity. In the case of the double buffering schedule, we simply overlap the execution of each computational kernel with loads for the input data blocks required by the next computational kernel.

In the case of the data sharing schedule, we only double buffer in the $A_{i,j}$ blocks required for each computational kernel's execution. For the $B_{i,j}$ blocks we perform some systolic data sharing amongst SPEs to alleviate the bandwidth pressure on main memory. Starting from the first matrix multiplication kernel, we load on the required $B_{i,j}$ block. However for the next *seven* matrix multiplication kernels on each SPE, an SPE gets the input $B_{i,j}$ block from the SPE logically next to it, in a pipe-like fashion. For every eighth matrix multiplication kernel executed on each SPE, we load a $B_{i,j}$ block from

main memory again, before repeating the pattern of sharing $B_{i,j}$ blocks. The appropriate $A_{i,j}$ block must be loaded on to be multiplied against each $B_{i,j}$ block, and the SPEs involved must be computing appropriate $C_{i,j}$ blocks which involve the same $B_{i,j}$ blocks in their computation. A moments thought about the block matrix multiplication algorithm, and how the same column of B is multiplied against n rows of A (assuming square $n \times n$ block matrices), should allow one to see how this sharing is possible. As each of our test matrix sizes is a multiple of $8 \times (64 \times 64)$, we have for simplicity fixed the input sizes such that this sharing can take place throughout the entirety of the schedule, as with our $n \times n$ block matrices n is always divisible by 8, the number of SPEs. The communication pattern is visualized in Figure 6.8. The purpose of this more involved communication pattern is to reduce the main memory bottleneck that was discussed in Section 3.5.1, the more data we can share amongst the SPEs the less we have to draw from main memory and slow down other transfers in the process.

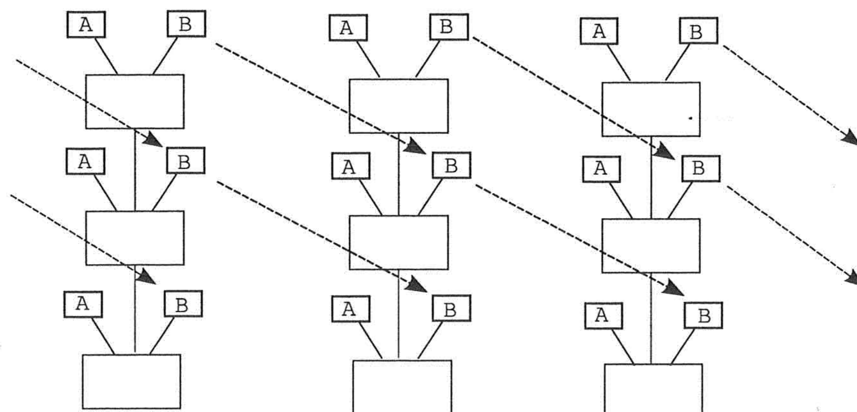

Figure 6.8: Data Sharing Communication Pattern

Two different computational kernels are used with each of these AVOp schedules, one of which implements the traditional $\mathcal{O}(n^3)$ complexity matrix multiplication algorithm (i.e. $C_{i,j} = \sum_{k=1}^{n} A_{i,k}B_{k,j}$). As was discussed in section 3.5.1, assuming we have 64×64 matrices this algorithm would require $64^3 = 262144$ floating point operations. As an SPE has 25.6 GFLOPS of performance, we would expect an implementation of this kernel to execute in $10,240$ nanoseconds.

The second computational kernel simulated is one which implements the Strassen algorithm[Str69], which through a series of operations on submatrices is able to achieve a lower complexity of $\mathcal{O}(n^{2.807})$[GSvdG95]. Assuming

64×64 size matrices, this would lead to approximately $64^{2.807} = 117475$ floating point operations being required to execute a Strassen matrix multiplication kernel. An implementation should then have a runtime of approximately $4,589$ nanoseconds. Keep in mind that we are not attempting to parallelize the Strassen algorithm itself across all SPEs as others have done[Ear08], we are simulating a kernel implementation of it which executes on a single SPE. The numerical stability of the Strassen algorithm is not guaranteed in the general case, the matrices involved must be well-conditioned, and for this reason other researchers have not, thus far, implemented it on the Cell/B.E.[SHW+08].

For each matrix multiplication kernel (cubic complexity and Strassen) and for each AVOp scheduling algorithm (double buffering and data sharing), we performed simulations for matrices of size 512×512, 1024×1024 and 2048×2048. The tests were done 5 times each, so the execution time of the Performance Simulator itself could be averaged (the simulated execution time is the same across all test runs). In Table 6.4.6, we present the simulated execution time measured in nanoseconds, the runtime of the simulator itself measured in seconds, the number of AVOps simulated, the AVOps simulated by the Performance Simulator per second and the nanoseconds of execution simulated per second of simulator runtime.

Cubic Complexity Kernel / Double Buffering AVOp Schedule			
Input Matrix Size	512×512	1024×1024	2048×2048
Simulated Execution Time (Nanoseconds)	715637.2	5492765.0	43239664.0
Runtime of Simulator (Seconds)	0.557	2.777	19.808
Number of AVOps	2688	20992	165888
AVOps Simulated per Second	4825.85	7559.24	8374.80
Nanoseconds Simulated per Second	1284806.46	1977949.23	2182939.42

Cubic Complexity Kernel / Data Sharing AVOp Schedule			
Input Matrix Size	512×512	1024×1024	2048×2048
Simulated Execution Time (Nanoseconds)	757565.6	5859873.0	46198156.0
Runtime of Simulator (Seconds)	1.1878	8.3504	65.217
Number of AVOps	4032	31744	251904

AVOps Simulated per Second	3394.51	3801.49	3862.55
Nanoseconds Simulated per Second	637788.85	701747.58	708375.98
Strassen Kernel / Double Buffering AVOp Schedule			
Input Matrix Size	512×512	1024×1024	2048×2048
Simulated Execution Time (Nanoseconds)	704288.4	5438928.0	42830080.0
Runtime of Simulator (Seconds)	0.591	3.014	22.167
Number of AVOps	2688	20992	165888
AVOps Simulated per Second	4548.22	6964.83	7483.56
Nanoseconds Simulated per Second	1191689.34	1804554.74	1932155.01
Strassen Kernel / Data Sharing AVOp Schedule			
Input Matrix Size	512×512	1024×1024	2048×2048
Simulated Execution Time (Nanoseconds)	418681.9	3152965.8	24536674.0
Runtime of Simulator (Seconds)	1.316	9.732	68.655
Number of AVOps	4032	31744	251904
AVOps Simulated per Second	3562.97	3261.82	3669.13
Nanoseconds Simulated per Second	318050.67	323979.22	357390.93

Table 6.1: Matrix Multiplication Performance Simulation Results

The first thing to notice is that the results seem to conform rather well with expectations of actual runtime. Matrix multiplication, in the traditional case of the cubic complexity algorithm, is a computationally intense problem that should be computation bound on the Cell/B.E.. That is that it should be possible for communication costs to be hidden more or less in their entirely with computation, as was discussed in Section 3.5.1. As such we would expect performance in the range of 204.8 GFLOPS; others in the literature executing

code similar to our cubic complexity double buffered solution, without the overhead of the AVOp and Runtime System abstraction, were able to achieve about 201 GFLOPS[CRDI07]. Our cubic complexity double buffered solution was able to achieve 187.54 GFLOPS in the 512×512 case, 195.48 GLOPS in the 1024×1024 case, and 198.66 GLOPS in the 2048×2048 case. These numbers were obtained by dividing the number of floats required for cubic complexity matrix multiplication by the simulated execution time of these test cases. Though we do not have execution results of our own or in the literature to compare the rest of the test cases to, these numbers seem very reasonable given the overhead of AVOps and the Runtime System and we can begin to state with some confidence that our simulator is reasonably accurate.

