University of Derby

A Data Dependency Recovery System for a Heterogeneous Multicore Processor

Haresh Singh Kainth

Doctor of Philosophy 2014

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Hardware and Conventional CMP versus the Cell . . . 6
- 1.3 Aims and Objectives of this Research . . . 9
- 1.4 Contributions . . . 9
- 1.5 Overview and Structure of this Thesis . . . 12

2 The Research Methodology **13**
- 2.1 Approach Taken for this Research . . . 14
  - 2.1.1 Quantitative and or Qualitative . . . 15
  - 2.1.2 Applications Used for the Research . . . 16
  - 2.1.3 Measurements . . . 16
  - 2.1.4 Value of Research . . . 17
  - 2.1.5 Equipment . . . 19
  - 2.1.6 Limitations . . . 19
- 2.2 Summary . . . 20

3 Thread-Level Speculation Theory and Application **21**
- 3.1 Execution Model . . . 22
- 3.2 Dependencies and Data Hazards . . . 25
- 3.3 Writeback Invalidation-Based Cache Coherence . . . 27
- 3.4 Thread-Level Speculation Schemes . . . 29
  - 3.4.1 Loop-Based Speculative Execution . . . 33
  - 3.4.2 Thread Spawning Policies . . . 34
  - 3.4.3 Manual Parallelisation . . . 35
  - 3.4.4 Hardware Support for Thread-Level Speculation . . . 36
  - 3.4.5 Compiler Support for Thread-Level Speculation . . . 41
  - 3.4.6 Speculation Types and Predication Techniques . . . 46

3.5 Summary . . . 48

4 Computer Architecture and the IBM Cell Broadband Engine 51
4.1 Instruction-Level, Data-Level and Thread-Level Parallelism . . 53
4.2 Automating and Manual Parallelism . . . 55
4.3 Microprocessor Architecture . . . 57
4.3.1 Amdahl's Law . . . 60
4.4 Synchronisation . . . 62
4.5 IBM Cell Broadband Engine . . . 64
4.5.1 PowerPC Processor Element . . . 68
4.5.2 Synergistic Processor Element . . . 70
4.5.3 Local Storage . . . 78
4.5.4 Element Interconnect Bus . . . 80
4.5.5 Memory Flow Controller . . . 81
4.5.6 Direct Memory Access . . . 82
4.5.7 Signal and Mailboxes . . . 84
4.5.8 Software Cache and Memory . . . 84
4.6 Summary . . . 86

5 Introducing the Lyuba-API Framework 88
5.1 Lyuba Framework . . . 88
5.2 L-API PPE Kernel . . . 90
5.2.1 Threads on the PPE . . . 91
5.2.2 Element State Containment . . . 93
5.2.3 Hardware Interfaces for Communication and Synchronisation . . . 94
5.2.4 Callback Functions . . . 96
5.2.5 Calculating Data Load and Store . . . 96
5.3 L-API SPE Kernel . . . 99
5.3.1 L-API Load and Store Constructs . . . 101
5.3.2 Mapping Array Segments . . . 105
5.4 Violation Detection and Resolution . . . 106
5.5 Worked Example . . . 108
5.5.1 `Main.c` on the PPE . . . 111
5.5.2 `fft.h` on the SPE . . . 112
5.5.3 L-API use of Low-Level Cell API Calls . . . 114
5.6 Summary . . . 119

6 Results and Analysis **121**
6.1 Results . . . 124
6.1.1 Loop Coverage Analysis . . . 129
6.1.2 Sparse Matrix Multiply Application . . . 129
6.1.3 Jacobi Successive Over-Relaxation Application . . . 134
6.1.4 Dense LU Matrix Factorisation Application . . . 142
6.1.5 Array 2D Copy Application . . . 144
6.1.6 Fast Fourier Transform Application . . . 149
6.2 Comparative Analysis . . . 158
6.3 Summary . . . 160

7 Conclusion **162**
7.1 Findings . . . 163
7.2 Contributions . . . 164
7.2.1 The Heterogeneous Approach . . . 165
7.2.2 A Unified Support . . . 166
7.2.3 A Comprehensive Evaluation of the L-API Framework 167
7.3 Hindrances to the L-API Framework . . . 168
7.4 Future Work . . . 171

References **173**

Appendices **193**

A Emphasis on common function calls with description **194**

B L-API PPE Code **197**
B.1 `common.hpp` . . . 197
B.2 `kernel_element_manager.hpp` . . . 210
B.3 `kernel_measurement_reading.hpp` . . . 221
B.4 `kernel_system_headers.hpp` . . . 225
B.5 `kernel.hpp` . . . 226
B.6 `main.cpp` . . . 285
B.7 `random.cpp` . . . 292

C L-API SPE Code **298**
C.1 `main.c` . . . 298
C.2 `kernel.h` . . . 299
C.3 `array.h` . . . 361

C.4 `fft.h` 362
C.5 `lu.h` 367
C.6 `sor.h` 371
C.7 `sparse.h` 372

D Additional Processor Results **375**

E Single SPE Processor Results without L-API **394**

List of Figures

1.1 Simple thread-level speculation - (A) Sequential program (B) Sequential flow is decomposed into two epochs. 4

3.1 Classical TLS execution model. (A) illustrates epoches executing with no violation. (B) illustrates multiple epoches with a violation occuring on epoch (E2), this results in violation recovery. 22
3.2 Epoch relationship model. 23
3.3 Committing results back to main memory. (A) illustrates the controller threads (CT) and their unique identifier (commit-back priority). (B) illustrates two important components; the controller queue and main memory. The queue shows the CTs and their current ordering. Before CT-3 commits back to main memory, the logically earlier CTS must commit their results back to main memory, whilst respecting the original sequential program semantics. 24

4.1 Cell Broadband Engine block diagram [IBM, 2007a,IBM, 1994]. Note: The EIB consists of four 16-byte-wide data rings: two running clockwise. Each ring potentially allows up to three concurrent data transfers, provided their paths do not overlap [Scarpino, 2008] 67
4.2 PowerPC Processor Element Block Diagram. (A) illustrates a simple overview of an PPE. (B) shows the memory subsystems on the PPE. 69
4.3 Synergistic Processing Element. (A) illustrates a simple overview of an SPE. (B) shows the odd and even pipeline on the SPE. . 72
4.4 Internal organisation of an SPE 76

5.1 PPE and SPE kernel overview. (A) shows the L-API kernel components on the PPE. (B) shows the L-API kernel components on the SPE. 89
5.2 Parallelising For loop across all SPEs. 90
5.3 Inner loop synchronisation. 103
5.4 SPE LS memory segmentation. 104
5.5 Global shared array across multiple SPEs. 105
5.6 Flowchart showing the different states for the violation analyser function. 106

6.1 Sparse matrix multiply application SPE execution time using a small dataset. Mean results after ten runs. 131
6.2 L-API PPE kernel functions for sparse matrix multiply application SPE execution time using a small dataset. Mean results after ten runs. 132
6.3 Sparse matrix multiply application SPE execution time using a large dataset. Mean results after ten runs. 133
6.4 L-API PPE kernel functions for sparse matrix multiply application SPE execution time using a large dataset. Mean results after ten runs. 133
6.5 Jacobi successive over-relaxation SPE execution time using a small dataset. 136
6.6 L-API PPE kernel functions for Jacobi Successive Over-Relaxation Application SPE Execution Time Using a Small Dataset. Mean results after ten runs. 137
6.7 Jacobi successive over-relaxation SPE execution time using a large dataset. 137
6.8 L-API PPE kernel functions for Jacobi successive over-relaxation application SPE execution time using a large dataset. Mean results after ten runs. 138
6.9 Dense LU matrix factorisation SPE execution time using a small dataset. 142
6.10 L-API PPE kernel functions for dense LU matrix factorisation application SPE execution time using a small dataset. Mean results after ten runs. 143
6.11 Dense LU matrix factorisation SPE execution time using a large dataset. 143

6.12 L-API PPE kernel functions for dense LU matrix factorisation Application SPE execution time using a small dataset. Mean results after ten runs. 144
6.13 Array 2D copy SPE execution time using a small dataset. . . . 146
6.14 L-API PPE kernel functions for array 2D copy application SPE execution time using a small dataset. Mean results after ten runs. 147
6.15 Array 2D copy SPE execution time using a medium dataset. . 147
6.16 L-API PPE kernel functions for array 2D Copy application SPE execution time using a medium dataset. Mean results after ten runs. 148
6.17 Array 2D copy SPE execution time using a large dataset. . . . 148
6.18 L-API PPE kernel functions for array 2D copy application SPE execution time using a large dataset. Mean results after ten runs. 149
6.19 FFT application SPE execution time using a small dataset. . . 155
6.20 L-API PPE kernel functions for FFT application SPE execution time using a small dataset. Mean results after ten runs. . 156
6.21 FFT application SPE execution time using a large dataset. . . 156
6.22 L-API PPE kernel functions for FFT application SPE Execution time using a large dataset. Mean results after ten runs. . 157

D.1 CPI results for `SPR_SM` application with L-API transformations.376
D.2 CPI results for `SPR_LG` application with L-API transformations.376
D.3 CPI results for `SOR_SM` application with L-API transformations.376
D.4 CPI results for `SOR_LG` application with L-API transformations.377
D.5 CPI results for `LUCPYMX_SM` application with L-API transformations. 377
D.6 CPI results for `LUCPYMX_LG` application with L-API transformations. 377
D.7 CPI results for `ARR_SM` application with L-API transformations.378
D.8 CPI results for `ARR_MM` application with L-API transformations 378
D.9 CPI results for `ARR_LG` application with L-API transformations.378
D.10 CPI results for `FFT_SM` application with L-API transformations.379
D.11 CPI results for `FFT_LG` application with L-API transformations.379
D.12 `SPR_SM` Instructions Usage Percentage per SPE pipeline unit. . 381
D.13 `SPR_LG` Instructions Usage Percentage per SPE pipeline unit. . 382
D.14 `SOR_SM` Instructions Usage Percentage per SPE pipeline unit. . 383

D.15 SOR_LG Instructions Usage Percentage per SPE pipeline unit. . 384
D.16 LUCPYMX_SM Instructions Usage Percentage per SPE pipeline unit. 385
D.17 LUCPYMX_SM Instructions Usage Percentage per SPE pipeline unit. 386
D.18 FFT_SM Instructions Usage Percentage per SPE pipeline unit. . 387
D.19 FFT_LG Instructions Usage Percentage per SPE pipeline unit. . 388
D.20 ARR_SM Instructions Usage Percentage per SPE pipeline unit. . 389
D.21 ARR_MM Instructions Usage Percentage per SPE pipeline unit. . 390
D.22 ARR_LG Instructions Usage Percentage per SPE pipeline unit. . 391

E.1 Single SPE Results for all applications without L-API transformations. 395

List of Tables

2.1 Hardware environment. 18
2.2 Software environment. 19

3.1 Hazards. 25
3.2 Common Dependencies. 26

4.1 Cell processor architecture overview. 63
4.2 PowerPC Standard Version 2.02. 68
4.3 SPU even and odd functional units. 73
4.4 MFC component description. 74

5.1 SPE main interface constructs prototypes. 98
5.2 Violation detection state labels. 107

6.1 Benchmark description. 123
6.2 Loop execution coverage (function names). 127
6.3 Loop execution coverage (average). 128
6.4 Completion state matrix for SciMark application without L-API transformations. Application with a Yes status for completion represents a successful execution. Partial status represents the incorrect result generated from execution. Failed status represents a complete failure of the application, execution failure. 158
6.5 L-API performance improvement matrix. 159

A.1 Violation, recovery, load and store function prototypes. 195
A.2 Request analyser callback function state Labels. 196

Abstract

Multicore processors often increase the performance of applications. However, with their deeper pipelining, they have proven increasingly difficult to improve. In an attempt to deliver enhanced performance at lower power requirements, semiconductor microprocessor manufacturers have progressively utilised chip-multicore processors.

Existing research has utilised a very common technique known as thread-level speculation. This technique attempts to compute results before the actual result is known. However, thread-level speculation impacts operation latency, circuit timing, confounds data cache behaviour and code generation in the compiler.

We describe an software framework codenamed Lyuba that handles low-level data hazards and automatically recovers the application from data hazards without programmer and speculation intervention for an asymmetric chip-multicore processor.

The problem of determining correct execution of multiple threads when data hazards occur on conventional symmetrical chip-multicore processors is a significant and on-going challenge. However, there has been very little focus on the use of asymmetrical (heterogeneous) processors with applications that have complex data dependencies.

The purpose of this thesis is to: (i) define the development of a software framework for an asymmetric (heterogeneous) chip-multicore processor; (ii) present an optimal software control of hardware for distributed processing and recovery from violations; (iii) provides performance results of five applications using three datasets. Applications with a small dataset showed an improvement of 17% and a larger dataset showed an improvement of 16% giving overall 11% improvement in performance.

Acknowledgements

I would like to thank the many people who have provided me with the support and encouragement to complete this thesis and my Ph.D. To all my family members, friends and associates I say thank you.

I would like to thank my father Joginder Singh Kainth, my mother Gurdev Kaur Kainth, my brother Omraj Singh Kainth and my sisters Anita & Jeevan Kaur Kainth and our family Akita (Simba) for there love and support.

To my loving Lyuba (Lyubov), you are my heart and my soul who gave me the energy and support when I needed it most.

I would like to thank all my family members including my dear and close brothers Karan Bubber, Kuldipak Marwaha and James Anderson Bell. You fine gentlemen have been my rock and gave me support 24 hours of every day. But of course, I could not have come this far and could not have completed this work without my profound appreciation to the many advisor's who have helped steer my path and I want to express to them my deepest gratitude to Mr Clifton Jones, Professor Richard Hill, Dr Victoria Carpenter, Dr Stuart Berry and Dr Ovidiu Bagdasar who led me to today's achievements.

I feel very fortunate to have done my doctorate under an advisor whom I hold in such high regard Mr Clifton Jones. I was amazingly fortunate to have an advisor who gave me the inspiration to solve different research problems and at the same time the guidance to recover when my steps faltered. Finally, I thank past and present doctoral students and all members of the research team at the University of Derby.

This dissertation is dedicated to my love Lyubov, family and friends.

Abbreviations

API Application Programming Interface

ARRLG Array 2D Copy Large Dataset

ARRMM Array 2D Copy Medium Dataset

ARRSM Array 2D Copy Small Dataset

AU Atomic Unit

BIC Bus Interface Control

CB Copy Back

CBE Cell Broadband Engine

CBFR Call Back Function Return

CMP Chip-Multicore Processor

CPU Central Processor Unit

CS Control Speculation

CTP Compute-Transfer Parallelism

DDRS Data Dependency Recovery System

DDS Data Dependence Speculation

DLP Data Level Parallelism

DMA Direct Memory Access

DMAE Direct Memory Access Engine

DMAQ Direct Memory Access Queue

DOVAC Data Output Violation Analysis – Creation

DOVAK Data Output Violation Analysis – Kill

DR Deregister

DSP Digital Signal Processor

DVS Data Value Speculation

EIB Element Interface Bus

EMC Element Manager Check

FFT Fast Fourier Transform

FFTLG Fast Fourier Transform Large Dataset

FFTSM Fast Fourier Transform Small Dataset

GPU Graphics Processing Unit

IO Input/Output

IE Insert Element

ILP Instruction Level Parallelism

IPC Instructions Per Cycle

L-API Lyuba-Application Programming Interface

LE Load Element

LS Local Storage

LUCPYMXLG Dense LU Matrix Factorisation Large Dataset

LUCPYMXSM Dense LU Matrix Factorisation Small Dataset

MAESC or M Monitor Array Element State Container

MAS Monitor Array State

MDT Memory Disambiguation Table

MFC Memory Flow Controller

MMIO Memory-Mapped Input/Output

MMU Memory Management Unit

NOC Network On-Chip

ORC Open Research Compiler

OS Operating System

PE Processor Element

PLP Process Level Parallelism

PPE PowerPC Processing Element

R Register

RA Request Analysis

RAM Random Access Memory

RAW Read After Write

RISC Reduced Instructions Set Computer

RMU Resource Management Unit

SCMP Speculative Chip-Multicore Processor

SCN SPU Control Unit

SFP SPU Floating-Point Unit

SFS SPU Odd Fixed-Point Unit

SFX SPU Even Fixed-Point Unit

SIMD Single Instruction Multiple Data

SLS SPU Load And Store Unit

SMP Symmetrical Multicore Processors

SORLG Jacobi Successive Over-Relaxation Large Dataset

SORSM Jacobi Successive Over-Relaxation Small Dataset

SPE Synergistic Processor Element

SPRLG Sparse Matrix Multiply Large Dataset

SPRSM Sparse Matrix Multiply Small Dataset

SPU Synergisitic Processor Unit

SR Service Request Load/Store

SS Synchronising Scoreboard

SUIF Stanford University Intermediate Representation

TLP Thread Level Parallelism

TT Thread Type

UAR Update And Reinsert

VA Violation Analyser

VM Virtual Machine

WAW Write After Write

Chapter 1

Introduction

1.1 Background

Complementary metal oxide semiconductor (CMOS) dies continue to shrink in dimensions simultaneously; traditional microarchitectural processors delivering limited performance increases with deeper pipelining have become increasingly expensive, despite their diminished performance improvements.

This is partly due to increased power consumption, that limited the use of uniprocessors and led to the use of semiconductors chip-multicore processors (CMPs) to deliver increased performance at lower power requirements.

Single CMPs are now commonplace in most modern desktop computers, supercomputers, workstations and games consoles [Akkary et al., 2008]. Chip-multicore processors developed by major processor manufacturers include the Intel© Xeon™, AMD FX©, Sun SPARC© and IBM Cell Broadband Engine©.

Improving workload throughput is relatively straightforward using CMPs because modern workloads tend to have a large degree of concurrency. CMP architectures provide multiple threads of execution, allow non-stalled threads to continue execution and determine independent memory access. The latter are serviced independently of stalled threads, whilst simultaneously, improv-

ing the use of limited resources and the off-chip bandwidth.

Refining the performance of a sequential program is predominantly achieved by extracting parallelism via threads.

Extracting parallelism from a sequential code stream can be accomplished by many existing mechanisms such as instruction-level parallelism (ILP), process-level parallelism (PLP), data-level parallelism (DLP) and thread-level parallelism (TLP). TLP can optimise the workloads through time-slicing scheduling.

Multitasking operating systems (OS) can further assist parallelism using the OS scheduler. The scheduler is able to use algorithms such as PLP which extracts parallelism from workloads. The PLP is able to parse high-level workloads (referred to as process) and inherently executes separate and possibly unrelated instructions (contained in a process) on hardware with multiple processing cores. This process is able to reduce the processing latency while effectively utilising available microprocessor computing resources. This approach is quite sufficient for workloads that do not exhibit complex interdependencies.

However, as hardware increases in complexity, it has now become possible to extract fine grain parallelism such as ILP. ILP performs low-level parallelism, whereby the processor is simultaneously loading, decoding, executing and writing back the instruction. This process is typically referred as pipelining. ILP implicitly exploits parallel operations, such as loops or straight-line code segments. This complex procedure is implemented in hardware and intentionally hidden from the programmer. However, ILP has been exploited to its fullest extent, and attempting to extract further ILP would result in overly complex designs [Rul et al., 2007].

Threads are typically small sequences of programmed instructions that are executed either independently or collectively. When threads begin to exhibit dependencies (including complex synchronisation and communication issues) with upon other, the complexity of the program also changes. Manag-

ing this complexity has led to a considerable volume of research (see Chapter 3) that considers software[1] and hardware. In particular, hardware solutions include the use of conventional symmetrical CMPs and symmetrical multicore processors (SMPs) [Hammond et al., 2000]. Such processors contain multiple processing cores that are based on a single silicon chip.[2]

In a conventional multicore processor, the processing elements are generally identical to one another and each core is efficiently designed to execute sequential binary. When sequential binaries contain threads that share data, the processor suffers locks and latency issues: before a thread is executed on a core, the processor must set up the core's registers, load the data and instructions and execute the thread's program. If a thread's instruction exhibits a dependency, the core is triggered to stop its current execution, store the current context (context image), handle the interrupt and finally restore the previous context; this process is called context switching. Context switching imposes latency and reduces the overall performance of the computation [David et al., 2007]. Latency is reduced if dependent threads are within the same stack space.

Major CMP manufacturers impose a restrictive design, whereby the cores on the same silicon chip cannot directly and efficiently communicate with one another. All communication is channelled through an on-chip bus that is external to the cores. This increases the contention on the bus and reduces the available bandwidth[3]. Until recently, symmetrical processors were considered the mainstream CMPs. High performance computing has become more dependent on commodity computing (i.e. conventional processors) and the increased use of cluster based parallel machines [Kistler et al., 2006].

In 2007, IBM and partners[4] introduced an asymmetrical CMP to mar-

[1] Manual schemes or an automation process through a compiler.

[2] Also known as a many-core processor.

[3] Other data and instructions are also placed on the same bus, such as memory address fetches, data instruction fetches and I/O interrupt calls.

[4] Jointly developed by a Sony, Toshiba and IBM alliance known as STI; see Chapter 4.

ket, the Cell Broadband Engine multicore processor [Gorder, 2007]. The underlying architecture and design of this CMP has presented a significant change. It exhibits increased flexibility, reduced complex communication flow and improved parallelism. The research conducted in this study presents the performance gained by using TLS as a concept [Steffan et al., 2000] on an asymmetric CMP. This fine-grained approach requires low-level programming to allow a speculative paradigm on an asymmetric CMP.

Higher-level parallelism is achieved with the support of the OS, which in particular, provides an interface and support for thread creation and scheduling. The thread creation process generally requires the programmer to extract parallelism from a program also ensuring that communication and synchronisation are handled properly. Such a manual processes is extensive and exhaustive.

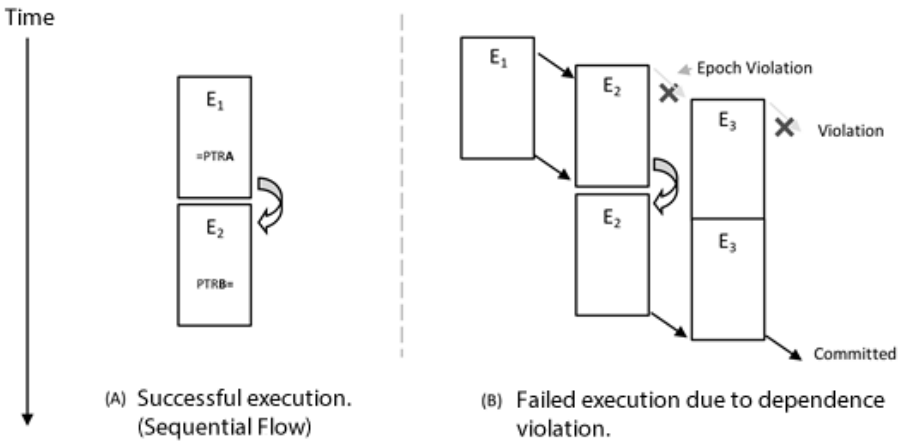


Figure 1.1: Simple thread-level speculation - (A) Sequential program (B) Sequential flow is decomposed into two epochs.

Section 3.2 outlines different forms of violations such as control specula-

tion [Aragon et al., 2006], data dependence and data value speculation [Hammond et al., 1998]. For example, a RAW (read after write) is considered a true data dependence violation, whereby two or more pointers and/or data locations are dependent upon the same memory location (see Figure 1.1).

A sequential program is executed in order, but the execution of TLS program is executed out of order. Nevertheless, phase data commit must respect the sequential counterpart. Therefore, speculative execution would change to in-order mode when data commit is required (respecting the sequential data commit process).

When dependencies occur, the TLS system must locate all versions of dependence and verify the violation. Hence, all data is held within the cache memory subsystem, and this presents a major bottleneck. Special-purpose hardware has been theoretically researched and created, for TLS combined with software. This assists dependency recovery (see Section 3.4.4).

The research presented in this thesis presents a new framework the Lyuba framework, and demonstrates the implementation of the L-API, which requires low programmer effort to extrapolate parallelism for common applications.

Other important factors to consider are Moore's law, power wall, frequency wall, complexity wall, cache locality and single thread performance within the multicore architecture. Processing cores have typically doubled in transistor count every two years resulting in the following observations: chip manufacturing has remained the same, that the cost of computer logic and memory circuitry has fallen dramatically. As both logic and memory components are integrated or placed closer together on more densely packed processors, this has shortened the length of the electrical paths between the components.

Another interesting observation is that the footprint of the processor has reduced, resulting in power reduction and lower cooling requirements. The interconnections on the integrated circuit are significantly more reliable than

past implementations [Stallings, 2009]. As the density of logic and the clock speed on a chip increases, so does power density (watts/cm^2) that has caused difficulty in dissipating the heat that is generated on high-density high-speed chips [Stallings, 2009]. The remainder of this chapter presents a brief discussion of issues surrounding the conventional CMP and the IBM Cell processor.

1.2 Hardware and Conventional CMP versus the Cell

In 1965, Gordon Moore described the exponential growth of transistor count on a single die; this came to be known as Moore's Law. With modern microprocessors encompassing an ever-increasing transistor count, there is now a situation where the main cause of loss of performance is the power constraints (albeit, partly due to the increase in the number of transistors). The design complexity of the overall microprocessor architecture also contributes to loss of performance.

Microprocessors are increasingly complex and exhibit enhanced functionality, ranging from simple microcode-based cores to complex out-of-order, fully pipelined cores housing extensive instruction sets including on-board caches and I/O and memory controllers that contain either single or multiple cores. While all these additional components provide a rich-feature set and improve the performance of a processor it has also increased the complexity. Such complexity has hindered the extraction of parallelism by programmers.

As mentioned previously, power consumption has constrained achievable peak computing performance. Thus, reducing power consumption has arguably become the highest priority for computer architects [Crowley and Turner, 2007, Qi and Zhu, 2011]. Furthermore, higher clock frequencies have resulted in signals propagating across a small area of a processors core, which means that only a limited amount of logic can be performed within a pipeline stage. Unfortunately, as a compensatory move, the numbers of pipeline regis-

ters have been increased, which further exacerbates the power usage problem and limits scaling of higher clock frequency.

Performance is not only constrained by the microprocessor, but also affected by the off-chip bandwidth, particularly the communication latency between the microprocessor and system memory. Ensuring that the microprocessor is able to access the required data (i.e. it can retrieve and send data to the off-chip memory) requires bandwidth. Lack of bandwidth causes processing bottlenecks, especially if the processors require a larger amount of data. This reduces the overall performance of a program.

A conventional multicore processor has a non-unified data and instruction cache (L1) private to each core, as well as a single shared unified cache (L2) that is visible to all cores contained on the same processor. Conventional CMPs suffer from latency, owing to context switching and bandwidth issues. Importantly, the cache memory system on the processor can also become a major bottleneck.

CMPs are equipped for parallel execution and generally provide excellent performance for multitasking operating systems. When a thread is spawned, the speculative system might create multiple versions of the same thread. The data and instructions of each thread are retrieved from the systems main memory and sent to the processor's cache memory. At this point, synchronisation is needed to ensure data validity. Multiple versions of the same data are checked in different cache memory systems, increasing additional overheads and reducing overall performance. Given this complexity, researchers have devised multiple schemes and extensions to current hardware components [Acosta and Liu, 2012].

One research object was to ensure that data is correctly scaled to all processing elements using an asymmetrical CMP and architecture. The IBM Cell is an asymmetrical processor with distinct architectural differences when compared to a symmetric CMP architectural design

The IBM Cell Broadband Engine (CBE) is a CMP that contains nine

processing elements. One of the processor elements on the Cell CMP is a conventional RISC PowerPC core (PPE) that handles the operating system runtime environment. The remaining processing elements are vector based RISC processor cores (SPEs) that are designed to handle the main computational tasks of a program. Each core can execute an individual program or work collectively on a single program. Any one of the SPEs can communicate with the others without communicating via the central processing element (PPE) by using the on-board memory flow controller-direct memory access (MFC-DMA) [Kahle et al., 2005].

The cache memory subsystem on a conventional symmetric processor reduces the average response time taken to obtain data from the main memory. This is achieved by storing frequently used copies of both data and instruction codes, although sufficient bandwidth is required to ensure that all processing cores are constantly busy with work. The Cell PPE has a similar cache implementation unlike the SPE. Each SPE has a large unified register file and is manually controlled by the programmer. Such an approach is significantly different from traditional symmetric processors as the cache memory is controlled by hardware algorithms. However, this manual approach does increase complexity but it also gives a programmer greater control.

On a traditional symmetric CMP, a thread must check multiple versions of its data held within multiple core cache units. This is followed by a broadcast message that interrupts all available cores. Once a core receives this interrupt, it will handle the interrupt appropriately before returning to its previous state. This procedure of stalling a core's current work causes additional overheads and reduces overall performance.

The Cell CMP introduces an unconventional communication mechanism known as the memory flow controller (MFC) on each SPE (see Section 4.5.5) that attempts to keep data on-chip and available for use by other processing elements. The Cell does not retain all data on-chip, in spite of attempting to keep as much of the communication on the processor while optimising efficient

use of available bandwidth. This memory controller handles all communication to and from the SPE and handles specific memory logic instructions. The MFC provides a flexible communication mechanism that assists parallelism and complex data flow.

1.3 Aims and Objectives of this Research

The aim of this study is to develop a framework that encompasses efficient algorithms for the underlying CMP. To accomplish this task, a review of the relevant background materials will be conducted, which will establish the current status of the research field with the following objectives:

1. To investigate current TLS models, schemes and proposals on traditional architectures.
2. To evaluate a mechanism (through the development of a software framework) that supports a non-speculative execution.
3. To analyse the results generated by the software tool and to verify any performance gains or losses.
4. To investigate the Cell CMP asymmetric applicability for non-speculative type execution and available programming methodologies.
5. To ensure that the software framework efficiently distributes the execution of threads on an asymmetric CMP while simultaneously handling dependencies through inter-thread communication using hardware mechanisms and software logic.

1.4 Contributions

A brief discussion was provided on parallelism, conventional hardware and, more specifically, multicore processors and a software approach for extracting

parallelism. Research on speculation analysis, including speculative hardware implementations to assist parallelisation, has been researched for conventional and specialist symmetrical processors, but asymmetric hardware has not been researched in-depth with or without the use of speculation.

There are considerable numbers of projects that specifically focus on developing methods to automatically parallelise applications (see Chapter 3) with a large number of projects focusing on hardware aided designs (see Section 3.4.4). Instead, this thesis investigates two other areas. Firstly, manual parallelisation coupled with programmer expertise such that design modifications can yield higher parallel performance. Secondly, another important criterion from this research is the understanding, development and analysis of a software based framework executing on a heterogeneous multicore processor; the IBM Cell Broadband Engine.

An important clarification that should be stated is the contribution from this study. Past and current researchers have focused meticulously on homogeneous multicore processors including specialist custom processors (see Chapters 3 for a detailed review).

At the time that this thesis was written, little or no research was found that accelerates the detection of data hazards on a heterogeneous microarchitecture and/or utilising microprocessor hardware to accelerate high-level code, specifically the IBM Cell processor.

For this study, a framework was developed known as the Lyuba framework (a non-speculative system). A programmer is able to interact with the framework through the L-API (Lyuba-application programming interface), see Section 5.1.

The framework targeted asymmetric multicore processors in particular the IBM Cell Broadband Engine (see Section 4.5). My research takes the underlying principles of thread-level speculation (TLS) see Section 3.4.

The research itself takes the execution model of traditional TLS as the underlying philosophy; however, no speculation mechanisms are coded or sup-

ported in the framework. The Cell BE processor has no hardware speculation support. However, the framework leverages the Cell memory subsystem and data transmission mechanisms (DMA, see Section 4.5.6) to access data by calculating the precise address of the data for all load and store operations (see Chapter 5 for a detailed discussion).

The study conducted in this thesis does focuses not only on hardware utilisation but also on the management of data, on-board communication and self-recovery using an asymmetric multicore processor. Together with manual parallelisation, this thesis also details how parallelism is extracted from sequential code. The research presented in this study can help direct potential and future related research to automate and improve performance for general and scientific applications towards the direction of the L-API framework.

This study presents manual parallelisation, which is applied to the selected benchmark using the L-API that supports and extracts TLP, overcoming obstacles that are prevalent in parallelisation.

This study presents the L-API framework using a selected benchmark. Performance results were obtained for five applications using multiple datasets and the applications with a small dataset showed an improvement of 17%, a larger dataset showed an improvement of 16% and the overall performance improvement was 11%.

The author of this research firmly believes in heterogeneous architecture and in order to effectively accelerate code on ever-increasingly complex multicore hardware, software engineers must take into consideration data hazards without the need for speculative hardware because speculation is CPU cycle intensive and limits computation performance.

1.5 Overview and Structure of this Thesis

Research from this study will target a specific asymmetric CMP and the remainder of this dissertation is structured as follows: Chapter 2 outlines the research methodology taken in this dissertation. Chapter 3 presents a discussion of past and current literature on thread-level speculation. Chapter 4 reviews architecture primitives for parallelism such as synchronisation, parallelism, and the most importantly an overview of the IBM Cell processor. Chapter 5 then showcases the Lyuba framework and the L-API. Chapter 6 outlines the results of the experiments. Finally Chapter 7 presents the conclusion of the study with projected future work.

This study does not modify existing compilers or use specialist TLS hardware such as the Hydra CMP [Hammond et al., 2000], however, presents the use of a mature compiler and a commercially available processor. The practical outcome of this study is to present a software-based library for the Cell processor.

Chapter 2

The Research Methodology

This chapter outlines the research problems and describes the processes of method selection and data analysis. An overview of the applications, hardware platform used during the research project, discusses research ethics and the value of the research undertaken. The research problems can be expressed as follows:

1. How can dependencies presented in complex code, which use both hardware and software, be handled?

2. How can workloads be evenly distributed across an asymmetrical processor?

It was necessary to select a set of applications, hardware and parameters (datasets) for the research aspect that involved parallelising applications using the TLS principles, the underlying philosophy of the Lyuba framework. For the purpose of testing, where appropriate, two datasets of varying sizes where chosen for each application. The rationale for selecting datasets is described in Chapter 6.

2.1 Approach Taken for this Research

An understanding of the characteristics of the application, hardware and the ability to extract TLP was sought before an application was transformed and executed on an asymmetric multicore processor. By understanding the complex interactions of an application, it was possible to profile the code and clearly identify which parts of the code were parallelisable. Combining the analysis of the benchmark applications with a greater understanding of the hardware assisted with the development and certification of a framework (see Chapter 5).

Of particular interest was the ability to show how asymmetric hardware can be used to assist parallelism. However, reliance on this asymmetric hardware became obvious. As a result, the framework developed was programmed in a modular format, such that all communication mechanisms are in fact modules. To address the research problems, an IBM Cell CMP was utilised. Due to its unique hardware properties such as its DMA communication mechanism, Mailboxes and Signals (see Sections 4.5.6 and 4.5.7 respectively). These properties assist with parallelism and are compatible with the framework.

The applications selected for testing were chosen because they were conveniently available. Moreover, the selected applications were useful to demonstrate transformation using the Lyuba framework constructs. The objectives of this research were to understand and capitalise on a given asymmetric hardware and to accelerate synchronisation and communication between multiple processing cores, while limiting the off-chip bandwidth for such mechanisms. Hence, a wide bandwidth was given for storing and retrieving data from system memory.

The methodology presented in Section 5.1 describes how the L-API constructs were applied to un-parallelised code and highlight the effort required of the programmer. By referring to the experiences of conducting parallelisation using the L-API, ways in which applications must be prepared in order

to correctly apply the L-API constructs are described.

2.1.1 Quantitative and or Qualitative

To conduct the research the following two methods were incorporated: quantitative[1] and qualitative.[2] It is important to recognise the application of both. Quantitative methods are used to gather quantitative data – measurable properties such as mathematical models, scientific theories and hypotheses from the investigation. The empirical observation for this study is based on the results from the computation taking place on an asymmetrical CMP. Participant observation is also considered as a branch of quantitative research, in which the following will be used:

1. C/C++ programming to code lightweight rollback routines for the Cell processor
2. Analysis of performance results
3. Tables and graphs to illustrate results and statistics

Qualitative methodology relies upon an in-depth understanding of the subject matter – a descriptive approach. Qualitative applicability for computing research is limited when research involves extensive and complicated scientific experiments. Since both methods identify key components, the study is naturally biased towards quantitative; however, qualitative methods were used for the literature review, so both methods were incorporated into this study.

[1] A systematic investigation of scientific properties, phenomena and their relationships, expressed through measurements and numerical analysis.

[2] An in-depth understanding of the subject matter.

2.1.2 Applications Used for the Research

A search was made for simple, standardised applications, freely available via the Internet, that represented general-purpose applications; the SciMark benchmark [Pozo and Miller, 2004] was chosen. The algorithms were extracted from the benchmark, and the constructs from the Lyuba framework were applied - see Chapter 5. The benchmark comprises five applications from floating point to integer applications. Even though for the number of applications is relatively small general-purpose applications, the importance is the quality of the data structures utilised in the algorithms and how such applications are transformed. These applications are representative and exemplify different types of computation including FFT, Gauss-seidel relaxation, sparse matrix-multiply, array copy algorithm extrapolated from the monte carlo integration application and dense LU factorisation.

2.1.3 Measurements

All applications use either two or three datasets and each application is run approximately 10 times. By running each application 10 times, a mean is generated from all the cycles to represent a normalised result. This approach allows clarity of performance and an assessment of the impact of the Lyuba framework. The dynamic behaviour of the application changed substantially during the course of the assessment when all datasets were applied. Hence, caution must be exercised when assessing the parallel performance of a test system by using a small section of the application run.

The IBM Cell Broadband Engine simulator [IBM, 2005] was used to measure the execution. Measuring the parallel performance of a system of multiple cores is a considerable challenge. A great deal of care was taken to accurately capture results using the Cell's high precision hardware counter. To capture the results of the entire Cell processor is extremely time-consuming. Hence, only segments of execution that were considered of greater importance

(ensuring internal validity[3]) were selected for measurement.

It has been usual to select several areas for measurements such as subroutines, in order that a pattern that closely approximates the behaviour of the entire execution would be seen. This is a satisfactory approach when assessing symmetric based hardware. However, the hardware used in this study was based on asymmetric hardware with similar or varied behaviour for all processing cores.

2.1.4 Value of Research

As mentioned in the introduction, the multicore era is rapidly becoming the de-facto standard, with future predications of 10−260 cores and more on a single-silicon [Battiwalla, 2009]. New software mechanisms, techniques and implementations will be needed to extract maximum parallelism and to ensure that resource utilisation is kept high while simultaneously handling complex dependencies. Parallel programming is not a new concept or a new idea but has existed since the dawn of the SMP (symmetrical multiprocessor) era and it has remained prevalent in the scientific and academic community. For example, current and popular threading models such as MPI [Karniadakis and Robert, 2003], OpenMP [Chapman et al., 1997] and Posix [Barney, 2010] already provide an adequate solution for extracting parallelism. These languages provides a communication mechanism for transferring instructions including data across multiple processing nodes. However, such languages do not handle complex dependencies within complex data structures and do not fully utilise the hardware capabilities of the CMP.

Interestingly, OpenCL has grown in strength due to its heterogeneous structure by encapsulating and distributing code to both CPU and GPU. However, the technology is young and still requires further analysis [Gaster, 2010]. The OpenCL API was not available when this study began but future

[3]There is an assurance that the observations in the sample group are accurate. This ensures that the assessment of characteristics was able to be defined for measurement.

versions of the L-API could possibly incorporate OpenCL technology.

The proposal by [Oancea and Mycroft, 2008] bears similarity to this study from a software perspective. The authors implement a lightweight TLS library as the underlying framework that handles dependencies and violations. This current study also proposes a rollback routine (derived from thread-level speculation) and, similar to [Oancea and Mycroft, 2008], it handles violations but it also exploits the underlying architecture. Currently no literature or studies have been found that attempt TLS or speculation-type execution for an asymmetrical CMP, specifically the Cell processor. The contributions from this study will support the asymmetric CMP programming knowledge pool and introduce a new parallelisation library to an asymmetric CMP.

Table 2.1: Hardware environment.

Machine environment	Specification
Primary machine	1. Intel Core 2 Quad 2. 4GB RAM 3. 100MB Ethernet network interface 4. Fedora 7 with IBM Cell development libraries
Test environment	1. Sony Playstation 3 20GB 2. IBM Cell 6-SPU 3. Fedora 9 with IBM Cell development libraries

Table 2.2: Software environment.

Software
1. Fedora 7[a]
2. IBM Cell SDK
3. IBM Cell Fuel System Simulator
4. Eclipse CDT Integrated Development Environment

[a]The official supported operating system for the IBM Cell SDK 3.0.

2.1.5 Equipment

The equipment used for this study is listed in Table 2.1. To develop for the Cell processor, the SDK provided by IBM was used since it has some open source elements such as the GNU C compiler. The rest of the SDK is closed source including the IBM XL compiler. Unfortunately, the SDK is only compatible with a Linux based operating system, necessitating its use. Both the SDK and the operating system are available from the Internet as freeware (Table 2.2 lists the main software tools used for this study).

2.1.6 Limitations

The number of available processing units limits the hardware used to test the framework. The thermal temperature readings were not accessible due to the hypervisor, obtaining the temperature parameters of the framework running on the processor could not be carried out.

2.2 Summary

The research methodology used is described. The development and testing environment, testing algorithms and the value of research were considered. This chapter also outlines the main limitations of the project, i.e. hardware resources. The layout of the remaining chapters is as follows: a literature review of the main field of study, followed by an overview of technologies employed including an understanding of the IBM Cell microprocessor. More importantly, the study will present the Lyuba framework (L-API) and this is followed by the results of the experiments. The final chapter will present a final conclusion make suggestions for future work.

Chapter 3

Thread-Level Speculation Theory and Application

This chapter investigates thread-level speculation (TLS) including background material and related works that support TLS. The rollback routines developed from this study are derived from earlier studies; however, before considering such routines and the L-API, the fundamental grounding and the theoretical basis of thread-level speculation (TLS) must be considered. A persistent bottleneck that impedes thread-level speculation is value communication, speculative states and synchronisation between speculative threads, whereby the difficulty remains in deciding how and when to apply the above methods. Next, this chapter will go on to describing how a system based on TLS philosophy can be practically implemented on an asymmetric platform. Originally invented by [Knight, 1986] there have been many systems and schemes in addition to implementations that have extended the basic concept of TLS.

However, to determine whether speculative based systems are viable and/or relevant on today's modern asymmetric processors, past and current TLS schemes and implementations must be investigated. The remaining sections will describe the fundamental parallelism techniques and then the classical

TLS execution model coupled with an evaluation of TLS schemes including hardware and software supported schemes; compiler and speculation types are also documented in this chapter.