The results also seem to accurately simulate the differences in algorithm execution time that we would expect. The execution times of each algorithm are depicted in Figure 6.9, where CC = cubic complexity, S = Strassen, DB = double buffering and DS = data sharing. We observe that in the case of the cubic complexity kernel, the data sharing AVOp schedule is actually *out-performed* by the less complex double buffering AVOp schedule. It is in fact the worst performing solution. This may seem surprising as the data sharing AVOp schedule was meant as an optimization, in that it alleviates the main memory bottleneck. The problem with applying this optimization in the case of the cubic complexity kernel is that the problem was already computation bound. This means optimizations to speed up main memory data transfers, such as the data sharing AVOp schedule, should not provide a performance improvement. What happens instead is that the overhead of executing AVOps to perform these data sharing transfers actually causes performance degradation. This makes sense, as the data sharing algorithm requires 5 AVOps to transfer a block of data from one SPE to another (SendSignal, WaitSignal, SendData, WaitData, WaitDMA) where the double buffering algorithm requires 2 AVOps to transfer a block of data from main memory (LoadMemory, WaitDMA).

Continuing to look at simulated execution times, we observe that the Strassen kernel and double buffering AVOp schedule test case only slightly outperforms the cubic complexity kernel and double buffering AVOp schedule case and performs far worse than the Strassen Kernel and data sharing case. Though the Strassen kernel has a far shorter runtime relative to the cubic complexity kernel, it is not surprising that it only marginally outperforms the cubic complexity kernel in the case of the double buffering AVOp schedule. Recall from Section 3.5.1 that there is *exactly* enough computation time in a cubic complexity matrix multiplication kernel to hide the communication latency. So though we describe the problem as computation bound because communication latency is hidden, it is as close as possible to being a com-

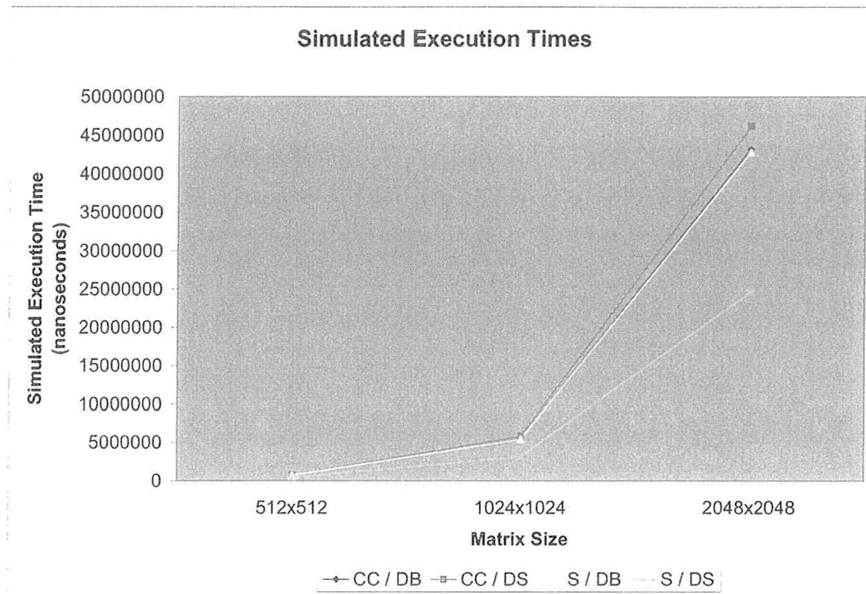

Figure 6.9: Simulated Execution Time

munication bound problem. This is why we do not see a large performance increase by switching from a cubic complexity kernel to a Strassen kernel in the case of double buffering AVOps, the bound becomes communication as soon as we make shorter computation runtime. It does improve marginally however, and this can be explained by the overhead of the AVOp and Runtime System abstraction. Though theoretically computation exactly hides communication costs, in practice the additional expense of interpreting and executing AVOps would make computation costs slightly more expensive than communication costs. As such the large decrease in computation cost provided by the Strassen kernel would be expected to improve runtime by that slight expense, and this is exactly what we observe.

Finally the fact that the case of the Strassen kernel and data sharing AVOps far outperforms all other test cases is completely expected. As discussed the Strassen kernel turns the problem into one that is communication bound, and if we then alleviate the main memory bottleneck by sharing data amongst SPEs we decrease data transfer latencies and improve performance.

The fact that these results are entirely consistent with what is expected when compared to one another provides evidence that the Performance Simulator meets its primary objective to allow a Coconut developer to contrast the effectiveness of different algorithms. What makes these test cases particularly interesting is that given a situation where the bound to program performance is essentially *both* communication and computation, we were able to demonstrate that schedules with optimizations aimed at only one of these bottlenecks to performance had only a small effect on performance, either positively or negatively. However we were also able to demonstrate that by combining the alleviation of the computation bottleneck (Strassen kernel) together with alleviation of the communication bottleneck (data sharing AVOp schedule), large performance gains could be expected.

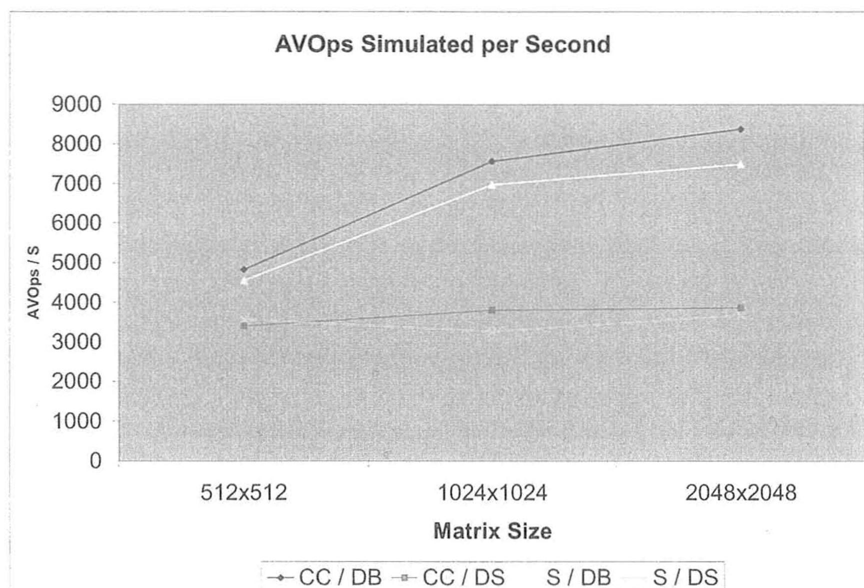

Figure 6.10: AVOps Simulated per Second

Another core objective for the Performance Simulator was that it be linearly bounded to the number of AVOps, as the Verification Tool is, for purposes of scalability to future architectures and so that its efficient runtime may open up opportunities to use the tool more actively in making scheduling decisions. In order to test that the Performance Simulator does in fact have linearly bounded runtimes with respect to AVOps, we observed the runtime of the Performance Simulator itself for each test case. We then divided the

number of AVOps for each test case by the runtime of the Performance Simulator, to give us the number of AVOps simulated per second. If the Performance Simulator is linearly bounded with respect to AVOps than we would expect, as the number of AVOps increases as the matrix size of each test case increases, for the number of AVOps simulated per second to either remain the same or perhaps increase. Observing the graph of AVOps simulated per second in Figure 6.10, this is precisely what we observe. It is not unexpected that different combinations of AVOp streams and computational kernels have different AVOp per second processing speeds; different patterns of AVOp streams will require different amounts of simulation processing as the machine state being simulated is different. That in all cases we observe linearly bounded runtimes as AVOps increase serves as evidence that though runtime may be effected by particular simulation schedules, the Performance Simulator itself is of linearly bounded runtime complexity.