3.1 Execution Model

Thread-level speculation (TLS) was originally proposed by Knight [Knight, 1986], and since then there have been many schemes, proposals and implementations that have extended the basic concept of TLS. This section describes the classical execution model for TLS, which is typically implemented in a compiler and executed in hardware. The basic overview of the execution model is discussed below and the remainder of this section evaluates different TLS proposals and schemes, types of dependencies, cache coherence and manual parallelisation.

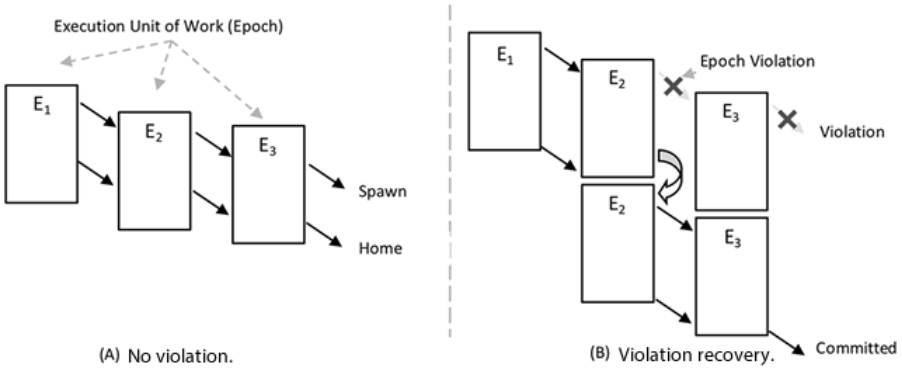


Figure 3.1: Classical TLS execution model. (A) illustrates epoches executing with no violation. (B) illustrates multiple epoches with a violation occuring on epoch (E2), this results in violation recovery.

Figure 3.1 depicts the classical execution model for TLS that has been adapted from [Steffan et al., 2005]. The following description is a high-level overview of TLS model. Firstly, a program is decomposed into units of work

named epochs (or speculative/worker threads), which may contain dependencies. When a microprocessor is executing instructions in the out-of-order manner, the microprocessor must respect dependencies between instructions (see Section 3.2).

The first spawned thread of a program will at all times remain non-speculative. Typically, a non-speculative thread has additional privileges as depicted in Figure 3.2. The first spawned thread is known as the controller-thread (`CT`) and is not speculative. The `CT` can start, stop or halt any of its worker threads (speculative threads/epochs) and commit final results back to memory.

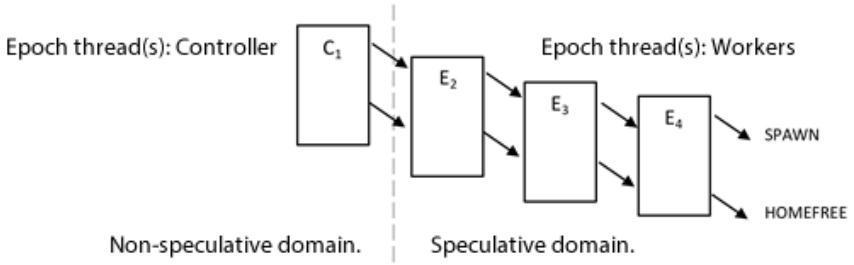


Figure 3.2: Epoch relationship model.

In Figure 3.2 the work crew - `E2`, `E3` and `E4` (executed speculatively) - are responsible to `C1` (a non-speculative thread). Once a speculative thread has completed its execution of work, it would contact the `CT` and acquire another unit of work. Once all available units of work are exhausted, the `CT` would then be ready to commit-back the results to main memory. If an epoch speculatively identifies a dependency, it would halt its execution and notify the `CT`.

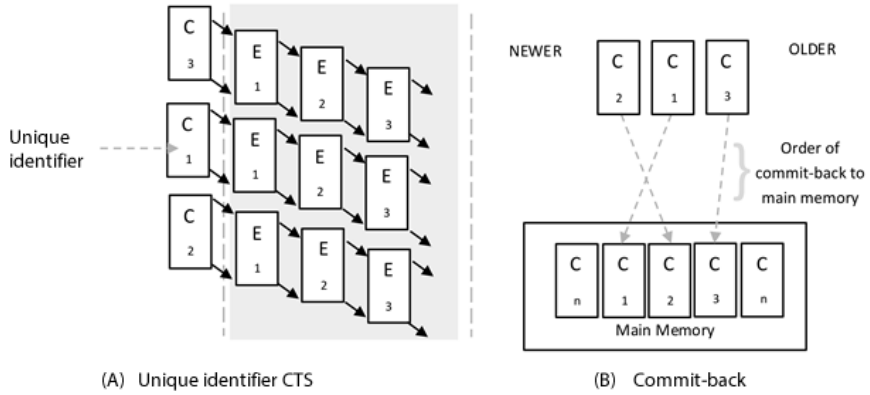


Figure 3.3: Committing results back to main memory. (A) illustrates the controller threads (CT) and their unique identifier (commit-back priority). (B) illustrates two important components; the controller queue and main memory. The queue shows the CTs and their current ordering. Before CT-3 commits back to main memory, the logically earlier CTS must commit their results back to main memory, whilst respecting the original sequential program semantics.

This is highlighted in Figure 3.3 where the CT would squash (i.e. stop) E2, including any logically later (following) violated epochs. The dependency is then corrected and all logically later epochs are restarted. The CT would then resolve the dependency in E2 and then finally restart the squashed epochs (E2, E3 and E4) as depicted in Figure 3.3 (B). Each speculative thread (epoch) has the capability to spawn new epochs through a spawning mechanism, which is described in more detail in Section 3.4.2.

A spawned epoch would obtain its initial parameters including the program counter (PC) and a unit of work from a logically earlier (preceding) epoch. When the CT has completed its work set, the results are then ready to be committed back to main memory. To ensure that the original sequential semantics are respected as described in Section 3.1, the CTs have a unique identifier that represents the logical-sequential order as depicted in Figure

3.3 (A).

Table 3.1: Hazards.

Structural	Resource conflicts/hardware contention. The multiple instructions require a hardware component that is only available to one instruction per cycle(s)
Data	Instruction dependency
Control	Alters the PC of a program by either a branch or other instructions

3.2 Dependencies and Data Hazards

This section explores dependencies that commonly occur within inter-thread communication. Table 3.1 presents three levels of hazards: structural hazards occur at the hardware layer and hazards such as resource contention generated by microprocessor resource management instructions and actual instruction hazards, whereby program instructions (statements) correspond with the data of a preceding statement; and control dependencies, that is control logic within code statements such as branching/`if`-statements.

Table 3.2: Common Dependencies.

Flow/true	Example: RAW (read-after-write), where a later epoch consumes a value/datum produced from an earlier epoch. However, the consumed value may have been updated by another (an earlier) epoch, hence, the consumed value becomes stale and incorrect.
Anti-dependence	Example: WAR (write-after-read), a later instruction affects (overwrites) the input value for another (earlier) instruction.
Output	Example: WAW (write-after-write), a later instruction affects (overwrites) the input value for another (earlier) instruction
Input/Output	When two or more instructions attempt to access the same file simultaneously

The most common dependencies are listed in Table 3.2. Desirable properties for speculative threads are high predictability for control and data dependencies [Krishnan and Torrellas, 1998, Marcuello et al., 1998]. Study by Rundberg [Rundberg and Stenstrom, 2001] the authors explore name dependencies and true data dependencies, using synchronisation atomic primitives, similar to the Zhai's research [Zhai et al., 2002,Zhai et al., 2004]. Their study brings into focus load and store instructions while resolving dependencies. Nonetheless, their base line scheme is memory intensive and may not be practical on modern CMPs.

A profiling approach in [Rul et al., 2007] examines memory dependencies between different functions using call graphs. Interprocedural data flow graphs are beneficial for viewing dependencies and determining memory access times of functions. The memory access times were dependent on other functions (similar to the producer and consumer structure). This approach was supported by cluster analysis of dependent functions and synchronisation. The final step was the testing of a compression algorithm from the SPEC CPU 2000 benchmark suite (bzip2). The end result was a speed-up

of 3.74 (compression) and 1.41 (decompression).

Despite this comprehensive analysis, the aforementioned study [Rul et al., 2007] did not explain the memory results. The use of these timings and other such measurements obtained from the study could have possibly improved performance but were not considered or taken into account. Furthermore, this study did not define possible cache issues, such as cache miss rates, which also adversely affect the performance of the speed-up. This research exploited a high-performance processor (Intel© Itanium© processor), which is widely used in high performance computing machines (i.e. supercomputers). However, such a processor is not common in the general commodity processor environment, so the authors' work is restricted to a smaller audience. One final shortcoming of the research was that the implementation was hardware dependent. The specific hardware required is not portable, adaptable or scalable to commodity processors without a special instruction set. To summarise, it is apparent that the fundamental issue with TLS proposals must overcome dependencies within instructions.

This section has highlighted the general dependencies such as RAW (most common dependency) and past TLS studies have extensively explored a processor's cache memory system. The following section explores the cache coherence protocol.

3.3 Writeback Invalidation-Based Cache Coherence

This section evaluates invalidation-based cache coherence schemes. The coherence protocol makes it possible to maintain consistency in cached systems. The motivation for invalidation-based cache coherence is to invalidate cached copies of a line before getting exclusive ownership of that line. The cache lines hold the most recent and valid data in the main system memory. Reported works that are heavily dependent and extensively utilise this protocol

include [Zhai et al., 2002, Warg and Stenstrom, 2008, Krishnan and Torrellas, 1999b, Krishnan and Torrellas, 1999a, Warg and Stenstrom, 2006].

Study by Steffan [Steffan et al., 2000] explore the use of a shared cache for TLS. Limiting the number of epochs accessing a single shared speculative cache line to one reduces the negative concurrency behaviour that generally afflicts multithreaded programs. The authors employ a specialist TLS hardware processor and such hardware-centric designs are considered to be more effective than software schemes. However, TLS hardware processors have additional overheads including maintaining different states of data in the cache and that the use of additional coprocessors requires additional resources to maintain different processor states. As a result such states affect power consumption, thermal issues and the hardware complexity of a CMP. Unfortunately, their scheme is not transferable to modern processor architectures, the applicability of their scheme cannot be adequately warranted by their results.

In [Tuah et al., 2002] the authors examine a cache memory subsystem and present a detailed analysis of cache use and in particular the use of prefetching. The authors develop a performance model by which their algorithm attempts to solve the knapsack problem [Hifi et al., 2008]. While the implementation is theoretically complete, it may not be practical or viable on modern processor architectures due to extensive modification of the prefetching scheme. Processor designers of modern multicore processors typically optimise the prefetching scheme, which is efficiently developed and works effortlessly. Another cache supporting mechanism that is implemented in both speculative and non-speculative CMPs is the snoop cache coherence protocol that is explored meticulously in [Kumar and Huggahalli, 2007, Gopal et al., 1998].

This section has outlined the importance of cache use for thread-level speculation. By extending the cache to accommodate additional cache line states, TLS researchers have been able to gain significant hardware control.

Consequentially additional hardware is required as more speculative data is processed, requiring more data to be placed within a cache of a processor's memory subsystem. However, it is not clear from literature if cache capacity could be a potential bottleneck. The remaining sections will explore TLS schemes.

3.4 Thread-Level Speculation Schemes

This section reviews different implementations encompassing both hardware and software approaches: [Knight, 1986, Steffan et al., 2000, Wang et al., 2008, Wang et al., 2009, Fung and Steffan, 2006a, Warg and Stenstrom, 2008, Steffan et al., 2005, Gupta and Nim, 1998, Krishnan and Torrellas, 1999b, Zilles and Sohi, 2002, Johnson et al., 2007, Prvulovic et al., 2001, Renau et al., 2005, Rauchwerger and Padua, 1995, Franklin, 1993, Garzaran et al., 2005, Yanagawa et al., 2003, Iwama et al., 2001, Olukotun et al., 1996, Gopal et al., 1998, Luo et al., 2009, Marcuello and Gonzalez, 2000, Marcuello and Gonzalez, 2002, Hammond et al., 1998, Martinez and Torrellas, 2003, Packirisamy et al., 2006, Oancea and Mycroft, 2008, Madriles et al., 2008, Wang et al., 2008, Miura et al., 2003]. The invalidation-based cache coherence protocol has been extensively researched (e.g. [Steffan et al., 2000, Steffan et al., 2005, Madriles et al., 2008, Xekalakis et al., 2009, Da Silva and Steffan, 2006]).

In [Steffan et al., 2000] the processor's cache coherence protocol (i.e. writeback invalidation-based cache coherence) has been rigorously extended to handle arbitrary memory access patterns such as array references (see Section 3.3). The authors only attempt to leverage the coherence scheme for the nearest/local cache unit (L1); their study does not clearly define hardware utilisation. However, such an approach could potentially scale to large processors (distributed environment). Moreover, performance results are only shown for loop-based algorithms and not complex pointer-based algorithms.

A similar implementation by [Krishnan and Torrellas, 1999a, Krishnan

and Torrellas, 1999b] modifies the coherence protocol to detect data dependencies (see Section 3.2). Their conclusions demonstrate sequential binaries decomposed using a TLS compiler into a binary state without the need for complete recompilation. The compiler inserts specific TLS codes [Zhai et al., 2004] to ensure that threads are handled correctly.

This approach is limited to static time compilation, where the actual program is analysed and speculative codes added before execution of the program. Unfortunately, applying speculative codes at runtime is considerably difficult due to hidden and often complex dependencies. In summary, the authors have not evaluated complex data structures and their approach does not execute multiple speculative programs, which is a serious limitation.

Should multiple speculative and non-speculative programs exist in the same domain, then a complex context switch is required since this directly increases latency and processor cycles, although their approach is in theory possible. However, their scheme is not really suitable for modern CMPs.

Study by [Packirisamy et al., 2006] used a dual speculative thread protocol that can overcome some of these limitations. The main outcome from their research was the ability to ensure that speculative and non-speculative threads are kept isolated. The approach can limit or reduce the bandwidth for speculative data traffic on the CPU by restricting speculative threads to execute on the same processing core, thereby limiting speculative data to registers and cache to a single core of the CPU.[1] However, the scalability implication of this approach applied to larger processor designs and distributed architectures is unclear.

Typically, a sequential program with TLS assistance would execute on a single processor using a non-unified L1 cache. Threads executing on multiple processor elements would disrupt cache locality resulting in defragmentation [Tam and Tam, 2003].

The study by Fung [Fung and Steffan, 2006b] reduced data defragmen-

[1]Provided the CPU has more than two processing cores.

tation while improving the cache locality problem. Their article illustrates where cache misses occur within the L2 cache. However, modifying cache policies by the programmer remains limited. Moreover, the authors only use parallel access patterns such as loops and this requires hardware support (see Section 3.4.1).

Hammond [Hammond et al., 1998] describes a TLS scheme that uses both hardware and software techniques to extract parallelism from a sequential binary. Once the sequential binary is decomposed (see Section 3.1), the speculative threads that contain dependencies can be handled by speculative assisted hardware – the Hydra speculative chip-multicore processor (SCMP) [Hammond et al., 2000]. The focus was placed upon memory accesses made by the speculative threads that might violate true dependencies (see Section 3.2).

However, it is unclear in what manner the Hydra processor handles and executes non-speculative code and requires significant hardware understanding. Whereby, additional hardware used in their scheme handles dependencies with the aid of a coprocessor (known as the speculation control coprocessor). Furthermore, cache memory – that is external to the coprocessor – was modified to store additional data (similar to the cache coherence protocol) resulting in a scheme that heavily relies on speculative hardware. The complexity and effectiveness of their research including hardware utilisation, remains unclear.

Another well-documented source of parallelism is loops. A significant amount of time is spent executing instructions within loops, explaining why loops have become a common source for parallelisation (see Section 3.4.1). In sharp contrast to earlier TLS research, which concentrated on low-level hardware-specific implementation (see Section 3.4.4), [Kazi and Lilja, 2001] implemented a software-heavy scheme. Their scheme supersedes Superthreaded Architecture [Tsai and Yew, 1996] such that the Kazi and Liljas implementation is a coarse-grained threaded pipelining system, which resembles [Tsai

et al., 1999]. Nevertheless, the approach used was implemented on a specific processor, the Hydra chip multiprocessor and, as such, cannot be ported to other architectures [Hammond et al., 1998]. From a practical perspective, the work of Kazi and Liljas is only applicable to proprietary CMP, bringing the importance, scalability and availability of their work to only a limited environment.

Work by Oancea [Oancea and Mycroft, 2008] also used thread management; however, their implementation differed from the research presented in this thesis in that it was hardware specific. In this thesis, a unique process for each processor element is created through the libspe2[2] library. This library mimics the actual processor element [Mols, 2009]. The [Oancea and Mycroft, 2008] study used functors (function objects; [Haendel, 2005]) to create new threads and handle the different stages of the execution of the thread. By way of comparison with the libspe2, the `spe_context_create` function creates an image of an SPE, which provides most of the capabilities of an SPE[3] (see Section 4.5.2 for an overview of the SPE).

Study by Oancea and Mycroft [Oancea and Mycroft, 2008] describe hardware resource utilisation. In particular, threads were not discussed in depth and the paper did not explain the impact of a large number of threads (i.e. optimum concurrent thread use). Such information is important since it allows the upper limit of active threads to be determined and supports the analysis of memory use and/or memory requirement. As the described system aims to support and resolve data hazards, addressing memory-related issues is essential for managing data hazards. If the scheme generated a significantly large number of threads, and each thread was allocated space within the stack, the impact on memory utilisation, which has a direct effect on the virtual memory (VM), physical system memory (RAM) and possibly on the locality of cache memory becomes pertinent. Such information would

[2]The libspe2 is a library interface developed and distributed by IBM.

[3]SPE is a simple in-order core processor, which is one of the central components of the IBM Cell Broadband Engine multicore processor.

highlight possible bottlenecks that adversely affect speed-up. While Oancea and Mycroft's scheme demonstrates potential, its implementation was only executed on a shared-memory multiprocessor system with no hardware specific details. Moreover, no consideration was given to the scalability of their scheme.

Studies by [Packirisam et al., 2008, Renau et al., 2005, Luo et al., 2009] examined TLS workload efficiency on a high performance SMT CMP processor using performance, power and thermal indicators as benchmarks. The scheme implemented a TLS-based compiler targeting Intel's Itanium architecture. Moreover, the paper presented an in-depth TLS comparative analysis of multiple SMT CMP configurations, using the SPEC benchmark. Once again, the scheme is restricted to a specific processor type from Intel© and it remains unclear how it transcribes to other processors.

3.4.1 Loop-Based Speculative Execution

Loops typically assert a deterministic structure and potentially amass a significant amount of parallelism, so loops have become an ideal source of parallelism, and is the significant coverage makes them an ideal source for parallelisation. Schemes from [Hammond et al., 1998, Kejariwal et al., 2006, Gupta and Nim, 1998, Martinez and Torrellas, 2003, Wang et al., 2008, Marcuello and Gonzalez, 2000, Tian et al., 2009, Oplinger and Lam, 2002, Li et al., 2005] exploit loops through different TLS techniques.

In [Wang et al., 2005] the authors propose an algorithm that simply selects loops that could be successfully parallelised, the output being an improved overall performance. The authors implement a loop selection algorithm using two parameters, speed-up and coverage of loops, that equate to an optimal selection of speculative parallelisable loops. But such an approach is limiting the scope of potential parallelisable code; furthermore their scheme extensively analyses the speed-up through the aid of profiling and compiler analysis for each loop. However, the results did not reflect the actual behaviour at

runtime – see Section 3.4.5. The authors have not identified overheads, although they have noted the behaviour of TLS loops that will vary across different invocations. Furthermore, their scheme can only improve the performance for some benchmarks.

Research by [Gupta and Nim, 1998] implements a similar approach, based on static program analysis [Rauchwerger and Padua, 1995] of potential speculative parallelisation of loops. The authors' scheme presents a composite of run-time tests for speculative loops. However, their approach is limited to dependence testing and has not captured instruction complexity such as pointer referencing, complex data structures within loops and control flow. Loops and pointer referencing (complex instructions) are closely related. For an in-depth analysis and discussion of loops, please see [Hammond et al., 1998, Gupta and Nim, 1998, Martinez and Torrellas, 2003, Wang et al., 2005].

3.4.2 Thread Spawning Policies

An important consideration in TLS is the thread spawning mechanism, and [Marcuello and Gonzalez, 2000, Marcuello and Gonzalez, 2002, Oancea and Mycroft, 2008, Madriles et al., 2008] assess thread-spawning techniques including thread ordering and predication techniques, while their scheme concentrates on thread spawning for loops with the use of hardware. Unfortunately, the authors have not clarified how branching hardware is modified to support their loop iteration, including continuation policies nor discussed the scalability of their approach.

The authors have analysed two specific spawning policies, namely sequential ordering policy (processor cores interconnected through a unidirectional ring topology, i.e. thread data is unidirectional) and unrestricted ordering policy (no strict organisation of processor cores). Although both spawning policies are dependent on the underlying architecture, it is not clear how well sequential ordering is implemented. However, to enhance their schemes, the authors implemented a value prediction scheme [Raman et al., 2008]. The

value predictors attempt to improve scheme performance.

A severe limitation of speculatively optimised loops is the limited research, investigation and handling of complex pointer referencing instructions that are external from loops – see Section 3.4.1.

3.4.3 Manual Parallelisation

An alternative approach to automatic compiler and hardware driven TLS parallelisation is manual parallelisation [Prabhu and Olukotun, 2003, Gupta and Nim, 1998, Cintra and Ferraris, 2003, Rauchwerger and Padua, 1995, Oancea and Mycroft, 2008]. Research by [Prabhu and Olukotun, 2003] manually inserts TLS codes into the program's source code. In order to support TLS codes, the approach relied extensively on the Hydra CMP [Hammond et al., 2000]. Using such a specialist CMP limits the applicability for a wider range of processors. Indeed, the research is only applicable to an experimental specialist processor and microarchitecture. Furthermore, single-level and loop-based algorithms are extensively tested; there is no indication as to how complex pointer reference code is handled.

Research by [Oancea and Mycroft, 2008, Wenjie et al., 2012] used a similar approach and proposed a TLS library. In theory, the solution by [Wenjie et al., 2012] has the potential to work well on most x86 architectures (CPUs) but at present it is unclear how their solution interfaces with hardware. The authors proposed a design from a high abstract layer that uses the non-standard Boost library for meta-programming [Gurtovoy and Abrahams, 2008]. This library contains extremely efficient algorithms. This fact might, however, necessitate higher memory and bandwidth use. Hence, research offers the promise of improvement if the design is implemented with the instruction set architecture (ISA) in mind and the use of PE's NOC are improved.

3.4.4 Hardware Support for Thread-Level Speculation

Many implementations rely on hardware mechanisms that complement TLS. This section will briefly analyse common hardware implementations, including TLS CMPs. It will also describe research that depends on hardware that has speculative support. Modifying cache lines through the coherence protocol is evaluated in Section 3.3. Research by [Gopal et al., 1998] proposed the use of a speculative versioning cache (SVC) rather than an address resolution buffer (ARB) [Franklin and Sohi, 1996].

SVC is similar to a snoop bus-based cache coherence system [Tanenbaum, 2005] but SVC increases the chip hardware complexity. Many researchers have proposed hardware support for speculative parallelisation (e.g. [Cintra and Ferraris, 2003, Krishnan and Torrellas, 1998, Oplinger and Lam, 2002,Zilles and Sohi, 2002,Steffan, 2003]). Multiscalar architecture [Franklin, 1993] was one of the first complete architectures for TLS. It selects tasks by traversing a control flow graph (CFG).

In [Marcuello and Gonzalez, 2000] the authors research endeavours to prove that speculative TLS CMP increases the performance of a speculative program. The authors confirm the use of speculative architectures, such as the multiscalar architecture [Franklin, 1993], SPSM [Krishnan and Torrellas, 1999a] and the Superthreaded Architecture [Tsai and Yew, 1996] provide the necessary hardware interfaces to forward and predict values produced by threads. Such an approach bears some similarities to an earlier study [Hammond et al., 1998]. However, in order to support such CMPs and speculative architectures, a TLS compiler such as the Stanford University intermediate format (SUIF) compiler [Wilson et al., 1994] is required. The compiler itself is limited to speculative architectures.

Studies by [Krishnan and Torrellas, 1999a,Krishnan and Torrellas, 1999b] proposed an extension of the processor elements register set. This makes the approach similar to the use of the cache coherence scheme. The authors examined the allocation of registers on each core and identified those registers

that store thread data. These registers are globally visible to all cores on the CMP such as a register passing buffer scheme proposed by [Hammond et al., 2000]. This indirect inter-processor communication reduces latency (processor cycles) when sending and retrieving data to and from neighbouring cores. However, it is unclear how global registers are protected from internal and global changes from a hardware perspective. Furthermore, allocated registers in the CMP are not directly connected to other allocated registers on their associated cores while all communication is channelled through the on-chip (NOC).

The scheme also supports a hardware synchronising scoreboard (SS) which is a decentralised structure embedded in each core. The SS stores thread data and allows synchronisation and communication between threads. The same outcomes can be achieved using the cache coherence protocol provided that thread data does not exceed the storage capacity of the cache. Unfortunately, this complicates data management. However, the authors demonstrated notable speed-ups using their scheme although, problems do exist such as scalability, hardware complexity (on-chip hardware use)[4] [Renau et al., 2005, Packirisam et al., 2008].

Steffan et al [Steffan et al., 2000, Steffan et al., 2005] implemented a traditional TLS model to detect data dependencies and violations using load and store address comparisons and at runtime as an indicator . Using the processor elements cache (cache coherence scheme; also known as the write-back invalidation-based cache coherence), the authors scheme was able to detect dependencies and resolve the violations, by modifying the appropriate cache line and/or restarting the thread. However, to obtain exclusive ownership of a cache line requires all other affected cache lines to be invalidated prior to modification being executed [Steffan et al., 2005]. No timing or overhead indications were available for analysis. Furthermore, the authors' scheme requires hardware support to store speculative state bits (an

[4]Energy consumption and efficiency were not considered.

extension to the existing cache line, MESI[5]) known as the ownership-required buffer (ORB) [Radulovic and Tomasevic, 2007].

The common held opinion is that current modern processors lack (or have insufficient) hardware speculation support. Hence, the scheme is limited to speculative type processors and will not scale to non-speculative processors due to its reliance on speculative hardware.

The study by Martinez [Martinez and Torrellas, 2003] concentrated on synchronisation and coined the phrase speculative synchronisation, while developing this unconventional synchronisation mechanism. The synchronisation primitives developed during the author's study allowed threads to continue speculative execution in critical sections. This required both hardware and compiler support to ensure that states of each speculative synchronisation were maintained or squashed as required.

Nevertheless, the cache feature set is extended to carry an additional bit for acquire and release logic. This simply changes the cache policy system that releases the cache line once the associated extra bit has been flagged with a release message, or when an attempt to acquire the cache line has been made by setting the additional extra bit to an acquire bit.

A scheme such as this extends the coherence protocol even though conventional synchronisation attempts to ensure that a critical region is not corrupted by concurrency and attempts to preserve the integrity of data retained in a critical region. It is not clear from the paper how the system ensures data integrity from concurrency. In addition, the author fails to provide details of the additional overheads for the hardware. The paper does, however, identify that no programmer intervention is needed and, therefore, their implementation remains transparent. Despite this, the scheme itself is only applicable to a specialised hardware and it is not clear how the scheme works for multiple or CMP-based processors.

The studies [Packirisamy et al., 2006, Hammond et al., 2000] presented

[5]Modified Exclusive Shared Invalid.

classical TLS implementations using common and typical hardware support; they explored different hardware mechanisms that are typically found on speculative CMPs, such as additional buffers, ORBs, ARBs and register passing buffers (RPBs). These mechanisms attempt to buffer speculative data at the same time as assisting the TLS scheme. Many researchers have been able to implement their TLS schemes through these hardware interfaces. The ARB [Franklin and Sohi, 1996, Gopal et al., 1998] provided hardware speculation support, with the design composed of a single shared buffer interconnecting all processor elements. This design creates significant bottlenecks and results in poor NOC utilisation of load and store transactions. The memory disambiguation table (MDT) was proposed by [Krishnan and Torrellas, 1999a, Akkary and Driscoll, 1998, Cintra et al., 2000], that is similar to a snoop-based coherence scheme but it is implemented as a decentralised scoreboard structure that threads use to synchronise and communicate register values. However, such an implementation requires another NOC/bus, and the schemes custom MDT NOC (similar to a cache memory subsystem with self-governing logic) can only transmit one word per cycle, a major limitation as compared with the IBM Cell EIB – see Section 4.5.4 – resulting in the MDT being a limitation.

The use of a cache controller for speculative states was examined by [Yanagawa et al., 2003], although their work resembles preceding studies, and it fails to address some fundamental issues, such as effective bandwidth utilisation and data throughput.

Interestingly, [Garzaran et al., 2005] endorses the use of hardware speculation to increase the performance of buffer use (i.e. such as ARB). Their scheme reduces overall performance when squashes are frequent, and it does not take into consideration the memory, power and thermal factors. The reader is referred to the [Renau et al., 2005, Packirisam et al., 2008] for a detailed analysis of these studies.

A study by [Garzaran et al., 2003] examined and presented a detailed

taxonomy of buffer use. Similarly to [Hammond et al., 2000, Krishnan and Torrellas, 1999a], unmodified speculative data (i.e. previous version of data) are stored in hardware, namely memory-system history buffer (MHB), that is similar to the work described by [Akkary and Driscoll, 1998] with the remaining TLS protocol implemented in software.

TLS hardware proposals and schemes used a single type of hardware and the speculative states that were visible on the hardware used multiple interfaces, such as traditional epoch identifiers, stored in a processor's core cache. The epoch identifiers (or task_IDs) are stored in the MHB (which represents a log of unmodified speculative data). In the meantime, the rest of the cache is used to store the remaining speculative data. In addition, the paper described modification of the TLB to store and route of speculative state data. The research is theoretically appealing as a quasi-hybrid approach. However, strict hardware requirements and excessive communication across the NOC could impede the overall performance, and this is likely to make this approach non-viable.

A true hybrid scheme described by [Miura et al., 2003] is broadly based on a multiscalar architecture [Franklin, 1993]; i.e. the use of specialist TLS hardware. Here, speculative CMP is expanded with a dedicated thread control unit (TCU) coprocessor. Furthermore, inter-thread communication such as register dependencies is handled in hardware, and speculative epochs are stored within hardware for fast retrieval. The shortcomings of this research include the lack of consideration for resource contention, such that simultaneous requests to the TCU from multiple processor elements could overload the TCU. The scheme is also restricted to specialist speculative CMP, potentially restricting its adaptability to other, more common architectures, such as commercial non-TLS CMPs.

Research by [Akkary et al., 2008] proposed a new microarchitecture – the TLS CMP processor that is similar to the superthreaded architecture. Inspired by speculative multithreading (SpMT) and a derivative of the multi-

scalar study [Franklin, 1993], this scheme supports the use of simple in-order cores (similar to that of IBM Cell SPUs). Perhaps the most important contribution this paper makes is the use of a control independent predication coprocessor to spawn threads. The results are promising, albeit that the code patterns required perfect parallelism. Whether actual production code exhibits perfect parallelism is yet to be determined.

It is assumed that production code does not have significant coverage of perfect parallelism. More importantly, since commercial and high performance CMPs have limited or, more commonly, no TLS logic hardware support; the work described previously has limited applicability, as it applies to only speculative based hardware. Deep speculation code paths that require additional speculation hardware support further limits the authors' work.

Wang [Warg and Stenstrom, 2006] focused on single threads and on hardware resources to minimise dependency associated with a speculative thread (conventional threads also exhibit similar overheads, such as register dependency). Moreover, the authors extended the cache coherency protocol to support additional cache line states. Although, the study simulates a processor that meets their research requirements, it is unclear if their approach can be considered innovative or to have improved upon research that existed at the time. Manual and hardware implementations have been popular with many researchers, and another avenue of TLS parallelisation is the use of compilers as discussed below.

3.4.5 Compiler Support for Thread-Level Speculation

This section will briefly evaluate common speculative compilers. The intention is not to provide an in-depth theory of compilation as this is beyond the scope of this study. The reader is referred to [Kennedy and Allen, 2001, Aho et al., 1986, Grune et al., 2000] for further information on compiler theory.

Despite considerable progress that has been made in the field of auto-

matically parallelising regular numeric programs, compilers have struggled to automatically parallelise non-numeric (irregular) programs. This is due to complex control flow and memory access patterns [Steffan et al., 2000]. Study by [Bhowmik and Franklin, 2004] implements a scheme based on an established speculative compiler, the SUIF compiler [Amarasinghe et al., 1995,Wilson et al., 1994]. The compiler translates the source into SUIF Intermediate Representation (IR) followed by various optimisations and produces an annotated program with profile information that is finally partitioned into threads.

The SUIF compiler can only execute multiple instructions on specific processors and make use of a profiler that is limited in determining the most likely paths in a program and which estimates the possible number of dynamic instructions per procedure, including function calls and iteration count within a loop. The authors use profile data and implement their own algorithms for thread generation, inter-thread data dependence modelling and program partitioning. It is assumed that the algorithms complement the SUIF compiler. However, it is not clear whether such a scheme is scalable and portable to non-speculative hardware.

Madriles et al [Madriles et al., 2008] explored the Mitosis framework [Gonzalez, 2010] in their studies. This framework is similar to Hydra and SUIF, in that it is a combination of a compiler and speculative hardware support [Hammond et al., 1998, Olukotun et al., 1996, Wilson et al., 1994, Amarasinghe et al., 1995,Garzaran et al., 2003] which enhances speculative threads. Interestingly, the Mitosis compiler generates speculative threads with precomputed thread data, such as register and memory values known as speculative pre-computation slice (p-slice). The framework utilises a speculative processor with additional speculation hardware support. The approach provides a tight-coupled architecture, where software and hardware work in cooperation; however, the research provides no benefit when current modern processors and micro-architectures are used. Hence, the proposed scheme is

not transferable, scalable or viable for the current generation of processors.

Moreover, [Miura et al., 2003] reported an intuitive design, which again is similar to designs that predate its report. The hardware manages execution and synchronisation of the speculative threads, and the compiler inserts the synchronisation points [Zhai et al., 2002], the speculative threads assist hardware control speculation to ensure validation of threads execution (i.e. validates data) and is followed by synchronisation that reduces misspeculation overheads from a software perspective. The drawback of this approach is that it relies heavily on hardware support. Indeed, the scheme can only be implemented on a multiscalar architecture, making the research somewhat redundant. The redundancy is due to multiscalar architecture not being available for research or commercial use, and its algorithms are solely dependent on hardware that is not transferable to alternative microarchitectures.

In [Liu et al., 2006] the authors detail another TLS based compiler known as POSH. This compiler embeds a significant amount of TLS instruction into the original binary code. Each TLS instruction is analysed further (the refinement stage), then the POSH compiler decides which TLS instruction is executed from the binary code. The POSH compiler profiles data to eliminate non-beneficial data and value prediction, see Section 3.4.6 – to determine possible speculative thread data. However, POSH is system specific and makes multiple assumptions of target hardware such as enforcement of dependency policy between tasks, yet POSH does not assume any specialist hardware to support transfer of data passed between registers, whereby that all data transactions are communicated through memory. The compiler also requires hardware to resolve dependency violations through memory squash and restart tasks accordingly.

Here, the compiler (POSH) spawns and commits instructions but hardware takes a considerable amount of responsibility for data dependency resolution. The results from their research show a cumulative growth in all

benchmarks, but it only seems natural progression such that the number of instructions increases, so will the number of commit instructions (completion of task). Therefore, POSH only scales logarithmically, which results in a deterministic growth as their results indicate. Their research does not indicate hardware overheads and performance impacts. Speed-up was calculated based on number of instructions over processors but how their research is applicable to multi-node CMP units is yet to be determined and in order to use the POSH compiler, TLS hardware is required which is unavailable for research or commercial use.

Wu et al [Wu et al., 2004] explored the open research compiler (ORC); the authors' work introduced compiler functional features and its internal workings. ORC is designed to work with popular front-end languages such as GNU C, FORTRAN and OpenMP. An addition to the compiler's traditional feature set was the use of control and data speculation support – see Section 3.4.6. Speculation improves instruction level parallelism (ILP) for conventional symmetrical CMPs with or without TLS-hardware support [Mahlke et al., 1992]. ORC was designed to match conventional architecture (i.e. Intel) specifications and includes the use of the Open64 platform [Lin et al., 2004]. The report gave no indication of whether the compiler is portable to many NUMA and/or non-NUMA platforms and lacked a clear explanation or demonstration of the compiler's capabilities in a multicore system. Whether ORC is a commercially viable and stable platform has yet to be determined. However, this research has potential and has been used extensively within the academic community [Du et al., 2004].

Another intuitive proposal by [Ro and Gaudiot, 2005] was inspired by the work of [Rabbah et al., 2004]. The proposal by [Ro and Gaudiot, 2005] describes SPEAR – speculative pre-execution assisted by compiler, and focuses in particular, on the issue queue that is utilised in superscalar architectures. The queue broadcasts (issues) operations (instructions) and the activated (selected) instructions are executed. Typically, an issue queue is a centralised

structure, and a large queue results in high hardware complexity that, ultimately, reduces the operating clock rate. To mitigate this, the paper implements the SPEAR compiler [Rabbah et al., 2004], which more effectively controls the issue queue. Despite performance gains, such an implementation requires hardware changes, more specifically circuit design changes, to achieve for non-speculative execution. Furthermore, the applicability of the design to current CMPs remains unknown. Hence, while it is promising, the viability of the design is unclear and of little use for the commercially dominant conventional CMPs.

In [Dou and Cintra, 2007] the authors give a detailed analysis of speculative parallelisation using a TLS compiler. Speculation overhead and scheduling restrictions, including cost performance models of execution timings are outlined, yet it is not clear whether the approach is applicable to conventional CMPs.

To date, compilers have been extensively modified to derive TLS compilers for process automation. Crucially, TLS compilers convey data between speculative threads, that is to say they are enhancing and optimisation compilers that exhibit TLS-type detection and execution. For examples, see [Li et al., 2005, Wu et al., 2004].

Studies by [Steffan et al., 2002, Zhai et al., 2002] explores optimisation of scalar value communication through synchronisation and the subsequent value forwarding (i.e. critical forwarding path). The compiler initiates the TLS procedure when a data dependency occurs through inter-thread communication. Initially loops were profiled to determine possible speculative regions. Once speculative regions were identified, TLS codes were inserted (synchronisation and value forwarding). TLS codes were shown to also interact with the underlying TLS hardware. Finally, the compiler regenerated the binary with the optimisations using GCC into MIPS binary. The authors concentrated on loops and the compilers use of a forwarding frame.[6] It is

[6] A forwarding frame is the allocated memory within a stack that supports the commu-

similar to the synchronising scoreboard (SS [Krishnan and Torrellas, 1999b]).

In addition, the compiler inserts synchronisation atomic commands (wait and signal) that are masked as gateway commands. This complicates the code. The authors applied their optimisations to the SUIF compiler [Wilson et al., 1994] but it is not clear whether the optimisation would be workable for other TLS compilers. A similar study by [Zhai et al., 2004] also optimises compilers. Here, instead of scalar communication, the authors, elaborating on their previous work [Zhai et al., 2002], encapsulated memory-resident values. Once again, synchronisation atomic commands and a forwarding frame were used. However, TLS codes were applied to real memory addresses of scalar values, allowing a higher level of parallelism without degrading the overall performance as TLS was applied to actually memory addresses.

3.4.6 Speculation Types and Predication Techniques

This section will explore the use of speculation and predication techniques. There are four main types of speculation, namely data value speculation (DVS), control speculation (CS), data dependence speculation (DDS) and software-only speculation [Fu et al., 1998]. Earlier studies embedded speculation into the hardware [Marcuello et al., 1998, Sohi and Roth, 2001, Krishnan and Torrellas, 1998], from a practical perspective; this type of hardware use increases circuit complexity and cost.

Although early studies extensively used hardware TLS implementations – see Section 3.4.4, DVS is still commonly used to improve the performance of a program [Steffan et al., 2002] by speculating a predicated value for inter-thread communication. This is only applicable when a thread may consume a variable that has a potential dependency trait. Using a predicted value rather than an actual value, the consumer thread can continue its execution [Rul et al., 2007]. Before committing, the consumer thread compares its results with predicted values. If the predicted value was incorrect, a violation results,

nication of variables.

the consumer thread is squashed and restarted using the correct value.

Software value prediction (i.e. DVS) is explored by [Li et al., 2005]. The approach described yields value prediction compilation without hardware support. This was achieved by determining critical values. Once values were determined, the TLS codes were inserted (predictors) and speculative regions highlighted, similarly to the work described by [Zhai et al., 2002, Zhai et al., 2004]. There was reliance on speculative processors and speculative-type compilers (ORC [Wu et al., 2004]). While the results are promising and there are performance gains, such an implementation still requires a certain degree of speculation hardware support. Thus, further research is required to determine whether such a design is practical on non-speculative CMP hardware.

Prefetching strategies used for pre-computation are examined by [Rabbah et al., 2004]. DVS was used for prefetching data and the study presents detailed analysis of the implemented algorithms. The effects of excessive rollback of threads were not examined. This means that ameliorating parallelism through data prefetching could only achieve limited parallelism due to the increase of data and memory traffic.

Research by [Tubella and Gonzalez, 1998, Marcuello et al., 1998] examined CS that potentially increases parallelism through branch prediction techniques. This is considered to be the most widely studied CS. CS monitors the control flow of an application and attempts to accurately predict the future path of the program. The technique is well suited to loops, that is, iterations and the execution of the loops. However, CS is also widely used in non-multithreaded architectures, albeit it has not seen a wide adoption in multithreaded and multicore processors. The authors presented details of the application of CS to multithreaded processors. However, CS relies upon the compiler to accurately locate the loop and transform each loop with CS codes.

The use of their compiler on a four-context multithreaded processor with

concurrent threads is demonstrated. Unfortunately, the way data dependencies occur within the loops, particularly if multiple threads are concurrently executing, was not demonstrated in either article. Moreover, there was no clear indication how multiple threads (CS loops) are managed should a dependency occur. Hence, the approach works well, but only when loops do not have data dependencies. The scheme does not require specialist TLS hardware even though the authors implement a current loop stack (CLS) that stores the current and predicted pointers to branches within the loops, which could be a pure software approach or a hardware mechanism. To summarise, the authors presented a unique CS-compiler based approach. However, for modern code patterns containing simple and/or complex dependencies, the approach is not applicable, partly because current multicore processors do not have the additional logic required for CS.