One other observation worth making is with respect to the nanoseconds of execution simulated per second of simulator execution. Noticing that it is difficult to spot any meaningful pattern in this statistic brings up an interesting point. In contrast to other possible simulators where it may be natural and interesting to compare the units of 'real time' it takes to simulate units of 'execution time', it does not make sense to do so with our Performance Simulator. This is because computational kernels embed so much potential simulated execution time, and at such a cheap cost. One could have an AVOp stream that consisted entirely of a single RunComputation instruction, executing a kernel that simulates hours of execution. However because simulating the kernel's execution is just a matter of applying a function that maps kernels to execution times, simulation could be done instantly. As such, for our Performance Simulator tool such a metric comparing simulated execution time to real time is not as valid a metric as it could be for other lower-level (though more powerful) simulators.

Network Simulation Performance Test

As a critical element of the Performance Simulator is successful simulation of the EIB network of the Cell/B.E., a performance test targeted towards the Network Simulator component is prudent. We test the Performance Simulator by checking against performance tests already published in the literature[CRDI07]. The authors of the paper "Cell Broadband Engine Architecture and its first implementation - A performance view" ran a set of experiments in which pairs of SPEs streamed data to one another; the results of this paper were reviewed in Section 3.5.3.

We reproduced this test with AVOps, pairing the same SPEs against the same SPEs and streaming data to one another, and then measuring the performance. Table 6.2 compares the results of our simulation testing to the experimental results on real hardware. Note that as was discussed in Section 6.4.3, before applying Algorithm 4 to determines transfer rates, the total bandwidth of the command bus itself (i.e. the EIB) can be lowered by optionally detecting and penalizing poor network traffic scenarios. We implemented a basic bandwidth linear penalty function which penalizes command bus performance based on how many unique transfer paths are overlapping, and how many network components are involved in overlapping paths. With this, we were able to identify and closely approximate the effects of the poor traffic configuration scenarios that were found in the experiments on real hardware, as can be seen in Table 6.2.

We stress that we are not claiming that we can identify or simulate exactly which network traffic scenarios will cause EIB bandwidth to drop well below peak performance. However given a traffic scenario which we know will result in poor network performance, we can identify it and account for it during simulation.

For the majority of the test cases, where the original experiments produced 197 GB/s of bandwidth, we came very close at 196.5 GB/s. For the outlying case where real hardware experiments produced 187 GB/s, there is not an abundance of overlapping transfers, so the simulation still came in at 196.5 GB/s. This could be accounted for by penalizing specifically this traffic pattern, but as this pattern may have simply been an aberration from the norm, and its lack of performance is not as severe as the other cases, we did not make a specific point to account for it.

Table 6.2: Network Bandwidth Simulation Test

Test Configuration	**EIB Bandwidth [CRDI07]**	**Simulated Bandwidth**
SPE1 ↔ SPE3, SPE5 ↔ SPE7, SPE0 ↔ SPE2, SPE4 ↔ SPE6	186	196.5
SPE0 ↔ SPE4, SPE1 ↔ SPE5, SPE2 ↔ SPE6, SPE3 ↔ SPE7	197	196.5
SPE0 ↔ SPE1, SPE2 ↔ SPE3, SPE4 ↔ SPE5, SPE6 ↔ SPE7	197	196.5
SPE0 ↔ SPE3, SPE1 ↔ SPE2, SPE4 ↔ SPE7, SPE5 ↔ SPE6	197	196.5
SPE0 ↔ SPE7, SPE1 ↔ SPE6, SPE2 ↔ SPE5, SPE3 ↔ SPE4	78	76.8
SPE0 ↔ SPE5, SPE1 ↔ SPE4, SPE2 ↔ SPE7, SPE3 ↔ SPE6	95	93.2
SPE0 ↔ SPE6, SPE1 ↔ SPE7, SPE2 ↔ SPE4, SPE3 ↔ SPE5	197	196.5

Chapter 7

Conclusion and Future Work

In conclusion, this thesis has documented the Performance Simulator tool created for the Coconut Multicore Framework (CMF) which targets the multicore layer of parallelism of the Cell/B.E.. This thesis has also documented the CMF itself and a literature review of relevant parallel computing, the Cell/B.E. and alternative Cell/B.E. program models. Performance testing of the Performance Simulator presented in Section 6.4.6 indicate that it meets its intended objectives of being a scalable, versatile and efficient compile-time method for comparing the performance of different CMF AVOp streams on Coconut developer workstations.

The Performance Simulator is a basically 'complete' tool in that it has met its original design objectives. However some of the more ambitious usages envisioned for the tool should be investigated, in particular its usage in scheduling algorithms themselves. As its runtime has been verified through performance testing to be computationally efficient, these ambitions seem increasingly sound. The Performance Simulator should also be extended to simulate AVOp execution on dual processor Cell/B.E. systems.

Future work on the CMF is required with respect to the generation of AVOp streams from higher-level constructs. Until this work is done, the CMF cannot be considered a fully complete developer to assembly code solution for the Cell/B.E. or other multicore architectures.

The generation of computational kernels from the ILP level of Coconut has held back performance testing of the CMF. However further performance analysis of the Performance Simulator against real results would be very helpful in determining that the tool is in fact simulating accurately. As these computational kernels become available, such performance testing of both the CMF and Performance Simulator must take place.

One feature relatively unique to the CMF is a high assurance of parallel

correctness, with a verified and peer reviewed algorithm having been made publicly available[AK08]. Though other solutions may either make assurances of parallel safety or provide functionality to ensure parallel safety, a published mechanism for how parallel safety is verified is atypical. It would be desirable to build on this success, by continually building new verification mechanisms for new constructs brought into the CMF. For example, verification of any future loop representation of AVOp streams.

We propose on the basis of the literature search conducted of alternative Cell/B.E. frameworks and models, that future research efforts for the CMF should focus on utilizing the correspondence between the virtual machine abstraction and ILP. Firstly because it is unlikely that in the near term a research project with the resources of Coconut can realistically compete as a general multicore solution with commercially developed and industry supported frameworks such as OpenMP, OpenCL and RapidMind. But also because this ILP correspondence is a feature unique to the CMF, and is an area which the Coconut team at present has a relatively high level of expertise. Due purely to its uniqueness, this line of research may also be of particular interest to the parallel computing community at large.