DDS refers to techniques that execute parts of code, exclusive of a comprehensive knowledge of data dependencies. Dependencies are typically identified in both compiler and hardware; in particular [Marcuello et al., 1998] attempted to extend DDS and used them together to speculate on dependences through memory. Memory references for which effective addresses have not been calculated or are unknown are referred to as ambiguous references. They introduced a novel architecture, the dependence speculative multithread architecture (DeSM), which attempts to predict inter-thread dependences when they are unknown for both register and memory dependencies. It is currently unknown how well the DeSM is able to cope on a true multicore processor, without any DeSM logic support.

3.5 Summary

This chapter has provided a background to TLS, that has existed for many years, in many different schemes and implementations that share a common theme of resolving dependencies. The use of TLS compilers and TLS

hardware has predominated prior research. Whether bespoke and specialist CMP or general purpose CMP was used, most research was conducted on symmetrical CMPs (with or without TLS hardware support). There has been no reported research that uses asymmetrical hardware for TLS or for implementations that resemble the TLS philosophy. As industry is moving towards energy-efficient processors and minimising their use of hardware that requires complex speculative support [Park et al., 2003,Kahle et al., 2005], it is placing more cores onto a single chip processor, emphasising power-saving logic without compromising on performance. TLS hardware has become less popular on conventional processors. It is believed that a lightweight speculation or TLS derivative schemes must be derived from software that works with generic hardware support.

The use of CMPs without TLS hardware (exploitation of on-chip hardware) has been reported. However, conventional CMPs do not support TLS hardware albeit that they provide some form of speculation [Kahle et al., 2005]. TLS processors that do implement deep speculation attempt to increase the number of transactions to cover memory latencies. An important observation is that speculative execution is inefficient. This inefficiency is due to the extra computation needed to preserve correct speculation. This, in turn, is proportional to the extra dynamic power needed because of the poor use of available conventional hardware support. As discussed in Section 3.4.4, such hardware requires the use of additional hardware and can be expensive and also increases power consumption. Current processor designers are more energy conscious and modern processors aim to use less power than their predecessors, yet have high compute-throughput with a significant reliance on software optimisation techniques [Shen and Lipasti, 2006].

As traditional cache-based memory hierarchies require data cache access, this creates a significant burden on the design [Zheng et al., 2011]. Data cache-hit and cache-miss detection embedded in the time-critical path requires page translation. This means that tag array contents must be com-

pared, to determine a cache hit. Attempting to overcome cache hit and miss, speculation is used together with an application program. This chapter has explored speculation techniques including hardware-based speculation, which increases complexity and dynamic power dissipation and requires many timing critical paths throughout the circuitry design. Speculation also impacts operation latency, circuit timing and design complexity and confounds data cache behaviour and code generation in the compiler. Finally, this chapter has introduced elementary terminology of the Cell Broadband Engine processor. Chapter 4 introduces the Cell processor and its use with software support to create a dynamic recovery system (DRS), which follows a very similar approach to TLS; however, the DVS is encapsulated in the Lyuba framework. The framework underlying philosophy is based on TLS when data recovery is initiated.

Chapter 4

Computer Architecture and the IBM Cell Broadband Engine

After describing thread-level speculation (TLS) in the previous chapter, this chapter will investigate the Cell Broadband Engine (BE) processor and highlighting the significant differences between a Cell processor and the conventional architecture of a chip multicore processor (CMP). However, this chapter will not focus on speculative processors, as these CMPs are not available for this study and are beyond scope for this chapter (see Chapter 3 for a brief overview of past TLS research based on speculative hardware).

After reviewing the Cell BE, the following chapter will delve into the experimental framework designed for the Cell processor. Section 4.5 gives a brief overview of the Cell processor considering the components of the Cell BE chip that are directly relevant to this study. Information on the other components of the Cell chip can be found in the referenced material. Previous research has focused on extracting parallelism from a higher level such as data- and thread-level parallelism.

This chapter will focus on the fundamental primitives of parallelism followed by a summary of Amdahl's Law and then a brief overview of synchronisation, an outline of symmetric and asymmetric architecture and finally

the IBM Cell processor itself.

The history of microprocessor architecture has been one of making use of an ever-increasing number of transistors to provide higher levels of performance. Microprocessor architectures including both symmetric and asymmetric multicore processors have been established for a considerable amount of time and now dominate most desktop and server environments.

Current symmetric multicore processors encompass a typical homogeneous design where all traditional processor elements (PEs) are coupled together on the same silicon chip (SoC[1]). These conventional CMPs are now increasingly facing performance limits such as memory latency, bandwidth and power constraints. Moreover, increasing the processors frequencies (clock rates) and deepening pipelines have shown diminishing results [Oracle, 2005, Laudon and Spracklen, 2007]. The other side of this architecture spectrum is asymmetrical multiprocessors such as the IBM Cell processor – see Section 4.4. The IBM Cell design implements many simpler cores that inherently provide high parallel performance as opposed to complex cores that provide a respectable serial performance.

In [Balakrishnan et al., 2005] the authors conclude that the performance of a symmetric CMP is beneficial regardless of whether the applications have parallel or serial regions of code. Moreover, the authors state that high-performance cores are still required, which could further increase the speed-up of both parallel and serial code. The popularity of symmetric homogeneous processors remains due to time-to-market pressures so reusing hardware designs allows non-recurring engineering costs; so simplifying an already complex design and programming for a symmetric processor is typically easier than an asymmetric processor such that all cores on a symmetric CMP are identical [Halfhill, 2007].

Both symmetric and asymmetric processors are going through dramatic changes by extracting more performance from each complex and power-

[1]System-on-chip.

hungry core, which has become a difficult proposition. However, incorporating multiple cores on a silicon die is relatively modest due to the continuing advances in process technology, therefore major microprocessor vendors are marketing multicore (CMPs) with two, eight and more cores; such microprocessors allow and support many threads executing simultaneously [Cho et al., 2008]. Clearly, packaging additional transistor counts within a processor core no longer increases performance; hence multiple cores are packaged into a single processor causing a dramatic increase in performance [Guccione, 2008].

4.1 Instruction-Level, Data-Level and Thread-Level Parallelism

This section will briefly explore the basic building blocks of parallelism, starting with instruction-level parallelism (ILP) [Laudon and Spracklen, 2007]. ILP is the lowest level to achieve parallelism by issuing multiple instructions per clock cycle [Lo et al., 1997]. The ILP technique has been widely researched [Hennessy and Patterson, 2007], with ILP being the primitive instruction issuing mechanism in superscalar processors. With ILP, there are known hazards that prevent an instruction in the instruction stream from executing.

Table 3.2 identifies the three fundamental hazards that are fully described in [Zaharieva-Stoyanova and Jantschii, 2003]. Both structural and control hazards are recovered in hardware and are typically not visible to programmers, but a data hazard is visible to the application layer. A great deal of research continues to investigate this particular area. By focusing on techniques such as speculation to overcome data hazards, many proposals have been introduced including speculative processors (see Section 3.4.4). Control hazards can cause a significant loss in performance. Moreover many schemes have been proposed from freezing or flushing the pipeline, to delay the branch

process [Hennessy and Patterson, 2007].

The data-level parallelism (DLP) [Pericas, 2003] paradigm uses vector instructions (vectorisation techniques) to concurrently execute a single instruction as multiple instances. Such a technique can produce a large amount of parallelism for many consecutive cycles [Espasa and Valero, 1997]. Another form of parallelism is thread-level parallelism (TLP), which is considerably more visible than ILP to a programmer. A programmer is able to create, control and destroy threads in TLP, while ILP is completely controlled within the CPU hardware. A thread is a unit of work with its own instructions and data. ILP exploits implicit parallel operations: it explicitly exploits multiple threads of execution that are intrinsically parallel.

TLP has been widely exploited, and its application to multiprocessors has been widely researched [Hennessy and Patterson, 2007]. The Cell architecture exploits ILP with scheduled power-aware multi-issue microarchitecture [Gschwind, 2007], and simultaneously the architecture supports the scheduling of parallelism between multiple execution units, which naturally allows dual instruction issue for both types of processing cores.

A detailed review of the above three forms of parallelism is beyond the scope of this study. However, most of the work on TLP has so far been applied to symmetric multiprocessors (SMPs); this study will extend TLP to an asymmetric multicore processor. The critical importance of TLP is to allow multiple threads of execution to exist and share functional units of a single processor. This form of sharing is similar to that of hardware independent pipeline overlapping [Hennessy and Patterson, 2007].

With ILP reducing memory latency by concurrently servicing multiple outstanding cache misses. However, when clusters of cache misses occur, this results in the CPU to reload those caches effected by cache miss by a sequence of memory accesses. This increases the memory latency and can potentially halt program execution on the processor until the cache has the correct data.

To reduce this limitation, the Cell cores support a stall-on-policy that allows applications to initiate multiple data cache reload operations through the use of simple deterministic scheduling rules such as overlapping memory accesses precipitately of data use. A new form of parallelism has been integrated into the Cell architecture, known as compute-transfer parallelism (CTP).

CTP exploits available memory bandwidth more efficiently by decoupling and parallelising data transfer and processing. CTP considers data movement as an unequivocally scheduled operation controlled by the program to improve data delivery frequency [Gschwind et al., 2007]. Furthermore, CTP independently sequences and targets block transfers of up to 16 KB compared to software-directed data prefetch that is only able to access small amounts of data per prefetch request. This study will focus on threads of execution and more specifically communication between threads on hardware.

4.2 Automating and Manual Parallelism

Section 4.1 briefly outlined different forms of parallelism mechanisms. The ability to automate this process is considerably more desirable and removes the burden of parallelising applications. However, to apply automation, the application itself must have the following three characteristics before automation can be easily parallelised: regular data access, few unpredictable branches in the instruction stream and comparatively independent tasks or phases (such as database applications). Such automation is encompassed within a compiler.

Applications such as dense matrix (floating point) applications tend to have a large degree independent processing and are suitable for automated parallelisation. It must be noted that the automation of parallelisation (within a compiler) is more suited to an application in which the write accesses to data form some sort of pattern. This pattern could naturally exist

or can be made such that the application can be distributed over independent memory locations [Prabhu, 2005].

There are a considerable number of applications that are difficult to parallelise such that a pattern does not present itself in the program structure. This is typically caused by applications with irregular control flow and complex data access. Furthermore unpredictable data access with dependent variables (dependencies) and branching, such as integer applications, are clearly not suitable for automated parallelisation. Therefore such applications with limited parallelism were implemented using data structures and algorithms that obscured inherent parallelism. The approach that a programmer chooses and the use of algorithms with low TLP can obscure attainable parallelism, in particular the use of recursion and iteration. Iterative loops such as `For` loops tend to exhibit control flow and data parallelism with each iteration.

All but the final iteration will occur regardless of the task being processed by the previous iteration. This entails the computation generally remaining independent between iterations. Conversely, recursion simplifies programming whereby control flow and data dependency depends on the results from the computation in the previous portion of the recursion, which is inherently difficult to predict and parallelise.

Applications with obscured parallelism have often been implemented on a uniprocessor platform. Applications targeting uniprocessors were able to frequently reuse variables; for example, applications implementing a stack-based algorithm with small working sets typically yields good data locality. Therefore, a programmer who attempts to extract parallelism from an application that is not inherently parallelisable must either implement a framework or redesign the algorithm to achieve parallelism. Furthermore, microprocessors that are multicore in design require algorithms to be designed to harness the increased computing resource even if this produces slightly less efficient and/or more complex code.

4.3 Microprocessor Architecture

It is well known that microprocessor designers can no longer just rely on higher clock rates, deeper pipelines or instruction-level parallelism (ILP) for meaningful performance gains [Claydon, 2007].

The need for energy efficient processors that operate with a low thermal envelope dissipated of a uniprocessor has continued to be a primary factor in the design and development of a microprocessor. Conversely, the impact performance imposed by memory latency and bandwidth, power and increase in chip size have resulted in diminished returns from increased processor frequencies achieved, by reducing the amount of work per cycle while increasing pipeline depth. Microprocessor development and fabrication must take into account the memory wall [Wulf and McKee, 1995] whereby latencies between memory and higher processor frequencies are increasing and the latency is becoming a definite limiter. For example, a multi-GHz processor is measured in the hundreds of cycles, combined with symmetric architecture with shared memory and main memory, effects the latency that can tend towards thousand processor cycles (or more).

Current and past research has a tendency to deploy frameworks on commercial and/or commodity multicore or symmetrical processors with conventional sequential programming semantics that sustain only a limited number of concurrent memory transactions.

In a sequential model, the assumption is that each instruction is completed before execution of the next instruction per thread. If data or instruction fetch issues are missed in the caches this would impact and result in access to main memory.

When TLS is incorporated into the runtime execution and some schemes requiring hardware support to maintain a speculative state can exacerbates power consumption, increased heat dissipation and resource utilisation. Moreover, a non-speculative state has to be maintained in order to safely continue processing. Dependencies occurring from any previous states that are missed

in caches require an even deeper speculation which results in significant overheads [Kahle et al., 2005]; overheads include state administration, recovery and the probability of useful work that is speculated decreases rapidly as processing time continues.

The IBM Cell architecture aims to improve the effective memory bandwidth achievable by improving the degree to which software can tolerate memory latency [Flachs et al., 2007]. Memory bandwidth limitations are latency-induced, and increasing memory bandwidth at the expensive of memory latency can be counter-productive. Therefore, microprocessor designers must recalculate and design the organisation of a processor that allows an increased memory bandwidth that effectively allows increased number of memory transactions in flight to reduce the memory wall effect [Kahle et al., 2005].

An associated factor to microprocessor design is power density in CMOS processors which has steadily increased and is proportional to processor frequency; moreover dual issue pipelines can sustain large number of instructions per cycle. However, processor frequency cannot fully be realised due to increased pipeline depth and power limitations, which equates to execution latency that may degrade performance. This is accompanied by an increase of data content which requires a large amount of processing, and so microprocessor designers clearly needed to rethink an alternative architecture to meet the demand of this increased computation requirement, hence multicore came into the mainstream [Gorder, 2007].

Microprocessors are unable to achieve their potential if there is not a constant data stream of work in the form of computer instructions. Anything that interrupts the flow of instructions to the microprocessor undermines the power of the processor [Stallings, 2009]. Microprocessor architects have developed many techniques to assist instruction processing, such as branch prediction, data flow analysis and speculative execution. Such sophisticated techniques are made necessary by the sheer power of the processor, and

these techniques attempt to assist the exploitation of the raw speed of the processor.

An important aspect in the understanding of resource availability for software engineering is the architectural composition of the target hardware, the IBM Cell. The Cell was developed using the state-of-the-art 90-nm process with silicon-on-transistor (SOI), low-k dielectrics and copper interconnects [Yang et al., 2004]. The microarchitecture incorporated many flexible elements such as reprogrammable and reconfigurable synergistic processors and/or input/output (I/O) elements that support many system configurations with one high-volume chip [Gorder, 2007].

Multicore processors are not a new concept but have existed in the academic field, in particular in the scientific environment [Geer, 2005]. Compared to a uniprocessor, a multicore processor allows multiple tasks to operate in true concurrency. However, the complexity of an individual core depends on the chip manufacture. The composition of processor interconnections and architectures is well established from symmetrical multiprocessor (SMP) to chip multicore processors (CMPs), but memory access from internal and external forces remains an important issue, and more specifically, the way programs can make effective use of available computing resources while trying to ensure that memory use remains optimum from a programmer's perspective is a particularly important issue.

Chip designers continue to improve capabilities and features and to increase the number of physical cores on a single processor; however, limitations still exist which include increasing the issue rates of a processor, which requires an increase of fetch instructions for data from memory. Moreover, the fetch access per cycle and branches per cycle need to increase. Further increases in the complexity of hardware could adversely reduce the maximum clock rate. Another key criteria is the power issue [Pedram, 1996] such that increasing the complexity of the circuitry and increasing the logic set would directly affect power consumption, which is a function of both static power

(power growing proportionally to the transistor count) and dynamic power (product of the number of transistors switching between states and the rate at with the switch occurs) [Hennessy and Patterson, 2007].

Increasing the efficiency by extracting a greater degree of parallelism is limited by power, and such a constraint has been considered and it is widely accepted that modern microprocessors are primarily power limited. As stated in Section 3.5, speculation does provide additional assistance and support for parallelism but the inefficiency of speculation overheads and the power issue from speculation support has resulted in limited or no support for, and integration of, speculative mechanisms in commercial multicore processors.

The performance between the processor and system memory has grown [Mahapatra and Venkatrao, 1999]. However, the emergence of compiler technologies attempt to enhance performance using instruction scheduling by exploiting pipelined architectures and handling register allocation to reduce the impact of processor-memory differences, and optimisations is beyond the scope of this thesis, but these topics are more fully described in [Hennessy and Patterson, 2007] and [Pas, 2002].

The next subsection will briefly explore Amdahl's law and its application to modern programming paradigms and processors and whether its application is still relevant (including applicable to asymmetric architecture).

$$S(p) = \frac{1}{s + (1 - s)/p} \tag{4.1}$$

4.3.1 Amdahl's Law

The application of Amdahl's law (see Equation 4.1) is widely cited, in fact Amdahl's law has become one of few laws in computer science concerning the percentage of a serial processing relative to the overall program execution time using a single processor.

In Equation 4.1, s represents the serial fraction and $1 - s$ represents the fraction of the application that can be parallelised. With P representing the

number of processors used to achieve the greatest speedup.

The law is independent of the number of processors available in the system [Shi, 1996]. Amdahl's law states that the attainable speed-up of a parallel algorithm is constrained by the percentage of the algorithm that is performed sequentially.

In [Koivisto, 2005], the authors attempt to apply the law to multicore processors and their article attempts to outline potential parallelism or using an assumption that no natural parallelism exists. Therefore, eliminating internal dependencies that typically exist in most algorithms potentially creates parallelism. In some cases, the original algorithm is parsed and transformed into a new algorithm to satisfy parallelism prerequisites [Trinder et al., 1993].

The principal use of Amdahl's law is to show the potential speed-up from a design perspective, equating to potential performance increase. If the speculative speed-up value is attainable and significant in value, then the algorithm should be considered for parallelism. However, measuring the performance (speed-up) is more complex for multicore architectures. The original incarnation of the law did not consider a finite number of processors, so an infinite number of processors can be used (the `N` argument in the speed-up equation [see above] is independent of the architecture). Furthermore [Koivisto, 2005] states,

> sequential performance has been historically easier to predict by examining code execution profiles, or timing code embedded for each execution unit.

Clearly, sequential code is considerably easier to analyse. However, when code exhibits dependencies, the analysis is complicated. An important criterion is that Amdahl's law considers code that is naturally parallelisable and does not take into account factors such as synchronisation and message passing.

Moreover, measuring and predicating the execution times for these factors is particularly difficult. This law has been applied in nearly every research

study in the previous chapter. The law itself has become quite adaptable to symmetrical homogeneous processors. However, as stated earlier, Amdahl's law does not take into account the factors, which are significant for asymmetric architecture. Some research has been undertaken to apply the law to asymmetric heterogeneous processors [Moncrieff et al., 1996, Paul and Meyer, 2007, Hill and Marty, 2008].

In [Paul and Meyer, 2007] the authors give an excellent discussion and emphase that Amdahl's law is based upon two assumptions – boundlessness and homogeneity; hence Amdahl's law could fail to be applicable to a single chip heterogeneous processor. Due to these limitations, [Paul and Meyer, 2007] states

> improvements are presumed to be isolated from one another. However, when resources must be viewed as a trade-off within a bounded (finite) space, this assumption no longer holds

This research study concurs with the analysis from [Paul and Meyer, 2007]. The Amdahl's law equation is too simple and does not take into account significant factors such as synchronisation and message passing, which are critical to any parallel execution. Ignoring these factors can hinder performance for an optimal designed algorithm for both symmetric and asymmetric microarchitectures. Amdahl's law highlights the upper bound of achievable speed-up but in reality a much lower speed-up is achieved (and in some cases no speed-up is achieved).

4.4 Synchronisation

This section will briefly outline the importance and application of synchronisation. This subsection supports the conclusion of the previous subsection and highlights the significance of synchronisation that a future speed-up equation must take into consideration. Parallelising an existing or new algorithm that must execute concurrently on CMP will inherently require some

type of synchronisation to control the relative order of thread execution and manage shared data [Roberts and Ahkter, 2006]. A significant amount of research has been conducted in regards to synchronisation [Cintra et al., 2000].

Common synchronisation primitives include mutual exclusion (mutex), semaphores, conditions and signals. This section will only investigate both mutual and condition synchronisations and will refer the reader to Section 4.5.7 for signals. Mutual exclusion is the process of protecting shared data multiple accesses by multiple threads. The use of mutexes helps to preserve data correctness and the integrity of the concurrent system [Butenhof, 1997]. The other cases are the condition primitives, which are used to communicate the condition of a particular state or region of data to other condition variables.

An example of such use is to signal a thread whether it can continue or must halt execution, depending on the signal type. This brief encounter with synchronisation is extremely vital to ensure that concurrent and halted threads are able to manipulate data correctly, preserving data and system integrity. Synchronisation is also embedded into speculative systems – see Section 4.4.

Table 4.1: Cell processor architecture overview.

Specification	Clock Rate	Bandwidth
Running speed	4 GHz+	
Memory bandwidth		25.6 GB/s
I/O bandwidth		78.6 GB/s
Single precision	4 GHz	256 GFLOPS
Double precision	4 GHz	25 GFLOPS
Transistors		235 Million

4.5 IBM Cell Broadband Engine

This section will investigate a subset of the Cell BE hardware components that are relevant to this study. Table 4.1 shows a brief specification of the Cell processor, but for a detailed analysis, see [Kahle et al., 2005, Shimpi, 2005, Blachford, 2005, Takahashi et al., 2005, Gschwind et al., 2006, Gschwind, 2006, Chen et al., 2005, Gschwind et al., 2007].

The IBM Cell Broadband Engine (CBE) is a heterogeneous multicore processor designed by IBM. The Cell architecture exploits multiple levels of system parallelism including IPL, TLP, DLP and compute-transfer parallelism (CTP). The fundamental characteristic that separates a Cell CMP from a conventional CMP[2] is the physical components, programmability and an asymmetric architecture [Kahle et al., 2005]. IBM collaborated with SCEI (Sony Computer Entertainment Incorporated) group and Toshiba, forming an alliance known as the STI[3] group. STI is responsible for the content, development and manufacturing of the Cell processor [Kahle et al., 2005]. The Cell BE processor was originally designed for media applications such as gaming, image processing, hi-definition television and computation-intensive applications [Srinivasan et al., 2005].

Interestingly, the unique characteristic of the Cell processor is the architectural composition of an SMT PowerPC core known as the PowerPC element (PPE), which is a modified general-purpose processor with eight[4] vector processor elements known as the synergistic processor element (SPE[5]). SPEs are designed to handle computationally intensive tasks with both types of processor executing in-order instructions. Moreover, the SPEs are independent processors, each running an independent application thread. Both

[2]Intel Multicore Processor.

[3]SCEI-Toshiba-IBM.

[4]Typically eight SPEs are found on the first- and second-generation Cell processors. The processor in the Playstation 3©has six SPEs due to manufacturing yield and one SPE is utilised by the Sony operating system.

[5]Also referred as the Synergistic Processing Unit (SPU).

processor core types share access to a common address space such as main memory and visibility of each others processor resources through direct memory addresses. Resources such as local store's, control registers and I/O devices. Moreover, the Cell architecture ensures data type compatibility across all processor elements, ensuring that operation semantics are respected to allow efficient communication of shared data types.

With high performance computing (HPC) processing exponential amounts of data in a manner that can be partitioned to enable parallel execution with a greater focus on throughput than general-purpose threads with complex branching schemes, the Cell processor fits into such an HPC environment. Having multiple lightweight threads and optimised software pipelines present both in media-rich and scientific applications allows improved and effective hardware utilisation. Moreover, such software design characteristics allow improved memory and bandwidth and utilisation compared to commodity memory systems. Interestingly, software that executes on processors that are typically designed to accelerate a single thread of execution by taking advantage of ILP, however, are less able to derive benefit from the new memory systems [Flachs et al., 2007].

A principal limiter to processor performance is memory latency whereby modern processors can typically lose up to 4000 instruction slots while waiting for data from main memory. Past designs attempted to reduce the wait cycle by increasing on-chip caches and reorder buffers that reduce the average latency, also maintaining instruction throughput while waiting for data from cache misses. Large caches are able to deliver high hit rates on large data structures, but the footprint of large on-chip caches consume large amounts of area that could be utilised for computational elements, and the reuse rate for much of the data is low. When a cache miss occurs, the processing can continue through branch predication to fill a reorder buffer but they are difficult to construct if they are to be large enough to continue through a main memory access [Flachs et al., 2007].

As processor performance becomes power limited, leakage current becomes an important performance issue. Performance per transistor is the principal motivation for heterogeneity, with each transistor now being only a few atomic levels thick and the channels being extremely narrow. These features improve transistor performance and increase transistor density, but they tend to also increase leakage current, which is proportional to the area of the processor. Vendors are continuing to try to extract more performance per transistor, but since the performance efficiency of caches and reorder buffers diminishes as the size increases, another approach is necessary.

The Cell presents unique communication mechanisms (element interface bus [EIB] and the memory flow controller [MFC]) as well as asymmetric design. Importantly, the Cell BE supports many system configurations within a single Cell BE chip. The principal goal that the Cell BE embodied was to improve the following factors: increase performance by reducing memory latency, increase bandwidth, improve efficiency of power use and function at lower operating temperatures [Kahle et al., 2005].

A design principle of the Cell processor was to enable it to run at high frequency with modest pipeline depths and to limit mechanisms such as register renaming and to have highly accurate branch predictors that are typically accommodated in conventional multicore processors. Reducing such mechanisms reduces architectural complexity where feasible, subject to latencies from basic resource decisions such as the large register file (2 KB) and a large local store (256 KB). All data transfers placed on the Cell's bus (EIB) are quadword aligned, thus eliminating complex data alignment execution that is typically associated with scalar data access simultaneously reducing the number of cycles needed to calculate the memory address of both data and instructions. For optimal data transfers, all data is aligned at 16 bytes, which can only be specified through software.

Current CMPs have hit many performance barriers such as memory latency and reduced memory bandwidth [Wulf and McKee, 1995], with sym-

metric shared memory processors exhibiting latencies near to that of a thousand processor cycles [Drepper, 2007] with frequency scaling diminishing and deeper pipelines being exhausted. The Cell BE has been designed to alleviate these problems and as a result the design of its CMP does not resemble or reflect current conventional CMPs.

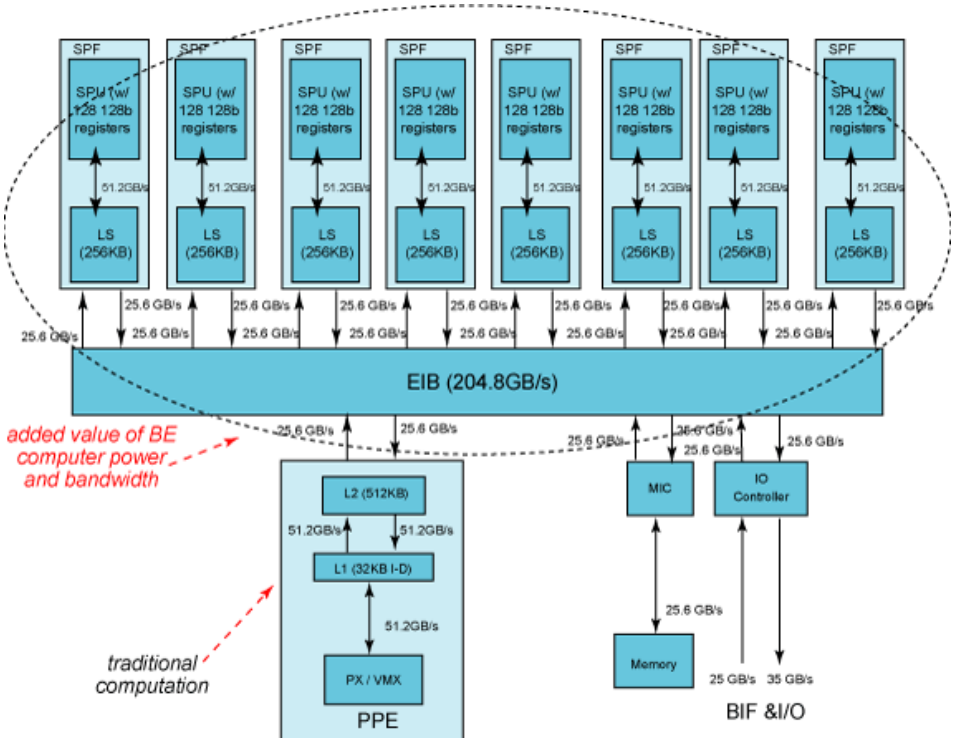


Figure 4.1: Cell Broadband Engine block diagram [IBM, 2007a, IBM, 1994]. Note: The EIB consists of four 16-byte-wide data rings: two running clockwise. Each ring potentially allows up to three concurrent data transfers, provided their paths do not overlap [Scarpino, 2008]

.

The Cell's design reduces control logic and circuit complexity and increases power efficiency [Blachford, 2005, Takahashi et al., 2005]. Figure 4.1 shows a block diagram that highlights the principal components that form

the Cell processor: the PowerPC processor element (PPE), eight synergistic processor elements (SPEs), the element interconnect bus (EIB), the memory interface controller (MIC) and the I/O interface.

The internal communication mechanisms allow the PPE to communicate with the SPEs through privileged-state and problem-state memory-mapped input/output (MMIO) registers supported by the memory flow controller (MFC) for each SPE. The Cell supports additional mechanisms for internal communication such as mailboxes, signal notification registers and direct memory access (DMA) [Bai et al., 2008].

Both PPE and SPEs share address translation and virtualisation memory architecture to support dynamic system partitioning. Moreover, the processors share system page tables and system functions such as interrupt management and data type and operation semantics to allow efficient data sharing among the processor elements to sustain data type compatibility. Moreover, it avoids duplicating capabilities across all execution contexts, hence using resources more efficiently.

Table 4.2: PowerPC Standard Version 2.02.

Specification
32-bit and 64-bit modes of operations – big-endian
Segments and pages allocated in virtual memory
64-bit effective address (EA) wide
32-bit instructions and word-aligned
64-bit wide registers
32x 64-bit general-purpose registers (GPRs)
32x 64-bit floating-point registers (FPRs)

4.5.1 PowerPC Processor Element

A clear distinction must be made between the following two terms: PowerPC element (PPE) and PowerPC. PowerPC refers to the microprocessor architec-

ture and not to a specific processor (IBM, 1994). The PowerPC architecture defines standards that depict the characteristics that all PowerPC systems must adhere to – see Table 4.2, which lists a few of these requirements. The PPE[6] is a dual-threaded (two concurrent hardware threads), 64-bit RISC general-purpose processor that conforms to the PowerPC Architecture version 2.02 [Frey, 2005, Gschwind, 2007], and the PPE delivers system-wide services, such as virtual memory management, exception handling, thread scheduling and other operating system services.

Furthermore, the PPE exhibits traditional processor characteristics such as a memory subsystem with separate L1 instruction and data caches (32-Kbyte each) and a unified 512-Kbyte L2 cache. The PPE is responsible for the main execution thread (main thread) and for managing the synergistic processor elements (SPEs). The additional coprocessors (SPEs) perform specialised execution which accelerates the overall execution of a program or task; for a more detailed analysis of the specification and standards, see [Frey, 2005, Scarpino, 2008].

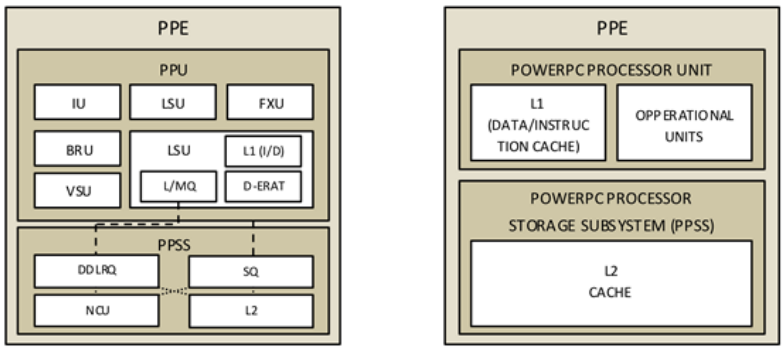


Figure 4.2: PowerPC Processor Element Block Diagram. (A) illustrates a simple overview of an PPE. (B) shows the memory subsystems on the PPE.

[6]Also referred as the PowerPC Processor Unit (PPU).

Figure 4.2 (A) shows a simple overview of a PPE that encompasses all the components such as the PPU and PPSS. The PPU of the PPE is responsible for all processing while the PPSS is responsible for data storage needed by the PPU. Figure 4.2 (B) depicts the PowerPC Processor Unit (PPU), which performs the instruction execution, also equipped with an L1[7] instruction and data cache. In addition to the instruction execution, the PPU can load 32 bytes and store 16 bytes, independently and memory coherently per processor cycle [Johns and Brokenshire, 2007].

Conventional x86 CMPs also employ a standard cache hierarchy with a non-unified L1 cache. However, the key difference is the PPU PPSS memory subsystem and its relationship to the PPU [Scarpino, 2008, IBM, 2007a].

Software is utilised to partition and target general-purpose computing threads, operating system (OS) tasks and computational tasks to a processing core customised for that particular task. For example, the OS is executed on the PPE while computational tasks are mapped to the SPEs. Such differentiation allows multiple schemes whereby tasks are executed on either or both the PPE and SPE and optimised for their respective workloads, and this enables significant improvements in performance per transistor [Flachs et al., 2007].

4.5.2 Synergistic Processor Element

The SPE architecture reduces area and power while facilitating improved performance by requiring software to solve difficult scheduling problems, such as data fetch and branch prediction. Software solves these problems by including explicit data movement and branch prediction directives in the instruction stream. As the OS is not able to run on the SPE, it is optimised for user-mode execution. The Cell BE introduces the synergistic processor element (SPE) which delivers the majority of the Cell BE's compute performance. The SPEs are accelerator cores implementing a novel, ubiquitously

[7] L1 cache represents the closest memory to the processor, Level 1 being the nearest.

data-parallel computing architecture based on the SIMD RISC system and explicit data transfer management.

An SPE is an autonomous processor element that stores its program and data into its associated local storage (LS) memory (see Section 4.5.3) and offers a new direction of parallelism by supporting autonomous compute and transfer threads within each SPE. Each SPE is fully integrated in the PowerPC architecture and shares virtual memory architecture coupled with a synergistic processor unit (SPU) and a synergistic memory flow controller (MFC).

Each SPE thread has the capability to execute independent compute and transfer sequences. The kernel/application layer interacts with an SPE thread that controls the entire SPE heterogeneous core. The SPE thread is created within the Cell SDK environment and allows the application to control the state of an SPE. The SPE threads are able to fetch their data from system memory by issuing DMA transfer commands independently [Gschwind et al., 2007].

However, an SPE has not been designed to run a complicated system such as an operating system. Therefore the SPE incorporates techniques that attempt to limit latency by using techniques such as software pipelining [Gschwind, 2007].

The hardware within an SPE is limited and cannot sufficiently execute tasks that require branching prediction or advanced caching, nor does an SPE have an out-order execution, separate fixed-point, floating-point or vector registers, but each SPE has a unified register file which is commonly found in traditional and conventional processing elements such as the PPE and PowerPC processors. Hence, as the SPE does not support the execution of a complete operating system, it therefore cannot allow a multithreaded environment and only has access to its own 256-KB local store (264 bytes via DMA). SPEs are efficient vector processing units which have been designed for accelerating numerical computation, predominantly optimised for

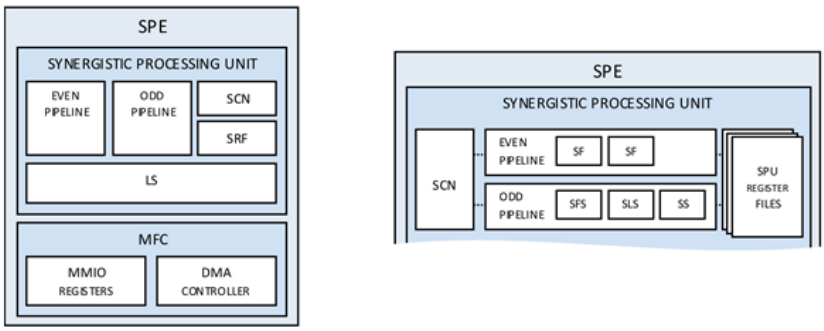


Figure 4.3: Synergistic Processing Element. (A) illustrates a simple overview of an SPE. (B) shows the odd and even pipeline on the SPE.

processing intensive application [Gschwind et al., 2007], more precisely

> SPEs in the Cell BE have been designed to bear the computational workload of an application [IBM, 2007b].

The SPE architecture is based upon the pervasively data parallel computing (PDPC) model, whereby the processor architecture exploits the wide data paths such as scalar and data-parallel SIMD execution on these wide data paths. Furthermore, having wider data paths potentially eliminates a considerable number of hardware overheads such as additional issue slots, separate pipelines and complex scalar units [Gschwind, 2007].

Moreover, the wide data paths in the SPE accommodate instruction messages from memory to the execution units available on the SPE. The SPEs adapt additional logic circuitry, which enables efficient communication with other processor elements (PE) through an advanced proxy controller known as the memory flow controller (MFC) (see Section 4.5.5); this proxy controller is leveraged and optimised for the Lyuba framework.

Figure 4.3 (A) shows a simplistic overall block diagram of a single SPE, clearly showing a unique design that is very unconventional compared to

a typical general-purpose processor. In fact, the SPE represents a similar design approach to a DSP processor, due to its distinct hardware logic, an on-board memory controller and the architectural design itself. Figure 4.3 (B) shows a more detailed view of the pipeline.

Table 4.3: SPU even and odd functional units.

Even pipeline	Odd pipeline
(SFX) SPU even fixed-point unit	(SFS) SPU odd fixed-point unit
(SFP) SPU floating-point unit	(SLS) SPU load and store unit
	(SSC) SPU channel and DMA unit

Each SPU is designed with an even and odd pipeline (see Table 4.3). The importance of this pipeline in relation to speculation is the ability to process, fetch and put data to and from memory, while traditional TLS based processors (see Section 3.4.4) require dedicated hardware. The Cell allows the programmer to create applications and use the on-board logic communication mechanisms for the needs of the application.

Table 4.4: MFC component description.

Component	Description
(DMAQ) Direct memory access queue	Storing up to 16 requests from the local SPU acquiesced through the SPU channel interface to the MFC. Also stores up to eight requests from remote SPEs and PPEs transferred through the memory mapped I/O interface.
(DMAE) Direct memory access engine	Controls the transfers of data blocks between the local store of SPE and system memory with transfers ranging from a single byte to 16 KB. For larger data transfers, the DMA is able to use DMA list commands that can be used to support non-contiguous data transfers.
(MMU) Memory management unit	Provides the ability to translate addresses between a processes effective address (EA) and the real memory address.
(RMT) Resource management unit	Enables locking of translations in the MMU and supports bandwidth reservation of the EIB.
(AU) Atomic unit	Enables snoop-coherent cache to implement load-and-reserve/store-conditional memory synchronisation that can be used to synchronise data between SPEs.
(BIC) Bus interface control	Assists the MFC to access the high-speed on-chip EIB. Also provides access to memory-mapped registers that provide an interface to distribute DMA requests from remote processor elements (PEs) and updates the virtual memory translations and to configure the MFC.

Table 4.4 outlines functional units that support the SIMD-RISC instruction set [Johns and Brokenshire, 2007]. The remaining functional unit, the SPU control unit (SCN), is associated with both pipelines, in addition to the SPU register file. The SCN performs management operations such as fetching and transmitting instructions to the functional execution units on the pipeline, and the SCN handles branching.

The LS is investigated in Section 4.5.3 and the MFC in Section 4.5.5. For a detailed analysis of the SPU components see [Torre, 2009, Johns and Brokenshire, 2007, Scarpino, 2008, Kahle et al., 2005]. Dual issue is limited to instruction sequences such that instructions are scheduled to match the resource profile of the SPU due to no instruction reordering being provided to increase the potential for multi-issue. Another limitation that SPE presents is that execution units are not duplicated, which could inherently increase multi-issue potential.

Figure 4.4: Internal organisation of an SPE

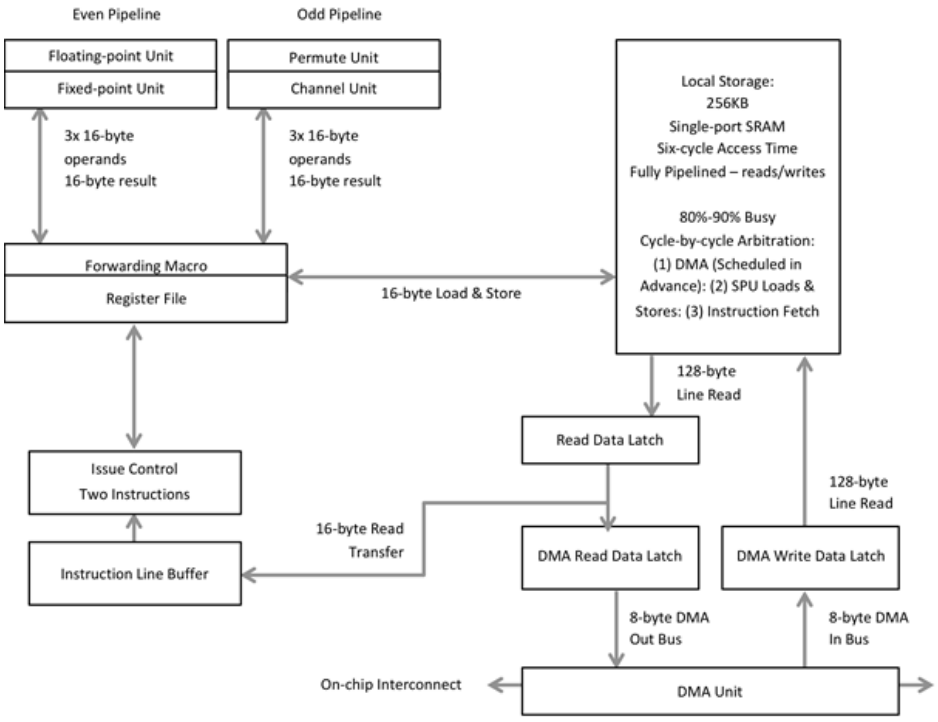


Figure 4.4 shows the organisation of an SPE with key bandwidth (per cycle) labelled between units. It uses 32 four-byte instruction groups which are fetched from the LS when idle; each fetch group is aligned to 64-byte boundaries to advance the effective instruction fetch bandwidth such that fetched lines are sent to the SPUs instruction line buffer (ILB) in two cycles, storing 3.5 fetched lines. All half-line hold instructions are sequenced into the issue logic, while another line holds the single entry software managed branch target buffer (SMBTB) [Flachs et al., 2007]. Two instructions are used for inline prefetching and are also sent at the same time from the ILB to the issue control unit.

All instructions that are issued by the SPE are completed in program

order; no reordering or renaming of instructions takes place. The SPE has a dual-issue pipeline that resembles VLIW architecture such that each SPE is able issue up to two instructions per cycle (IPC) to nine execution units. Pairs of instruction can be issued together if the first instruction is initiated with an even address and channelled to an even pipeline unit while the second instruction is sent to the odd pipeline. The execution units on each SPEs are allocated to pipelines, to allow is to maximise dual-issue efficiency for a variety of workloads and allow very high performance [Flachs et al., 2007]. The micro-architecture of the SPE pipelines simplifies resource and dependency checking, and reduction of hardware-logic devoted to instruction sequencing and control.

Conventional CMPs are complex in design and operation. By utilising the accessible transistor resources on a single wafer or chip; this enables multiple threads of execution to retain large amounts of data on the processors primary register file.

Each SPU has simpler pipeline architecture and requires less energy to operate [Gschwind et al., 2006]. It should be mentioned that the Cell allows different programming models whereby the SPEs perform compute-intense operations such as media and numerical data processing while the PPE executes more traditional instructions such as branching, coordinating SPE tasks and handling requests from SPEs and other elements of the Cell processor.

The PPE can also execute complex systems such as the operating system [Williams et al., 2006]. Conventional processors support scalar types with scalar hardware including the PPE, but the SPU does not have separate hardware support for scalar processing and relies on the compiler to transform scalar code into vector code which is aligned and optimised for SPU execution.

4.5.3 Local Storage

The local storage (LS) is a 256 KB unified register file (128-bit wide bus) that is located on each SPE. The unified register file stores all data types such as integer values, single- and double-precision floating-point values, Boolean values and addresses. SPE load and store instructions are performed within a local address space and not in system address space [Flachs et al., 2007]. The local address space is untranslated, unguarded, and non-coherent with respect to the system address space, and it is serviced by the LS. Furthermore, the register file is able to provide single quad-word element, two 64-bit double-word elements, four 32-bit word elements, eight 16-bit half-word elements, 16-byte elements or a vector of 128 single-bit elements [Gschwind, 2007] and [Scarpino, 2008].

The LS memory address is also memory-mapped input/output (MMIO) to the main memory, and each LS has a separate memory address that is visible only to its associated SPE and is not coherent in the system. The most distinctive characteristic of the SPE LS is the unification of both instructions and data stored within the same register file. It is important not to confuse the LS with the L2 cache, which is typically a unified memory subsystem, the cache is managed by hardware cache policies data locality [Drepper, 2007,Furber, 2000] which determines what data is placed into the cache. The SPE can access the system memory through asynchronous DMA operations through the MFC (see Section 4.5.5).

To clarify, the LS is not cache memory but a unified register file, which only supports 128-entry and stores all types of data including condition statements, counters and static pre-computed branch link addresses.

The advantage of such a design is that it allows a compiler to fit compute-intensive programs into the register files, while supporting such large register lines that help to reduce considerably hardware administration overheads such as register renaming, which is another technique implemented in conventional processors to allow large numbers of instructions to be processed

simultaneously [Kahle et al., 2005]. However, very little effort is needed to make efficient use of the LS, even though the LS reduces the hardware complexity and increases performance of the SPE. Nevertheless, the LS does provide significant advantages when vector units are employed in the instruction stream of an application; optimised code for the SPE may present more opportunities for increased resource utilisation and increased computation and performance throughput.

With conventional CPUs use caches between processor and main memory to reduce latency for data fetch and store. Operating directly on main memory is considerably slower than using registers, hence additional hardware circuitry is needed to cache data and/or instructions, to speed up the process [Blachford, 2005]. Further exacerbating the situation is the use of coherence protocols to ensure data validity across the system (including additional caches and the main system memory). The Cell SPE's use of an LS and not a cache reduces the complexity and static power use; such an approach is quite radical and presents a distinct architectural difference including for code design and programming model paradigms.

As with conventional processors interfacing with cache memory, the SPE registers interface with their LS, delivering data to the SPE registers at an exceptional rate of 16 bytes (128 bits) per cycle with an upper bound of 64 GB per second. In comparison with conventional processors, these processors can also transfer a substantial amount of data, but such transfers occur in short bursts (a couple of hundred cycles at best, which further exhibits the memory wall issue).

The SPU does not support a separate scalar register file, as this would complicate data routing for source operands, additional multiplexers are required which causes an increase in latency and places additional load and store commands on the bus.

The register file is extremely versatile flexible, such that a program (including instructions) is able to utilise all 128 entries to store data values,

and the register file is fully symmetric from an architecture perspective. The Cell SPE architecture does not enforce or encourage hard wired specific register values that require expensive handling during instruction decoding and register file access or in bypass and forward logic [Gschwind, 2007]. This is because the LSs are not caches but a single register file located on each SPE that allows both compiler and programmer (application developer) to allocate the available SPE resources to the needs of an application, thus improving programmability and resource efficiency. The SPU register file does not require complex circuitry to support features such as coherence protocols and with this feature removed, the Cell SPE is able to scale further, straightforwardly.

The LS supports direct loads, stores and instruction fetches that complete with a fixed delay and without raising exceptions, and this simplifies the SPU core design and provides predictable real-time behaviour; such as design reduces the area and power of the core; moreover, the SPU core is able to operate at a higher frequency [Flachs et al., 2007].

4.5.4 Element Interconnect Bus

The increase of multiple compute cores on CMPs has resulted in a greater focus on the on-chip network (OCN) that interconnects the various resources on the Cell CMP such as the memory and computational units [Ainsworth and Pinkston, 2007]. The OCN, specially the element interconnect bus (EIB), is designed to be wire-efficient to improve throughput (effective bandwidth) and reduce possible end-to-end latency (communication delay) for a given cost in relation to the area, power, complexity and other such constraints.

However, the EIB is a high performance internal interconnect bus that integrates all the internal processor elements and ultimately is the heart of the Cell processor and enables all communication to take place among the PPE, SPEs, main system memory and external I/O. The EIB core network fabric must support high data rates to avoid possible bottlenecks, given that each

of the 12 elements of the EIB interconnects is capable of 51.2 GB/s aggregate insertion and reception bandwidth. OCNs in other mainstream commercial multicore processors have considerably lower bandwidth; typically less than 15 GB/s per core. The four unidirectional rings support 16-byte-wide data rings, with a shared command bus and a central data arbiter over which 128-byte packets are transmitted.

The Cell's OCN has a peak bandwidth of 204.8 GB/s[8] and has separate communication paths for commands such as data transfer requests to or from other elements on the bus [Gschwind et al., 2007]. As shown in Figure 4.1, the data transfers occur across four unidirectional rings, which are accompanied by a communication controller and a high-bandwidth bus, interfaces that are all integrated on-chip [Kistler et al., 2006].

With multiple data paths and a high bandwidth, the EIB can handle up to 100 DMAs with on-chip interconnect sources and sinks of 16 bytes of data every other cycle [Flachs et al., 2007]. Traditional modern multicore processors implement a crossbar OCN that is simpler in design and was considered for the Cell, but the limited die area and wiring constraints was considered unfeasible and provided limited scalability, which led to the implementation of the EIB for the Cell [Ainsworth and Pinkston, 2007].

4.5.5 Memory Flow Controller

The memory flow controller (MFC) serves as the data transfer engine and facilitates the following functions: a direct memory access (DMA) controller, a memory management unit (MMU), a bus interface unit and an atomic unit for synchronisation. The MFC interfaces the SPU with the main storage

[8] [Kistler et al., 2006] states "the EIB unit can simultaneously send and receive 16 bytes of data every bus cycle. The EIBs maximum data bandwidth is limited by the rate at which addresses are snooped across all units in the system, which is one address per bus cycle. Each snooped address request can potentially transfer up to 128 bytes, so in a 3.2 GHz Cell processor, the theoretical peak data bandwidth on the EIB is 128 bytes x 1.6 GHz = 204.8 GB/s".

(effective address – EA) through SPU channels. The MFC initiates this communication by transferring DMA commands and data between the local storage (LS) of the SPU and the main storage.

Furthermore, each SPE supports mailboxes, signal-notification messaging between the SPEs, PPE and other devices. Also, the PPE and other devices (including other SPEs) communicate to the target SPE with the associated MFC through memory-mapped input/output (MMIO) registers which are associated with the (target SPE) SPU channels. Large or bulk data transfers are typically initiated through the SPU channels within the SPU or MMIO by the PPE. MFC programs can be constructed which instruct the SPE to perform sequences of block transfers via transfer lists that consist of up to 2K block transfer operations.

4.5.6 Direct Memory Access

The direct memory access (DMA) is the main unit that provides communication for data between different SPE LSs and from an SPE LS to the main system memory effective address space as specified by the Power architecture [Kahle et al., 2005]. The DMA is a vital communication mechanism between the SPE and the EIB (see Section 4.5.4). The communication is initiated by the SPU sending a request to the MFC with a single DMA such as a PUT or a GET command.

The DMA primary operations are the PUT and GET commands that are further interleaved with fence and barrier mechanisms [Scarpino, 2008, IBM, 2007b]. The SPU is able to transport data in 1, 2, 4, 8 and 16 bytes or 16-byte multiples up to a maximum of 16 KB (16,383 bytes). Also, the transfer is performed at byte alignment, with a 128-byte boundary being the most efficient. Small data sizes are also supported whereby the associated data types that are naturally compatible such as 1-byte chars aligned on 1-byte boundary, 2-byte shorts aligned on 2-byte boundary and so forth. Each SPE supports 8-byte-wide inbound and outbound data busses allowing the

SMA engine to support transfer requests generated by both the local SPU and the requesters external to the local SPE. The external request queue holds external programmed requests or incoming EIB read or write received requests that are translated addresses within the system real-address range assigned to the LS.

The DMA engine can process up to 16 commands simultaneously, perform scatter and gather operations from system memory and set up a complex set of status reporting and notification mechanisms. Each command can fetch up to 16 KB of data, but each data transfer is divided into 128-byte packets for the on-chip interconnect while supporting up to 16 packets in flight at a time; the Cell's DMA commands are significantly richer than a standard set of cache prefetch instructions. Software is able to achieve a much higher bandwidth utilisation through the DMA engine than traditional hardware prefetch engine. The DMA has a higher fraction of useful data transmitted across the processor than a speculative prefetch engine design.

Using the SPEs DMA component for synchronisation messages, the Cell is able to reduce power consumption by queuing multiple messages on the channel interface on the SPE (via DMA). Such a process does not incur delay, until the channel capacity is exhausted, but each device is able to allocate one or more channels whereby messages can be sent to or from the SPU. When a channel becomes exhausted (no slots are available), the write or read instructions stall and the SPU stalls in a low-power wait mode until the device becomes ready. Interestingly, the channel wait mode can often be a substitute for polling and represents significant power savings [Flachs et al., 2007]. Channels on the SPU are accessed through three instructions: read channel, write channel and read channel count (measures channel capacity). Before a request is sent to the bus (EIB), the memory management unit (MMU) translates all DMA request addresses while software is able to check or be notified when requests or groups of requests are completed.

4.5.7 Signal and Mailboxes

Each SPE has two inbound (32-bit register) signal notification channels. Both signal channels can be configured for logical and mode one-to-one signalling or overwrite many-to-one signalling. The configuration of the signal modes is done when the SPE context is initialised. The signal mechanism is considered to be more system aware, such that signals can be transmitted between SPE and other components also used for DMA notifications that are coupled with tag group identifiers. The Cell provides a custom 32-bit communication mechanism between an SPE and the PPE mailbox. Each SPE has three mailboxes: one inbound and two outbound.

Firstly, the inbound mailbox is a 32-bit queue storing four 32-bit messages from the PPE, with non-interrupting and interrupting outbound mailbox supporting one 32-bit message each. Mailboxes are optimised for communication between an SPE and the PPE. Furthermore, an SPE is able to write to its associated outgoing mailbox with the intended recipient consuming the message at the appropriate time.

4.5.8 Software Cache and Memory

Certain applications tend to operate more effectively in cache-based processors. A PPE has a cache but the SPEs have LSs that are not caches. With processing distributed between both PPE and SPEs, the Cells SPUs have LSs that can emulate cache memory. Furthermore, the programmer can specify either all or part of the LS to be allocated as software cache memory [Scarpino, 2008]. Using the SPE LS with copy-in copy-out semantics does not require coherence maintenance, performed on the primary memory operand repository (LS).

With the software cache providing greater application flexibility, the Cell further attempts to increase efficiency of the memory subsystem by allowing the memory address translation to be performed during block transfer

operations.

In [Gschwind et al., 2007] the author highlights significant advantages that include a single address translation which can be used to translate operations corresponding to an entire page for access during data transfer instead of during each memory operand access. This also leads to a significant reduction in ERAT and TLB access, thereby reducing power dissipation and eliminating expensive time-critical ERAT miss logic from the critical operand access path. Transferring data in blocks therefore improves efficiency in terms of utilisation, and can further optimise the Cell's memory interface bandwidth. By using a fixed protocol [Gschwind et al., 2007] the overhead per request can be remunerated over a bigger set of user data, thereby reducing the overhead per use.

Area and power efficiency are important enablers for multicore designs that take advantage of parallelism in applications, where performance is power limited. Every design choice must trade off the performance that a prospective feature would bring versus the prospect of omitting the feature and devoting the area and power towards higher clock frequency or more SPE cores per Cell processor chip.

Power efficiency drives a desire to replace event and status polling performed by software during synchronisation mechanisms that allow for lower power waiting. Understanding power, energy and thermal issues for multicore processors is beyond the scope of this thesis and is described in more depth by [Moncrieff et al., 1996] where the authors explore the design space related to physical cores, L2 cache size, processor complexity and power constraints. Furthermore, in their paper, the authors outline the efficiency of power and thermal issues in conjunction with modern applications that exploit TLP.

4.6 Summary

This chapter has primarily focused on computer architecture and the IBM Cell processor. Firstly, identifying fundamental parallelism, with the outcome of current and future parallelism must come from TLP from a software perspective. With processors constantly increasing in compute ability – that is, increasing the number of processing cores, microprocessor architects are increasingly aware of power limitations; in fact, both static and dynamic power consumption is the primary focus, hence microprocessor architects are always increasing efficiency, by reducing unneeded circuitry that does not improve the overall efficiency or reducing power consumption.

The SPE has no additional reorder buffers, register-rename units and commit buffers that reduce core power dissipation. However, the resource profile is known to the compiler and is able to schedule instructions to the resource profile. Moreover, ILP is utilised in the Cell architecture which attempts to reduce power inefficiency of wide issue architectures as no execution units with their inherent static and dynamic power dissipation are added for marginal performance increase [Gschwind et al., 2007]. Speculative hardware increases circuit complexity; increase of dynamic power use with overheads and ultimately speculative hardware does not give results. The SPE and its architectural specification were inherently designed and optimised for low complexity and low area implementation.

This chapter also delved into Amdahl's law that calculates the speed-up by calculating the percentage of serial processing relative to the overall program execution time using a single processor. However, its simplicity does not take into account significant factors such as synchronisation and message passing. The chapter concludes with an overview of the IBM Cell microprocessor, which consists of two different core types in order to maximise the available system performance.

The Cell has been designed to address the diminishing returns obtainable from a frequency-oriented single core perspective by exploiting appli-

cation parallelism and embracing CMP [Gschwind et al., 2007]. The LS abstraction provides a dense, single-ported operand data storage with deterministic access latency compared with speculative hardware caches that provide non-deterministic timing; moreover, the LS provides the ability to perform software-managed data replacement for workloads with predictable data access patterns which attempts to further reduce long latency of memory operations and gives flexibility to both program and programmer.

The Cell architecture presents a unique change from conventional microprocessors and gives more visibility of hardware capabilities to the programmer, while increasing the flexibility of programming models including the RPC model to functional processing pipelines of several accelerator cores. Ultimately, it is the responsibility of the programmer to harness the computing power as well as control of the system. Moreover, the Cell architecture places an increased emphasis on compiler optimisation to generate more efficient and optimised machine object code, that is increase compiler efficiency in register allocation.

> Parallel execution becomes energy-efficient because the efficiency of the core is increased by dual issuing instructions: instead of incurring static power for an idle unit, the execution is performed in parallel, leading directly to a desirable reduction in energy-delay product [Gschwind et al., 2007].

The next chapter presents our implementation of a dynamic recovery system that is influenced by traditional speculation execution systems. However, no use of speculation is implemented moreover, this research capitalised the hardware feature set to enhance the overall execution of the program while dynamically handling violations when they are generated.

Chapter 5

Introducing the Lyuba-API Framework

This chapter will investigate the Lyuba framework, manual programming and the L-API that details the main functions and description of the high-level execution model. Furthermore, this chapter details both PPE and SPE kernels whilst integrating a concise explanation of the Cell hardware and how the Lyuba framework will leverage the Cell hardware.

5.1 Lyuba Framework

To support the data dependency recovery system (DDRS) for a heterogeneous multicore processor such as the IBM Cell (see Chapter 4) the framework must detect data dependence violations at runtime, which involves comparing load and store requests from multiple on-chip SPEs.

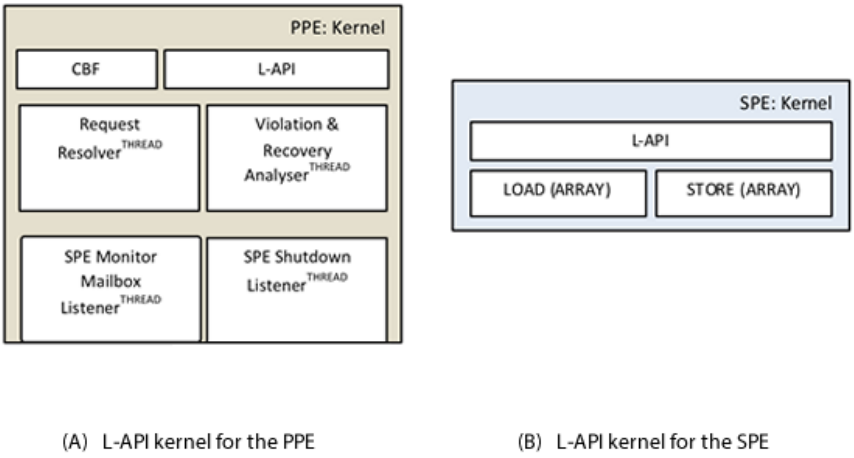


Figure 5.1: PPE and SPE kernel overview. (A) shows the L-API kernel components on the PPE. (B) shows the L-API kernel components on the SPE.

Figure 5.1 shows the kernels for both PPE and SPE. The kernels have been designed to interface directly to the hardware layer while ensuring correct operation of the entire program; such an implementation is similar to the task-based approach of [Keller and Varbanescu, 2010, Skovhede et al., 2010]. When developing the framework, a number of issues were taken into consideration to ensure that the interface between hardware components remained compatible: managing memory allocation of structures, creating and managing threads and recovery from violations.

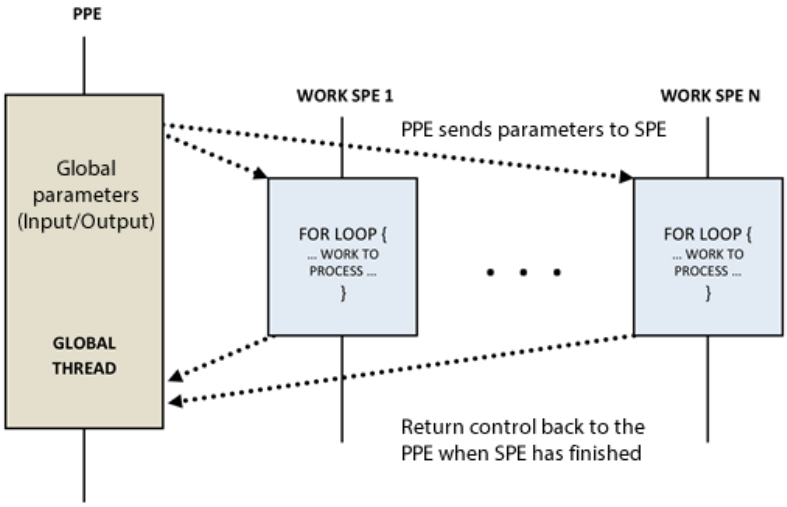


Figure 5.2: Parallelising `For` loop across all SPEs.

Figure 5.2 shows the organisation and parallelisation of a `For` loop. The PPE generates the preamble data or the prerequisites for each SPE. Each SPE is then started and obtains its specific parameters from the PPE and then returns control back to the PPE when an SPE has completed its work of execution. This section begins with a description of the PPE kernel, outlining the primary operations and supporting infrastructure of the SPE kernel, followed by a description of the SPE kernel, and then exploring the interface between the kernels, including the software interface to the Cell hardware.

5.2 L-API PPE Kernel

Typically, literature places meticulous focus on thread creation, optimisation, execution and forwarding the thread state. Past TLS implementations have focused on compiler optimisation and use of hardware (specialist TLS hard-

ware) instructions. The following subsection explores the design interface of the L-API PPE kernel.

5.2.1 Threads on the PPE

Typical use of threads in TLS is to enclose work to be done into a thread, known as an epoch. With each epoch is maintained, controlled and advanced through each stage of its pipeline by TLS hardware; this behaviour is typical of most TLS software and hardware schemes. The execution model from this study is not very different from typical software TLS schemes. Let us first consider the category of threads: system and parallel threads. System threads control the execution flow and provide service to any SPE L-API functions that have placed a request within the system and parallel threads represent epoch threads. System threads within the Cell environment (PPE domain) maintain the state of each SPE context – see [Scarpino, 2008]. Moreover, the PPE threads provide additional services that include monitoring requests made from the SPE threads (contexts), and they register and deregister regions, initialise context environmental data and ensure safe shutdown of the system.

Creation of threads remains the fundamental building block, hence exposing the POSIX library. The Cell SDK encompasses the POSIX library and is used for the creation of SPE context threads. The L-API framework makes effective use of Pthreads (POSIX threads[1]) that is available on most platforms. It is important to note that one aim of this research is not only to make use of available CMP hardware but also make succinct use of available software libraries, when applicable.

However, by creating threads for specific operations that occur, i.e. recovery threads, the system overall execution time will increase due to the creation process. The creation of threads being relatively uncomplicated can also lead to performance degradation of the entire system. Each thread in-

[1]The C POSIX library is a language-independent library.

curs overheads, so the L-API framework only creates threads when the state of program execution changes.

Another criterion involved in the development of the L-API framework is the scaling effect for threads, more specifically the synchronisation process between threads. Execution of threads with independent epochs typically increases parallelism [Butenhof, 1997]. Therefore, the system will only create an upper bound number of threads per each function equal to the number of SPEs initialised in the framework. The maximum number of SPEs initialised is the total number of available SPEs from the hardware abstraction layer (HAL). See Section 1.4 for maximum available SPEs used for this research.

Listing 5.1: STL containers and iterators.

```
1 /*A list to hold that status information about the SPE
      */
2 std::list<spe_status_t> _SPEStatusList;
3 std::list<spe_status_t>::iterator
      _SPEStatusListIterator;
4
5 /*A list to hold the work regions*/
6 std::list<context_params_t> _WorkList;
7 std::list<context_params_t>::iterator _WorkListIterator
      ;
8
9 /* Queue to store all requests made from SPEs */
10 std::list<request_message_t> _RequestList;
11 std::list<request_message_t>::iterator
      _RequestListIterator;
12
13 std::deque<recovery_info_t> _RecoveryQueue;
14 std::deque<recovery_info_t>::iterator
      _RecoveryQueueIterator;
15
16 /* List to hold the number
17 of regions created on the system */
18 std::list<region_complete_t> _RegionList;
```

```
19 std::list<region_complete_t>::iterator
       _RegionListIterator;
20
21 /* List to hold the registered regions */
22 std::list<loop_t> _LoopRegionList;
23 std::list<loop_t>::iterator _LoopRegionListIterator;
24
25 /*Map to store all element objects*/
26 std::multimap<int, element_t> _ElementMap;
27 std::multimap<int, element_t>::iterator
       _ElementMapIterator;
```

5.2.2 Element State Containment

This section discusses constant and temporary data stored in the L-API system. The previous chapter explained the difference between speculative hardware and non-speculative hardware, but the L-API integrates the standard template library (STL) to store and load temporary and constant data. Listing 5.1 shows the internal function calls that store different types of data into their associated standard template library (STL) containers.

Utilising STL containers with templates provides a very powerful compile-time polymorphic system [beach, 2008]. The L-API framework uses multiple STL list containers and a single STL deque (double-ended queue) and an STL multimap container. As requests are placed, the system would generate specific data (parse the data into a template structure, e.g. `element_t`) and store the template element into a defined data containers i.e. `_ElementMap`. To traverse through the STL data containers, the framework uses STL's bidirectional iterators container `_type<properties>::iterator_type iterator _name`.

With multiple threads accessing the structures concurrently, such actions as modify, removing and inserting elements at any position in the container, the framework uses a non-LIFO structure, a list, deque or multimap. By

using such containers, data and index integrity of the containers is maintained and allows the threads to search for the required element near an asymptotic upper bound $O(N)$. To control access and ensure that each STL container is indexed properly and to reduce data corruption, all containers are coupled with POSIX synchronisation locks, i.e. mutex variables [Barney, 2010].

Each Pthread accessing any STL container must firstly obtain access rights to the container by locking the mutex. If a thread cannot lock a mutex, the thread will wait until the mutex becomes available. Once a thread has locked the STL container's mutex, it can then access the container and continue with the thread's instructions. Just as thread creation causes overheads, so do mutexes. If multiple threads try to gain access to a shared area, the mutex will only allow a single thread to gain access. An improvement would be to use read/write mutexes. However, this would increase the complexity of execution and result in poor performance.

The following subsections outline how the framework interacts with the hardware components used for communication and synchronisation between the processing cores, how the systems resolves the location of data and the methodology of the L-API system interacting with the Cell callback routines.

5.2.3 Hardware Interfaces for Communication and Synchronisation

Section 4.5.7 provides a detailed overview of the hardware mechanisms used for communicating data among SPEs and the PPE. This section will outline the software interface used by the L-API to access the hardware communication layer. The following function monitors the SPE mailbox:

```
1 void *ptf_SPEMonitorMailbox(void *arg)
```

The L-API system will constantly check the effective address (EA) memory address of a specified SPE MMIO to determine if an SPE has placed

data into its outgoing mailbox slot. If the function detects a message in the specified SPE's MMIO EA, the function will access the SPE's outgoing mailbox slot and consume the data.

Listing 5.2: Checking SPE mailbox status and retrieving data.

```
1 //Read the status of an SPEs mailbox
2 _i_Return = spe_out_mbox_status(ps_SPEContext
3 [spe_number[0]].context);
4
5 //Read the contents of the mailbox
6 spe_out_mbox_read(ps_SPEContext
7 [spe_number[0]].context, &_i_Return, 1);
```

The code in Listing 5.2 is used by the kernel on the PPE accessing an SPE's memory-mapped input/output (MMIO) memory address space to determine if an SPE has placed data in its outgoing mailbox slot (line 2). If a message exists in an SPE's outgoing mailbox slot the result is placed in a temporary integer variable `_i_Return`. This variable is then compared using a conditional expression, should the variable contain an integer value of zero; the system has not detected any messages and therefore the function will start the checking routine again.

Listing 5.3: Signal interface - write only.

```
1 //Read the status of an SPEs mailbox
2 spe_signal_write(ps_SPEContext[_SPEStatusListIterator->
      spe].context, SPE_SIG_NOTIFY_REG_1, SPE_SHUTDOWN);
```

However, if an integer value of one is detected, then the system is able to retrieve the data from the SPE's outgoing mailbox slot (line 2 in Listing 5.3).

The signal hardware mechanism that is also employed in the Lyuba framework is only able to send a single 32-bit packet to an SPE. In Figure 5.3, the

code demonstrates how the PPE communicates with an SPE. The message from the PPE is sent to the designated SPE's inbound signal slot. It should be noted that the principal uses of such hardware interfaces is for on-chip communication, supporting the research aims and objectives, see Section 1.3. Utilising available hardware reduced potential synchronisation overhead from system and parallel threads, reduce off-chip memory bandwidth and exploit available hardware mechanisms.

5.2.4 Callback Functions

As described in Section 4.5.2, an SPE's pipeline is considerably simpler in design; hence it does not support many components that are traditionally established in a conventional processor.

Therefore, to extend the SPE's ability to handle certain functions that are more efficient to run on a general-purpose processor, the IBM Cell SDK provides a callback function [Bartlet, 2007] which is similar to the remote processor protocol (RPC) [Bloomer, 1992]. The callback functionality is used quite extensively by all SPEs; see Section 5.2.4 for an analysis of callback utilisation. Conversely, the callback functionality enables each SPE to extend its operational ability, but this would increase overheads for the PPE and therefore additional resources are needed. Furthermore, if callback is used greatly, this could prolong the execution of PPE threads.

The callback functions are typically prioritised given priority. This is due to the nature of the L-API system, whereby the system ensures that all SPE's tasks are processed immediately, or quickly as possible, and to keep the SPEs busy at all times.

5.2.5 Calculating Data Load and Store

This section explores the process of calculating the data load and store regions, which an SPE can load and store data (undertaken in the L-API PPE

kernel). Currently the system supports a master input and output domain. Also, the system supports three additional auxiliary domains (bidirectional use). Therefore, each request is associated with one domain.

```
1 ptf_SPERequestResolverScalar(void *arg)
```

The Request Resolver function handles the main task of detecting RAW violations within requests. Firstly, the function will determine if an element packet exists, and then if not, an element is created using the request packet's data, and inserted into the correct data container – see Section 5.2.2. Secondly, using the element packet, the system will determine the source and destination iterations.

Each request has both a who (iteration number) created the request and where (iteration number) data needs to be loaded or stored. Thirdly, the system determines the request type: load or store. Fourthly, before continuing with the request, the system checks if a RAW (see Section 3.2) violation has occurred and finally, if a violation is not detected, the system will complete the request, otherwise the recovery method is initiated. Provided no violations occur, the system can issue the SPE to either load or store data from another's SPE LS or the EA.

To determine the absolute memory location (effective address or LS address) the system will determine if an element exists in the system. An element is only created if it is not present in the system for the iteration that created the request. Moreover, another element is created for the iteration that has the real location of the data or where the data is stored. If an element already exists for the required iteration (source or destination), the element resolves the iteration commit stage, that is, is data committed to the EA or active in an SPE LS. Therefore, using the iteration's element object, the system can then determine the location of the data to be stored or retrieved. The results are then sent back to the SPE that created the request. The SPE is then able to retrieve or send the data using the information received by the L-API PPE kernel.

Table 5.1: SPE main interface constructs prototypes.

Prototype and description
`void Region(num)` Informs the PPE kernel that the SPE has entered a region
`void RegionEnd()` Informs the PPE kernel that the SPE has reached the end of the region, and now the SPE will wait for a synchronisation signal before moving on to the next line of user space code
`void OuterLoop(id, rs, re, itr)` All outer loops are enclosed with an outer loop function
`void OuterLoopEnd()` Specifies the end of the outer loop region
`void InnerLoop(id, rs, re, itr)` All inner loops are enclosed with an inner loop function
`void InnerLoopEnd()` Specifies the end of the inner loop region

Listing 5.4: Initialising region parameters and distribution.

```
1 /*main.c on the SPE*/
2 Region(1);
3 function_1();
4 RegionEnd;
5
6 Region(2);
7 function_2();
8 RegionEnd;
```

5.3 L-API SPE Kernel

The previous sections have established the initialising code on the PPE. This section will outline the L-API interface for the code that requires parallelisation. Currently, the L-API only supports `For` loops with future revisions of the L-API framework supporting many more structures.

All transformed code is placed into a function that is called from the SPE main.c file. Listing 5.4 shows the function placed between region constructs (see Table 5.1). These constructs help the L-API to identify which function are being called and identify different regions (programs) executing on the Cell processor simultaneously.

Listing 5.5: Example Bubble Sort Code Illustrating Regionable Code.

```
1 /*example code - double bubble sort*/
2
3 //NON-REGION CODE - NON-PARALLELISABLE
4 int y;
5 int x;
6
7 //REGION 1 CODE - PARALLELISABLE
8 for(int x=0; x<n; x++)  {
9   for(int y=0; y<n-1; y++) {
10        if(array[y]>array[y+1]) {
11          int temp = array[y+1];
12          array[y+1] = array[y];
13          array[y] = temp;
14        }
15       }
16 }
17
18 //NON-REGION CODE - NON-PARALLELISABLE
19 printf("%i\n", y);
20
21 //REGION 2 CODE - PARALLELISABLE
```

```
22 for(int y=0; y<n; y++)  {
23   for(int x=0; x<n-1; x++) {
24         if(array[x]>array[x+1]) {
25           int temp = array[x+1];
26           array[x+1] = array[x];
27           array[x] = temp;
28         }
29       }
30 }
```

Listing 5.5 shows an untransformed bubble sort code. However, the importance is where parallelism can be gained. As the L-API currently supports `For` loop statements, lines 8 - 16 and lines 22 - 30 are parallelised. Moreover, within the parallelisable regions of code, L-API load/store constructs are applied to the arrays. `For` loops typically exhibit a greater degree of parallelism [Chapman et al., 1997] and are regions which are parallelised. The next stage decomposes the `For` loop into primitive code blocks by applying L-API constructs to the actual code.

Listing 5.6: L-API SPE transformed code.

```
1 OuterLoop(0, Lower_IPC, Upper_IPC, i);
2 for(i=Lower_IPC; i<Upper_IPC; i++) {
3
4     InnerLoop(0, _i_SizeBeginPrimary, _i_SizeEndPrimary,
          i);
5     for(j=_i_SizeBeginPrimary; j<_i_SizeEndPrimary; j++)
          {
6         Store_DDA(lu, i, j, Load_DDA(A, i, j));
7     }
8     InnerLoopEnd; //<------ BARRIER CONSTRUCTS (INNER
          LOOP)
9
10 }
11 OuterLoopEnd; //<------ BARRIER CONSTRUCTS (OUTER LOOP)
```

5.3.1 L-API Load and Store Constructs

Listing 5.6 illustrates a `For` loop that has been transformed using the L-API constructs. Shown are the main constructs that are applied to unmodified code. The barrier constructs are primarily used for synchronisation between other SPEs and the PPE kernels. However, the main focus in this subsection is the load and store constructs: `InnerLoop` and `InnerLoopEnd` constructs which are used to synchronise with the L-API PPE kernel.

When an SPE enters an `InnerLoop` region, the L-API SPE kernel will register the loop with the PPE kernel. The `InnerLoop` construct notifies the global index (managed by the PPE kernel) of the active loop region. Although the L-API PPE kernel will already have a list of iteration groups, by implementing a registration process the L-API PPE kernel is then able to identify which SPE is processing the iteration group (identifier).

This registration process helps to identify which SPEs are active, and the information is used to resolve a request in regards to where data is placed or retrieved: the SPE LS or the main system memory (EA). Once a loop has

completed its operation successfully, it will then submit a prefetch request to the L-API PPE kernel for the next set of loop parameters. Once the parameters are received, the SPE will then change mode and wait for the loop barrier synchronisation. The loop barrier synchronisation notifies that the SPE has completed its loop region and waits for an `SPE_CONTINUE` signal from the L-API PPE kernel. Once all SPEs have completed their loop regions, the L-API PPE kernel is then able to send a signal `SPE_COMPLETE` message to all SPEs that are registered to the identical region.

The `OuterLoop` and `OuterLoopEnd` constructs are similar to their `InnerLoop` and `InnerLoopEnd` counterparts but have one variation. Whereas inner loops are only responsible for their immediate `For` loop, the `OuterLoopEnd` constructs are responsible for the entire region. Therefore, the outer loops will deregister the entire region and begin to process the next region (if applicable). The local store (LS) is the only memory that is directly accessible by an SPU - see Section 4.5.2 and 4.5.3 for a detailed review of the SPU and the local storage. The L-API SPE kernel has two main regions as depicted in Figure 5.4. As LS is limited to 256 KB, the L-API kernels and framework must be optimised in order to make full and efficient use of the available memory space.

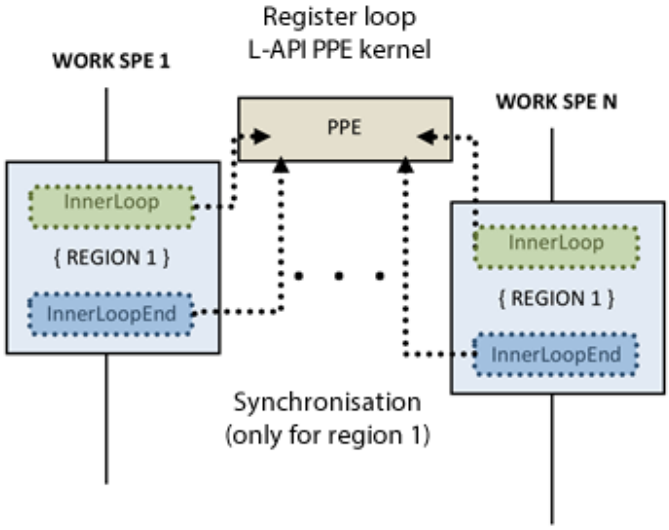


Figure 5.3: Inner loop synchronisation.

Figure 5.3 highlights the synchronisation of inner loops. When a inner loop construct is executed (`InnerLoop`) the SPE kernel notifies the PPE kernel that a loop is about to be executed on the calling SPE, the loop is registered on the PPE kernel. The PPE sends the appropriate parameters back to the SPE using in-built communication mechanisms (see Section 4.4) such as loop number and inner loop start and end values. Once the inner loop has reached its `InnerLoopEnd` construct, the SPE kernel then deregisters the loop from the PPE kernel.

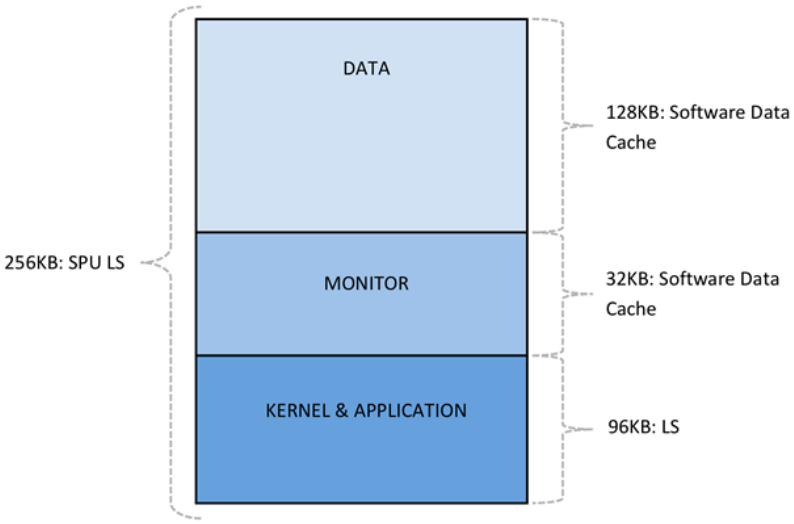


Figure 5.4: SPE LS memory segmentation.

The L-API SPE kernel segments its LS into three segmentations. Figure 5.4 shows an SPE LS segmented into three regions. The largest segment is software cache that only supports a single data type such as double or integer (depending on the application used). A smaller segment is also a software cache, which only stores data into an integer array that is globally available in the EA. Finally, the third segment is not a software cache but unmodified LS area where the L-API SPE kernel and application exist. The kernel and the application remain in this area of the memory and are not replaced by any cache policies.

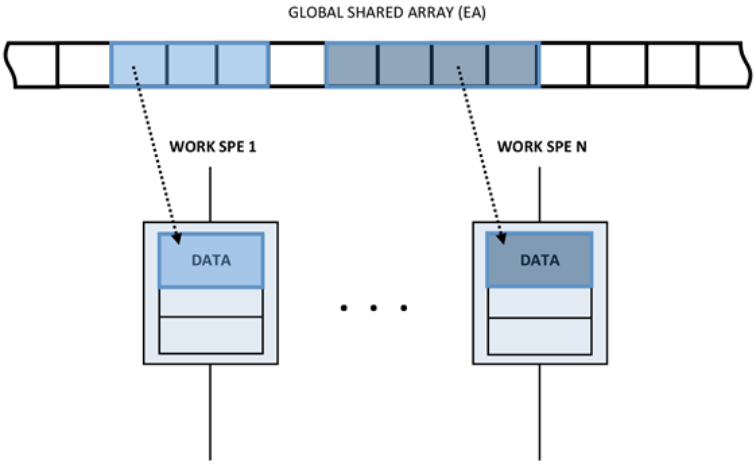


Figure 5.5: Global shared array across multiple SPEs.

5.3.2 Mapping Array Segments

When an SPE kernel is initialised, the kernel will generate two software caches of predetermined data types. Each SPE is then able to cache data from the EA;[2] however, the data being cached into an SPU LS must be of the same data type. Figure 5.5 illustrates a global shared array that is accessible by any SPU software cache. The data is cached using the Cell SDK's in-built functions [IBM, 1994]. Furthermore, the software cache is able to commit the data back to the global shard array memory address by flushing the software cache, again using the Cell SDK's in-built functions. Once a region of global memory is cached in an SPE LS, the cacheable region is still accessible and visible in the effective address but cannot be modified by any SPE or the PPE unless the L-API PPE kernel deems it is safe to do so.

[2]Effective Address.

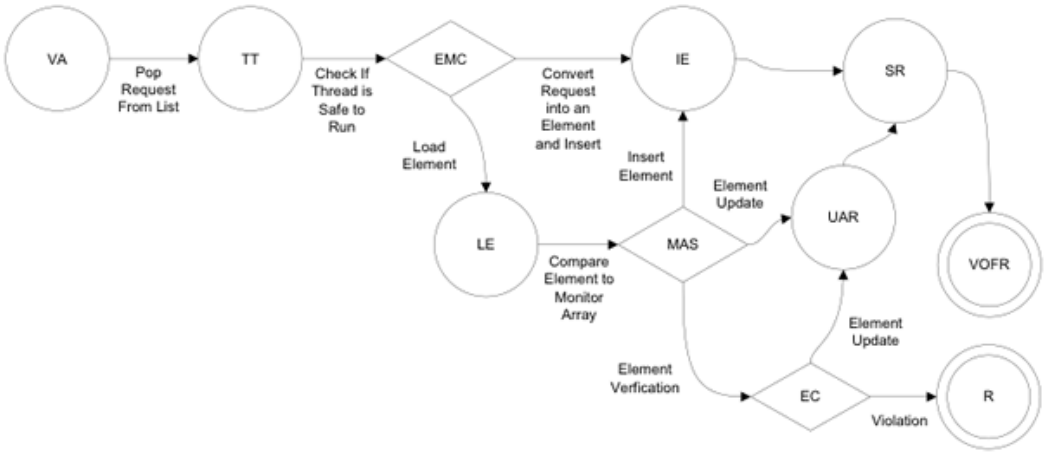


Figure 5.6: Flowchart showing the different states for the violation analyser function.