Bibliography

[AAA+02] NR Adiga, G Almasi, GS Almasi, Y Aridor, R Barik, D Beece, R Bellofatto, G Bhanot, R Bickford, M Blumrich, AA Bright, J Brunheroto, C Caşcaval, J Casta nos, W Chan, L Ceze, P Coteus, S Chatterjee, D Chen, G Chiu, TM Cipolla, P Crumley, KM Desai, A Deutsch, T Domany, MB Dombrowa, W Donath, M Eleftheriou, C Erway, J Esch, B Fitch, J Gagliano, A Gara, R Garg, R Germain, ME Giampapa, B Gopalsamy, J Gunnels, M Gupta, F Gustavson, S Hall, RA Haring, D Heidel, P Heidelberger, LM Herger, D Hoenicke, RD Jackson, T Jamal-Eddine, GV Kopcsay, E Krevat, MP Kurhekar, AP Lanzetta, D Lieber, LK Liu, M Lu, M Mendell, A Misra, Y Moatti, L Mok, JE Moreira, BJ Nathanson, M Newton, M Ohmacht, A Oliner, V Pandit, RB Pudota, R Rand, R Regan, B Rubin, A Ruehli, S Rus, RK Sahoo, A Sanomiya, E Schenfeld, M Sharma, E Shmueli, S Singh, P Song, V Srinivasan, BD Steinmacher-Burow, K Strauss, C Surovic, R Swetz, T Takken, RB Tremaine, M Tsao, AR Umamaheshwaran, P Verma, P Vranas, TJC Ward, M Wazlowski, W Barrett, C Engel, B Drehmel, B Hilgart, D Hill, F Kasemkhani, D Krolak, CT Li, T Liebsch, J Marcella, A Muff, A Okomo, M Rouse, A Schram, M Tubbs, G Ulsh, C Wait, J Wittrup, M Bae, K Dockser, L Kissel, MK Seager, JS Vetter, and K Yates. An overview of the BlueGene/L supercomputer. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–22, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[ABC+06] K. Asanovic, R. Bodik, B. Catanzaro, et al. The landscape of parallel computing research: A view from berkeley. Technical report, University of California at Berkeley, 2006. Technical Report No. UCB/EECS-2006-183.

[ACK+04] Christopher Kumar Anand, Jacques Carette, Wolfram Kahl, Cale Gibbard, and Ryan Lortie. Declarative assembler. SQRL Report 20, Software Quality Research Laboratory, McMaster University, October 2004. available from http://sqrl.mcmaster.ca/sqrl_reports.html.

[ACVP06] G. Amit, Y. Caspi, R. Vitale, and A.T. Pinhas. Scalability of multimedia applications on next-generation processors. *Multimedia and Expo, 2006 IEEE International Conference on*, pages 17–20, July 2006.

[ADE+01] Vishal Aslot, Max J. Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. SPEComp: A new benchmark suite for measuring parallel computer performance. In *WOMPAT '01: Proceedings of the International Workshop on OpenMP Applications and Tools*, pages 1–10, London, UK, 2001. Springer-Verlag.

[AG91] Ishfaq Ahmad and Arif Ghafoor. Semi-distributed load balancing for massively parallel multicomputer systems. *IEEE Trans. Softw. Eng.*, 17(10):987–1004, 1991.

[AJ88] R. Agrawal and H.V. Jagadish. Partitioning techniques for large-grained parallelism. *Computers, IEEE Transactions on*, 37(12):1627–1634, Dec 1988.

[AK07a] Christopher Kumar Anand and Wolfram Kahl. Code graph transformations for verifiable generation of SIMD-parallel assembly code. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance, Third Intl. Symp., AGTIVE 2007, Participants' Proceedings*, pages 213–228, 2007.

[AK07b] Christopher Kumar Anand and Wolfram Kahl. A domain-specific language for the generation of optimized SIMD-parallel assembly code. SQRL Report 43, McMaster University, May 2007. available from http://sqrl.mcmaster.ca/sqrl_reports.html.

[AK07c] Christopher Kumar Anand and Wolfram Kahl. Multiloop: Efficient software pipelining for modern hardware. In *CASCON '07: Proc. 2007 Conference of the Center for Advanced Studies on Collaborative Research*, pages 260–263, New York, 2007. ACM.

[AK08] Christopher K. Anand and Wolfram Kahl. Synthesising and verifying multi-core parallelism in categories of nested code graphs. In Michael Alexander and William Gardner, editors, *Process Algebra for Parallel and Distributed Processing*. Chapman & Hall/CRC, 2008.

[AL07] Anant Agarwal and Markus Levy. The kill rule for multicore. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 750–753, New York, NY, USA, 2007. ACM.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[And92] Birger Andersen. Fine-grained parallelism in Ellie. *J. Object Oriented Program.*, 5(3):55–62, 1992.

[AP07a] Thomas William Ainsworth and Timothy Mark Pinkston. On characterizing performance of the cell broadband engine element interconnect bus. In *NOCS '07: Proceedings of the First International Symposium on Networks-on-Chip*, pages 18–29, Washington, DC, USA, 2007. IEEE Computer Society.

[AP07b] T.W. Ainsworth and T.M. Pinkston. Characterizing the Cell EIB on-chip network. *Micro, IEEE*, 27(5):6–14, Sept.-Oct. 2007.

[Bac78] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM*, 21(8):613–641, 1978.

[Bar93] B. Lewis Barnett, III. An ethernet performance simulator for undergraduate networking. *SIGCSE Bull.*, 25(1):145–150, 1993.

[BCG+95] Prithviraj Banerjee, John A. Chandy, Manish Gupta, Eugene W. Hodges IV, John G. Holm, Antonio Lain, Daniel J. Palermo, Shankar Ramaswamy, and Ernesto Su. The paradigm compiler for distributed-memory multicomputers. *Computer*, 28(10):37–47, 1995.

[BCG+06] Brian Bouzas, Robert Cooper, Jon Greene, Michael Pepe, and Myra Jean Prelle. Multicore framework: An API for programming heterogeneous multicore processors. Technical report, Mercury Computer Systems, Inc., 2006.

[BDCW92] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. PROTEUS: a high-performance parallel-architecture simulator. In *SIGMETRICS '92/PERFORMANCE '92: Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 247–248, New York, NY, USA, 1992. ACM.

[BHK07] I. Baumgart, B. Heep, and S. Krause. OverSim: A flexible overlay network simulation framework. In *IEEE Global Internet Symposium, 2007*, pages 79–84, May 2007.

[BKP07] Nikhil Bansal, Tracy Kimbrel, and Kirk Pruhs. Speed scaling to manage energy and temperature. *J. ACM*, 54(1):1–39, 2007.

[BL89] B.L. Bodnar and A.C. Liu. Modeling and performance analysis of single-bus tightly-coupled multiprocessors. *Computers, IEEE Transactions on*, 38(3):464–470, Mar 1989.

[BLK+07] Alfredo Buttari, Piotr Luszczek, Jakub Kurzak, Jack Dongarra, and George Bosilca. SCOP3: A rough guide to scientific computing on the PlayStation 3. version 1.0. Technical Report UT-CS-07-595, Innovative Computing Laboratory, University of Tennessee Knoxville, May 2007.

[BM03] Simonetta Balsamo and Moreno Marzolla. A simulation-based approach to software performance modeling. *SIGSOFT Softw. Eng. Notes*, 28(5):363–366, 2003.