5.4 Violation Detection and Resolution

The preceding sections explored the application of the L-API framework including the constructs. Another important component of the framework is the detection of violations that may occur. Each load and/or store request in an iteration is checked (both PPE and SPE analyse the request) for a dependency violation. If the request is to load or store data into a slot that exists outside of the calling SPEs range (see Section 5.3) then the PPE determines if the destination source is ready for data replacement or retrieval. If so, the source SPE then places or retrieves the data from the destinations, which could be either the EA or an SPE's LS. Should a dependency violation occur (see Section 3.2) then the PPE initiates the recovery process. Figure 5.6 outlines the process taken for a violation detection and recovery.

Table 5.2: Violation detection state labels.

Tag	Description
VA	Violation analyser
TT	Thread type
EMC	Element manager check
IE	Insert element
LE	Load element
UAR	Update and reinsert
SR	Service request - load or store
MAS	Monitor array state
C or EC	Element check
R	Recovery
PE	Pop request from the request list
TR	Type of request
SR	Store resolver
LR	Load resolver
CR	Complete request

Table 5.2 provides the key (labels) to the different state names in Figure 5.6. All requests to the PPE are placed in a container (list) with a kernel thread running asynchronously that checks the list at a predetermined interval period. If a list has an element[3], the kernel thread pops the element off the list and determines what type of element (request) requires resolution. If the request type is an insert element (IE) then it's a store request (SR). The SR then checks the destination and attempts to store the data, unless the destination is not ready; the SR thread will wait until the slot is available.

However, if the request type is a load request (LR) than additional checks are required. The request data that requires the load is firstly checked in the monitor array state (MAS). The MAS identifies all the load values in the application and determines if it is safe to load by firstly marking the

[3]A request from the SPE is an object that is structured and is readable by the PPE. The request on the PPE is known as an element.

destination slot (that requires the load) as busy. This change in the MAS ensures that all current or future requests that require load or store to the marked slot as unavailable. Once marked as busy, the data is checked and the auxiliary information stored against the slot. The auxiliary information states the last known iterations to have loaded or stored and compares that data to the active request (element). If the destination slot has data that was stored by an iteration that is greater than the element's iteration then a violation has occurred. This violation than triggers the recovery process. Otherwise, no violation has occurred and the MAS auxiliary is updated and saved (SR). The PPE then sends a command to the SPE to show where to load the data and authorises the calling SPE to continue.

5.5 Worked Example

The previous sections explored the inner workings of the L-API but without a real example. By using a real worked example with annotation of L-API constructs with reference to Cell-specific code, allows a clearer understanding of the fundamentals of the L-API.

Using the fast Fourier application from the SciMark benchmark [Pozo and Miller, 2004] as an example, this section will present the process of applying the L-API constructs, which results in a transformed version of the application. This process will aid the reader in understanding the application of L-API constructs to existing code. The fast Fourier application [Pozo and Miller, 2004] will execute on both PPE and SPE but in order to distribute the computation across all available processors, the code has to be transformed using the L-API constructs.

The first stage in the transformation process is to set up the prerequisites that all SPEs require before processing iterations. Compile-time constants allows computations to take place at compile time, this improves performance as they don't allow arbitrary execution at construction and can therefore be

used at places where code is not required for recompilation. To improve the performance of the L-API, a considerable number of compile-time constants are used as an optimisation technique.

Both array sizes and loop bounds use compile-time constants and ensures the L-API is able to immediately access the appropriate memory addresses at runtime. Moreover, iterations of loops (loop bounds are computed immediately with compile-time constants) are accurately distributed across the multiple SPE's, which further allows the framework to generate all the required memory addresses at compile time, this ensures that the L-API is able to allocate the correct amount of memory and memory addresses. This allows appropriate validation rules on both PPE and the SPE kernels to analyse, store and retrieve data at runtime without the need to calculate the base addresses of arrays and loops are created and traversed with pre-computed bounds; this reduces overhead at runtime.

Note, the L-API only supports `For` loops (see Section 6.1.1). Once the prerequisites are established on the PPE, then actual computation code is extrapolated from the untransformed code and exported to the SPE, simultaneously applying L-API constructs. The next subsections will present these important concepts.

Listing 5.7: `Main.c`

```
1 #include "kernel.hpp"
2
3 int main(int argc, char *argv[]) {
4
5   // Step 1.0 - Set data size
6     int FFT_size = FFT_SIZE;  // TINY_FFT_SIZE (Small)
            FFT_SIZE (Medium)       LG_FFT_SIZE (Large)
7
8     // Step 2.0 - Set up the application attributes
          needed by the L-API framework
9   SingleArrayInput = (double *)RandomVector(2*FFT_size,
        R);
```

```
10    SingleArrayOutput = (double *)malloc(2*FFT_size*
          sizeof(float));
11
12    // Step 3.0 - Set the size of a global array monitor,
           whereby all SPEs can monitor and update
13    MonitorSize(2*FFT_size);
14
15    // Step 3.1 - Assign values and parameters to the L-
          API attributes
16    BenchmarkAttribute.bench_name = FFT;
17    BenchmarkAttribute.bench_id   = 0;
18    BenchmarkAttribute.total_iterations =
          fft_only_int_log2(2*FFT_size);
19    BenchmarkAttribute.size_1_dataset = FFT_size;
20    BenchmarkAttribute.aux = (unsigned long long)
          SingleArrayInput;
21    BenchmarkAttribute.aux2 = (unsigned long long)
          SingleArrayOutput;
22    BenchmarkAttribute.input = (unsigned long long)
          SingleArrayInput;
23    BenchmarkAttribute.output = (unsigned long long)
          SingleArrayOutput;
24    BenchmarkAttribute.io_array_type = SINGLE_ARRAY;
25
26    // Step 4.0 - (Instantiate) Insert the attributes
          into the framework
27    InsertBenchmark(BenchmarkAttribute);
28
29    // Step 5.0 - Start
30    SystemRun;
31
32    // Step 6.0 - Once completed, system safely exits
33    return 0;
34 }
```

5.5.1 `Main.c` on the PPE

Listing 5.7 displays `Main.c` code on the PPE. As mentioned in the above section, the L-API has prerequisites in order for the framework to operate. This information is established in the `Main.c` file on the PPE. The code in the `Main.c` file is ordered (see the `Steps` in code) whereby, the user must insert code in a specific order as shown in the above listing.

Line 6 specifies the number of elements to be processed which equates to the number of `For` loop iterations, so the variable `FFT_size` is a compile-time constant. Line 9 sets the address of a one-dimensional array which inputs data that requires processing. This is followed by line 10, where the `SingleArrayOutput` pointer is assigned the address of the output array where all processed data will be stored.

The L-API stores all iteration state data in a separate array which is referred to in line 13. The monitor array size (compile-time constant) must be equal to or greater than the number of elements to be processed.

The `BenchmarkAttribute` object contains approximately nine member attributes. The `bench_name` accepts a string value which must be unique and the id member must also contain a integer-only unique value. If there are multiple applications being executed on the same Cell processor, the framework uses the id value as a prefix for each generated DMA call. This allows the framework to determine which application-type that generated the request and therefore handle the request accordingly.

Line 18 specifies the `BenchmarkAttribute.total_iterations` that assigns the total number of iterations which typically equals the set data size variable (`FFT_size`). However, in this application, the number of iterations required is double the size of `FFT_size`. The `BenchmarkAttribute.aux` and `.aux2` (lines 20 and 21) assign the address of both input and output arrays. These arrays must be formatted as unsigned long long to ensure that the Cell's memory address reference is correctly addressed. Line 24 assigns a macro named `SINGLE_ARRAY` that specifies the type of array required on the

SPE. Once the `BenchmarkAttribute` is populated, it is then inserted into the system (line 24), followed by the run command (`SystemRun`) on line 30.

5.5.2 `fft.h` on the SPE

The complete SPE FFT transformed code is listed in Appendix C.4. Before any L-API constructs are applied, the user must insert the `#include "kernel.h"` header file into the C header file on the SPE[4]. Not all functions require transformation (application of L-API constructs). As noted in the earlier sections, the L-API currently supports `For` loops and variables that require pointers to array addresses. These arrays either have data that requires loading for processing or output arrays where data is stored.

Listing 5.8: Partial SPE code - load data from a pointer.

```
1 double *data = Load_DPTR(0);
```

When an SPE context is created and instantiated, the PPE kernel passes all prerequisite information to the SPE. Listing 5.8 highlights this important application with the `Load_DPTR(0)` function. The `(0)` argument is the index at which the address is passed. The function passes the address of the array where data is kept for processing. Using the SPE's overlay functionality, parts of the input array are pushed onto the SPE data array area (see Section 5.3.2).

Listing 5.9: Partial SPE code - `OuterLoop` barrier.

```
1 /* this loop executed int_log2(N) times */
2 OuterLoop(0, Lower_IPC, Upper_IPC, bit);
3 for(bit = Lower_IPC; bit < Upper_IPC; bit++, dual *= 2)
      {
4   w_real = 1.0;
5     .....
```

[4]To set up a Cell project please read the documentation in [Scarpino, 2008].

```
6 }
7 OuterLoopEnd;
```

Listing 5.9 shows the first important application of the L-API, to apply the `OuterLoop()` function at the beginning of a `for` loop. The first parameter specifies the loop number, followed by the starting iteration number (`Lower_IPC`) then end iteration number (`Upper_IPC`) and finally the relative index variable (bit). The bit variable only contains the current (active) iteration that is being processed: which aids the framework in determining which iteration is being processed of the `For` loop. The `OuterLoopEnd` barrier informs the framework on the SPE and PPE kernel that a loop has come to an end.

Listing 5.10: Partial SPE code - `InnerLoop` barrier.

```
1 InnerLoop(0, _i_SizeBeginPrimary, _i_SizeEndPrimary,
      bit);
2 for(a=_i_SizeBeginPrimary, b = _i_SizeBeginPrimary; b <
      _i_SizeEndPrimary; b += 2 * dual) {
3
4   i = b;
5   .....
6 }
7 InnerLoopEnd
```

If an algorithm exhibits internal loop(s), then an inner loop construct must be applied as shown in Listing 5.10. The `InnerLoopEnd` barrier informs the framework on the SPE and PPE kernel that a loop has come to an end. Another important application of the L-API framework is single value loads and stores.

Listing 5.11: Partial SPE code - `Load` data.

```
1 z1_real = LoadD(data,j);
```

Listing 5.11 shows the load construct used to load data from an array. The data parameter specifies the address of the array with argument `j` specifying the index (position). The `LoadD(,)` construct then utilises the specified arguments to load the data.

Listing 5.12: Partial SPE code - `Store` data.

```
1 StoreD(data, j,    LoadD(data,i) - wd_real);
```

Storing data is shown in Listing 5.12. The argument data specifies which array is used to store the data to, argument `j` specifies the index in the array and the final argument specifies the actual data that requires storing.

5.5.3 L-API use of Low-Level Cell API Calls

This section will briefly delve into the low-level Cell API calls made from the L-API layer. A primary notification method between an SPE and PPE is the use of Cell's RPC[5], mailboxes and signal notification registers.

Each SPE has three main communication portals to the PPE. This section will firstly highlight remote procedure call (RPC) and the following two subsections briefly discuss mailboxes and signals respectively. RPC can be considered as function offloading, whereby the PPE is able to perform the function more efficiently due to increased hardware functionality compared to the SPE pipeline.

Listing 5.13: Remote procedure call (RPC).

```
1 void __send_to_ppe(unsigned int signalcode, unsigned
      int opcode, void *data) {
2   if(_i_LoopRestart==FALSE) {
3     unsigned int combined = ((opcode <<24) | ((unsigned
          int)data & 0x00FFFFFF));
4
```

[5]Remote procedure call.

```
5       vector unsigned int stopfunc = {
6         signalcode,
7         combined,
8         0x4020007F,
9         0x35000000
10      };
11
12      void (*f) (void) = (void *)&stopfunc;
13      asm("sync");
14      f ();
15      errno = ((unsigned int *) data)[3];
16    }
17
18    _i_LoopRestart=FALSE;
19  }
```

Listing 5.13 shows the internal workings of the `__send_to_ppe()` function. The function itself requires three arguments when it is called. The first argument specifies the signalcode that notifies the PPE signal type, `opcode` refers to the operation code that should remain as `(0)` and finally the data argument that passes a data structure to the PPE.

Listing 5.14: `InnerLoopEnd` macro - SPE (`kernel.h`)

```
1 #define InnerLoopEnd    (f_LoopEnd(
      _i_ActiveInnerLoopIdentifier, INNER))
```

For example, when an SPE reaches a loop barrier macro such as the `InnerLoopEnd` (see Listing 5.14), this initiates the L-API SPE kernel to inform the L-API PPE kernel that an SPE `For` loop has come to an end and update the containers (see Section5.2.2) as necessary (from the PPE perspective).

The macro shown in Listing 5.14 is placed in the transformed user code. The macro itself calls an internal function, `f_LoopEnd()` (see Listing 5.15)

with the macro passing its argument into a function. The `_i_ActiveInner Loop_Identifier` parameter contains the loop number and the second argument defaults the loop deregistering type as inner loop (`INNER`).

Listing 5.15: Loop end function - SPE (`kernel.h`).

```
1 void f_LoopEnd(unsigned id, unsigned level) {
2   f_GetLastLoopParameters(id, level);
3   f_LoopBarrier(id, level);
4   f_DeregisterLoop(id, level);
5 }
```

Listing 5.15 shows another layer of abstraction whereby the three methods contain both further abstraction and direct hardware-code access. The `f_GetLastLoop()` function consists of three function calls.

Listing 5.16: `f_GetLastLoopParameters()` function - SPE (`kernel.h`)

```
1 void f_GetLastLoopParameters(int id, unsigned level) {
2   loop_t s_LoopInformation;
3
4   s_LoopInformation.spe = _i_SPEIdentifier;
5   s_LoopInformation.level = level;
6   s_LoopInformation.type = COPY_BACK;
7   s_LoopInformation.id = (id-1);
8
9   //Send struct to PPE
10  __send_to_ppe(0x2126, 0, &s_LoopInformation);
11
12  if(id > 0) {
13    _i_LoopStart = s_LoopInformation.pos_start;
14    _i_LoopEnd = s_LoopInformation.pos_end;
15  }
16 }
```

Listing 5.16 shows the `f_GetLastLoopParameters()` that passes two parameters (`id` and `level`). The parameters are captured into a structure (`s_LoopInformation`) with additional information. These additional parameters include SPE-specific information such as the identifier (`_i_SPEIdentifier`), the loop number (`level`), request type (`COPY_BACK`[6]) and the previous loop identifier (which notifies the previous loop number[7]). This pre-computed value assists the PPE kernel to quickly perform a hazard check by ensuring that there are no current loop completions from other SPEs that may have caused a violation - see Section 3.2.

This structure is then passed using an RPC function the `__send_to_ppe()` function as discussed above. The `0x02126` is a hexadecimal standardised code, whereby both the PPE and SPE understand that the RPC is a request type. Once the structure is sent to the PPE, the kernel then waits for a signal.

Listing 5.17: `f_LoopBarrier()` function - SPE (`kernel.h`).

```
1 int f_LoopBarrier(int loop, int level) {
2   spe_sig_ob_t s_SPESignal;
3
4   s_SPESignal.id = loop;
5   s_SPESignal.loop_level = level;
6   s_SPESignal.spe = _i_SPEIdentifier;
7   __send_to_ppe(0x2127, 0, &s_SPESignal);
8
9   if(s_SPESignal.id!=SPE_CONTINUE)
10    COMListen(2);
11
12  return 0;
13 }
```

[6] `COPY_BACK` informs the PPE kernel that the SPE has completed its execution of the loop and requires the data to be written back.

[7] The previous loop number is a pre-computed value based the completed loop number.

The SPE kernel is notified through communication mechanisms (see Section 4.5.7) from the PPE. The SPE is then allowed to continue with the next set of instructions. Listing 5.17 shows the steps taken by the SPE to wait for a signal. An `spe_sig_ob_t` structure (instantiated as an object [`s_SPESignal`]) collates specific information such as loop number that was processed, its level and the SPE number and then passed to the RPC function (`__send_to_ppe`) and then the SPE waits for its signal hardware. Until a signal is received, the SPE polls its signal hardware at a pre-determined interval while running in lower power. This reduction in power consumption reduces the overall processor power envelope [Takahashi et al., 2005].

Listing 5.18: `COMListen()` macro that calls the `f_StopAndListen()` function - SPE (`kernel.h`).

```
1 unsigned f_StopAndListen(unsigned channel) {
2   ...
3   f_SignalMonitor(channel); //will return 0 when done
4   ...
5 }
```

Listing 5.18 partially shows the function's implementation (see Appendix C.4).

Listing 5.19: `f_SignalMonitor()` function - SPE (`kernel.h`).

```
1 /*Wait on signal register numbered #*/
2 unsigned f_SignalMonitor(unsigned int channel) {
3   if(channel==1) { do {} while(!spu_stat_signal1()); }
4   if(channel==2) { do {} while(!spu_stat_signal2()); }
5   return 0;
6 }
```

The `f_SignalMonitor()` function in Listing 5.19 requires a channel number that can only be either `1` or `2`[8]. When a channel number is specified, the appropriate `if`-statement is executed. The `spu_stat_signal1()`

[8]Each SPU has two signal channels [Gschwind et al., 2006].

or `spu_stat_signal2()` intrinsic Cell function is called. While no signal is received on the specified channel, the SPU will stall (wait) until a signal is received. Once a signal is received on the required channel number, the function will then return (exit) and the SPU is allowed to continue with the next instruction.

The reason that the L-API called the Loop Barrier function is to ensure that all SPEs have reached the same loop level. This ensures that all SPEs are synchronised and start on the next set of instructions in a ordered manner. The final function call in Listing 5.15 is the `f_DeregisterLoop()` function that uses an RPC function to inform the PPE kernel that the SPE has finished processing a `for`-loop.

5.6 Summary

To conclude this chapter has clearly outlined the Lyuba framework and the L-API that harnesses the Cell hardware capabilities, thereby extending the available hardware primitives as required by the L-API. Due to the architecture of the Cell processor and the programming paradigm, it is considerably different from that of a conventional programming environment where basically one compiler is required to transform code into machine level code. The Cell requires two compilers, hence the control and computation is split, with the latter executed on the SPEs.

The L-API has two main kernel images, again for the PPE and the SPEs. Moreover, the SPE kernels are identical, including the application/computation code. It should be noted that when SPEs are waiting for a signal or waiting for their outgoing mailbox messages to be consumed, the SPE will automatically reduce power by automatically switching to sleep mode. This reduces both static and dynamic power [Takahashi et al., 2005]. Another advantage of the Cell architecture is the memory address visibility; hence the location of data is visible but is only accessible through the kernels. There-

fore, the need of speculation is removed, but the philosophy of TLS remains, with each SPE context thread controlled – that is, stopped and re-executed – as needed, depending on violation detection. The following chapter will explore the results obtained from our experiments, with analysis.

Chapter 6

Results and Analysis

In Chapter 5, the L-API framework was discussed. This chapter presents and discusses experimental results, in particular, the execution times for each SPE and the impact of the transformations on load and store results. Following this, for each application, a detailed explanation of its parallelisation is reviewed. Having described parallelism, further results and observations derived from the experiments are discussed. Sequential algorithms are typically evaluated in terms of their execution time.

The execution time of parallel algorithms not only depends on the input size but also the number of processing elements used. Therefore, a parallel algorithm cannot be calculated in isolation from a parallel architecture without some loss in precision. An intuitive measure for analysing performance has been to use the wall clock time to solve a given problem on a given parallel platform. However, such a performance indicator cannot be applied to other problems with large processor element configurations. Paradoxically, using an increased number of hardware resources for computation may not enable a parallel program to run exponentially faster due to the overhead of parallelisation.

Both multicore and parallel systems require their processing elements to interact and communicate data with each other. Therefore, the communi-

cation of data between processing elements, whereby processor elements idle caused by load imbalance, synchronisation and the presence of serial code paths within a program and possibly hardware resource retention [Grama et al., 2003].

Table 6.1: Benchmark description.

Application	Description	Input type	Transformed lines of code	Dataset
SPR_{SM}	Sparse matrix multiply	Small test input	40/43	1000×1000 with 5000 Non-Zeros
SPR_{LG}	Sparse matrix multiply	Large test input	40/43	100000×100000 with 1000000 Non-Zeros
SOR_{SM}	Jacobi successive over-relaxation	Small test input	42/60	100×100 Grid
SOR_{LG}	Jacobi successive over-relaxation	Large test input	42/60	1000×1000 Grid
$LUCPYMX_{SM}$	Dense LU matrix factorization	Small test input	101/115	100×100 using partial pivoting
$LUCPYMX_{LG}$	Dense LU matrix factorization	Large test input	101/115	1000×1000 using partial pivoting
ARR_{SM}	Array 2D copy	Small test input	78/58	50 elements
ARR_{MM}	Array 2D copy	Medium test input	78/58	100 elements
ARR_{LG}	Array 2D copy	Large test input	78/58	1000 elements
FFT_{SM}	Fast Fourier transform	Small test input	163/177	2^{10} (=1024) Complex numbers
FFT_{LG}	Fast Fourier transform	Small test input	163/177	2^{20} (=1048576) Complex numbers

6.1 Results

The SciMark is a freely available benchmark consisting of five benchmarks [Pozo and Miller, 2004] that are specifically designed to test the performance of a CPU. The array of applications contained in the benchmark are representative of CPU-intensive workloads executed on high-performance processors and memory systems. For this study, the benchmark contained applications that last at least a few thousand instructions. Anything below this is considered too short an interval, compared to input/output device interaction with peripheral systems.

Therefore, only four applications from the SciMark benchmark were used for experimentation:

- Fast Fourier transform
- Jacobi successive over-relaxation
- Sparse matrix multiply
- Dense LU matrix factorisation

The Monte Carlo integration was not considered due to lack of sufficient thread-level parallelism (see Section 4.1). However, the Monte Carlo integration algorithm did rely upon an array copy algorithm that was clearly suitable for the experiment, the array 2D copy algorithm was extrapolated. Multiple input datasets were used for some of the applications, otherwise a large dataset was used by default.

Table 6.1 highlights specific parameters including a comparison of line numbers of untransformed and transformed code. Applying L-API transformations only increases the overall line number by an average of 2%. However, in order to support SPE, the logic code on the PPE is significantly larger (see Appendix B).

Each of the SciMark applications comes with two or three standard input datasets (and execution parameters) a test dataset (small): a training data

set (medium) and reference (large) data sets. Where applicable, all datasets were used for this research.

Due to long execution times for the large datasets, complete execution was still required to provide a detailed analysis and to determine completion status. For each application in the benchmark, approximately 10 iterations of the application were executed and analysed by extrapolating results from the L-API framework.

Parallelising the applications from this benchmark gives a good indication of performance output, resource utilisation and complexity of parallelising general-purpose applications. Another reason for using SciMark is that current and future researcher can easily and freely obtain the benchmark without incurring any financial cost.

L-API transformations were applied to each application using real hardware and the accompanying GNU compiler for the IBM Broadband Engine processor. The applications were profiled to determine the percentage of time spent in each basic block and function. Datasets were then executed on evenly distributed samples of execution from throughout the entire reference runs on all available SPEs. The samples that were generated were those that spanned almost the complete execution time. In other words, effectively the entire execution time of the program can be attributed to the L-API functions and loop execution. All results presented below are based upon the coverage and execution for each application in the benchmark.

It is important to understand the scope of applications selected because this defines the applicability and the limitations of the research experience gained using the L-API framework for parallelisation. Table 6.1 contains five applications from the SciMark benchmark that are coded in C/C++. L-API currently supports both C/C++ exclusively.

The L-API framework permits other researchers to utilise the results and method from this study to improve their techniques for derivative works on heterogeneous multicore systems.

The remainder of this chapter[1] outlines each application with L-API transformation analyses of the results gained from each application. Each application was initially parallelised using base parallelisation of loops and automatic load-store regions using L-API transformation. The following subsections detail the results for each application.

At the time of writing, this work is deemed to be a unique contribution, as no previous or current studies were found using a heterogeneous processor coupled with SciMark benchmark.

Complete program execution times include the contributions not only of primary subroutines such as load-store functions, but also of all other subroutines that are called throughout the program execution.

[1]See Appendix D for additional SPU results.

Table 6.2: Loop execution coverage (function names).

Application	Function Name
Sparse matrix multiply	`void SparseCompRow_matmult()`
Jacobi successive over-relaxation	`void SOR_execute()`
Array 2D copy	`void LU_copy_matrix()`
	`int LU_factor()`
Fast Fourier transform	`static void FFT_transform_internal(int direction)`
	`int FFT_bitreverse()`
	`void FFT_inverse()`

Table 6.3: Loop execution coverage (average).

	SPR	SOR	LUCPYMX	ARR	FFT
Number of outer loops	1	1	1 1	1	3 0 0
Execution time (%) outer loops	3%	3%	3% 3%	57%	32% 99% 99%
Number of inner loops	2	2	1 4	1	3 0 0
Execution time (%) inner loops	15% 59%	12% 68%	71% 15% 41% 12% 30%	12%	22% 7% 26% 0% 0%

6.1.1 Loop Coverage Analysis

Table 6.2 identifies the main methods that contain significant parallelism and where L-API functions were applied, and table 6.3 identifies key metric results of all applications in the SciMark benchmark that were parallelised using the L-API (see Chapter 4):

1. Most applications have one outer loop that varies in execution time (percentage) apart from dense LU factorisation that has two outer loops.

2. Sparse matrix multiply, Jacobi successive over-relaxation and dense LU factorisation applications outer loops account for 3% and array 2D copy equates to approximately 57% of overall execution time on the SPEs.

3. Fast Fourier transform applications have three outer loops (over three functions) that vary in execution percentile:

   (a) 32% (`static void FFT_transform_internal(int direction)`)

   (b) 99% (`int FFT_bitreverse()`)

   (c) and 99% (`void FFT_inverse()`), respectively.

In all applications, the inner loops outweigh in percentile coverage and are where significant processing takes place.

6.1.2 Sparse Matrix Multiply Application

The SPR application uses an unstructured sparse matrix stored in compressed-row format. This kernel exercises indirection addressing and non-regular memory references [Pozo and Miller, 2004].

Listing 6.1: Sparse application SPE code.

```
1 void SparseCompRow_matmult() {
```

```
2   unsigned int p;
3   unsigned int r;
4   unsigned int i;
5   double sum;
6   double *y = DATA_PTR_SECONDARY(___CACHE_DOUBLE, 0);
        //OUTPUT
7   double *x = DATA_PTR_PRIMARY(___CACHE_DOUBLE, 0);
        //INPUT
8
9   unsigned int rowR;
10  unsigned int rowRp1;
11
12  OuterLoop(0, Lower_IPC, Upper_IPC, p); <-[3%]
13  for(p=Lower_IPC; p<Upper_IPC; p++) {
14    InnerLoop(0, _i_SizeBeginPrimary, _i_SizeEndPrimary
          , r); <-[15%]
15    for (r=_i_SizeBeginPrimary; r<_i_SizeEndPrimary; r
          ++) {
16      sum = 0.0;
17
18      rowR = Load_AUX(2, r);
19      rowRp1 = Load_AUX(2, r+1);
20
21      InnerLoop(1, rowR, rowRp1, i); <-[59%]
22      for(i=rowR; i<rowRp1; i++) {
23        sum = sum + LoadD(x, Load_AUX(3, i)) * Load_AUX
              (1, i);
24      }
25      InnerLoopEnd;
26
27      StoreD(y, r, sum);
28    }
29    InnerLoopEnd;
30  }
31  OuterLoopEnd;
32 }
```

The `SparseCompRow_matmult()` (Listing 6.1) function requires the parallelising three loops: an outer loop and two inner loops. The first loop on line 12 is set as the outer loop with an L-API transformation applied, that consumes approximately 3% of the overall execution time. The second loop is parsed as an inner loop that consumes approximately 15% on line 14. The internals of the second loop have two load constructs and a single store construct. The second inner loop has multiple load constructs with approximately 59% consumed of each execution time.

Each loop is associated with a chunk of iterations that are calculated by the PPE L-API kernel. The consumption of all load requests either resides on the active SPE or in the LS of the remaining SPEs or in main memory. The L-API load instruction also performs checks to ensure the right data is used. Once the inner most loop completes, the PC performs a store L-API routine that simply stores data into the correct iteration.

All L-API loops perform an associated end loop routine that registers the end of loop execution with the L-API PPE kernel on the PPE. The remaining percentage of execution time is reserved for variable state and context maintenance, including monitoring mailbox, signal registers and PPE kernel maintenance routines.

Figure 6.1: Sparse matrix multiply application SPE execution time using a small dataset. Mean results after ten runs.

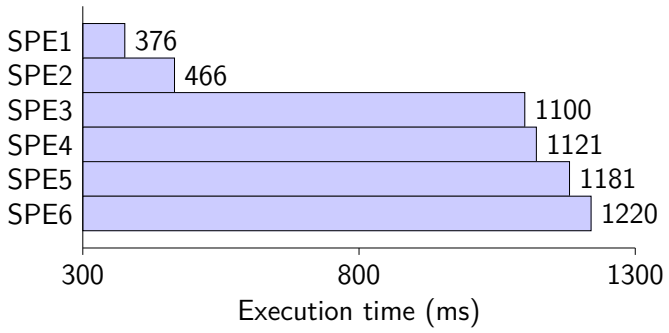


Figure 6.2: L-API PPE kernel functions for sparse matrix multiply application SPE execution time using a small dataset. Mean results after ten runs.

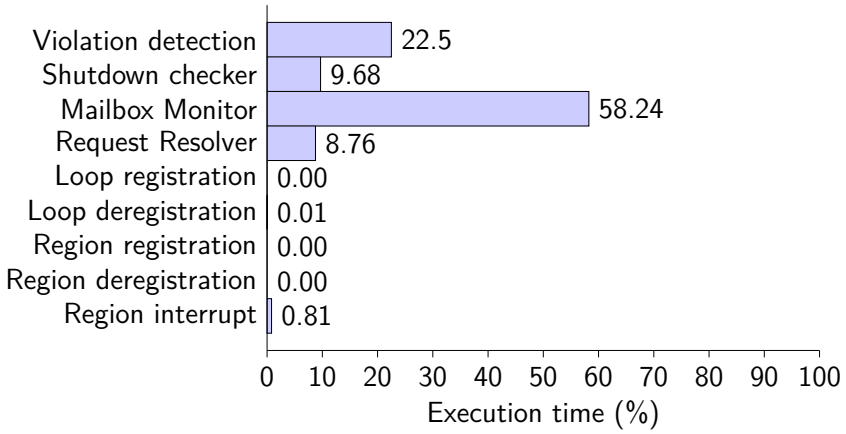
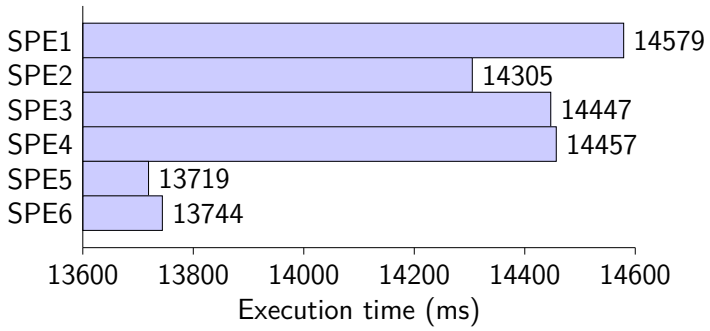


Figure 6.1 provides the computation times for all SPEs (mean time over 10 test runs) using a small dataset. SPEs 3 to 6 consume a similar amount of time, but SPEs 1 and 2 consume significantly less.

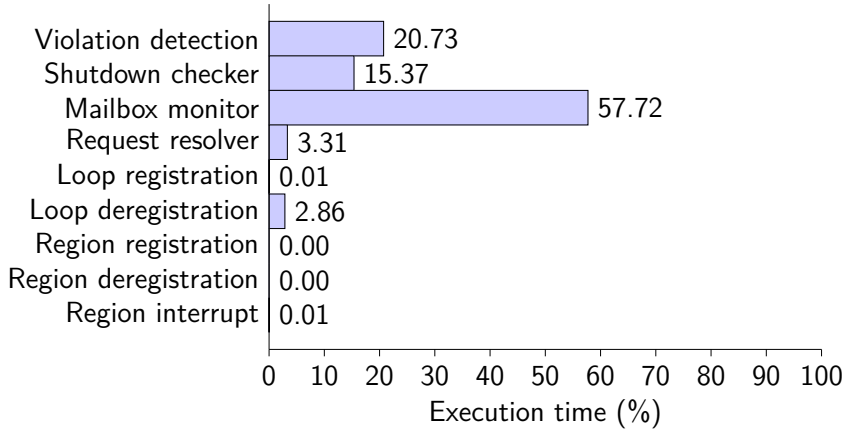
Accompanying the execution timings are the service callbacks for the sparse application (small dataset) in Figure 6.2. This figure shows the utilisation of callback functions performed on the PPE. Approximately 58% of execution time was devoted to mailbox monitoring, 22.5% used for recovery (violation detection), approximately 10% for checking shutdown requests (shutdown checker) and less than 0.9% used for the remaining functions. Even though the figure shows 0.00% for some methods, raw-results show some execution time for loop registration and deregistration but this occurs very quickly and does not consume significant amount of execution time.

Figure 6.3: Sparse matrix multiply application SPE execution time using a large dataset. Mean results after ten runs.



Combining results from Figure 6.1 and Figure 6.2 illustrates the difference in the SPEs execution time. A violation must have occurred which affected SPEs 3 – 6 and therefore the resolution and correct values with rollback must naturally increase overall execution time, as per Section 5.4.

Figure 6.4: L-API PPE kernel functions for sparse matrix multiply application SPE execution time using a large dataset. Mean results after ten runs.



Increasing the dataset to a larger input as shown in Figure 6.3, showed a reversed trend whereby, SPEs 1 – 4 show an increased execution time

compared to SPEs 5 - 6.

Utilisation of PPE functions which is shown in Figure 6.4 shows a similar outcome to a smaller dataset (see Figure 6.2), resulting in a deterministic pattern.

6.1.3 Jacobi Successive Over-Relaxation Application

The SOR (Jacobi successive over-relaxation) application operates on two datasets: 100×100 (SOR_SM) and 1000×1000 (SOR_LG) grids. Such an application can be used to solve a Laplace equation in 2D with Drichlet boundary conditions [Pozo and Miller, 2004].

Listing 6.2: Jacobi successive over-relaxation SPE code.

```
1 #include "kernel.h"
2
3 void SOR_execute() {
4   unsigned int M_size_start = _i_SizeBeginPrimary;
5   unsigned int M_size_end = _i_SizeEndPrimary;
6
7   unsigned int N_size_start = _i_SizeBeginSecondary;
8   unsigned int N_size_end = _i_SizeEndSecondary;
9
10  if(N_size_start==0) {
11    N_size_start++;
12  }
13
14  if(M_size_start==0) {
15    M_size_start++;
16  }
17
18  double omega = 1.25;
19
20  double omega_over_four = omega * 0.25;
21  double one_minus_omega = 1.0 - omega;
22
```

```
23   unsigned int p;
24   unsigned int i;
25   unsigned int j;
26
27   double *Gi;
28   double *Gim1;
29   double *Gip1;
30
31   OuterLoop(0, Lower_IPC, Upper_IPC, p); <-[3%]
32   for(p=Lower_IPC; p<Upper_IPC; p++) {
33
34     InnerLoop(0, M_size_start, M_size_end, i); <-[12%]
35     for(i=M_size_start; i<M_size_end; i++) {
36       Gi = Load_DPTR(i);
37       Gim1 = Load_DPTR(i-1);
38       Gip1 = Load_DPTR(i+1);
39
40       InnerLoop(1, N_size_start, N_size_end, j);
             <-[68%]
41       for(j=N_size_start; j<N_size_end; j++) {
42         StoreD(Gi, j, omega_over_four * (LoadD(Gim1, j)
               + LoadD(Gip1, j) + LoadD(Gi, j-1) + LoadD(
             Gi, j+1) + one_minus_omega * LoadD(Gi, j)))
             ;
43       }
44       InnerLoopEnd;
45     }
46     InnerLoopEnd;
47   }
48   OuterLoopEnd;
49 }
```

The `SOR_execute()` (Listing 6.2) function comprises three loops that are parallelised using the SPE L-API kernel constructs, as per Section 5.2. Overall, the format is similar to the sparse matrix application whereby there is a single outer loop with two inner loops. The outer loop consumes approxi-

mately 3%, followed by an inner loop that consumes 12% of execution time which performs three L-API load routines. The most that the inner loop consumes is approximately 68% of execution time, performing a store routine with five load routines.

Figure 6.5: Jacobi successive over-relaxation SPE execution time using a small dataset.

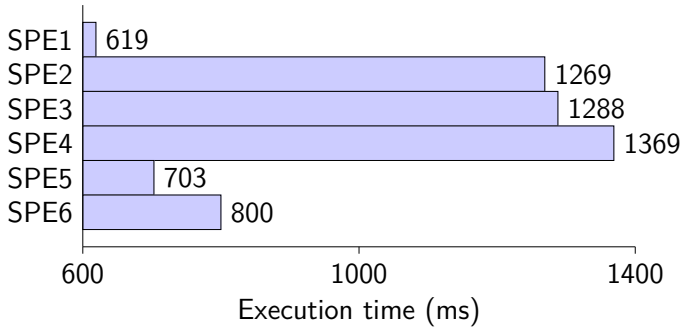


Figure 6.5 showcases results for all SPEs for the Jacobi application. SPEs 1, 5 and 6 show average execution times of 619ms, 703ms and 800ms, respectively. SPEs 2 to 4 show a larger average execution times of 1268ms, 1287ms and 1369ms. A reasonable conclusion to the increased execution time is due to recovery, whereby, a violation occurred and effected SPEs 2 to 4 (see Figure 6.6).

Figure 6.6: L-API PPE kernel functions for Jacobi Successive Over-Relaxation Application SPE Execution Time Using a Small Dataset. Mean results after ten runs.

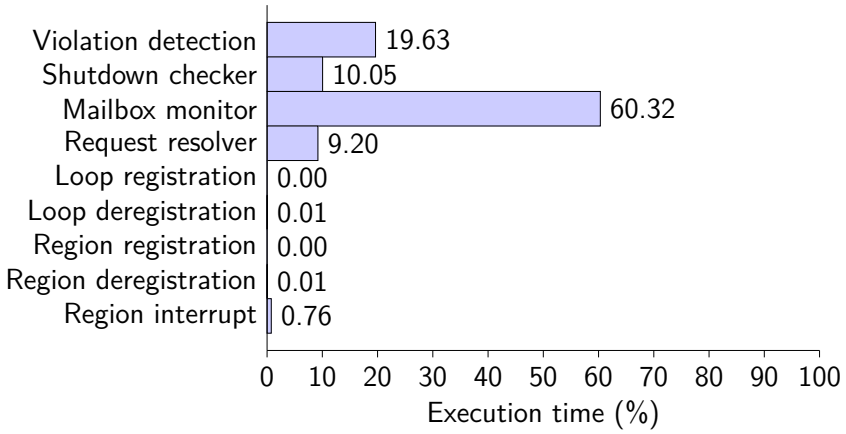
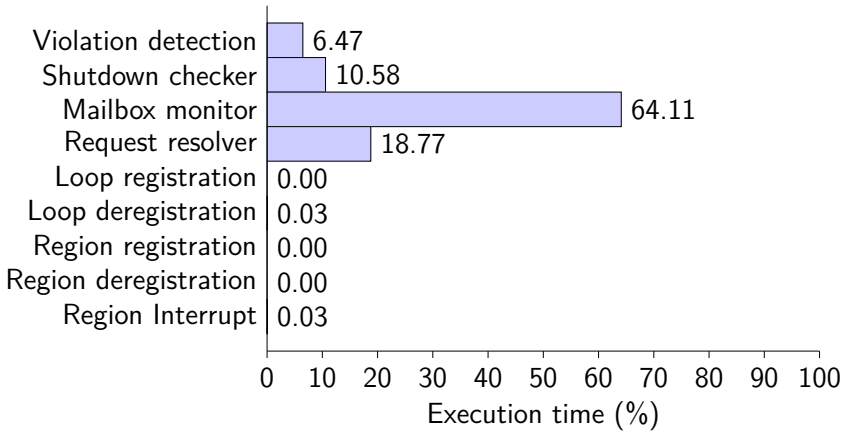


Figure 6.6 demonstrates the execution distribution as a percentile. Approximately 19.63% of execution on the PPE was devoted to resolution of violations (violation detection) and a larger percentage of 60% used for mailbox monitoring. The distribution is similar to the sparse matrix application (see Figure 6.2).

Figure 6.7: Jacobi successive over-relaxation SPE execution time using a large dataset.

Increasing the data input shows a distribution that tends to uniformity i.e. all SPEs complete in just over 30 seconds as shown in Figure 6.7.

Figure 6.8: L-API PPE kernel functions for Jacobi successive over-relaxation application SPE execution time using a large dataset. Mean results after ten runs.



The results from Figure 6.8 show a similar trend. However, there is a significant reduction in recovery (violation detection) by 13%. This could be explained by early violation detection and reducing the likelihood of violations at later stages of the application runtime, that is, increasing the dataset results in early violation detection and recovery.