[Bok81] S.H. Bokhari. On the mapping problem. *Computers, IEEE Transactions on*, C-30(3):207–214, March 1981.

[BPBL06] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. CellSs: a programming model for the cell be architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 86, New York, NY, USA, 2006. ACM.

[BPE+04] Patrick Bohrer, James Peterson, Mootaz Elnozahy, Ram Rajamony, Ahmed Gheith, Ron Rockhold, Charles Lefurgy, Hazim

Shafi, Tarun Nakra, Rick Simpson, Evan Speight, Kartik Sudeep, Eric Van Hensbergen, and Lixin Zhang. Mambo: a full system simulator for the PowerPC architecture. *SIGMETRICS Perform. Eval. Rev.*, 31(4):8–12, 2004.

[BTK06] Rainer Butchy, Jie Tao, and Wolfgang Karl. Automatic data locality optimization through self-optimization. In *Self-Organizing Systems*, pages 187–201. Springer Berlin / Heidelberg, 2006.

[BW93] Eric A. Brewer and William E. Weihl. Developing parallel applications using high-performance simulation. In *PADD '93: Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, pages 158–168, New York, NY, USA, 1993. ACM.

[BWSF06] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich. Ray tracing on the Cell Processor. *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 15–23, Sept. 2006.

[CCZ04] David Callahant, Bradford L. Chamberlaint, and Hans P. Zimaj. The Cascade high productivity language. *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, pages 52–60, 2004.

[CDG+93] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. *Supercomputing '93. Proceedings*, pages 262–273, Nov. 1993.

[CDJ+91] R. G. Covington, S. Dwarkadas, J. R. Jump, S. Madala, and J. B. Sinclair. The efficient simulation of parallel computer systems. *International Journal in Computer Simulation*, 1:31–58, 1991.

[CDW94] Jaeyong Choi, Jack J. Dongarra, and David W. Walker. Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. Technical report, University of Tennessee, 1994.

[CG94] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in c++ programs. In *POPL '94: Proceedings of the*

21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 397–408, New York, NY, USA, 1994. ACM.

[CG08] Michael Creel and William L. Goffe. Multi-core CPUs, clusters, and grid computing: A tutorial. *Comput. Econ.*, 32(4):353–382, 2008.

[CH07] P. Conway and B. Hughes. The AMD Opteron Northbridge Architecture. *Micro, IEEE*, 27(2):10–21, March-April 2007.

[Cha99] Xinjie Chang. Network simulations with OPNET. In *Simulation Conference Proceedings, 1999 Winter*, volume 1, pages 307–314 vol.1, 1999.

[CHB07] Eric Cheung, Harry Hsieh, and Felice Balarin. Framework for fast and accurate performance simulation of multiprocessor systems. In *HLDVT '07: Proceedings of the 2007 IEEE International High Level Design Validation and Test Workshop*, pages 21–28, Washington, DC, USA, 2007. IEEE Computer Society.

[CHKW08] Catherine H. Crawford, Paul Henning, Michael Kistler, and Cornell Wright. Accelerating computing with the cell broadband engine processor. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 3–12, New York, NY, USA, 2008. ACM.

[Chr96] D. Christie. Developing the AMD-K5 architecture. *Micro, IEEE*, 16(2):16–27, Apr 1996.

[CHV04] Chen-Yong Cher, Antony L. Hosking, and T. N. Vijaykumar. Software prefetching for mark-sweep garbage collection: hardware analysis and software redesign. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 199–210, New York, NY, USA, 2004. ACM.

[CK88] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2(2):151–169, 1988.

[CKPK90] George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputer performance evaluation and the Perfect Benchmarks. *SIGARCH Comput. Archit. News*, 18(3b):254–266, 1990.

[CMS01] Steve Carr, Jean Mayo, and Ching-Kuang Shene. Race conditions: a case study. *J. Comput. Small Coll.*, 17(1):90–105, 2001.

[CRDI07] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: a performance view. *IBM J. Res. Dev.*, 51(5):559–572, 2007.

[CS06] Tong Chen and Zehra Sura. Optimizing the use of static buffers for dma on a cell chip. In *In The 19th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2006*, 2006.

[CSY90] Ding-Kai Chen, Hong-Men Su, and Pen-Chung Yew. The impact of synchronization and granularity on parallel systems. *SIGARCH Comput. Archit. News*, 18(3a):239–248, 1990.

[Dal09] Jason Dale. Re: Cell be eib, 2009. personal e-mail correspondance.

[DB00] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. *SIGARCH Comput. Archit. News*, 28(5):202–211, 2000.

[Dei05] Steven J. Deitz. *High-level programming language abstractions for advanced and dynamic parallel computations.* PhD thesis, University of Washington, Seattle, WA, USA, 2005. Chair-Snyder, Lawrence.

[DG90] Helen Davis and Stephen R. Goldschmidt. Tango: A multiprocessor simulation and tracing system. Technical report, Stanford University, Stanford, CA, USA, 1990.

[DHN94] Phillip M. Dickens, Philip Heidelberger, and David M. Nicol. A distributed memory LAPSE: parallel simulation of message-passing programs. *SIGSIM Simul. Dig.*, 24(1):32–38, 1994.

[Din89] Anne Dinning. A survey of synchronization methods for parallel computers. *Computer*, 22(7):66–77, 1989.

[DLP03] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: Past, present, and future. concurrency and computation: Practice and experience. *Concurrency and Computation: Practice and Experience*, 15:2003, 2003.

[DM98] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

[DPA99] Murthy Durbhakula, Vijay S. Pai, and Sarita Adve. Improving the accuracy vs. speed tradeoff for simulating shared-memory multiprocessors with ILP processors. In *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, page 23, Washington, DC, USA, 1999. IEEE Computer Society.

[ea95] Z. Vranesic et al. The NUMAchine multiprocessor. Technical report, Technical Report CSRI-324, 1995.

[EA03] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252, New York, NY, USA, 2003. ACM.

[Ear08] Tyler J. Earnest. Strassen's algorithm on the cell broadband engine. *Department of Computer Science and Electrical Engineering, University of Maryland Baltimore County*, 2008. http://www.mc2.umbc.edu/papers.

[Eng00] Ralf S. Engelschall. Portable multithreading - the signal stack trick for user-space thread creation. In *In Proc. USENIX Tech. Conf*, pages 239–250, 2000.

[EVS98] Roger Espasa, Mateo Valero, and James E. Smith. Vector architectures: past, present and future. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 425–432, New York, NY, USA, 1998. ACM.

[FA99] Cormac Flanagan and Martín Abadi. Types for safe locking. In *ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems*, pages 91–108, London, UK, 1999. Springer-Verlag.

[FB92] E.S.T. Fernandes and F.M.B. Barbosa. Effects of building blocks on the performance of super-scalar architectures. *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on*, pages 36–45, 1992.

[Fek09] Alan D. Fekete. Teaching about threading: where and what? *SIGACT News*, 40(1):51–57, 2009.

[FHK+06] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: programming the memory hierarchy. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83, New York, NY, USA, 2006. ACM.

[FR96] Michael J. Flynn and Kevin W. Rudd. Parallel architectures. *ACM Comput. Surv.*, 28(1):67–70, 1996.

[Fre96] Vincent W. Freeh. A comparison of implicit and explicit parallel programming. *J. Parallel Distrib. Comput.*, 34(1):50–65, 1996.

[FRR+07] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Manticore: a heterogeneous parallel language. In *DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 37–44, New York, NY, USA, 2007. ACM.

[FVP06] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. Fft program generation for shared memory: Smp and multicore. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 115, New York, NY, USA, 2006. ACM.

[Gar07] Bryan Gardiner. Astrophysicist replaces supercomputer with eight playstation 3s, 2007. http://www.wired.com/techbiz/it/news/2007/10/ps3_supercomputer/.

[GcSS+05] Ralf Gruber, Marie christine Sawley, Basile Schaeli, Ali Tolou, and Marc Torruella. Towards an intelligent grid scheduling system. In *Parallel Processing and Applied Mathematics, LNCS 3911*, pages 751–757. Springer Verlag, 2005.

[Gen02] Wolfgang Gentzsch. Grid computing: A new technology for the advanced web. In *IWCC '01: Proceedings of the NATO Advanced Research Workshop on Advanced Environments, Tools, and Applications for Cluster Computing-Revised Papers*, pages 1–15, London, UK, 2002. Springer-Verlag.

[GG74] S. W. Galley and Robert P. Goldberg. Software debugging: the virtual machine approach. In *ACM '74: Proceedings of the 1974 annual ACM conference*, pages 395–401, New York, NY, USA, 1974. ACM.

[GG97] José González and Antonio González. Speculative execution via address prediction and data prefetching. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 196–203, New York, NY, USA, 1997. ACM.

[GGC05] Diwaker Gupta, Rob Gardner, and Ludmila Cherkasova. XenMon: QoS monitoring and performance profiling tool. Technical report, HP Laboratories Palo Alto, 2005.

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface.* MIT Press, Cambridge, MA, USA, 1994.

[GNS07] Michael T. Goodrich, Michael Nelson, and Nodari Sitchinava. Sorting in parallel external-memory multicores. Technical report, University of California, Irvine, 2007.

[GP85] D.D. Gajski and Jib-Kwon Peir. Essential issues in multiprocessor systems. *Computer*, 18(6):9–27, 1985.

[GP95] Milind Girkar and Constantine D. Polychronopoulos. Extracting task-level parallelism. *ACM Trans. Program. Lang. Syst.*, 17(4):600–634, 1995.

[Gsc07] Michael Gschwind. The cell broadband engine: exploiting multiple levels of parallelism in a chip multiprocessor. *Int. J. Parallel Program.*, 35(3):233–262, 2007.

[GSvdG95] Brian Grayson, Ajay P Shah, and Robert A. van de Geijn. A high performance parallel strassen implementation. Technical report, University of Texas at Austin, Austin, TX, USA, 1995.

[Gus88] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, 1988.

[GW97] Robert A. Van De Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. Technical report, Concurrency: Practice and Experience, 1997.

[Has03] David A. Hastings. Experience teaching hands-on parallel computing at a small college. *J. Comput. Small Coll.*, 18(3):62–67, 2003.

[HB06] Lorin Hochstein and Victor R. Basili. An empirical study to compare two parallel programming models. In *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 114–114, New York, NY, USA, 2006. ACM.

[HFM88] Debra Hensgen, Raphael Finkel, and Udi Manber. Two algorithms for barrier synchronization. *Int. J. Parallel Program.*, 17(1):1–17, 1988.

[Hil07] J. Hill. Scientific programming on the cell using alf. Technical report, Technical Report from the HPCx Consortium, 2007.

[HL00] Tony Hey and David Lancaster. The development of parkbench and performance prediction. *Int. J. High Perform. Comput. Appl.*, 14(3):205–215, 2000.

[HL08] Maurice Herlihy and Victor Luchangco. Distributed computing and the multicore revolution. *SIGACT News*, 39(1):62–72, 2008.

[HlJT93] Steven Huss-lederman, Elaine M. Jacobson, and Anna Tsao. Comparison of scalable parallel matrix multiplication libraries. In *in Proceedings of the Scalable Parallel Libraries Conference, Starksville, MS*, pages 142–149. Society Press, 1993.

[HMS+09] S. D. Hammond, G. R. Mudalige, J. A. Smith, S. A. Jarvis, J. A. Herdman, and A. Vadgama. WARPP: a toolkit for simulating high-performance parallel scientific codes. In *Simutools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 1–10, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[HNO97] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997.

[Hof06] Hans Peter Hofstee. Real-time supercomputing and technology for games and entertainment. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 199, New York, NY, USA, 2006. ACM.

[HS86] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[HSN+04] A. Halaas, B. Svingen, M. Nedland, P. Saetrom, Jr. Snove, O., and O.R. Birkeland. A recursive MISD architecture for pattern matching. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(7):727–734, July 2004.

[Hu61] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, 1961.

[IBM06] IBM. IBM BladeCenter QS20, 2006. http://www-03.ibm.com/technology/ splash/qs20/pdf/qs20_datasheet.pdf.

[IBM07a] IBM. ALF for Cell BE Programmer's Guide and API Reference, 2007.

[IBM07b] IBM. Software development kit for multi-core acceleration version 3.0., Oct. 2007.

[IBM08a] IBM. IBM BladeCenter QS21, 2008. ftp://ftp.software.ibm.com/common/ ssi/pm/sp/n/bld03006usen/BLD03006USEN.PDF.

[IBM08b] IBM. IBM BladeCenter QS22, 2008. ftp://ftp.software.ibm.com/common/ ssi/pm/sp/n/bld03019usen/BLD03019USEN.PDF.

[IBM08c] IBM. Software development kit for multicore acceleration: Programming tutorial, 2008. Version 3.0.

[IY00] Felix P. Muga II and William Emmanuel S. Yu. A proposed topology for a 192-processor symmetric cluster with a single-switch delay, 2000. Proceedings of the First Philippine Computing Science Congress, Manila, Philippines.

[JB07] C. R. Johns and D. A. Brokenshire. Introduction to the cell broadband engine architecture. *IBM J. Res. Dev.*, 51(5):503–519, 2007.

[JFL98] Minwen Ji, Edward W. Felten, and Kai Li. Performance measurements for multithreaded programs. *SIGMETRICS Perform. Eval. Rev.*, 26(1):161–170, 1998.

[JGMR07] D. Jimenez-Gonzalez, X. Martorell, and A. Ramirez. Performance analysis of cell broadband engine for high memory bandwidth applications. *Performance Analysis of Systems and Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 210–219, April 2007.

[JSK+06] Stephen A. Jarvis, Daniel P. Spooner, Helene N. Lim Choi Keung, Junwei Cao, Subhash Saini, and Graham R. Nudd. Performance prediction and its use in parallel and distributed computing systems. *Future Gener. Comput. Syst.*, 22(7):745–754, 2006.

[KA99] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[KAC06] Wolfram Kahl, Christopher Kumar Anand, and Jacques Carette. Control-flow semantics for assembly-level data-flow graphs. In Wendy MacCaull, Michael Winter, and Ivo Düntsch, editors, *RelMiCS 2005*, volume 3929 of *LNCS*, pages 147–160. Springer, 2006.

[KAD09] Jakub Kurzak, Wesley Alvaro, and Jack Dongarra. Optimizing matrix multiplication for a short-vector simd architecture - cell processor. *Parallel Comput.*, 35(3):138–150, 2009.

[KD09] Jakub Kurzak and Jack Dongarra. QR factorization for the cell broadband engine. *Sci. Program.*, 17(1-2):31–42, 2009.