Listing 6.3: Dense LU matrix factorisation application SPE code.

```
1 #include "kernel.h"
2
3 void LU_copy_matrix() {
4   double **A = (double **)Load_AUX_PTR2(1, 0, 0);
5   double **lu = (double **)Load_AUX_PTR2(2, 0, 0);
6   unsigned int i;
7   unsigned int j;
8   double data;
9
```

```
10    OuterLoop(0, Lower_IPC, Upper_IPC, i); <-[3%]
11    for(i=Lower_IPC; i<Upper_IPC; i++) {
12
13      InnerLoop(0, _i_SizeBeginPrimary, _i_SizeEndPrimary
            , i); <-[71%]
14      for(j=_i_SizeBeginPrimary; j<_i_SizeEndPrimary; j
            ++) {
15        Store_DDA(lu, i, j, Load_DDA(A, i, j));
16      }
17      InnerLoopEnd;
18    }
19    OuterLoopEnd;
20  }
21
22  int LU_factor() {
23    double **A = (double **)Load_AUX_PTR2(1, 0, 0);
24    int *pivot = (int *)Load_AUX_PTR(3, 0);
25    unsigned int minMN; // = M < N ? M : N;
26    unsigned int j = 0;
27    double *Aii;
28    double *Aj;
29    double AiiJ;
30    double ab;
31    int jj;
32    double recp;
33    double t;
34    unsigned int i;
35    unsigned int jp;
36    unsigned int k;
37    unsigned int ii;
38
39    OuterLoop(0, Lower_IPC, Upper_IPC, j); <-[26%]
40    for(j=Lower_IPC; j<Upper_IPC; j++) {
41
42      /* find pivot in column j and  test for singularity
            . */
43      jp=j;
```

```
44      t = fabs(Load_AUX2(1, j, j));
45
46      InnerLoop(0, j+1, _i_SizeEndPrimary, i); <-[15%]
47      for(i=j+1; i<_i_SizeEndPrimary; i++) {
48        ab = fabs(Load_AUX2(1, i, j));
49        if(ab > t) {
50          jp = i;
51          t = ab;
52        }
53      }
54      InnerLoopEnd;
55
56      StoreI(pivot, j, jp);
57
58      if(jp != j) {
59              /* swap rows j and jp */
60              double *tA = Load_AUX_PTR2(1, j, 0); //A
61              Store_DDA(A, j, 0, Load_DDA(A, jp, 0));
62              Store_DDA(A, jp, 0, tA[0]);
63          }
64
65          if(j<_i_SizeEndPrimary-1) { /* compute
                elements j+1:M of jth column  */
66              /* note A(j,j), was A(jp,p) previously
                    which was */
67              /* guaranteed not to be zero (Label #1)
                            */
68
69              recp =  1.0 / Load_DDA(A, j, j);
70
71              InnerLoop(1, j+1, _i_SizeEndPrimary, k);
                    <-[31%]
72              for (k=j+1; k<_i_SizeEndPrimary; k++) {
73                Store_DDA(A, k, j, Load_DDA(A, j, j) *
                      recp);
74              }
75              InnerLoopEnd;
```

```
76      }
77
78      if (j < minMN-1) {
79                  /* rank-1 update to trailing submatrix:
                           E = E - x*y; */
80                  /* E is the region A(j+1:M, j+1:N) */
81                  /* x is the column vector A(j+1:M,j) */
82                  /* y is row vector A(j,j+1:N)        */
83
84                  InnerLoop(3, j+1, _i_SizeEndPrimary, ii);
                        <-[12%]
85                  for (ii=j+1; ii<_i_SizeEndPrimary; ii++)
                        {
86                      Aii = Load_AUX_PTR(1, ii);//
                            Load_DPTR(ii); //A
87                      Aj = Load_AUX_PTR(1, j); //Load_DPTR(
                            j); //A
88                      AiiJ = LoadD(Aii, j);
89
90                      InnerLoop(4, j+1, _i_SizeEndSecondary
                            , ii); <-[20%]
91                      for (jj=j+1; jj<_i_SizeEndSecondary;
                            jj++) {
92                        StoreD(Aii, jj, LoadD(Aii, jj) -
                              AiiJ * LoadD(Aj, jj));
93                      }
94                      InnerLoopEnd;
95          }
96          InnerLoopEnd;
97      }
98  }
99  OuterLoopEnd;
100
101 return 0;
102 }
```

6.1.4 Dense LU Matrix Factorisation Application

The LUCPYMX application (dense LU matrix factorisation) computes the LU factorisation of a dense matrix using partial pivoting. The algorithm is the right-looking version of LU with rank-1 updates [Pozo and Miller, 2004]. The `LU_copy_matrix()` (Listing 6.3) function comprises five loops that are parallelised using the SPE L-API kernel constructs, as per Section 5.2.

The application comprises two functions that have multiple-loops L-API transformations. Firstly, the `LU_copy_matrix()` function contains a single outer loop that consumes 3% of overall SPE execution time and a further 71% of execution time is consumed by a single inner loop that simply performs a store routine on a double array with a single load routine. The second function in the application is the `LU_factor()`, which comprises a single outer loop and four inner loops. The outer loop consumes 16% of execution time with the first inner loop consuming 15%, second inner loop consuming 41%, third inner loop consuming 12% and the fourth (inner most) loop (embedded in the third loop) consuming approximately 20% of execution time.

Figure 6.9: Dense LU matrix factorisation SPE execution time using a small dataset.

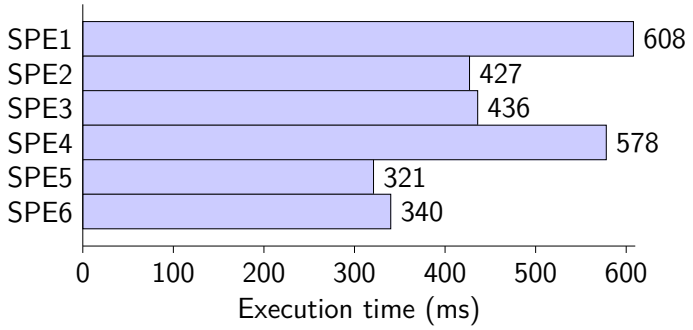
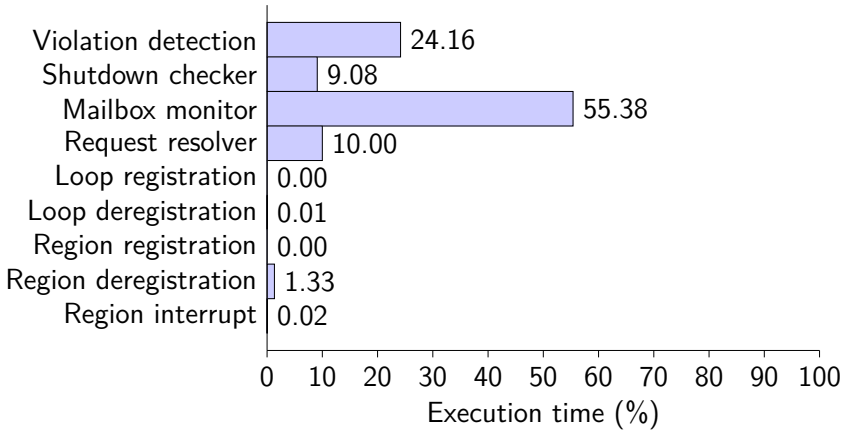


Figure 6.9 provides the computation times for all SPEs for the dense LU matrix factorisation application. SPEs 1 – 4 show a much larger execution time when compared to SPEs 5 – 6. Figure 6.10 will provide the most likely

Figure 6.10: L-API PPE kernel functions for dense LU matrix factorisation application SPE execution time using a small dataset. Mean results after ten runs.



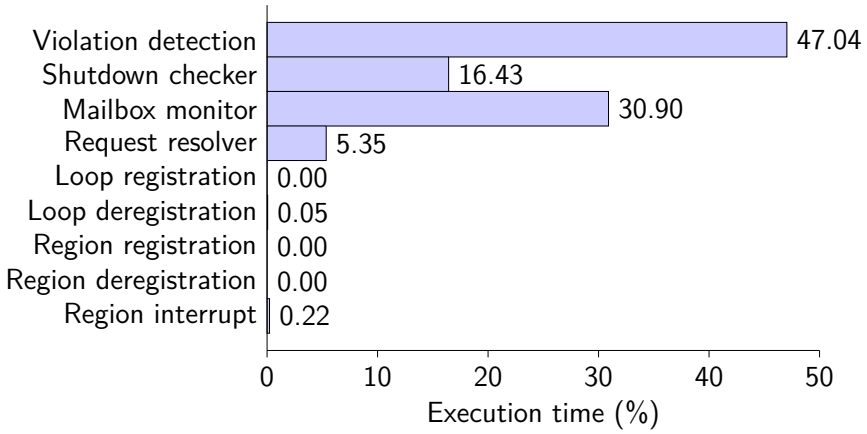
conclusion.

Figure 6.10 shows the execution in percentage format. The L-API PPE kernel consumed approximately 24% of execution time for recovery (violation detection) with a greater percentile (55%) monitoring mailboxes. Increasing the dataset presented a more uniformed distribution in SPE execution time.

Figure 6.11: Dense LU matrix factorisation SPE execution time using a large dataset.

Increasing the dataset size, significantly modifies the execution pattern,

Figure 6.12: L-API PPE kernel functions for dense LU matrix factorisation Application SPE execution time using a small dataset. Mean results after ten runs.



as shown in Figure 6.11. The execution times are fairly uniform, with all SPEs completing around a mean value of 2565ms. However, increasing the dataset size has also increased the violation recovery rate.

As shown in Figure 6.12, the L-API PPE kernel experiences a large percentage (47%) for recovery (violation detection) which explains the increased execution time in Figure 6.11 when compared to the results in Figure 6.9; similarly the increase in dataset size reflects the increased computation time.

$$y = \sqrt{1 - x^2} \tag{6.1}$$

6.1.5 Array 2D Copy Application

The ARR (Array 2D Copy) application is a simple 2D copy array algorithm. The array is extracted from the Monte Carlo integration application that approximates the value of pi by computing the integral of the quarter circle, Equation 6.1 on [0,1]. It chooses random points with the unit square and computes the ratio of those within the circle. The algorithm exercises random

number generators, synchronised function calls and function in lining.

Listing 6.4: Array 2D copy application SPE code.

```
1 #include "kernel.h"
2
3 void Array2D_double_copy() {
4     unsigned int remainder = _i_SizeEndSecondary & 3;
                /* N mod 4; */
5     unsigned int i=0;
6     unsigned int j=0;
7     double *Bi;
8     double *Ai;
9
10    OuterLoop(0, _i_SizeBeginPrimary, _i_SizeEndPrimary
          , i); <-[12%]
11    for (i=_i_SizeBeginPrimary; i<_i_SizeEndPrimary; i
          ++) {
12        Bi = Load_AUX_PTR(0, i);  //  double **B
13        Ai = Load_AUX_PTR(1, i);  //  double **A
14
15        InnerLoop(1, remainder, _i_SizeEndSecondary, j)
              ; <-[57%]
16        for (j=remainder; j<_i_SizeEndSecondary; j+=4)
              {
17            Store_SA_AUX(1, Ai, j, Load_SA_AUX(0, Bi, j
                  ));
18            Store_SA_AUX(1, Ai, j+1, Load_SA_AUX(0, Bi,
                      j+1));
19            Store_SA_AUX(1, Ai, j+2, Load_SA_AUX(0, Bi,
                      j+2));
20            Store_SA_AUX(1, Ai, j+3, Load_SA_AUX(0, Bi,
                      j+3));
21        }
22        InnerLoopEnd;
23    }
24    OuterLoopEnd;
```

```
25 }
```

The `Array2D_double_copy()` function (Listing 6.1.5) comprises two loops, which are parallelised using the SPE L-API kernel constructs (see Section 5.2). The application comprises two loops with a single outer loop consuming 12% of execution time that consists of two load pointer routines and a single inner loop consuming approximately 57%, which has three store routines with an associated one of three load routines.

Figure 6.13: Array 2D copy SPE execution time using a small dataset.

Figure 6.13 provides the computation times for all SPEs for the array 2D copy application. SPEs 1 – 4 have a much higher computation time when compared to SPEs 5 – 6.

Figure 6.14: L-API PPE kernel functions for array 2D copy application SPE execution time using a small dataset. Mean results after ten runs.

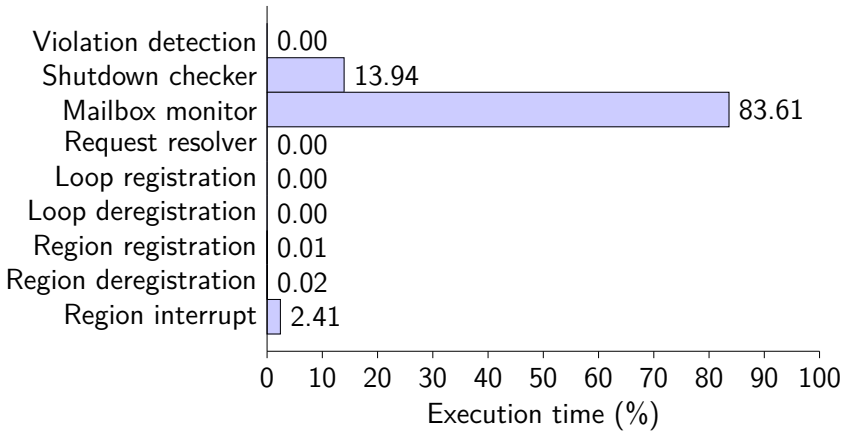
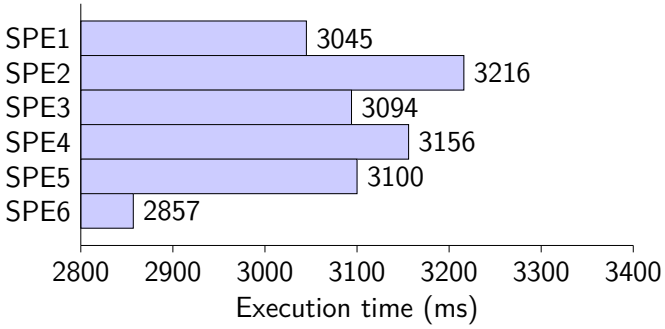


Figure 6.14 shows a higher utilisation for shutdown requests (Shutdown Checker), and a 13.94% utilisation for recovery and small percentage for the remaining functions. Therefore, SPEs 1 – 4 complete their execution approximately at the same rate as SPE 4 – 5. However, SPEs that complete their copy process must wait until all SPEs have completed execution.

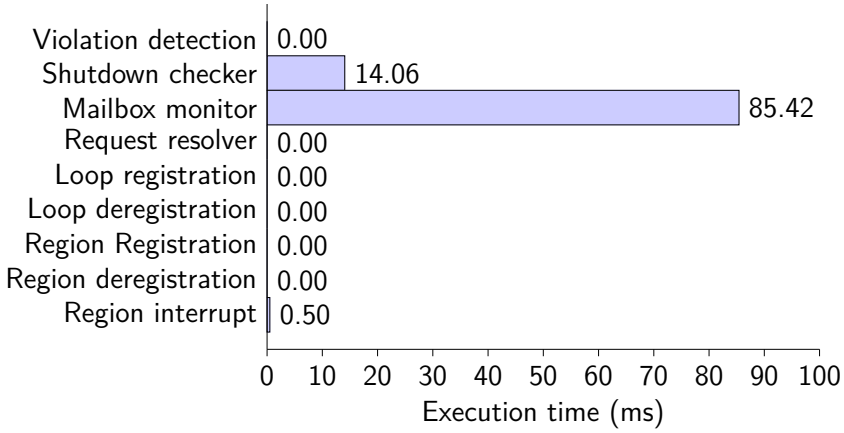
Figure 6.15: Array 2D copy SPE execution time using a medium dataset.



Increasing the data input size (medium) as shown in Figure 6.15 presents a similar result to a small dataset. However, SPEs 1 – 5 show similar execution

time with an average completion time of 3122 ms and SPE 6 with only 8.5% lower execution time of 2857 ms.

Figure 6.16: L-API PPE kernel functions for array 2D Copy application SPE execution time using a medium dataset. Mean results after ten runs.



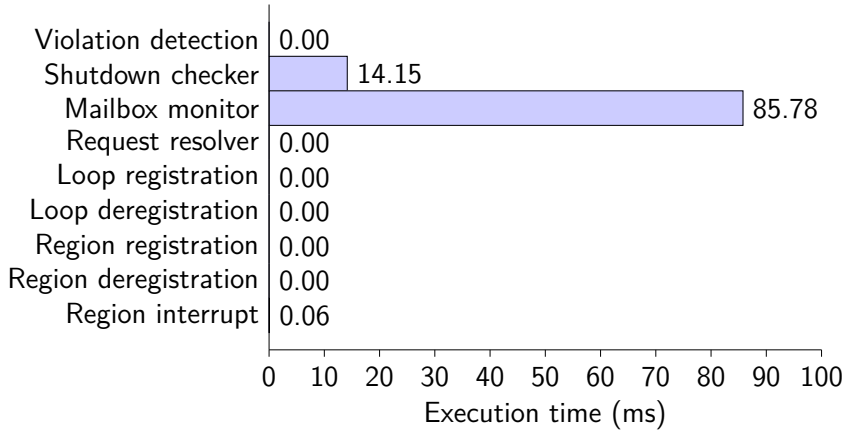
Results from Figure 6.16 do not show a significant change in PPE utilisation when compared to the results in Figure 6.14.

Figure 6.17: Array 2D copy SPE execution time using a large dataset.

Figure 6.17 shows an unified distribution of execution time. Therefore, increasing the dataset (large) the PPE is able to service all SPE requests at

similar timings.

Figure 6.18: L-API PPE kernel functions for array 2D copy application SPE execution time using a large dataset. Mean results after ten runs.



Results from Figure 6.18 do not show a significant change in PPE utilisation. The results in Figure 6.14 and Figure 6.16 result in similarity in execution time for all kernel functions.

6.1.6 Fast Fourier Transform Application

The FFT (fast Fourier transform) application performs a one-dimensional forward transform of 4000 complex numbers, demonstrating the use of complex arithmetic, shuffling, non-constant memory references and trigonometric functions. The application is split into two main functions, with the first part performing bit-reversal portion (no flops) and the second part performing the actual $Nlog(N)$ computational steps [Pozo and Miller, 2004].

Listing 6.5: FFT Application SPE code.

```
1 #include "kernel.h"
2 #include <simdmath.h>
3
```

```
4 #define PI  3.1415926535897932
5
6 static void FFT_transform_internal(int direction) {
7     /* bit reverse the input data for decimation in
          time algorithm */
8     double *data = Load_DPTR(0);
9
10    unsigned int bit;
11    unsigned int dual = 1;
12
13    double w_real;
14    double w_imag;
15    unsigned int a;
16    unsigned int b;
17
18    double theta;
19    double s;
20    double t;
21    double s2;
22
23    int i;
24    int j;
25
26    double wd_real;
27    double wd_imag;
28
29    double tmp_real;
30    double tmp_imag;
31
32    double z1_real;
33    double z1_imag;
34
35
36    FFT_bitreverse();
37
38    //printf("FFT_bitreverse done %i \n",
          _i_SPEIdentifier);
```

```
39
40      /* apply fft recursion */
41      /* this loop executed int_log2(N) times */
42
43      OuterLoop(0, Lower_IPC, Upper_IPC, bit); <-[32%]
44      for(bit = Lower_IPC; bit < Upper_IPC; bit++, dual
            *= 2) {
45          w_real = 1.0;
46          w_imag = 0.0;
47
48          theta = 2.0 * direction * PI / (2.0 * (double)
                dual);
49          s = sin(theta);
50          t = sin(theta / 2.0);
51          s2 = 2.0 * t * t;
52
53          InnerLoop(0, _i_SizeBeginPrimary,
                _i_SizeEndPrimary, bit); <-[22%]
54          for(a=_i_SizeBeginPrimary, b =
                _i_SizeBeginPrimary; b < _i_SizeEndPrimary;
                 b+= 2 * dual) {
55              i = b;
56              j = (b + dual);
57              wd_real = LoadD(data, j);
58              wd_imag = LoadD(data, j+1);
59              StoreD(data, j,  LoadD(data, i+1) - wd_real
                    );
60              StoreD(data, j+1, LoadD(data, i+1) -
                    wd_imag);
61              StoreD(data, i,   LoadD(data, i) + wd_real)
                    ;
62              StoreD(data, i+1, LoadD(data, i+1) +
                    wd_imag);
63          }
64          InnerLoopEnd;
65
66          /* a = 1 .. (dual-1) */
```

```
67          InnerLoop(1, 1, dual, a); <-[7%]
68          for(a = 1; a < dual; a++) {
69              /* trignometric recurrence for w-> exp(i
                    theta) w */
70              tmp_real = w_real - s * w_imag - s2 *
                    w_real;
71              tmp_imag = w_imag + s * w_real - s2 *
                    w_imag;
72              w_real = tmp_real;
73              w_imag = tmp_imag;
74          }
75          InnerLoopEnd;
76
77          InnerLoop(2, _i_SizeBeginPrimary,
                _i_SizeEndPrimary, b); <-[26%]
78          for(b = _i_SizeBeginPrimary; b <
                _i_SizeEndPrimary; b += 2 * dual) {
79              i = 2*(b + a);
80              j = 2*(b + a + dual);
81              z1_real = LoadD(data,j);
82              z1_imag = LoadD(data,j+1);
83              wd_real = w_real * z1_real - w_imag *
                    z1_imag;
84              wd_imag = w_real * z1_imag + w_imag *
                    z1_real;
85              StoreD(data, j,   LoadD(data,i) - wd_real);
86              StoreD(data, j+1, LoadD(data,i+1) - wd_imag
                    );
87              StoreD(data, i,   LoadD(data, i) + wd_real)
                    ;
88              StoreD(data, i+1, LoadD(data, i+1) +
                    wd_imag);
89          }
90          InnerLoopEnd;
91      }
92      OuterLoopEnd;
93  }
```

```
94
95 int FFT_bitreverse() {
96   int N = _i_SizeEndPrimary; //_i_SizeEndPrimary;
97   double *data = Load_AUX_PTR(1, 0);// Load_DPTR(0);
98
99   /* This is the Goldrader bit-reversal algorithm */
100  unsigned int n=N/2;
101  unsigned int nm1 = n-1;
102  unsigned int i=0;
103  unsigned int j=0;
104  int ii;
105    int jj;
106    unsigned int k;
107    double tmp_real;
108    double tmp_imag;
109
110    OuterLoop(0, 0, nm1, i); <-[99%]
111    for(i=0; i < nm1; i++) {
112
113        /*int ii = 2*i; */
114        ii = i << 1;
115
116        /*int jj = 2*j; */
117        jj = j << 1;
118
119        /* int k = n / 2 ; */
120        k = n >> 1;
121
122        if (i < j) {
123          tmp_real = Load_AUX(1, ii); //LoadD(data, ii)
               ;
124            tmp_imag = Load_AUX(1, ii+1); //LoadD(data,
                 ii+1);
125            Store_AUX(1, ii, Load_AUX(1, jj));
126            Store_AUX(1, ii+1, Load_AUX(1, jj+1));
127            Store_AUX(1, jj, tmp_real);
128            Store_AUX(1, jj+1, tmp_imag);
```

```
129             }
130
131             while (k <= j && (k!=0 && j!=0)) {
132                 /*j = j - k ; */
133                 j -= k;
134
135                 /*k = k / 2 ;  */
136                 k >>= 1;
137             }
138
139             j += k;
140         }
141         OuterLoopEnd;
142
143         return 0;
144 }
145
146 void FFT_inverse() {
147   int N = _i_SizeEndPrimary;
148   double *data = DATA_PTR_PRIMARY(___CACHE_DOUBLE, 0);
149   int n = N/2;
150   double norm = 0.0;
151   unsigned int i;
152
153   FFT_transform_internal(+1);
154
155   /* Normalize */
156   norm=1/((double) n);
157
158   OuterLoop(0, _i_SizeBeginPrimary, _i_SizeEndPrimary,
          i); <-[99%]
159   for(i=_i_SizeBeginPrimary; i<_i_SizeEndPrimary; i++)
          {
160     StoreD(data, i, LoadD(data, i) * norm);
161   }
162   OuterLoopEnd;
163 }
```

The `FFT_transform_internal()` function (Listing 6.5) comprises three functions that are parallelised using the SPE L-API kernel constructs - see Section 5.2. The FFT application is built from three functions: the first function, `static void FFT_transform_internal(int direction)`, consists of a single outer loop that consumes approximately 32% of execution time followed by an inner loop that performs a six load routines and four store routines that consumes 22% of execution time. A second inner loop performs no L-API load or store functions, followed by a third inner loop that is similar to the first inner loop that consumes approximately 26% of execution time. The second function `int FFT_bitreverse()` consists of a single loop with multiple load and store routines which consumes approximately 99% of execution time. The final function in the application is the `void FFT_inverse()` which all consists of a single loop that consumes 99% of the execution time.

Figure 6.19: FFT application SPE execution time using a small dataset.

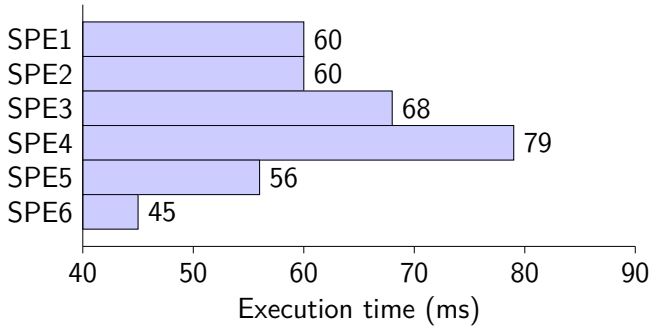
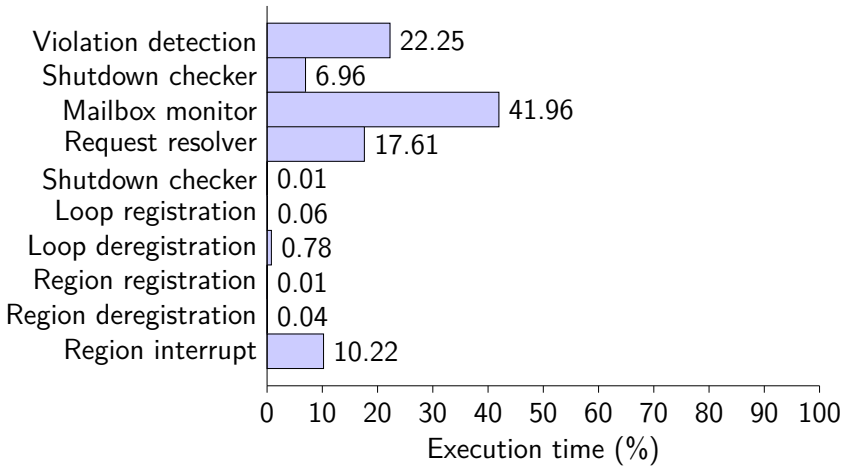


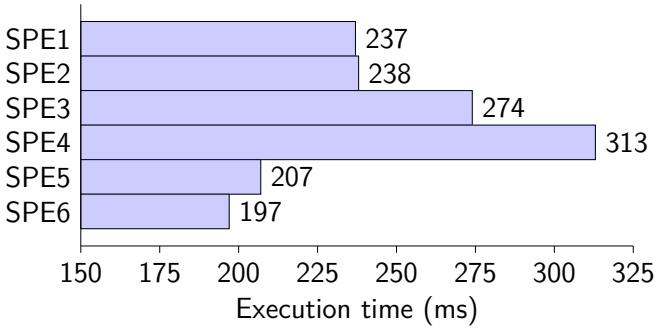
Figure 6.19 provides the computation times for all SPEs for the FFT application. SPEs 1 - 3 and 5 complete execution at an average of 61 ms; SPE 4 completes in a much higher time of 79 ms and SPE 6 completes in a much lower time of 45 ms.

Figure 6.20: L-API PPE kernel functions for FFT application SPE execution time using a small dataset. Mean results after ten runs.



Results from Figure 6.20 show a sporadic change in PPE utilisation. Recovery (violation detection) consumes an approximate 22.25%, shutdown requests (shutdown checker) utilises 6.96%, mailbox monitor utilises a larger 41.96%, region interrupt consumes 10.22% and the remaining functions consume less than 1%.

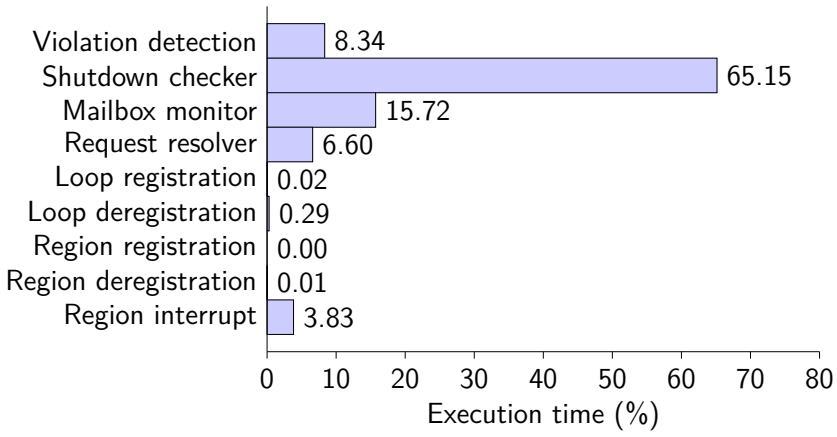
Figure 6.21: FFT application SPE execution time using a large dataset.



Increasing the dataset size increases the computation time for each SPE, as shown in Figure 6.21. In comparison with Figure 6.19, the increases in

computation times follow a similar trend with the only difference being the increased time per SPE.

Figure 6.22: L-API PPE kernel functions for FFT application SPE Execution time using a large dataset. Mean results after ten runs.



However, the results from Figure 6.22 reflect a different trend when compared to Figure 6.20, with mailbox monitoring consuming a greater percentage for a small dataset, the current results for a larger dataset, the mailbox monitoring is significantly reduced to 15.72%, with shutdown request consuming the largest percentage of 65.15%.

Table 6.4: Completion state matrix for SciMark application without L-API transformations. Application with a Yes status for completion represents a successful execution. Partial status represents the incorrect result generated from execution. Failed status represents a complete failure of the application, execution failure.

Application	Dataset	Completed	Description
Sparse matrix multiply	Small	Yes	Passed
Sparse matrix multiply	Large	Yes	RAW
Jacobi successive over-relaxation	Small	Partial	RAW
Jacobi successive over-relaxation	Large	Partial	RAW
Dense LU matrix factorisation	Small	Yes	Passed
Dense LU matrix factorisation	Large	No	Failed
Array 2D copy matrix	Small	Yes	Passed
Array 2D copy matrix	Medium	Partial	RAW
Array 2D copy matrix	Large	Partial	RAW
Fast Fourier transform	Small	Yes	Passed
Fast Fourier transform	Large	No	Failed

6.2 Comparative Analysis

The previous subsections provide results for all applications using the L-API framework. This section explores the same applications without utilising the L-API framework, thus presenting results and observations when attempting to apply the original benchmark code to a single SPE – see Appendix E[2].

Table 6.4 lists the results when strictly applying the SciMark benchmark to the IBM Cell processor without any L-API transformations. All applications with a small dataset successfully complete execution without any issues. However, increasing the dataset size for all applications resulted in either a partial or failed execution.

Increasing the dataset size resulted in either RAW or complete application failure whereby the system was unable to recover (application crashed), causing the default operating system operation of application process thread

[2]See Figure E.1.

termination. This behaviour was most common throughout the test runs. On a few occasions, most applications did complete successfully with a large dataset. However, this occurrence was very rare and therefore the application was considered to have failed.

In contrast, the array 2D copy matrix application completed on all datasets, but increasing the dataset size did cause some of the SPEs to use the wrong data due to RAW hazards. In each cases, the active SPE would use stale data when it should have waited to use the updated data which resulted in partial completion but an incorrect final result.

Table 6.5: L-API performance improvement matrix.

Application	Dataset	L-API
Sparse matrix multiply	Small	22.5%
Sparse matrix multiply	Large	20.73%
Jacobi successive over-relaxation	Small	19.63%
Jacobi successive over-relaxation	Large	6.47%
Dense LU matrix factorisation	Small	24.16%
Dense LU matrix factorisation	Large	6.47%
Array 2D copy matrix	Small	0%
Array 2D copy matrix	Medium	0%
Array 2D copy matrix	Large	0%
Fast Fourier transform	Small	22.25%
Fast Fourier transform	Large	8.34%

Applying the L-API framework (transformations to the code) resulted all in applications executing successfully and, more importantly, Table 6.5 outlines the actual performance improvement gained when using the L-API for each application.

Taking all the L-API results, divided by the total number of applications, gives an average 11% recovery improvement and all application completed successfully when using the L-API framework, excluding those applications where no improvement at all was seen, the speed increased on average by 16.3%. Another observation made affecting the framework is the dataset

size, such that, increasing the dataset reduces the recovery time for each application, as fewer violations are encountered when compared to a small dataset. The framework overhead is consistently higher, with increased violation recovery, but still resulting in a successfully completed application.

6.3 Summary

All applications in the SciMark benchmark were simple to parallelise using the L-API constructs. L-API kernels, operating on both processor types, functions consistently, with kernel functions operating as designed. Loops were successfully transformed with constructs, with the coupling of load and store constructs resulting in the degradation of the overall performance.

Interestingly, in order for correct use of data by an SPE, load and store constructs were absolutely imperative and therefore SPEs did result in a suspended state whereby the entire SPE context would wait until a signal message was received from either an SPE or the PPE. This is normal architectural behaviour when employing SPE mailboxes. This can be considered as a benefit because of the reduced overall power consumption when an SPE is in a suspended state. However, no other work can be done by the SPE until a signal arrives. An SPE would typically wait for a signal notification, otherwise mailboxes are utilised while the SPE context is in an active processing state.

With the use of a single SPE the results reflected more favourably towards non-L-API transformed code due to large overheads. Results from L-API SPE kernels for each application show a greater dependence on the odd-pipeline (see Section 4.5 and Appendix D), moreover such functions (L-API SPE kernel functions) would operate more consistently on traditional processor architectures. However, the original design of the SPEs was to accelerate mathematical code and it was not designed for the general-purpose algorithms that are prevalent in the SPE L-API kernels.

When comparing both non- and with L-API versions of the applications, the comparisons differed quite significantly. Executing the SciMark framework on multiple SPEs resulted in either successful, partial or failed execution. However, applying the L-API framework resulted in all applications executing successfully with an overall 11% improvement recovery rate.

Chapter 7

Conclusion

Chip-multicore processor dies continue to shrink in dimensions, while traditional microarchitectural processors that are used to deliver higher performance, with deeper pipelining, have become increasingly expensive, despite their diminished performance improvements. Workloads with inherent parallelism are able to take advantage of such processors. This thesis proposes software support for a heterogeneous chip-multicore processor using a bespoke software framework (L-API) that empowers programmers to harness the underlying hardware through simple API calls.

The two fundamental research questions that have driven this study are: how can dependencies presented in complex code, which use both hardware and software, be handled and how can workloads be evenly distributed across an asymmetrical processor.

Dependencies are handled by the L-API framework that handles low-level data hazards and automatically recovers the application from such hazards without programmer intervention.

The L-API framework ensure workloads are distributed across an asymmetrical processor by decomposing `for` loop iterations into groups. The total number of groups generated by the L-API framework equals the total number of SPEs available on the Cell processor. Each group is then controlled by

the PPE while ensuring workloads (groups) are maintained and monitored by the PPE L-API kernel.

This thesis is validated through a detailed evaluation of SciMark [Pozo and Miller, 2004] benchmark applications measured on a realistic IBM simulator and the IBM Cell chip-multicore processor.

The next section reviews in detail the findings and contributions from this thesis and then discuss possible directions for future work. This is followed by summarising the most important lessons to be taken from this research.

7.1 Findings

The results and analysis presented in Chapter 6 provide a detail analysis of the L-API framework applied to five applications from the SciMark benchmark. This section will briefly describe the findings captured from the analysis presented in the previous chapter.

The findings are grouped as follows:

1. Without the L-API framework, not all applications were able to complete using all available datasets.

2. When the L-API framework is applied to any application from the SciMark benchmark using any dataset, all applications complete without error.

3. Meaningful violations are detected by locating precise memory addresses. This is achieved through Cell's DMA mechanism coupled with L-API framework that tracks in real-time the location of all data variables used (provided those variables are registered with L-API) that may or may not produce a data violation.

4. Applications using a small dataset saw an improvement of 17% in completion. Completion is defined as successful violation detection, res-

olution and rollback (if applicable) when compared to a non-L-API version.

5. Applications using a larger dataset saw an improvement of 16% in completion when compared to a non-L-API transformation. Completion is defined as successful violation detection, resolution and rollback (if applicable) when compared to a non-L-API version.

6. Overall improvement recovery rate was 11% (mean).

7. Not all applications were successfully completed when the L-API is not applied. Also, for some applications that did complete successfully, the final calculated result was incorrect (see Table 6.4).

8. The L-API utilised a significant number of available resources from the Cell microprocessor for the L-API framework.

9. The L-API framework is modular in software design but the communication mechanisms are heavily reliant on the Cell's underlying communication mechanisms, in particular DMA. However, through additional development, non-homogeneous processes can be supported.

The following section presents the main contributions from this research.

7.2 Contributions

The contributions of this study are located in the nexus between hardware and software, this is hypothesised into three groups:

1. A software-only approach does not take into account the functionality of hardware and therefore, a brute-force approach is implemented; moreover the results obtained may not reflect positively, and therefore the conclusion of hardware inability is the root cause.

2. A hardware-only approach using speculative assistance and cache line policies accelerates detection and enables correct dispersion of tasks across the processing cores. The programmer does not have any significant control over hardware operation (see Section 3.4.4).

3. A hybrid approach allows researchers to focus on the cooperation between software and hardware to perform the required task and accelerate the performance from both aspects more effectively (see Chapter 3).

Only when both elements work in harmony with a degree of synchronisation towards a mutual goal is a task completed. Hardware and software need to coexist and function more effectively on current and new generation of multicore processors. Developing software that aids the detection of parallelism, and hardware to provide the transits, means that data movements across components can be better achieved through a greater understanding of both hardware functionality and software capability to enhance the given functionality from the available hardware.

The true test of any framework is its application to real-world code and systems. The Cell CMP presents itself as a challenge. However, due to its structure and hardware a programmer can easily visualise and implement systems to extrapolate potential performance while retaining software communication through software control more readily. The contribution offered by the framework presented in this research attempts to demonstrate this marriage and in doing so further enhances the point of heterogeneous microprocessor architecture.

7.2.1 The Heterogeneous Approach

Compared to previous approaches that utilise TLS, this thesis contributes a cooperative approach that exploits the respective strengths of software and hardware and fortifies the interface between them. In other words, maximum

utilisation of the hardware is the priority; this creates efficiency savings with speed-up as a result. This could potentially facilitate current processes to be completed using more energy-efficient processors, saving time, power and money; playing a part in reducing greenhouse emissions and for this reason the ability to control hardware was vital and why the Cell was chosen. This thesis utilises most of the available components of the Cell processor to offload the logic overheads from the main processor.

Many proposals utilising generic processors have limited hardware control (see Section 3.4.4), requiring a greater dependency on abstracted software libraries with modified compiler support.

Through a new software framework whose instructions harness the underlying hardware, this in turn frees the developer and hardware from the burden of breaking programs into threads and to track registers from dependencies between them, while empowering the software (L-API) to deterministically parallelise programs without the cost and overheads of speculation.

This cooperative approach has many advantages over those that used either software or hardware in isolation, and it allows the implementation of aggressive software optimisations that harness the underlying hardware and distribution logic computation ,and minimises both the complexity of confusing logic and the external costs of speculative instructions and additional computation overheads.

7.2.2 A Unified Support

Fully utilising the strengths of both hardware and software assists in the creation of efficient code and is considered to be vitally important. Thread-level speculation has matured but its premise of improved detection of data hazards (see Chapter 3) is quite limited. Speculation does not take full consideration of hardware traits and power consumption. Results from past research (see Section 3.1, 3.3, 3.4 and 4.3) use the SPEC benchmark which itself is oriented to scientific applications rather than everyday uses. More-

over, commercial code and general-purpose code have more in common and therefore benchmarks such as SPEC cannot be considered to be an ideal measure of performance.

Previous TLS studies are supported by hardware speculation but there is no hardware support of this nature used in this thesis which is unique because the implementation of the framework is modularised and can be adapted to the architectures with appropriate modification (however, some considerable effort is required but implementation is achievable).

This thesis has demonstrated software and hardware working in tandem in order to achieve a greater level of utilisation and there after focus on true performance achievement. With homogeneous multicore processors, a significant amount of hardware control is not visible to the programmer and therefore they have to rely on software libraries to achieve the required task, it is common to as seen in many past and current studies.

Moreover, heterogeneous processors such as the IBM Cell processor allow significant hardware access to better parallelise and utilise the available hardware. Traditional multicore processors (symmetric) have complex cache policies which are hidden from the programmer, and the application implementation must rely on given hardware logic to achieve a greater degree of parallelism which is not always the case (see Section 3.4.4).

7.2.3 A Comprehensive Evaluation of the L-API Framework

Another key contribution is the framework itself – see Chapter 5. It is designed to be simple to use by the programmer, while simultaneously harnessing the underlying hardware for bespoke software logic control and computation. Recovery is dealt with almost instantaneous, proper coordination of data repositories in both off- and on-chip memory on multiple processors.

Past research focused mainly on detailed speculation, which in most cases did not resolve hazards without increased overheads; the framework pre-

sented here provides comprehensive data distribution and hazard recovery without the need of any forms of speculation. The L-API framework exploits the underlying heterogeneous hardware efficiently.

7.3 Hindrances to the L-API Framework

During this study a number of limitations and hindrances were found, which will be addressed in this section. SciMark (a benchmark with multiple applications) is inherently straightforward to parallelise using the constructs from the Lyuba-Application Programming Interface (L-API) library[1]. However, analysis of results has identified an area of improvement - the SPU pipeline constrains and limits due to it not being a true dual issue pipeline[2] (as found in general-purpose processors).

It was found that hardware itself created two bottlenecks. Firstly, due to the hardware limitations[3] of the SPE the L-API interface it is limited in its ability to process `if` statements, with associated branching, in comparison to other types of processors. Secondly, the code relating to mathematical operations should be optimised to better exploit hardware within the SPUs pipeline through the Cell SDK. Techniques such as branch processing and branch prediction instructions (for branching) should be enforced, notifying the SPU to stop fetching instructions and start processing instructions at a new program counter (new address) of instructions. Use of conditional (dependence on branch address comparison) and unconditional (constant branch address) would aid unnecessary `if` statement instructions and therefore reduce overall BR functional unit usage.

[1] see Chapter 5 for the L-API interface.

[2] see Chapter 6 and subsection 4.5.2.

[3] The IBM Cell has been specifically designed for fast vector processing and the SPEs are not ideally suited for general purpose code statements - `if` statements. These large numbers of questions challenge the SPE. However, due to the limited availability of heterogeneous microprocessors, the Cell was the only choice due to its availability and the options to program it.

However, the results do not show a huge improvement in performance of processed code. Also, research presents a framework that efficiently utilises the available hardware that can assist software frameworks to allow the extrapolation of sequential code. Research from past studies (see Chapter 3) places a greater focus on hardware implementations through a coarse-grained threaded pipelining system with software assistance; however, such research does not detail specific hardware resource utilisation.

The PPE element requires additional functional assets in order to cope with a large number of requests emanating from SPEs. If SPUs had a true dual-issue pipeline and did not share certain hardware resources while allowing multiple threads (minimum of two threads) to execute in parallel, then the issues outlined above could be alleviated to some degree.

Furthermore, L-API kernels functioning on the SPEs are increasingly too dependent on the PPE to resolve simple queries; hence, the waiting times (results[4]) of the SPE were significantly large. Another limitation was measuring the power usage of the processor when executing applications in the SciMark benchmark; unfortunately avail such a resource was unable for this study.

Taking into consideration all the factors involved throughout this research, the findings are as follows:

1. Developing a framework to detect hazards on a multicore processor is complex.

2. Developing on a heterogeneous multicore processor such as the IBM Cell Broadband Engine was difficult to begin with, but once I understood the concepts and abstraction of the different available libraries for the Cell, programming became less complex.

3. The Cell itself is more suited to fast SIMD vector type code, especially for the SPEs.

[4]See Chapter 6 for results and observations.

4. Hardware synchronisation is imperative to support thread-shared and global variables. Mutexes provide course-grained locks, but for fine-grain control, hardware synchronisation locks provide more control.

5. The L-API did not fully utilise all applicable resources of the SPE's pipeline. Further optimisation of the L-API could attain a greater utilisation of code with mathematical operators. In other words, code that exhibits extensive mathematical operation; the L-API can detect hazards with such operators. Currently, the L-API only protects the application from hazards that are within shared or global variables.

6. Instead of focusing on speed-up equations, I believe the focus for measurement must be shifted from Amdhal's law to microprocessor hardware utilisation. This metric provides a more realistic picture and can enhance the way systems use resources that are underutilised and attempt to optimise software components of a system that over-utilise hardware components.

7. The L-API over-utilised the PPE such that each SPE would send multiple requests for an active iteration; this must be reduced to one request for one iteration of a loop. As each SPE waited for a response, the PPE was not able to process all requests immediately. Therefore, the next version of the L-API must delegate response to in active SPEs, possibly using the overlay technique.

8. This needs to be a focus on modularity in software designs and possibly the application of L-API to other platforms including graphic processor units (GPUs).

However, problems do arise in heterogeneous processors: controlling the actions of specific components becomes complex, ensuring utilisation remains consistent and resource management may become tedious including, ensuring that the pipelines are kept busy with correct types of work. These software

and hardware limitations are evident in the current generation of the IBM Cell processor – see Chapter 4.

7.4 Future Work

Heterogeneous architectures are beginning to have a greater presence in both commercial and mobile platforms including graphic processor units (GPUs). Future work could entail the use of either parts of, or the complete, L-API framework. Due to its modular design and ease of use a greater focus can be placed on the development and enhancements of data hazard detection and recovery on such architectures as GPUs. The importance of hardware control should be considered by the programmer to further extrapolation of parallelism and improve hardware utilisation, accordingly.

As more scientific data is processed on GPUs due to vector-type data (inherently GPUs have a considerable number of vector processing units). Due to L-API modular design, the framework (L-API) has an opportunity to harness and provide value using GPUs. GPUs by design are heterogeneous, and the applicability of the L-API framework to such an architecture is well suited. However, the L-API currently supports scalar data types such as arrays (see Chapter 5), and therefore the framework must implement vector data which will require further enhancement to support the new capability. Another potential prospect is the ability to incorporate existing technologies that enhance the L-API framework.

Cuda and OpenCL technology allows programmers to efficiently distribute workloads across multiple devices. However, they only provide mechanisms for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processor units (GPUs), DSPs and other processors, using traditional languages such as C, C++ and Fortran. For data dependence checking and recovering, additional mechanisms are required to deal with such hazards.

As proved in this study, the L-API is able to successfully recover from violations and produce the correct results at the end of execution. Coupling these sets of technologies (L-API with Cuda and/or OpenCL) can further enhance L-API to support multiple platforms using GPUs.

References

[Acosta and Liu, 2012] Acosta, E. and Liu, A. (2012). A pipeline virtual environment architecture for multicore processor systems. *The Visual Computer*, 28(11):1099–1114.

[Aho et al., 1986] Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques and Tools*. Addison Wesley.

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

[Akkary and Driscoll, 1998] Akkary, H. and Driscoll, M. A. (1998). A dynamic multithreading processor. In *MICRO*, pages 226–236.

[Akkary et al., 2008] Akkary, H., Jothi, K., Retnamma, R., Nekkalapu, S., Hall, D., and Shahidzadeh, S. (2008). On the potential of latency tolerant execution in speculative multithreading. In Watson, I. and El-Shishiny, H., editors, *IFMT*, ACM International Conference Proceeding Series, page 3. ACM.

[Amarasinghe et al., 1995] Amarasinghe, S. P., Anderson, J.-A. M., Lam, M. S., and Tseng, C.-W. (1995). An overview of the suif compiler for scalable parallel machines. In *PPSC*, pages 662–667.

[Aragon et al., 2006] Aragon, J., Gonzalez, J., and Gonzalez, A. (2006). Control speculation for energy-efficient next-generation superscalar processors. *IEEE Trans. Computers*, 55(3):281–291.

[Bai et al., 2008] Bai, S., Zhou, Q., Zhou, R., and Li, L. (2008). Barrier synchronization for cell multi-processor architecture. In *Ubi-Media Computing, 2008 First IEEE International Conference on*, pages 155–158.

[Balakrishnan et al., 2005] Balakrishnan, S., Rajwar, R., Upton, M., and Lai, K. (2005). The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 506–517, Washington, DC, USA. IEEE Computer Society.

[Barney, 2010] Barney, B. (2010). Posix threads programming.

[Bartlet, 2007] Bartlet, J. (2007). Changes in libspe: How libspe2 affects cell broadband engine programming.

[Battiwalla, 2009] Battiwalla, X. (2009). Xerxes future predictions - computing power.

[beach, 2008] beach, C. (2008). Code beach.

[Bhowmik and Franklin, 2004] Bhowmik, A. and Franklin, M. (2004). A general compiler framework for speculative multithreaded processors. *IEEE Trans. Parallel Distrib. Syst.*, 15(8):713–724.

[Blachford, 2005] Blachford, N. (2005). ps3coderz.com.

[Bloomer, 1992] Bloomer, J. (1992). *Power Programming with RPC*. Number 978-0937175774. O'Reilly Media, 1 edition.

[Butenhof, 1997] Butenhof, D. R. (1997). *Programming with POSIX Threads*. Number 978-0201633924. Addison-Wesley Professional, 1 edition.

[Chapman et al., 1997] Chapman, B., Jost, G., van der Pas, R., and Kuck, D. J. (1997). *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 1 edition.

[Chen et al., 2005] Chen, T., Ragavan, R., Dale, J., and Iwata, E. (2005). Cell broadband engine architecture and its first implementation.

[Cho et al., 2008] Cho, S., Li, T., and Mutlu, O. (2008). Guest editors' introduction: Interaction of many-core computer architecture and operating systems. *IEEE Micro*, 28(3):2–5.

[Cintra et al., 2000] Cintra, M., Martinez, J., and Torrellas, J. (2000). Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In Berenbaum, A. D. and Emer, J. S., editors, *ISCA*, pages 13–24. IEEE Computer Society.

[Cintra and Ferraris, 2003] Cintra, M. H. and Ferraris, D. R. L. (2003). Toward efficient and robust software speculative parallelization on multiprocessors. In Eigenmann, R. and Rinard, M. C., editors, *PPOPP*, pages 13–24. ACM.

[Claydon, 2007] Claydon, P. (2007). Multicore future is right now.

[Crowley and Turner, 2007] Crowley, P. and Turner, J. (2007). On the use of general-purpose multi-core processors in networking devices. *Washington University in St. Louis*.

[Da Silva and Steffan, 2006] Da Silva, J. and Steffan, J. G. (2006). A probabilistic pointer analysis for speculative optimizations. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 416–425, New York, NY, USA. ACM.

[David et al., 2007] David, F. M., Carlyle, J. C., and Campbell, R. H. (2007). Context switch overheads for linux on arm platforms. In *Experimental Computer Science*, page 3. ACM.

[Dou and Cintra, 2007] Dou, J. and Cintra, M. (2007). A compiler cost model for speculative parallelization. *TACO*, 4(2).

[Drepper, 2007] Drepper, U. (2007). What every programmer should know about memory.

[Du et al., 2004] Du, Z.-H., Lim, C.-C., Li, X.-F., Yang, C., Zhao, Q., and Ngai, T.-F. (2004). A cost-driven compilation framework for speculative parallelization of sequential programs. In Pugh, W. and Chambers, C., editors, *PLDI*, pages 71–81. ACM.

[Espasa and Valero, 1997] Espasa, R. and Valero, M. (1997). Exploiting instruction and data level parallelism in future high performance processors. *IEEE Micro*, 17:20–27.

[Flachs et al., 2007] Flachs, B., Asano, S., Dhong, S. H., Hofstee, H. P., Gervais, G., Kim, R., Le, T., Liu, P., Leenstra, J., Liberty, J. S., Michael, B., Oh, H.-J., Mueller, S. M., Takahashi, O., Hirairi, K., Kawasumi, A., Murakami, H., Noro, H., Onishi, S., Pille, J., Silberman, J., Yong, S., Hatakeyama, A., Watanabe, Y., Yano, N., Brokenshire, D. A., Peyravian, M., To, V., and Iwata, E. (2007). Microarchitecture and implementation of the synergistic processor in 65-nm and 90-nm SOI. *IBM Journal of Research and Development*, 51(5):529–544.

[Franklin, 1993] Franklin, M. (1993). The multiscalar architecture. Technical report.

[Franklin and Sohi, 1996] Franklin, M. and Sohi, G. S. (1996). Arb: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45:552–571.

[Frey, 2005] Frey, B. (2005). Powerpc architecture book, version 2.02.

[Fu et al., 1998] Fu, C. Y., Jennings, M. D., Larin, S. Y., and Conte, T. M. (1998). Software-only value speculation scheduling. Technical report.

[Fung and Steffan, 2006a] Fung, S. L. C. and Steffan, J. G. (2006a). Improving cache locality for thread-level speculation. In *IPDPS*. IEEE.

[Fung and Steffan, 2006b] Fung, S. L. C. and Steffan, J. G. (2006b). Improving cache locality for thread-level speculation. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 32–32, Washington, DC, USA. IEEE Computer Society.

[Furber, 2000] Furber, S. (2000). *ARM system-on-chip architecture*. Pearson Education Limited, 2 edition.

[Garzaran et al., 2005] Garzaran, M. J., Prvulovic, M., and Llaberia, J. M. (2005). Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *ACM Trans. Archit. Code Optim.*, 2(3):247–279.

[Garzaran et al., 2003] Garzaran, M. J., Prvulovic, M., Vinals, V., Llaberia, J. M., Rauchwerger, L., and Torrellas, J. (2003). Using software logging to support multi-version buffering in thread-level speculation. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, page 170, Washington, DC, USA. IEEE Computer Society.

[Gaster, 2010] Gaster, B. R. (2010). Introductory tutorial to opencl.

[Geer, 2005] Geer, D. (2005). Industry trends: Chip makers turn to multicore processors. *IEEE Computer*, 38(5):11–13.

[Gonzalez, 2010] Gonzalez, A. (2010). Speculative threading: Creating new methods of thread-level parallelization.

[Gopal et al., 1998] Gopal, S., Vijaykumar, T. N., Smith, J. E., and Sohi, G. S. (1998). Speculative versioning cache. In *HPCA*, pages 195–205. IEEE Computer Society.

[Gorder, 2007] Gorder, P. F. (2007). Multicore processor for science and engineering.

[Grama et al., 2003] Grama, A., Gupta, A., Karypis, G., and Kumar, V. (2003). *Introduction to Parallel Computing*, volume 2. Addison Wesley, 2nd edition.

[Grune et al., 2000] Grune, D., Bal, H. E., Jacobs, C. J., and Langendoen, K. G. (2000). *Modern Compiler Design*. John Wiley and Sons Ltd.

[Gschwind, 2006] Gschwind, M. (2006). Chip multiprocessing and the cell broadband engine. In *Conf. Computing Frontiers*, pages 1–8. ACM.

[Gschwind, 2007] Gschwind, M. (2007). The cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, 35(3):233–262.

[Gschwind et al., 2007] Gschwind, M., Erb, D., Manning, S., and Nutter, M. (2007). An open source environment for cell broadband engine system software. *Computer*, 40(6):37–47.

[Gschwind et al., 2006] Gschwind, M., Hofstee, H. P., Flachs, B. K., Hopkins, M., Watanabe, Y., and Yamazaki, T. (2006). Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(2):10–24.

[Guccione, 2008] Guccione, S. A. (2008). Hardware/software trade-offs in multicore architectures.

[Gupta and Nim, 1998] Gupta, M. and Nim, R. (1998). Techniques for speculative run-time parallelization of loops. In *SC*, page 12. IEEE.

[Gurtovoy and Abrahams, 2008] Gurtovoy, A. and Abrahams, D. (2008). The boost c++ metaprogramming library.

[Haendel, 2005] Haendel, L. (2005). The function pointer tutorials.

[Halfhill, 2007] Halfhill, T. R. (2007). The future of multicore processors.

[Hammond et al., 2000] Hammond, L., Hubbert, B. A., Siu, M., Prabhu, M. K., Chen, M. K., and Olukotun, K. (2000). The stanford hydra cmp. *IEEE Micro*, 20(2):71–84.

[Hammond et al., 1998] Hammond, L., Willey, M., and Olukotun, K. (1998). Data speculation support for a chip multiprocessor. In Bhandarkar, D. and Agarwal, A., editors, *ASPLOS*, pages 58–69. ACM Press.

[Hennessy and Patterson, 2007] Hennessy, J. L. and Patterson, D. A. (2007). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann.

[Hifi et al., 2008] Hifi, M., Mhalla, H., and Sadfi, S. (2008). An adaptive algorithm for the knapsack problem: perturbation of the profit or weight of an arbitrary item. *European Journal of Industrial Engineering*, 2(2):134–152.

[Hill and Marty, 2008] Hill, M. D. and Marty, M. R. (2008). Amdahl's law in the multicore era. *Computer*, 41(7):33–38.

[IBM, 1994] IBM (1994). *PowerPC Architecture: A Specification for the New Family of RISC Processors*. Morgan Kaufmann Publishers.

[IBM, 2005] IBM (2005). Ibm full-system simulator for the cell broadband engine processor.

[IBM, 2007a] IBM (2007a). Cell broadband engine architecture.

[IBM, 2007b] IBM (2007b). *SPE Runtime Management Library Version 2.2*. IBM.

[Iwama et al., 2001] Iwama, C., Barli, N. D., Sakai, S., and Tanaka, H. (2001). Improving conditional branch prediction on speculative multithreading architectures. In Sakellariou, R., Keane, J. A., Gurd, J. R., and Freeman, L., editors, *Euro-Par*, volume 2150 of *Lecture Notes in Computer Science*, pages 413–417. Springer.

[Johns and Brokenshire, 2007] Johns, C. R. and Brokenshire, D. A. (2007). Introduction to the cell broadband engine architecture. *IBM Journal of Research and Development*, 51(5):503–519.

[Johnson et al., 2007] Johnson, T. A., Eigenmann, R., and Vijaykumar, T. N. (2007). Speculative thread decomposition through empirical optimization. In Yelick, K. A. and Mellor-Crummey, J. M., editors, *PPOPP*, pages 205–214. ACM.

[Kahle et al., 2005] Kahle, J. A., Day, M. N., Hofstee, H. P., Johns, C. R., Maeurer, T. R., and Shippy, D. (2005). Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5).

[Karniadakis and Robert, 2003] Karniadakis, G. E. and Robert, M, K. I. (2003). *Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and their Implementation*. Cambridge University Press,.

[Kazi and Lilja, 2001] Kazi, I. and Lilja, D. (2001). Coarse-grained thread pipelining: a speculative parallel execution model for shared-memory multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 12(9):952–966.

[Kejariwal et al., 2006] Kejariwal, A., Tian, X., 0015, W. L., Girkar, M., Kozhukhov, S., Saito, H., Banerjee, U., Nicolau, A., Veidenbaum, A. V., and Polychronopoulos, C. D. (2006). On the performance potential of different types of speculative thread-level parallelism: The dl version of

this paper includes corrections that were not made available in the printed proceedings. In Egan, G. K. and Muraoka, Y., editors, *ICS*, page 24. ACM.

[Keller and Varbanescu, 2010] Keller, J. and Varbanescu, A. L. (2010). Performance impact of task mapping on the cell be multicore processor. In Varbanescu, A. L., Molnos, A. M., and van Nieuwpoort, R., editors, *ISCA Workshops*, volume 6161 of *Lecture Notes in Computer Science*, pages 13–23. Springer.

[Kennedy and Allen, 2001] Kennedy, K. and Allen, J. R. (2001). *Optimising Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Inc., 1 edition.

[Kistler et al., 2006] Kistler, M., Perrone, M., and Petrini, F. (2006). Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23.

[Knight, 1986] Knight, T. F. (1986). An architecture for mostly functional languages. In *LISP and Functional Programming*, pages 105–112.

[Koivisto, 2005] Koivisto, D. (2005). What amdahl's law can tell us about multicores and multiprocessing.

[Krishnan and Torrellas, 1998] Krishnan, V. and Torrellas, J. (1998). Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In *Proc. of 1998 Int. Conf. on Supercomputing*, pages 85–92.

[Krishnan and Torrellas, 1999a] Krishnan, V. and Torrellas, J. (1999a). A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. Computers*, 48(9):866–880.

[Krishnan and Torrellas, 1999b] Krishnan, V. and Torrellas, J. (1999b). The need for fast communication in hardware-based speculative chip multiprocessors. In *IEEE PACT*, pages 24–33. IEEE Computer Society.

[Kumar and Huggahalli, 2007] Kumar, A. and Huggahalli, R. (2007). Impact of cache coherence protocols on the processing of network traffic. In *MICRO*, pages 161–171. IEEE Computer Society.

[Laudon and Spracklen, 2007] Laudon, J. and Spracklen, L. (2007). The coming wave of multithreaded chip multiprocessors. *International Journal of Parallel Programming*, 35(3):299–330.

[Li et al., 2005] Li, X.-F., Yang, C., Du, Z.-H., and Ngai, T.-F. (2005). Exploiting thread-level speculative parallelism with software value prediction. In Srikanthan, T., Xue, J., and Chang, C.-H., editors, *Asia-Pacific Computer Systems Architecture Conference*, volume 3740 of *Lecture Notes in Computer Science*, pages 367–388. Springer.

[Lin et al., 2004] Lin, J., Hsu, W.-C., Yew, P.-C., Ju, Dz-Ching, R., and Ngai, T.-F. (2004). A compiler framework for recovery code generation in general speculative optimizations. In *IEEE PACT*, pages 17–28. IEEE Computer Society.

[Liu et al., 2006] Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., and Torrellas, J. (2006). Posh: a tls compiler that exploits program structure. In Torrellas, J. and Chatterjee, S., editors, *PPOPP*, pages 158–167. ACM.

[Lo et al., 1997] Lo, J. L., Eggers, S. J., Emer, J. S., Levy, H. M., Stamm, R. L., and Tullsen, D. M. (1997). Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Trans. Comput. Syst.*, 15(3):322–354.

[Luo et al., 2009] Luo, Y., Packirisamy, V., Hsu, W.-C., Zhai, A., Mungre, N., and Tarkas, A. (2009). Dynamic performance tuning for speculative threads. In *ISCA*, pages 462–473. ACM.

[Madriles et al., 2008] Madriles, C., García-Quiñones, C., Sánchez, J., Marcuello, P., González, A., Tullsen, D. M., Wang, H., and Shen, J. P. (2008). Mitosis: A speculative multithreaded processor based on precomputation slices. *IEEE Trans. Parallel Distrib. Syst.*, 19(7):914–925.

[Mahapatra and Venkatrao, 1999] Mahapatra, N. R. and Venkatrao, B. (1999). The processor-memory bottleneck: problems and solutions. *Crossroads*, 5(3es).

[Mahlke et al., 1992] Mahlke, S. A., Chen, W. Y., mei W. Hwu, W., Rau, B. R., and Schlansker, M. S. (1992). Sentinel scheduling for VLIW and superscalar processors. In Flahive, B. and Wexelblat, R. L., editors, *ASPLOS*, pages 238–247. ACM Press.

[Marcuello and Gonzalez, 2000] Marcuello, P. and Gonzalez, A. (2000). A quantitative assessment of thread-level speculation techniques. In *IPDPS*, pages 595–. IEEE Computer Society.

[Marcuello and Gonzalez, 2002] Marcuello, P. and Gonzalez, A. (2002). Thread-spawning schemes for speculative multithreading. In *HPCA*, pages 55–64. IEEE Computer Society.

[Marcuello et al., 1998] Marcuello, P., Gonzalez, A., and Tubella, J. (1998). Speculative multithreaded processors. In *Proceedings of the 12th international conference on Supercomputing*, ICS '98, pages 77–84, New York, NY, USA. ACM.

[Martinez and Torrellas, 2003] Martinez, J. F. and Torrellas, J. (2003). Speculative synchronization: Programmability and performance for parallel codes. *IEEE Micro*, 23(6):126–134.

[Miura et al., 2003] Miura, H., Hung, L. D., Iwama, C., Tashiro, D., Barli, N. D., Sakai, S., and Tanaka, H. (2003). Compiler-assisted thread level

control speculation. In *Euro-Par*, volume 2790 of *Lecture Notes in Computer Science*, pages 603–608. Springer.

[Mols, 2009] Mols, E. (2009). A cellfs implementation for the x86 architecture. *Enschede 10th Twente Student Conference on IT*.

[Moncrieff et al., 1996] Moncrieff, D., Overill, R. E., and Wilson, S. (1996). Heterogeneous computing machines and amdahl's law. *Parallel Comput.*, 22(3):407–413.

[Oancea and Mycroft, 2008] Oancea, C. E. and Mycroft, A. (2008). Software thread-level speculation: an optimistic library implementation. In *Proceedings of the 1st international workshop on Multicore software engineering*, IWMSE '08, pages 23–32, New York, NY, USA. ACM.

[Olukotun et al., 1996] Olukotun, K., Nayfeh, B. A., Hammond, L., Wilson, K. G., and Chang, K. (1996). The case for a single-chip multiprocessor. In Dally, B. and Eggers, S. J., editors, *ASPLOS*, pages 2–11. ACM Press.

[Oplinger and Lam, 2002] Oplinger, J. T. and Lam, M. S. (2002). Enhancing software reliability with speculative threads. In Gharachorloo, K., editor, *ASPLOS*, pages 184–196. ACM Press.

[Oracle, 2005] Oracle (2005). Changing the economics and ecology of the data center with innovative sparc technology.

[Packirisam et al., 2008] Packirisam, V., Lu, Y., lung Hun, W., Antonia Zha, P.-c. Y., and fook Ngai, T. (2008). Efficiency of thread-level speculation in smt and cmp architectures - performance, power and thermal perspective.

[Packirisamy et al., 2006] Packirisamy, V., Wang, S., Zhai, A., Hsu, W.-C., and Yew, P.-C. (2006). Supporting speculative multithreading on simultaneous multithreaded processors. In Robert, Y., Parashar, M., Badrinath, R., and Prasanna, V. K., editors, *HiPC*, volume 4297 of *Lecture Notes in Computer Science*, pages 148–158. Springer.

[Park et al., 2003] Park, I., Ooi, C. L., and Vijaykumar, T. N. (2003). Reducing design complexity of the load/store queue. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 411–, Washington, DC, USA. IEEE Computer Society.

[Pas, 2002] Pas, R. v. d. (2002). Memory hierarchy in cache-based systems.

[Paul and Meyer, 2007] Paul, J. M. and Meyer, B. H. (2007). Amdahl's law revisited for single chip systems. *International Journal of Parallel Programming*, 35(2):101–123.

[Pedram, 1996] Pedram, M. (1996). Power minimization in IC design: principles and applications. *ACM Trans. Design Autom. Electr. Syst.*, 1(1):3–56.

[Pericas, 2003] Pericas, M. (2003). Simultaneous multithreading: Present developments and future directions.

[Pinkston and Prasanna, 2003] Pinkston, T. M. and Prasanna, V. K., editors (2003). *High Performance Computing - HiPC 2003, 10th International Conference, Hyderabad, India, December 17-20, 2003, Proceedings*, volume 2913 of *Lecture Notes in Computer Science*. Springer.

[Pozo and Miller, 2004] Pozo, R. and Miller, B. (2004). Scimark 2.0.

[Prabhu, 2005] Prabhu, M. K. (2005). *Parallel Programming Using Thread-Level Speculation*. PhD thesis.

[Prabhu and Olukotun, 2003] Prabhu, M. K. and Olukotun, K. (2003). Using thread-level speculation to simplify manual parallelization. In Eigenmann, R. and Rinard, M. C., editors, *PPOPP*, pages 1–12. ACM.

[Prvulovic et al., 2001] Prvulovic, M., Garzarán, M. J., Rauchwerger, L., and Torrellas, J. (2001). Removing architectural bottlenecks to the scalability of speculative parallelization. In Stenstrom, P., editor, *ISCA*, pages 204–215. ACM.

[Qi and Zhu, 2011] Qi, X. and Zhu, D. (2011). Energy efficient block-partitioned multicore processors for parallel applications. *J. Comput. Sci. Technol.*, 26(3):418–433.

[Rabbah et al., 2004] Rabbah, R. M., Sandanagobalane, H., Ekpanyapong, M., and Wong, W.-F. (2004). Compiler orchestrated prefetching via speculation and predication. In Mukherjee, S. and McKinley, K. S., editors, *ASPLOS*, pages 189–198. ACM.

[Radulovic and Tomasevic, 2007] Radulovic, M. and Tomasevic, M. (2007). Towards an improved integrated coherence and speculation protocol. In *The International Conference*, pages 405–412.

[Raman et al., 2008] Raman, E., Vachharajani, N., Rangan, R., and August, D. I. (2008). Spice: speculative parallel iteration chunk execution. In Soffa, M. L. and Duesterwald, E., editors, *CGO*, pages 175–184. ACM.

[Rauchwerger and Padua, 1995] Rauchwerger, L. and Padua, D. (1995). The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 218–232, New York, NY, USA. ACM.

[Renau et al., 2005] Renau, J., Strauss, K., Ceze, L., Liu, W., Sarangi, S. R., Tuck, J., and Torrellas, J. (2005). Thread-level speculation on a cmp can be energy efficient. In Arvind and Rudolph, L., editors, *ICS*, pages 219–228. ACM.

[Ro and Gaudiot, 2005] Ro, W. W. and Gaudiot, J.-L. (2005). A low-complexity issue queue design with speculative pre-execution. In Bader, D. A., Parashar, M., Varadarajan, S., and Prasanna, V. K., editors, *HiPC*, volume 3769 of *Lecture Notes in Computer Science*, pages 353–362. Springer.

[Roberts and Ahkter, 2006] Roberts, S. and Ahkter, J. (2006). *Multi-Core Programming : Increasing Performance through Software Multi-threading*. Number 978-0976483243. Intel Corporation, 1 edition.

[Rul et al., 2007] Rul, S., Vandierendonck, H., and Bosschere, K. D. (2007). Function level parallelism driven by data dependencies. *SIGARCH Computer Architecture News*, 35(1):55–62.

[Rundberg and Stenstrom, 2001] Rundberg, P. and Stenstrom, P. (2001). An all-software thread-level data dependence speculation system for multiprocessors. *J. Instruction-Level Parallelism*, 3.

[Scarpino, 2008] Scarpino, M. (2008). *Programming the Cell Processor: For Games, Graphics, and Computation*. Prentice Hall, 1 edition.

[Shen and Lipasti, 2006] Shen, J. P. and Lipasti, M. H. (2006). *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Higher Eduction.

[Shi, 1996] Shi, Y. (1996). Reevaluating amdahl's law and gustafson's law.

[Shimpi, 2005] Shimpi, A. L. (2005). Understanding the cell microprocessor.

[Skovhede et al., 2010] Skovhede, K., Larsen, M. N., and Vinter, B. (2010). Extending distributed shared memory for the cell broadband engine to a channel model. In *PARA (1)*, volume 7133 of *Lecture Notes in Computer Science*, pages 108–118. Springer.

[Sohi and Roth, 2001] Sohi, G. S. and Roth, A. (2001). Speculative multithreaded processors. (4):66–73.

[Srinivasan et al., 2005] Srinivasan, V., Santhanam, A. K., and Srinivasan., M. (2005). Cell broadband engine processor dma engine, part 1: The little engines that move data.

[Stallings, 2009] Stallings, W. (2009). *Computer Organization and Architecture: International Version Designing for Performance*. Pearson, 8th edition.

[Steffan, 2003] Steffan, J. G. (2003). *Hardware Support for Thread-Level Speculation*. PhD thesis, School of Computer Science, Pittsburgh: Carnegie Mellon University.

[Steffan et al., 2000] Steffan, J. G., Colohan, C. B., Zhai, A., and Mowry, T. C. (2000). A scalable approach to thread-level speculation. In Berenbaum, A. D. and Emer, J. S., editors, *ISCA*, pages 1–12. IEEE Computer Society.

[Steffan et al., 2002] Steffan, J. G., Colohan, C. B., Zhai, A., and Mowry, T. C. (2002). Improving value communication for thread-level speculation. In *HPCA*, pages 65–75. IEEE Computer Society.

[Steffan et al., 2005] Steffan, J. G., Colohan, C. B., Zhai, A., and Mowry, T. C. (2005). The stampede approach to thread-level speculation. *ACM Trans. Comput. Syst.*, 23(3):253–300.

[Takahashi et al., 2005] Takahashi, O., Cottier, S. R., Dhong, S. H., Flachs, B. K., and Silberman, J. (2005). Power-conscious design of the cell processor's synergistic processor element. *IEEE Micro*, 25(5):10–18.

[Tam and Tam, 2003] Tam, A. and Tam, D. (2003). Improving data locality on thread-level speculation.

[Tanenbaum, 2005] Tanenbaum, A. S. (2005). *Structured Computer Organisation*. Prentice Hall.

[Tian et al., 2009] Tian, C., Feng, M., Nagarajan, V., and Gupta, R. (2009). Speculative parallelization of sequential loops on multicores. *International Journal of Parallel Programming*, 37(5):508–535.

[Torre, 2009] Torre, M. A. (2009). Word in the space.

[Trinder et al., 1993] Trinder, P. W., Hammond, K., Loidl, H.-W., and L., S. (1993). Algorithm + strategy = parallelism. *Cambridge University Press*.

[Tsai et al., 1999] Tsai, J.-Y., Huang, J., Amlo, C., Lilja, D. J., and Yew, P.-C. (1999). The superthreaded processor architecture. *IEEE Trans. Computers*, 48(9):881–902.

[Tsai and Yew, 1996] Tsai, J.-Y. and Yew, P.-C. (1996). The superthreaded architecture: thread pipelining with run-time data dependence checking and control speculation. In *IEEE PACT*, pages 35–46. IEEE Computer Society.

[Tuah et al., 2002] Tuah, N. J., Kumar, M., Venkatesh, S., and Das, S. K. (2002). Performance optimization problem in speculative prefetching. *IEEE Trans. Parallel Distrib. Syst.*, 13(5):471–484.

[Tubella and Gonzalez, 1998] Tubella, J. and Gonzalez, A. (1998). Control speculation in multithreaded processors through dynamic loop detection. In *HPCA*, pages 14–23. IEEE Computer Society.

[Wang et al., 2008] Wang, C., Wu, Y., Borin, E., Hu, S., Liu, W., fook Ngai, T., and Fang, J. (2008). New slicing algorithms for parallelizing single-threaded programs. pages 20–27.

[Wang et al., 2009] Wang, C., Wu, Y., Borin, E., Hu, S., Liu, W., Sager, D., and fook Ngai, Tin, F. J. (2009). Dynamic parallelization of single-threaded binary programs using speculative slicing. In *ICS*, pages 158–168. ACM.

[Wang et al., 2005] Wang, S., Dai, X., Yellajyosula, K., Zhai, A., and Yew, P.-C. (2005). Loop selection for thread-level speculation. In *LCPC*, volume 4339 of *Lecture Notes in Computer Science*, pages 289–303. Springer.

[Warg and Stenstrom, 2006] Warg, F. and Stenstrom, P. (2006). Dual-thread speculation: Two threads in the machine are worth eight in the bush. In *SBAC-PAD*, pages 91–98. IEEE Computer Society.

[Warg and Stenstrom, 2008] Warg, F. and Stenstrom, P. (2008). Dual-thread speculation: A simple approach to uncover thread-level parallelism on a simultaneous multithreaded processor. *International Journal of Parallel Programming*, 36(2):166–183.

[Wenjie et al., 2012] Wenjie, T., Yiping, Y., and Feng, Z. (2012). A hierarchical parallel discrete event simulation kernel for multicore platform. *Cluster Computing*, pages 1–9.

[Williams et al., 2006] Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., and Yelick, K. A. (2006). The potential of the cell processor for scientific computing. In *Conf. Computing Frontiers*, pages 9–20. ACM.

[Wilson et al., 1994] Wilson, R. P., French, R. S., Wilson, C. S., Amarasinghe, S. P., Anderson, J. M., Tjiang, S. W. K., Liao, S.-W., Tseng, C.-W., Hall, M. W., Lam, M. S., and Hennessy, J. L. (1994). Suif: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29:31–37.

[Wu et al., 2004] Wu, C., Lian, R., Zhang, J., Ju, R., Chan, S., Liu, L., Feng, X., and Zhang, Z. (2004). An overview of the open research compiler. In Eigenmann, R., Li, Z., and Midkiff, S. P., editors, *LCPC*, volume 3602 of *Lecture Notes in Computer Science*, pages 17–31. Springer.

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

[Xekalakis et al., 2009] Xekalakis, P., Ioannou, N., and Cintra, M. (2009). Combining thread level speculation helper threads and runahead execu-

tion. In Gschwind, M., Nicolau, A., Salapura, V., and Moreira, J. E., editors, *ICS*, pages 410–420. ACM.

[Yanagawa et al., 2003] Yanagawa, Y., Hung, L. D., Iwama, C., Barli, N. D., Sakai, S., and Tanaka, H. (2003). Complexity analysis of a cache controller for speculative multithreading chip multiprocessors. In [Pinkston and Prasanna, 2003], pages 393–404.

[Yang et al., 2004] Yang, H., Malik, R., Narasimha, S., Li, Y., Divakaruni, R., Agnello, P., Allen, S., Antreasyan, A., Arnold, J., Bandy, K., Belyansky, M., Bonnoit, A., Bronner, G., Chan, V., Chen, X., Chen, Z., Chidambarrao, D., Chou, A., Clark, W., Crowder, S., Engel, B., Harifuchi, H., Huang, S., Jagannathan, R., Jamin, F., Kohyama, Y., Kuroda, H., Lai, C., Lee, H., Lee, W.-H., Lim, E., Lai, W., Mallikarjunan, A., Matsumoto, K., Mcknight, A., Nayak, J., Ng, H., Panda, S., Rengarajan, R., Steigerwalt, M., Subbanna, S., Subramanian, K., Sudijono, J., Sudo, G., Sun, S.-P., Tessier, B., Toyoshima, Y., Tran, P., Wise, R., Wong, R., Yang, I., Wann, C., Su, L., Horstmann, M., Feudel, T., Wei, A., Frohberg, K., Burbach, G., Gerhardt, M., Lenski, M., Stephan, R., Wieczorek, K., Schaller, M., Salz, H., Hohage, J., Ruelke, H., Klais, J., Huebler, P., Luning, S., van Bentum, R., Grasshoff, G., Schwan, C., Ehrichs, E., Goad, S., Buller, J., Krishnan, S., Greenlaw, D., Raab, M., and Kepler, N. (2004). Dual stress liner for high performance sub-45nm gate length SOI CMOS manufacturing. In *Electron Devices Meeting, 2004. IEDM Technical Digest. IEEE International*, pages 1075–1077.

[Zaharieva-Stoyanova and Jantschii, 2003] Zaharieva-Stoyanova, E. and Jantschii, L. (2003). Detection of software data dependency in superscalar computer architecture execution. *Proceedings of the 4th international conference conference on Computer systems and technologies: e-Learning*, pages 107–112.

[Zhai et al., 2002] Zhai, A., Colohan, C. B., Steffan, J. G., and Mowry, T. C. (2002). Compiler optimization of scalar value communication between speculative threads. In Gharachorloo, K., editor, *ASPLOS*, pages 171–183. ACM Press.

[Zhai et al., 2004] Zhai, A., Colohan, C. B., Steffan, J. G., and Mowry, T. C. (2004). Compiler optimization of memory-resident value communication between speculative threads. In *CGO*, pages 39–52. IEEE Computer Society.

[Zheng et al., 2011] Zheng, L., Dong, M., Ota, K., Jin, H., Guo, S., and Ma, J. (2011). Energy efficiency of a multi-core processor by tag reduction. *J. Comput. Sci. Technol.*, 26(3):491–503.

[Zilles and Sohi, 2002] Zilles, C. and Sohi, G. (2002). Master/slave speculative parallelization. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 35, pages 85–96, Los Alamitos, CA, USA. IEEE Computer Society Press.

Appendices

Appendix A

Emphasis on common function calls with description

Table A.1: Violation, recovery, load and store function prototypes.

Prototype & Description
`1 void *ptf_SPEDataOutputViolationAnalyser(void *arg)` Analyse each load or store requests placed are rechecked for violation. If no violation is found then service request, otherwise create a recovery thread for system recovery.
`1 void *ptf_SPERecovery(void *unused)` The recovery begins in this thread function. Only one instance of this thread function can exist in the runtime environment. Typical responsibility of the function is to stop all SPEs, recalculate region parameters and restart the system from the new recovered state.
`1 void *ptf_SPEMonitorMailbox(void *arg)` Each SPE will place a value of the iteration it has successfully processed. The PPE L-API kernel will read the value from the SPE's outgoing mailbox and update the iteration completion log structure.
`1 void *ptf_SPERequestResolverScalar(void *arg)` Each request placed in the system is channelled to this function. Here, the request is analysed and serviced only if no violations have been detected. The Output Violation array is also updated for each iteration that has been processed.
`1 void *ptf_SPERequestResolverScalar(void *arg)` Determines where the location of data loads or store.

Table A.1 shows additional information of L-API functions from the PPE kernel. These low-level functions provide auxiliary logic for the L-API, by abstracting low-level calls (embedded into auxiliary functions) assists L-API to harness the underlying hardware functionality of the Cell microprocessor.

Table A.2: Request analyser callback function state Labels.

Tag	Description
RA	Request analysis
RG	Register
DR	Deregister
CB	Copy back
M or MAESC	Monitor array element state container
DOVA-C	Data output violation analysis - creation
DOVA-K	Data output violation analysis - kill
CBFR	Callback function return

Table A.2 shows the different state labels/stages used by the request analyser function. The reader is referred to Chapter 5 for details on request recovery.