[KDH+05] J. A. Kahle, N. 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*, 49(4/5):589–604, July/September 2005.

[Kes88] Srinivasan Keshav. REAL: A network simulator. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1988.

[KJS+07] Arun Kumar, Naresh Jayam, Ashok Srinivasan, Ganapathy Senthilkumar, Pallav K. Baruah, Shakti Kapoor, Murali Krishna, and Raghunath Sarma. Feasibility study of MPI implementation on the heterogeneous multi-core cell be architecture. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 55–56, New York, NY, USA, 2007. ACM.

[KMVR90] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory architectures. *SIGPLAN Not.*, 25(3):177–186, 1990.

[Koc07] Ken Koch. Roadrunner system overview, October 2007. http://www.lanl.gov/roadrunner/rrperfassess.shtml.

[KPH+98] K. Keeton, D.A. Patterson, Yong Qian He, R.C. Raphael, and W.E. Baker. Performance characterization of a quad pentium pro SMP using OLTP workloads. *Computer Architecture, 1998. Proceedings. The 25th Annual International Symposium on*, pages 15–26, Jun-1 Jul 1998.

[KPP06] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, 2006.

[Kri01] S. Krishnaprasad. Uses and abuses of amdahl's law. *J. Comput. Small Coll.*, 17(2):288–293, 2001.

[KSK+07] Arun Kumar, Ganapathy Senthilkumar, Murali Krishna, Naresh Jayam, Pallav K. Baruah, Raghunath Sharma, Ashok Srinivasan, and Shakti Kapoor. A buffered-mode MPI implementation for the Cell BE processor. In *ICCS '07: Proceedings of the 7th international conference on Computational Science, Part I*, pages 603–610, Berlin, Heidelberg, 2007. Springer-Verlag.

[Lab09] Standford University Computer Systems Laboratory. Sequoia, 2009. http://www.stanford.edu/group/sequoia.

[LGV+05] R. Lottiaux, P. Gallard, G. Vallee, C. Morin, and B. Boissinot. OpenMosix, OpenSSI and Kerrighed: a comparative study. *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, 2:1016–1023 Vol. 2, May 2005.

[LKN96] Hsueh Lu, Philip N. Klein, and Robert H.B. Netzer. Detecting race conditions in parallel programs that use one semaphore. Technical report, Brown University, Providence, RI, USA, 1996.

[LR97] Daniel Leibholz and Rahul Razdan. The alpha 21264: A 500 mhz out-of-order execution microprocessor. In *COMPCON '97: Proceedings of the 42nd IEEE International Computer Conference*, page 28, Washington, DC, USA, 1997. IEEE Computer Society.

[LRF97] Hyuk-Jae Lee, James P. Robertson, and José A. B. Fortes. Generalized cannon's algorithm for parallel matrix multiplication. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 44–51, New York, NY, USA, 1997. ACM.

[LRW91] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. *SIGOPS Oper. Syst. Rev.*, 25(Special Issue):63–74, 1991.

[Lut96] Mark Lutz. *Programming python*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[Mar07] Grant Martin. *A Power and Energy Perspective on MultiProcessors*. Springer Netherlands, 2007.

[McC06] M. D. McCool. Data-parallel programming on Cell BE and the GPU using the Rapidmind development platform. *GSPx Multicore Applications Conference*, 2006.

[McK04] Sally A. McKee. Reflections on the memory wall. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, page 162, New York, NY, USA, 2004. ACM.

[McM09] Robbie McMahon. OpenCL on the playstation 3. *Loyola University Chicago*, 2009. http://sites.google.com/site/openclps3/project-proposal.

[M.J72] Flynn M.J. Some computer organizations and their effectiveness. *IEEE Trans. on Comp.*, C-21:948–960, 1972.

[MML07] C. Mueller, B. Martin, and A. Lumsdaine. CorePy: High-productivity Cell/B.E. programming, 2007. In Proc. 1st STI/Georgia Tech Workshop on Software and Applications for the Cell/B.E. Processor.

[MMSC99] Berna L. Massingill, Timothy G. Mattson, Beverly A. Sanders, and Timothy G. Mattson Intel Corporation. Patterns for parallel application programs, 1999. Proc. 6th Pattern Languages of Programs Workshop (PLoP99).

[Moi97] Mark Moir. Practical implementations of non-blocking synchronization primitives. In *PODC '97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 219–228, New York, NY, USA, 1997. ACM.

[Mon08] Matthew Monteyne. RapidMind multi-core development platform, 2008. http://www.rapidmind.net/pdfs/WP_RapidMindPlatform.pdf.

[MPV93] Mayan Moudgill, Keshav Pingali, and Stamatis Vassiliadis. Register renaming and dynamic speculation: an alternative approach. In *MICRO 26: Proceedings of the 26th annual international symposium on Microarchitecture*, pages 202–213, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[MRF+00] Shubhendu S. Mukherjee, Steven K. Reinhardt, Babak Falsafi, Mike Litzkow, Mark D. Hill, David A. Wood, Steven Huss-Lederman, and James R. Larus. Wisconsin Wind Tunnel II: A fast, portable parallel architecture simulator. *IEEE Concurrency*, 8(4):12–20, 2000.

[MRR03] Wolfgang Müller, Wolfgang Rosenstiel, and Jürgen Ruf, editors. *SystemC: methodologies and applications.* Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[MS04] Allen D. Malony and Sameer S. Shende. Overhead compensation in performance profiling. In *In Proceedings of the 10th International Euro-Par Conference on Parallel Processing (Euro-Par 04*, pages 119–132. Springer-Verlag, 2004.

[MSL+07] Kent-Andre Mardal, Ola Skavhaug, Glenn T. Lines, Gunnar A. Staff, and Asmund Odegard. Using Python to solve partial dif-

ferential equations. *Computing in Science and Engg.*, 9(3):48–51, 2007.

[MSVV00] Silvia M. Müller, Per Stenström, Mateo Valero, and Stamatis Vassiliadis. Parallel computer architecture. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 537–538, London, UK, 2000. Springer-Verlag.

[MT04] Michael McCool and Stefanus Du Toit. *Metaprogramming GPUs with Sh.* AK Peters Ltd, 2004.

[Mud06] Jayaram Mudigonda. *Addressing the memory bottleneck in packet processing systems.* PhD thesis, University of Texas at Austin, Austin, TX, USA, 2006. Adviser-Vin, Harrick M.

[Mun08] Aaftab Munshi. OpenCL - parallel computing on the GPU and CPU, 2008. http://s08.idav.ucdavis.edu/munshi-opencl.pdf.

[MWHL06] Michael D. McCool, Kevin Wadleigh, Brent Henderson, and Hsin-Ying Lin. Performance evaluation of GPUs using the Rapid-Mind development platform. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 181, New York, NY, USA, 2006. ACM.

[Naf06] S. Naffziger. High-performance processors in a power-limited world. *VLSI Circuits, 2006. Digest of Technical Papers. 2006 Symposium on*, pages 93–97, 0-0 2006.

[NH00] Kazunori Nishihara and Takaai Hiramatsu. Condition variable to synchronize high level communication between processing threads, February 2000.

[NL91] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8):52–60, Aug 1991.

[NM92] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.

[OIS+06] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. MPI Microtask for programming the cell broadband engine processor. *IBM Syst. J.*, 45(1):85–102, 2006.

[ONH+96] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *SIGOPS Oper. Syst. Rev.*, 30(5):2–11, 1996.

[OOS+08] Kevin O'Brien, Kathryn O'Brien, Zehra Sura, Tong Chen, and Tao Zhang. Supporting OpenMP on cell. In *IWOMP '07: Proceedings of the 3rd international workshop on OpenMP*, pages 65–76, Berlin, Heidelberg, 2008. Springer-Verlag.

[PCK07] Gabriele Program Chair-Keller. Haskell '07: Proceedings of the acm sigplan workshop on haskell workshop, 2007.

[PZOL01] S. Parthasarathy, M. J. Zaki, M. Ogihara, and W. Li. Parallel data mining for association rules on shared memory systems. *Knowl. Inf. Syst.*, 3(1):1–29, 2001.

[RD00] Michiel Ronsse and Koen De Bosschere. Non-intrusive on-the-fly data race detection using execution replay, Nov 2000. In Proceedings of Automated and Algorithmic Debugging.

[RPK00] S.K. Raman, V. Pentkovski, and J. Keshava. Implementing streaming SIMD extensions on the Pentium III processor. *Micro, IEEE*, 20(4):47–57, Jul/Aug 2000.

[RSB94] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A convex programming approach for exploiting data and functional parallelism on distributed memory multicomputers. *Parallel Processing, 1994. ICPP 1994. International Conference on*, 2:116–125, Aug. 1994.

[SBG+02] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P.N. Strenski, and P.G. Emma. Optimizing pipelines for power and performance. *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, pages 333–344, 2002.

[Sch09] Scott Schneider. Cellgen, 2009. http://people.cs.vt.edu/ scschnei/cellgen/.

[Sha] Agam Shah. PC World - Intel acquires software company RapidMind. http://www.pcworld.com/ article/170632/intel_acquires _software_company_rapidmind.html.

[Shi07] Tomonori Shindou. How far has cell dna been passed on? interview with toshiba spursengine developer, October 2007. http://techon.nikkeibp.co.jp/english/NEWS_EN/20071017/140756/.

[SHW+08] T. Schneider, T. Hoefler, S. Wunderlich, T. Mehlan, and W. Rehm. An optimized ZGEMM implementation for the Cell BE. In *Proceedings of the 9th Workshop on Parallel Systems and Algorithms (PASA)*, Feb. 2008.

[SKGF08] Sriram Swaminarayan, Kai Kadau, Timothy C. Germann, and Gordon C. Fossum. 369 tflop/s molecular dynamics simulations on the Roadrunner general-purpose heterogeneous supercomputer. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–10, Piscataway, NJ, USA, 2008. IEEE Press.

[SKP06] Sayantan Sur, Matthew J. Koop, and Dhabaleswar K. Panda. High-performance and scalable mpi over infiniband with reduced memory usage: an in-depth performance analysis. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 105, New York, NY, USA, 2006. ACM.

[SKST08] Vipin Sachdeva, Michael Kistler, Evan Speight, and Tzy-Hwa Kathy Tzeng. Exploring the viability of the cell broadband engine for bioinformatics applications. *Parallel Computing*, 34(11):616 – 626, 2008. High-Performance Computational Biology.

[Smi88] J. E. Smith. Characterizing computer performance with a single number. *Commun. ACM*, 31(10):1202–1206, 1988.

[Son09] Sony. Unit sales of hardware (since april 2006), 2009. http://www.scei.co.jp/corporate/data/bizdataps3_sale_e.html.

[SP88] J.E. Smith and A.R. Pleszkun. Implementing precise interrupts in pipelined processors. *Computers, IEEE Transactions on*, 37(5):562–573, May 1988.

[Spr07] J. Spray. Lattice QCD on the cell processor. *University of Edinburgh*, 2007. http://www2.epcc.ed.ac.uk/msc/dissertations/dissertations-0607/8991210-27b-d07rep1.2.pdf.

[SS95a] Brian Schmidt and V. S. Sunderam. Empirical analysis of overheads in cluster environments. *Concurrency: Practice and Experience*, 6:1–32, 1995.

[SS95b] J.E. Smith and G.S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609–1624, Dec 1995.

[SSOG93] Jaspal Subhlok, James M. Stichnoth, David R. O'Hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. *SIGPLAN Not.*, 28(7):13–22, 1993.

[SSS+04] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Trans. Archit. Code Optim.*, 1(1):94–125, 2004.

[Sto77] H.S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *Software Engineering, IEEE Transactions on*, SE-3(1):85–93, Jan. 1977.

[Str69] V. Strassen. Gaussian elimination is not optimal. *Numerical Mathematics*, 13:354–356, 1969.

[SWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. *SIGARCH Comput. Archit. News*, 20(1):5–44, 1992.

[SYR+08] Scott Schneider, Jae-Seung Yeom, Benjamin Rose, John C. Linford, Adrian Sandu, and Dimitrios S. Nikolopoulos. A comparison of programming models for multiprocessors with explicitly managed memory hierarchies. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 131–140, New York, NY, USA, 2008. ACM.

[Tha06] Wolfgang Thaller. Explicitly staged software pipelining. Master's thesis, McMaster University, Department of Computing and Software, 2006. http://sqrl.mcmaster.ca/~anand/papers/ThallerMScExSSP.pdf.

[Top08] Top500.org. Jaguar chases Roadrunner, but cant grab top spot on latest list of worlds top500 supercomputers, 2008. http://www.top500.org/lists/2008/11/press-release.

[UK88] A. K. Uht and J. F. Kolen. On the combination of hardware and software concurrency extraction methods. *SIGMICRO Newsl.*, 19(1-2):53–57, 1988.

[vdS93] Aad J. van der Steen. The benchmark of the EuroBen group. In *Computer benchmarks*, pages 165–175. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 1993.

[VKJ+07] M. K. Velamati1, A. Kumar, N. Jayam, N. G. Senthilkumar, P. K. Baruah, R. Sharma, S. Kapoor, and A. Srinivasan. Optimization of collective communication in intra-cell MPI, 2007.

[VRCG+99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

[WMZ+08] Qianxiang Wang, Na Meng, Zhiyi Zhou, Jinhui Li, and Hong Mei. Towards SOA-based code defect analysis. In *SOSE '08: Proceedings of the 2008 IEEE International Symposium on Service-Oriented System Engineering*, pages 269–274, Washington, DC, USA, 2008. IEEE Computer Society.

[Wol04] Wayne Wolf. The future of multiprocessor systems-on-chips. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 681–685, New York, NY, USA, 2004. ACM.

[WSO+07] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. Scientific computing kernels on the cell processor. *Int. J. Parallel Program.*, 35(3):263–298, 2007.

[YKJ02] Yi-ran Sun, Shashi Kumar, and Axel Jantsch. Simulation and evaluation of a network on chip architecture using. In *Proc., IEEE NorChip Conference*, 2002.

[ZHNB06] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar RAM-CPU cache compression. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 59, Washington, DC, USA, 2006. IEEE Computer Society.

[ZR04] Jingren Zhou and Kenneth A. Ross. Buffering databse operations for enhanced instruction cache performance. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 191–202, New York, NY, USA, 2004. ACM.

[ZSLW92] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous distributed shared memory. *IEEE Trans. Parallel Distrib. Syst.*, 3(5):540–554, 1992.

5649 76