Appendix B

L-API PPE Code

B.1 common.hpp

```
1 #ifndef _COMMON_HPP_
2 #define _COMMON_HPP_
3
4 #ifndef NULL
5 #define NULL 0
6 #endif
7
8 #define THREAD_USLEEP_DEFAULT      usleep(10000);
9
10 #define CELL_MEMALIGN_16          4
11 #define CELL_MEMALIGN_128         7
12
13 #define TRUE                  1
14 #define FALSE                 0
15
16 #define STORE                 200
17 #define LOAD                  201
18
19 #define STORE_AUX               202
20 #define LOAD_AUX                203
```

```
21
22 #define LOOP_RESTART            204
23
24 #define SPE_STOP                300
25 #define SPE_START               301
26 #define SPE_HALT                302
27 #define SPE_SHUTDOWN            303
28 #define SPE_CONTINUE            304
29 #define SPE_RESTART             305
30 #define SPE_STANDBY             306
31 #define SPE_DEPENDENCY          307
32 #define SPE_NO_DEPENDENCY       308
33
34 #define SPE_REQUEST_COMPLETE    309
35 #define SPE_COMMIT_RANGE        310
36 #define SPE_INIT_MODE           311
37
38 #define EMPTY                   312
39 #define MODIFIED                313
40 #define SPE_BUSY                314
41 #define NONE                    315
42
43 #define EA                      316
44 #define LS                      317
45
46 #define SPE_COMPLETE            318
47 #define SPE_WRITEBACK           319
48
49 #define SPE_WRITEBACK_COMPLETE  322
50
51 #define SPE_SHUTDOWN_REQUEST    323
52
53
54 #define REQUEST_RESOLVE         330
55 #define SPE_WORK_PROCESSED      331
56
57 #define BUFFER1                 335
```

```
58 #define BUFFER2                 336
59
60 #define THREAD_EXIT             340
61 #define THREAD_PAUSE            341
62 #define THREAD_RESTART          342
63
64 #define FUNCTION_FAILED         -1
65 #define FUNCTION_EXIT           0x003
66
67 #define REGISTER                400
68 #define DEREGISTER              401
69 #define COPY_BACK               402
70 #define INNER                   403
71 #define OUTER                   404
72
73 #define REGION_PROCESSING       412
74 #define REGION_SYNC             413
75 #define REGION_COMPLETED        414
76 #define REGION_FAILED           415
77
78 #define INPUT                   416
79 #define OUTPUT                  417
80
81 #define INTERNAL_STORE          418
82 #define EXTERNAL_STORE          419
83 #define NO_STORE                420
84
85 #define DOUBLE_ARRAY            0x001
86 #define SINGLE_ARRAY            0x002
87
88 #define START                   0x003
89 #define STOP                    0x004
90
91 #define SPU_ELEMENT             0x005
92 #define PPU_ELEMENT             0x006
93
94 #define SIGNAL1_CL              0x009
```

```
95 #define SIGNAL2_CL              0x010
96 #define SIGNAL1_QC              0x011
97 #define SIGNAL2_QC              0x012
98
99 #define LOAD_E                  0x007
100 #define STORE_E                 0x008
101
102 #define MAILBOX_INTERRUPT       0x013
103 #define PROGRAM_COMPLETE_EXECUTION      0x014
104 #define CALLBACK                0x015
105
106 #define THREAD_DOVA             0x016
107 #define THREAD_R                0x017
108 #define THREAD_ASC              0x018
109 #define THREAD_RRS              0x019
110 #define THREAD_MM               0x026
111
112 #define CALLBACK_SR             0x020
113 #define CALLBACK_R              0x021
114 #define CALLBACK_RC             0x022
115 #define CALLBACK_RC1            0x023
116 #define CALLBACK_RCS            0x024
117 #define CALLBACK_RL             0x025
118 #define CALLBACK_RR             0x027
119
120 typedef struct measure_tag {
121   unsigned int name;     //LOAD_E || STORE_E || SIGNAL1_CL ||
122                //SIGNAL2_CL || SIGNAL1_QC || SIGNAL2_QC ||
123                //CALLBACK_INTERRUPT || MAILBOX_INTERRUPT
124   unsigned int spe;
125   unsigned int element;    //SPU_ELEMENT or PPU_ELEMENT
126 //  unsigned long long value;    //decrement value (SPU) or
      decrement value (PPU)
127   unsigned long long begin;
128   unsigned long long end;
129 } measure_t;
130
```

```
131
132
133
134
135 typedef struct parameters_tag {
136   unsigned int spe;
137   unsigned int begin;
138   unsigned int end;
139 } parameters_t;
140
141 typedef struct {
142   int iteration_owner;
143   unsigned int request_type;
144   unsigned int spe;
145   int last_load;
146   int last_store;
147   int exclusive;
148   int exist;
149     unsigned short load;
150     unsigned short store;
151   unsigned int region;
152   unsigned int loop_level;
153   unsigned int level;
154   double data;
155   int aux;
156   unsigned index_a;
157   unsigned index_b;
158   unsigned io_array_type;
159
160   unsigned int outer_level;
161 } element_t;
162
163 typedef struct loop_tag {
164   unsigned int done;
165   unsigned int outer_level;
166   unsigned int level;
167   unsigned int spe;
```

```
168   unsigned int id;
169   unsigned int pos_start;
170   unsigned int pos_end;
171   unsigned int type;
172 } loop_t;
173
174 typedef struct spe_sig_ob_tag {
175   unsigned int id;
176   unsigned int spe;
177   unsigned int loop_level;
178 } spe_sig_ob_t;
179
180 typedef struct spe_area_tag {
181   unsigned long long reg1[8];
182   unsigned long long reg2[8];
183 } spe_area_t;
184
185 typedef struct temp_storage_tag {
186   int spe;
187   int size;
188   unsigned int addr;
189   unsigned long long next_spe_addr;
190   unsigned long long prev_spe_addr;
191 } tmp_storage_t;
192
193 typedef struct spe_sig_tag {
194   int spe_id_exclude;
195   int region;
196 } spe_sig_t;
197
198 typedef struct _control_block {
199   unsigned long long ea_addr[4];
200   int id_addr[4];
201 } control_block_t;
202
203 typedef struct _control_block_monitor {
204   unsigned long long ea_in;
```

```
205   unsigned long long ea_out;
206   int pad;
207   int unused[3];
208 } control_block_monitor_t;
209
210 typedef struct request_message_tag {
211   double data;
212   int data_type;
213   unsigned long long data_address;
214   int region_number;
215   int request_type;
216   unsigned int spe_sid;
217   unsigned int owner_itr;
218   unsigned int where_itr;
219   unsigned int loop_level;
220   unsigned int level;
221   unsigned loop_type;
222   int aux;
223   unsigned index_a;
224   unsigned index_b;
225   unsigned io_array_type;
226
227   unsigned int outer_level;
228 } request_message_t;
229
230 typedef struct context_ls_tag {
231   unsigned long long ea_addr[5];
232   unsigned long long pad[3];
233 } context_ls_t;
234
235 typedef struct transmit_report_tag {
236   unsigned int type;
237   unsigned int spe;
238   unsigned int result;
239   unsigned int handle;
240   unsigned int data_from_spe;
241   unsigned long addr;
```

```
242   unsigned int region;
243   unsigned int monitor;
244 } transmit_report_t;
245
246 typedef struct monitor_report_tag {
247   unsigned int spe;
248   unsigned int loop;
249   unsigned int region;
250   unsigned int monitor;
251 } monitor_report_t;
252
253 typedef struct recovery_info_tag {
254   element_t element_data_from;
255   element_t element_to_update;
256 } recovery_info_t;
257
258 typedef struct ipc_info_tag {
259   unsigned int spe;
260   unsigned long long ipc_addr;
261 } ipc_info_t;
262
263 typedef struct region_complete_tag {
264   unsigned short connected;
265   unsigned int spe;
266   unsigned int region;
267   unsigned int complete;
268   int commit_back;
269 } region_complete_t;
270
271 typedef struct spe_info_init_tag {
272   unsigned int spe;
273   unsigned long long ea_addr[5];
274   unsigned long long pad[3];
275 } spe_info_init_t;
276
277 typedef struct register_shared_variable_tag {
278   int data_type;
```

```
279   unsigned int data_region;
280   unsigned long long data_address;
281   unsigned int spe;
282 } register_shared_variable_t;
283
284 typedef struct spe_status_tag {
285   unsigned long long ipc;
286   unsigned long long r3;
287   unsigned long long r4;
288   unsigned long long r5;
289   int status;
290   unsigned int spe;
291   int counter;
292 } spe_status_t;
293
294 typedef struct data_io_tag {
295   unsigned int data_type;
296   unsigned int region;
297   void *input;
298   void *output;
299 } data_io_t;
300
301 typedef struct context_params_tag {
302   unsigned int       final;
303   unsigned int       completed;
304   int             assigned;
305   unsigned int       bench;
306   unsigned int       found;
307   unsigned long long     spe_r3;
308   unsigned long long     spe_r4;
309   unsigned long long     spe_r5;
310   unsigned int       type;
311   unsigned int       region_total_spe_count;
312   unsigned int       spe;
313   unsigned int       region;
314   unsigned long long     ea_in;
315   unsigned long long     ea_out;
```

```
316     unsigned int        size;
317     unsigned int        size2;
318     unsigned int        itr_begin;
319     unsigned int        itr_end;
320     unsigned int        size_begin;
321     unsigned int        size_end;
322     unsigned int        size_begin_2;
323     unsigned int        size_end_2;
324     unsigned int        itr_start;
325     unsigned int        itr_total;
326     unsigned int        parent_loop;
327     unsigned int        child_loop;
328     unsigned long long      ipc_internal_counter;
329     unsigned long long      ea_aux;
330     unsigned long long      ea_aux2;
331     unsigned long long      ea_aux3;
332     unsigned int        io_array_type;
333     unsigned long long      store_array;
334 } context_params_t;
335
336 typedef struct params_t {
337   unsigned int bench_name;
338   unsigned int bench_id;
339   unsigned int size;
340   unsigned int start_iterations;
341   unsigned int total_iterations;
342   unsigned int size_1_dataset;
343   unsigned int size_2_dataset;
344   unsigned long long input;
345   unsigned long long output;
346   unsigned long long aux;
347   unsigned long long aux2;
348     unsigned long long aux3;
349   unsigned io_array_type;
350 } param_io_t;
351
352
```

```
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369 ////////////////////////////////// Benchmark constants
        //////////////////////////////////
370
371 #define FFT    0x801
372 #define SOR    0x802
373 #define SPARSE  0x803
374 #define LU     0x804
375 #define MONTE 0x805
376 #define ARRAY 0x806
377
378 const double RESOLUTION_DEFAULT = 2.0;  /* secs (normally 2.0)
      */
379 const int RANDOM_SEED = 101010;
380
381 /* default: small (cache-contained) problem sizes */
382
383  const  int FFT_SIZE = 1024;  /* must be a power of two */
384  const  int SOR_SIZE = 100; /* NxN grid */
385  const  int SPARSE_SIZE_M = 1000;
386  const  int SPARSE_SIZE_nz = 5000;
387  const  int LU_SIZE = 100;
```

```
388
389 /* large (out-of-cache) problem sizes */
390
391  const  int LG_FFT_SIZE = 1048576;  /* must be a power of two */
392  const  int LG_SOR_SIZE = 1000;  /*  NxN grid  */
393  const  int LG_SPARSE_SIZE_M = 100000;
394  const  int LG_SPARSE_SIZE_nz = 1000000;
395  const  int LG_LU_SIZE = 1000;
396
397 /* tiny problem sizes (used to mainly to preload network classes
         */
398 /*                      for applet, so that network download
    times    */
399 /*                      are factored out of benchmark.)
                  */
400 /*
    */
401  const  int TINY_FFT_SIZE = 16;  /* must be a power of two */
402  const  int TINY_SOR_SIZE = 10; /* NxN grid */
403  const  int TINY_SPARSE_SIZE_M = 10;
404  const  int TINY_SPARSE_SIZE_N = 10;
405  const  int TINY_SPARSE_SIZE_nz = 50;
406  const  int TINY_LU_SIZE = 10;
407
408 static int fft_only_int_log2(int n) {
409     int k;
410     k = 1;
411
412     int log;
413     log = 0;
414
415     for(/*k=1*/; k < n; k *= 2, log++);
416       if(n != (1 << log)) {
417       printf("PPE == FFT: Data length is not a power of 2!: %d"
              ,n);
418            exit(1);
```

```
419         }
420
421     return log;
422 }
423
424
425
426 #ifndef NULL
427 #define NULL 0
428 #endif
429
430
431 double** new_Array2D_double(int M, int N)
432 {
433     int i=0;
434     int failed = 0;
435
436     double **A = (double**) malloc(sizeof(double*)*M);
437     if (A == NULL)
438         return NULL;
439
440     for (i=0; i<M; i++)
441     {
442         A[i] = (double*) malloc(N * sizeof(double));
443         if (A[i] == NULL)
444         {
445             failed = 1;
446             break;
447         }
448     }
449
450     /* if we didn't successfully allocate all rows of A      */
451     /* clean up any allocated memory (i.e. go back and free  */
452     /* previous rows) and return NULL                        */
453
454     if (failed)
455     {
```

```
456             i--;
457             for (; i<=0; i--)
458                 free(A[i]);
459             free(A);
460             return NULL;
461         }
462         else
463             return A;
464 }
465
466 void Array2D_double_delete(int M, int N, double **A)
467 {
468     int i;
469     if (A == NULL) return;
470
471     for (i=0; i<M; i++)
472         free(A[i]);
473
474     free(A);
475 }
476 ////////////////////////////////////// ^ Benchmark constants ^
        //////////////////////////////////////
477
478 #endif /*COMMON_INFO_H_*/
```

B.2 kernel_element_manager.hpp

```
1 #ifndef KERNEL_ELEMENT_MANAGER_HPP_
2 #define KERNEL_ELEMENT_MANAGER_HPP_
3
4 #include "kernel_system_headers.hpp"
5
6 /*pair<Iter, Iter> range = my_multimap.equal_range("Group1");
7 int total = accumulate(range.first, range.second, 0);*/
8
9 /*Map to store all element objects*/
```

```
10 std::multimap<int, element_t>         _ElementMap;
11 std::multimap<int, element_t>::iterator _ElementMapIterator;
12
13 pthread_mutex_t m_Exclusive = PTHREAD_MUTEX_INITIALIZER;
14 pthread_mutex_t m_LockMap = PTHREAD_MUTEX_INITIALIZER;
15 pthread_mutex_t m_Find = PTHREAD_MUTEX_INITIALIZER;
16
17 class ElementManager {
18 public:
19   void InsertElement(int key, element_t obj);
20   void ReinsertElement(int key, element_t obj);
21
22   void DeleteAllElements(void);
23   void Remove(element_t obj);
24   void RemoveByIndex(int key);
25   int CheckByIndex(int key);
26   element_t GetElementByIndex(int key);
27
28   int WaitAndEnableExclusive(element_t obj, unsigned int type);
29   int WaitAndDisableExclusive(element_t obj, unsigned int type);
30   unsigned int Find(element_t obj, unsigned int type);
31   int GetElementOwner(element_t obj, unsigned int type);
32   element_t GetElement(element_t obj, unsigned int type);
33
34   unsigned long int GetSize(void);
35
36   element_t GetElementByAuxIndex(int aux_key, int key);
37   int ExistElementByAuxIndex(int aux_key, int key);
38   int RemoveElementByLoopRange(unsigned int level);
39   int RemoveElementByLoopRangeIdentifer(unsigned int level, int
       key);
40 };
41
42 #endif /*KERNEL_ELEMENT_MANAGER_HPP_*/
43
44 unsigned long int ElementManager::GetSize(void) {
45   return _ElementMap.size();
```

```
46 }
47
48 void ElementManager::InsertElement(int key, element_t obj) {
49   pthread_mutex_lock(&m_LockMap);
50   _ElementMap.insert(std::pair<int, element_t>(key, obj));
51   pthread_mutex_unlock(&m_LockMap);
52
53
54 }
55
56 void ElementManager::DeleteAllElements(void) {
57   pthread_mutex_lock(&m_LockMap);
58   _ElementMap.erase(_ElementMap.begin(), _ElementMap.end());
59   pthread_mutex_unlock(&m_LockMap);
60 }
61
62 void ElementManager::ReinsertElement(int key, element_t obj) {
63   Remove(obj);
64   InsertElement(key, obj);
65 }
66
67 void ElementManager::RemoveByIndex(int key) {
68   element_t *_e_tmp;
69
70   for(_ElementMapIterator = _ElementMap.begin();
        _ElementMapIterator != _ElementMap.end(); ++
        _ElementMapIterator) {
71     _e_tmp = &_ElementMapIterator->second;
72
73     if(_e_tmp->iteration_owner==key) {
74       _ElementMap.erase(_ElementMapIterator);
75     }
76
77   }
78 }
79
80 element_t ElementManager::GetElementByIndex(int key) {
```

```
81    element_t *_e_tmp;
82
83    for(_ElementMapIterator = _ElementMap.begin();
          _ElementMapIterator != _ElementMap.end(); ++
          _ElementMapIterator) {
84      _e_tmp = &_ElementMapIterator->second;
85
86      if(_e_tmp->iteration_owner==key) {
87        break;
88      }
89    }
90
91    return *_e_tmp;
92 }
93
94
95 element_t ElementManager::GetElementByAuxIndex(int aux_key, int
       key) {
96    element_t *_e_tmp;
97
98    for(_ElementMapIterator = _ElementMap.begin();
          _ElementMapIterator != _ElementMap.end(); ++
          _ElementMapIterator) {
99      _e_tmp = &_ElementMapIterator->second;
100
101     if(_e_tmp->iteration_owner==key && _e_tmp->aux==aux_key) {
102       return *_e_tmp;
103       break;
104     }
105
106   }
107
108   return *_e_tmp;
109 }
110
111 int ElementManager::ExistElementByAuxIndex(int aux_key, int key)
        {
```

```
112   element_t *_e_tmp;
113
114   for(_ElementMapIterator = _ElementMap.begin();
          _ElementMapIterator != _ElementMap.end(); ++
          _ElementMapIterator) {
115     _e_tmp = &_ElementMapIterator->second;
116
117     if(_e_tmp->iteration_owner==key && _e_tmp->aux==aux_key) {
118       return TRUE;
119       break;
120     }
121
122   }
123
124   return FALSE;
125 }
126
127 int ElementManager::CheckByIndex(int key) {
128   element_t *_e_tmp;
129   int _i_Return = FALSE;
130
131   for(_ElementMapIterator = _ElementMap.begin();
          _ElementMapIterator != _ElementMap.end(); ++
          _ElementMapIterator) {
132     _e_tmp = &_ElementMapIterator->second;
133
134     if(_e_tmp->iteration_owner==key) {
135       _i_Return = TRUE;
136       break;
137     }
138
139   }
140
141   return _i_Return;
142 }
143
144
```

```
145 void ElementManager::Remove(element_t obj) {
146   pthread_mutex_lock(&m_LockMap);
147
148   if(Find(obj, obj.io_array_type)==TRUE) {
149     int i_ElementNumber;
150     i_ElementNumber = GetElementOwner(obj, obj.io_array_type);
151     _ElementMap.erase(_ElementMap.find(i_ElementNumber));
152   }
153
154   pthread_mutex_unlock(&m_LockMap);
155 }
156
157 unsigned int ElementManager::Find(element_t obj, unsigned int
      type) {
158   element_t *_e_tmp;
159   unsigned int _i_Return = FALSE;
160
161   std::multimap<int, element_t>::iterator
        _InternalElementMapIterator;
162
163   for(_ElementMapIterator = _ElementMap.begin();
        _ElementMapIterator != _ElementMap.end(); ++
        _ElementMapIterator) {
164     _e_tmp = &_ElementMapIterator->second;
165     _InternalElementMapIterator = _ElementMapIterator;
166
167     if(type==SINGLE_ARRAY) {
168       if(_e_tmp->index_a==obj.index_a) {
169         _i_Return = TRUE;
170         goto __FIND_EXIT;
171       }
172     }
173
174     if(type==DOUBLE_ARRAY) {
175       if((_e_tmp->index_a==obj.index_a) && (_e_tmp->index_b==obj
            .index_b)) {
176         _i_Return = TRUE;
```

```
177             goto __FIND_EXIT;
178           }
179         }
180       }
181
182     __FIND_EXIT:
183       return _i_Return;
184   }
185
186   int ElementManager::RemoveElementByLoopRange(unsigned int level)
          {
187     element_t *_e_ptr;
188
189     for(_ElementMapIterator = _ElementMap.begin();
            _ElementMapIterator != _ElementMap.end(); ++
            _ElementMapIterator) {
190       _e_ptr = &(_ElementMapIterator->second);
191
192       if(_e_ptr->outer_level==level) {
193         Remove(*_e_ptr);
194       }
195     }
196
197     return 0;
198   }
199
200   int ElementManager::RemoveElementByLoopRangeIdentifer(unsigned
        int level, int key) {
201     element_t *_e_ptr;
202
203     for(_ElementMapIterator = _ElementMap.begin();
            _ElementMapIterator != _ElementMap.end(); ++
            _ElementMapIterator) {
204       _e_ptr = &(_ElementMapIterator->second);
205
206       if(_e_ptr->outer_level==level && _e_ptr->iteration_owner==
              key) {
```

```
207         Remove(*_e_ptr);
208       }
209     }
210
211     return 0;
212 }
213
214 int ElementManager::GetElementOwner(element_t obj, unsigned int
      type) {
215     element_t *_e_tmp;
216     unsigned int _i_Return;
217
218     std::multimap<int, element_t>::iterator
          _InternalElementMapIterator;
219
220     for(_ElementMapIterator = _ElementMap.begin();
          _ElementMapIterator != _ElementMap.end(); ++
          _ElementMapIterator) {
221       _e_tmp = &(_ElementMapIterator->second);
222       _InternalElementMapIterator = _ElementMapIterator;
223
224       if(type==SINGLE_ARRAY) {
225         if(_e_tmp->index_a==obj.index_a) {
226           _i_Return = (*_e_tmp).iteration_owner;
227           goto __GETELEMENTOWNER_EXIT;
228         }
229       }
230
231       if(type==DOUBLE_ARRAY) {
232         if((_e_tmp->index_a==obj.index_a) && (_e_tmp->index_b==obj
              .index_b)) {
233           _i_Return = (*_e_tmp).iteration_owner;
234           goto __GETELEMENTOWNER_EXIT;
235         }
236       }
237     }
238
```

```
239   __GETELEMENTOWNER_EXIT:
240   return _i_Return;
241 }
242
243 element_t ElementManager::GetElement(element_t obj, unsigned int
        type) {
244   element_t *_e_tmp;
245
246   std::multimap<int, element_t>::iterator
          _InternalElementMapIterator;
247
248   for(_ElementMapIterator = _ElementMap.begin();
          _ElementMapIterator != _ElementMap.end(); ++
          _ElementMapIterator) {
249     _e_tmp = &_ElementMapIterator->second;
250     _InternalElementMapIterator = _ElementMapIterator;
251
252     if(type==SINGLE_ARRAY) {
253       if(_e_tmp->index_a==obj.index_a) {
254         goto __GETELEMENT_EXIT;
255       }
256     }
257
258     if(type==DOUBLE_ARRAY) {
259       if((_e_tmp->index_a==obj.index_a) && (_e_tmp->index_b==obj
              .index_b)) {
260         goto __GETELEMENT_EXIT;
261       }
262     }
263   }
264
265   __GETELEMENT_EXIT:
266   return *_e_tmp;
267 }
268
269 int ElementManager::WaitAndEnableExclusive(element_t obj,
        unsigned int type) {
```

```
270   element_t *_e_tmp;
271   std::multimap<int, element_t>::iterator
          _InternalElementMapIterator;
272
273   for(_InternalElementMapIterator = _ElementMap.begin();
          _InternalElementMapIterator != _ElementMap.end(); ++
          _InternalElementMapIterator) {
274     _e_tmp = &_InternalElementMapIterator->second;
275
276     if(type==SINGLE_ARRAY) {
277       if(_e_tmp->index_a==obj.index_a) {
278         goto __EXCLUSIVE_EXIT;
279       }
280     }
281
282     if(type==DOUBLE_ARRAY) {
283       if((_e_tmp->index_a==obj.index_a) && (_e_tmp->index_b==obj
              .index_b)) {
284         goto __EXCLUSIVE_EXIT;
285       }
286     }
287   }
288
289   __EXCLUSIVE_EXIT:
290   if(_e_tmp->exclusive == FALSE) {
291     do {
292       THREAD_USLEEP_DEFAULT
293       _e_tmp = &_InternalElementMapIterator->second;
294     } while(_e_tmp->exclusive == TRUE);
295
296     pthread_mutex_lock(&m_LockMap);
297     _e_tmp->exclusive = TRUE;
298     pthread_mutex_unlock(&m_LockMap);
299   }
300
301   return 0;
302 }
```

```
303
304
305 int ElementManager::WaitAndDisableExclusive(element_t obj,
      unsigned int type) {
306   element_t *_e_tmp;
307   std::multimap<int, element_t>::iterator
        _InternalElementMapIterator;
308
309   for(_ElementMapIterator = _ElementMap.begin();
        _ElementMapIterator != _ElementMap.end(); ++
        _ElementMapIterator) {
310     _e_tmp = &_ElementMapIterator->second;
311     _InternalElementMapIterator = _ElementMapIterator;
312
313     if(type==SINGLE_ARRAY) {
314       if(_e_tmp->index_a==obj.index_a) {
315         goto __DEXCLUSIVE_EXIT;
316       }
317     }
318
319     if(type==DOUBLE_ARRAY) {
320       if((_e_tmp->index_a==obj.index_a) && (_e_tmp->index_b==obj
            .index_b)) {
321         goto __DEXCLUSIVE_EXIT;
322       }
323     }
324   }
325
326   __DEXCLUSIVE_EXIT:
327   if(_e_tmp->exclusive != FALSE) {
328     do {
329       THREAD_USLEEP_DEFAULT
330       _e_tmp = &_ElementMapIterator->second;
331     } while(_e_tmp->exclusive != FALSE);
332
333     pthread_mutex_lock(&m_LockMap);
334     _e_tmp->exclusive = FALSE;
```

```
335       pthread_mutex_unlock(&m_LockMap);
336     }
337
338     if(_e_tmp->exclusive == TRUE) {
339       do {
340         THREAD_USLEEP_DEFAULT
341         _e_tmp = &_InternalElementMapIterator->second;
342       } while(_e_tmp->exclusive == FALSE);
343
344       pthread_mutex_lock(&m_LockMap);
345       _e_tmp->exclusive = FALSE;
346       pthread_mutex_unlock(&m_LockMap);
347     }
348
349     return 0;
350 }
```

B.3 kernel_measurement_reading.hpp

```
1 #ifndef KERNEL_MEASUREMENT_READING_HPP_
2 #define KERNEL_MEASUREMENT_READING_HPP_
3
4 #include "kernel_system_headers.hpp"
5
6 std::multimap<unsigned int, measure_t>
      __MeasurementReadingMap;
7 std::multimap<unsigned int, measure_t>::iterator
      __MeasurementReadingMapIterator;
8
9 class MeasurementReading {
10 public:
11   int Insert(measure_t mreading);
12   int CreateReport(unsigned int specifier);
13 };
14
15 #endif /*KERNEL_MEASUREMENT_READING_HPP_*/
```

```
16
17 int MeasurementReading::Insert(measure_t mreading) {
18   __MeasurementReadingMap.insert(std::pair<unsigned int,
       measure_t>(mreading.name, mreading));
19   return 0;
20 }
21
22 int MeasurementReading::CreateReport(unsigned int specifier) {
23   measure_t s_measure;
24
25 //  FILE *p_FilenameOutput;
26 //  char *p_Mode = "w+";
27 //  p_FilenameOutput=fopen("/home/harry/workspace/D_PPU/Results.
     txt", p_Mode);
28
29   std::pair<std::multimap<unsigned int, measure_t>::iterator,
       std::multimap<unsigned int, measure_t>::iterator>
       __MeasurementReadingMapIteratorRange;
30
31   __MeasurementReadingMapIteratorRange = __MeasurementReadingMap
       .equal_range(specifier);
32
33   printf("\n\nResults:\t");
34   switch(specifier) {
35     case SPU_ELEMENT:
36       break;
37
38     case PPU_ELEMENT:
39       break;
40
41     case LOAD_E:
42       printf("Load");
43       break;
44
45     case STORE_E:
46       printf("Store");
47       break;
```

```
48
49      case MAILBOX_INTERRUPT:
50        printf("Mailbox Interrupt");
51        break;
52
53      case THREAD_DOVA:
54        printf("ptf_SPEDataOutputViolationAnalyser");
55        break;
56
57      case THREAD_R:
58        printf("ptf_SPERecovery");
59        break;
60
61      case THREAD_ASC:
62        printf("ptf_SPEAllShutdownCheck");
63        break;
64
65      case THREAD_MM:
66        printf("ptf_SPEMonitorMailbox");
67        break;
68
69      case THREAD_RRS:
70        printf("ptf_SPERequestResolverScalar");
71        break;
72
73      case CALLBACK_SR:
74        printf("cf_SPEShutdownRequest");
75        break;
76
77      case CALLBACK_R:
78        printf("cf_SPERequest");
79        break;
80
81      case CALLBACK_RC:
82        printf("cf_SPERequestComplete");
83        break;
84
```

```
85      case CALLBACK_RR:
86        printf("cf_SPERegionRequest");
87        break;
88
89      case CALLBACK_RC1:
90        printf("cf_SPERegionComplete");
91        break;
92
93      case CALLBACK_RCS:
94        printf("cf_SPERegionCompleteSignal");
95        break;
96
97      case CALLBACK_RL:
98        printf("cf_SPERegisterLoop");
99        break;
100
101     case PROGRAM_COMPLETE_EXECUTION:
102       printf("PROGRAM_COMPLETE_EXECUTION");
103
104     default:
105       break;
106   }
107
108   //Newline
109   printf("\n\n");
110
111   /* Write data to output file */
112   for(std::multimap<unsigned int, measure_t>::iterator
        __MeasurementReadingMapIteratorInternal =
        __MeasurementReadingMapIteratorRange.first;
113                                    __MeasurementReadingMapIteratorInternal
                                        !=
                                       __MeasurementReadingMapIteratorRange
                                       .second;
114                                    ++
                                       __MeasurementReadingMapIteratorInternal
                                       ) {
```

```
115
116         s_measure = (*___MeasurementReadingMapIteratorInternal).
                second;
117         printf("%i,%i,%i,%llu,%llu\n",  s_measure.name, s_measure.
                spe, s_measure.element, s_measure.begin, s_measure.end)
                ;
118     }
119
120
121 //  fclose(p_FilenameOutput);
122     return 0;
123 }
```

B.4 kernel_system_headers.hpp

```
1 #ifndef KERNEL_SYSTEM_HEADERS_HPP_
2 #define KERNEL_SYSTEM_HEADERS_HPP_
3
4 /*Standard C headers*/
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <math.h>
8 #include <stdint.h>
9 #include "malloc.h"
10
11 /*Standard C++ headers*/
12 #include <list>
13 #include <deque>
14 #include <vector>
15 #include <map>
16 #include <queue>
17 #include <exception>
18 #include <cstring>
19
20 /*IBM Cell BE headers*/
21 #include <libspe2.h>
```

```
22 #include <ppu_intrinsics.h>
23 #include <altivec.h>
24
25 /*Pthread headers*/
26 #include <pthread.h>
27
28 /*Local headers*/
29 #include "common.hpp"
30 #include "random.hpp"
31
32 #endif /*KERNEL_SYSTEM_HEADERS_HPP_*/
```

B.5 kernel.hpp

```
1 #ifndef _KERNEL_H_
2 #define _KERNEL_H_
3
4 #include "kernel_system_headers.hpp"
5 #include "kernel_element_manager.hpp"
6 #include "kernel_measurement_reading.hpp"
7
8 typedef struct {
9   int sid;
10   void *argp;
11     void *envp;
12     pthread_t thread;
13     spe_context_ptr_t context;
14 } context_setup_t;
15
16 /*A list to hold that status information about the SPE*/
17 std::list<spe_status_t>          _SPEStatusList;
18 std::list<spe_status_t>::iterator    _SPEStatusListIterator;
19
20 /*A list to hold the work regions*/
21 std::list<context_params_t>        _WorkList;
22 std::list<context_params_t>::iterator _WorkListIterator;
```

```
23
24 /* Queue to store all requests made from SPEs */
25 std::list<request_message_t>       _RequestList;
26 std::list<request_message_t>::iterator  _RequestListIterator;
27
28 std::deque<recovery_info_t>        _RecoveryQueue;
29 std::deque<recovery_info_t>::iterator _RecoveryQueueIterator;
30
31 /* List to hold the number of regions created on the system */
32 std::list<region_complete_t>       _RegionList;
33 std::list<region_complete_t>::iterator  _RegionListIterator;
34
35 /* List to hold the registered regions */
36 std::list<loop_t>              _LoopRegionList;
37 std::list<loop_t>::iterator        _LoopRegionListIterator;
38
39 /*Typedef'd variable*/
40 typedef unsigned int spe_offset_t;
41
42
43 /*Linker to the SPU program*/
44 extern spe_program_handle_t D_SPU;
45
46
47 /*Structures*/
48 context_setup_t    *ps_SPEContext;
49 spe_stop_info_t    *ps_StopInfo;
50 spe_info_init_t    *ps_SPEInitialise;
51 context_params_t   *ps_ContextParameters    __attribute__((aligned
       (16)));
52 context_params_t   *ps_SPEParameters       __attribute__((aligned
       (16)));
53 request_message_t    s_RequestMessage      __attribute__((aligned
       (16)));
54 context_ls_t     s_SPEContextLS       __attribute__((aligned(16)))
       ;
55 context_ls_t     s_SPELS           __attribute__((aligned(16)));
```

```
56 context_params_t   s_RegionParameters;
57 monitor_report_t   s_Report;
58 spe_area_t        s_SPEArea;
59 parameters_t      *ps_MonitorParameters;
60 measure_t        s_Measure;
61
62 /*The BenchmarkAttribute is used to input the benchmark settings
       */
63 param_io_t               BenchmarkAttribute;
64 #define BenchmarkReport(x)      (kern.GenerateReport((x)))
65
66 /*Pthread's*/
67 pthread_t *pt_SPEResolveDual;
68 //pthread_t t_SPEResolve;
69 pthread_t t_SPEShutdownMonitor;
70 pthread_t t_SPEElementUpdater;
71 pthread_t *pt_SPEMailboxMonitor;
72 pthread_t *pt_ElementAnalyser;
73
74 /*Pthread sync data types*/
75 pthread_mutex_t m_SPEShutdown      = PTHREAD_MUTEX_INITIALIZER;
76 pthread_mutex_t m_SPESID          = PTHREAD_MUTEX_INITIALIZER;
77 pthread_mutex_t m_SPEIPCUpdater      = PTHREAD_MUTEX_INITIALIZER;
78 pthread_mutex_t m_RequestQueueLock     =
       PTHREAD_MUTEX_INITIALIZER;
79 pthread_mutex_t m_SPEHasShutdown     = PTHREAD_MUTEX_INITIALIZER;
80 pthread_mutex_t m_RegionComplete     = PTHREAD_MUTEX_INITIALIZER;
81 pthread_mutex_t m_RegionParameters     =
       PTHREAD_MUTEX_INITIALIZER;
82 pthread_mutex_t m_RegisterVariable     =
       PTHREAD_MUTEX_INITIALIZER;
83 pthread_mutex_t m_RequestComplete      =
       PTHREAD_MUTEX_INITIALIZER;
84 pthread_mutex_t m_RegionSignal       = PTHREAD_MUTEX_INITIALIZER;
85 pthread_mutex_t m_SPESignal         = PTHREAD_MUTEX_INITIALIZER;
86 pthread_mutex_t m_LoopRegion        = PTHREAD_MUTEX_INITIALIZER;
87 pthread_mutex_t m_Recovery          = PTHREAD_MUTEX_INITIALIZER;
```

```
88 pthread_mutex_t m_Insert          = PTHREAD_MUTEX_INITIALIZER;
89 pthread_mutex_t m_SPEIPC          = PTHREAD_MUTEX_INITIALIZER;
90
91 pthread_mutex_t m_MonitorArrayMonitor =
       PTHREAD_MUTEX_INITIALIZER;
92 pthread_mutex_t m_MeasureReading     = PTHREAD_MUTEX_INITIALIZER;
93
94
95 /* Callback functions */
96 int cf_SPEShutdownRequest(void *ls_base_tmp, unsigned int data);
97 int cf_SPERequest(void *ls_base_tmp, unsigned int data);
98 int cf_SPERequestComplete(void *ls_base_tmp, unsigned int data);
99 int cf_SPERegionRequest(void *ls_base_tmp, unsigned int data);
100 int cf_SPERegionComplete(void *ls_base_tmp, unsigned int data);
101 int cf_SPERegisterR3(void *ls_base_tmp, unsigned int data);
102 int cf_SPERegisterIPC(void *ls_base_tmp, unsigned int data);
103 int cf_SPERegionCompleteSignal(void *ls_base_tmp, unsigned int
       data);
104 int cf_SPERegisterLoop(void *ls_base_tmp, unsigned int data);
105
106 int cf_MeasurementReading(void *ls_base_tmp, unsigned int data);
107
108 /*Function prototypes*/
109 void *ptf_SPEContext(void *arg);
110 void *ptf_SPEAllShutdownCheck(void *unused);
111 void *ptf_SPEMonitorMailbox(void *arg);
112 void *ptf_SPERequestResolverScalar(void *arg);
113 void *ptf_SPERecovery(void *unused);
114 void *ptf_SPEDataOutputViolationAnalyser(void *arg);
115
116 /*Monitor variables*/
117 volatile unsigned int i_AllSPEShutdownFlag;
118 volatile unsigned int i_ResolveScalarRequestFlag;
119 volatile unsigned int i_MailboxMonitorFlag;
120 volatile unsigned int i_UpdaterFlag;
121 volatile unsigned int *i_ElementAnalyserFlag;
122
```

```
123 unsigned int _i_BaseSize;
124 unsigned int i_ActiveSPECount;
125 unsigned int *pi_SPENumber;
126 unsigned int *pi_SPEIPC;
127 unsigned int *pi_MonitorCompleteFlag;
128
129 unsigned int _i_ConstructorCount = 0;
130 int _i_RequestCounter = 0;
131
132 /*Master Input and Output*/
133 double *SingleArrayInput;
134 double *SingleArrayOutput;
135 double **DoubleArrayInput;
136 double **DoubleArrayOutput;
137 double *val;
138 int *col;
139 int *row;
140
141 /*Store element array*/
142 int *MonitorArray;
143
144
145 #define InsertBenchmark(x)       kern.InsertBenchmarkParameters(x
      )
146 #define InitialiseSystem       Kernel kern
147 #define SystemRun          kern.Run()
148 #define MonitorSize(x)         kern.SetMonitorArraySize(x)
149
150
151 class Kernel {
152   /*Internal kernel functions*/
153   unsigned int k_GetSPEContexts(void);
154   unsigned int k_GetAvailableSPECount(unsigned int info, int cpu
        );
155   unsigned int k_GetLastProcessedIPC(unsigned int id);
156   unsigned int k_Partition(context_params_t regionParameters);
157   unsigned int k_WaitForRegionCompletion(unsigned int id);
```

```
158   void k_StartAllSPE(void);
159   void k_RunSPE(int id);
160   void k_JoinSPEContexts(int id);
161   void k_DestroySPE(int id);
162   void k_LoadSPEProgram(int id);
163   void k_CreateSPEContext(int id, unsigned int flags);
164   void k_SignalSPEStart(void);
165   void k_Recovery(recovery_info_t recovery);
166   void k_AssignSPELS(int id);
167
168 public:
169   Kernel();
170   ~Kernel() {}
171   void Run(void);
172   void SystemInitialise(void);
173   void InsertBenchmarkParameters(param_io_t ioParameters);
174   int IterationBelongsToSPE(element_t *obj);
175   int IterationNumber(element_t *obj);
176   void SetMonitorArraySize(int size);
177
178   int InsertMetricResult(unsigned int name, unsigned long start,
          unsigned long end);
179   int GenerateReport(unsigned int specifier);
180 };
181
182 #endif
183
184 //Global kernel class
185 Kernel kern;
186 ElementManager em;
187 MeasurementReading mr;
188
189
190
191
192
193
```

```
//
    ////////////////////////////////////////////////////////////////

//////////////////////// KERNEL FUNCTIONS
    ///////////////////////
//
    ////////////////////////////////////////////////////////////////

Kernel::Kernel() {
  SystemInitialise();
}

int Kernel::GenerateReport(unsigned int specifier) {
  mr.CreateReport(specifier);
  return 0;
}

int Kernel::InsertMetricResult(unsigned int name, unsigned long
    start, unsigned long end) {
  THREAD_USLEEP_DEFAULT
  pthread_mutex_lock(&m_MeasureReading);

  s_Measure.name = name;
  s_Measure.element = PPU_ELEMENT;
  s_Measure.begin = start;
  s_Measure.end = end;
//  s_Measure.value = (end-start);

  mr.Insert(s_Measure);

```

```
225   pthread_mutex_unlock(&m_MeasureReading);
226   return 0;
227 }
228
229 void Kernel::SetMonitorArraySize(int size) {
230   _i_BaseSize = size;
231
232   MonitorArray = (int *)malloc(size*sizeof(int));
233
234   pthread_mutex_lock(&m_MonitorArrayMonitor);
235   for(int i = 0; i < size; i++) {
236     MonitorArray[i] = NO_STORE;
237   }
238   pthread_mutex_unlock(&m_MonitorArrayMonitor);
239 }
240
241 void Kernel::Run(void) {
242   unsigned int i;
243
244   // 1 - Load the programs into the SPE contexts
245   for(i=0; i<i_ActiveSPECount; i++) { kern.k_LoadSPEProgram(i);
          }
246
247   // 2 - Run the SPE contexts
248   for(i=0; i<i_ActiveSPECount; i++) { kern.k_RunSPE(i); }
249
250   // 3 - Listen for MASTER SHUTDOWN = COMPLETE
251   while(i_AllSPEShutdownFlag != TRUE) { THREAD_USLEEP_DEFAULT };
252
253   for(i=0; i<i_ActiveSPECount; i++) {
254     i_ElementAnalyserFlag[i] = THREAD_EXIT;
255 //     pthread_cancel(pt_ElementAnalyser[i]);
256   }
257
258   i_MailboxMonitorFlag = i_ResolveScalarRequestFlag =
          i_UpdaterFlag = FALSE;
259
```

```
260   // 4 - Join all SPEs
261   for(i=0; i<i_ActiveSPECount; i++) { kern.k_JoinSPEContexts(i);
          }
262
263   // 5 - Destroy all SPEs
264   for(i=0; i<i_ActiveSPECount; i++) { kern.k_DestroySPE(i); }
265 }
266
267 void Kernel::SystemInitialise(void) {
268   if(_i_ConstructorCount!=1) {
269     _i_ConstructorCount++;
270
271     unsigned i;
272     i_ActiveSPECount = 6;//k_GetAvailableSPECount(
          SPE_COUNT_USABLE_SPES, 0);
273
274 //     i_ElementAnalyserFlag = FALSE;
275     i_MailboxMonitorFlag = TRUE;
276     i_ResolveScalarRequestFlag = TRUE;
277
278     spe_callback_handler_register((void *)cf_SPEShutdownRequest,
              0x14, SPE_CALLBACK_NEW);
279     spe_callback_handler_register((void *)cf_SPERequest,
          0x15, SPE_CALLBACK_NEW);
280     spe_callback_handler_register((void *)cf_SPERegisterIPC,
            0x16, SPE_CALLBACK_NEW);
281     spe_callback_handler_register((void *)cf_SPERegionRequest,
              0x18, SPE_CALLBACK_NEW);
282     spe_callback_handler_register((void *)cf_SPERegionComplete,
              0x19, SPE_CALLBACK_NEW);
283     spe_callback_handler_register((void *)cf_SPERegisterR3,
            0x21, SPE_CALLBACK_NEW);
284     spe_callback_handler_register((void *)cf_SPERequestComplete,
              0x23, SPE_CALLBACK_NEW);
285     spe_callback_handler_register((void *)cf_MeasurementReading,
              0x25, SPE_CALLBACK_NEW);
286     spe_callback_handler_register((void *)cf_SPERegisterLoop,
```

```
            0x26, SPE_CALLBACK_NEW);
287     spe_callback_handler_register((void *)
            cf_SPERegionCompleteSignal,    0x27, SPE_CALLBACK_NEW);
288
289
290     ps_ContextParameters = (context_params_t *) realloc(
            ps_ContextParameters, i_ActiveSPECount * sizeof(
            context_params_t));
291     ps_SPEInitialise = (spe_info_init_t *) realloc(
            ps_SPEInitialise, i_ActiveSPECount * sizeof(
            spe_info_init_t));
292     ps_SPEContext = (context_setup_t *) realloc(ps_SPEContext,
            i_ActiveSPECount * sizeof(context_setup_t));
293     pt_SPEResolveDual = (pthread_t *)malloc(sizeof(pthread_t)*
            i_ActiveSPECount);
294     pt_SPEMailboxMonitor = (pthread_t *)malloc(sizeof(pthread_t)
            *i_ActiveSPECount);
295     pi_SPEIPC = (unsigned int *) malloc(sizeof(unsigned int)*
            i_ActiveSPECount);
296     pi_SPENumber = (unsigned int *) malloc(sizeof(unsigned int)*
            i_ActiveSPECount);
297     pt_ElementAnalyser = (pthread_t *)malloc(sizeof(pthread_t)*
            i_ActiveSPECount);
298     ps_MonitorParameters = (parameters_t *)malloc(sizeof(
            parameters_t)*i_ActiveSPECount);
299
300     i_ElementAnalyserFlag = (unsigned int *)malloc(sizeof(
            i_ElementAnalyserFlag));
301
302     for(i = 0; i < i_ActiveSPECount; i++) { pi_SPEIPC[i] = 0;
            i_ElementAnalyserFlag[i] = FALSE; }
303
304     pi_SPENumber = new unsigned int [i_ActiveSPECount];
305     ps_StopInfo = new spe_stop_info_t [i_ActiveSPECount];
306
307     for(i=0; i<i_ActiveSPECount; i++) {
308       pi_SPENumber[i] = i;
```

```
309       k_CreateSPEContext(i, SPE_CFG_SIGNOTIFY1_OR |
              SPE_EVENTS_ENABLE | SPE_MAP_PS);
310       ps_SPEContext[i].argp = (void *)pi_SPENumber[i];
311       ps_SPEContext[i].sid = i;
312     }
313
314     for(i=0; i<i_ActiveSPECount; i++) {
315       k_AssignSPELS(i);
316     }
317
318     spe_status_t spe_status;
319
320     /*Setup the status for each SPE*/
321     for(i=0; i<i_ActiveSPECount; i++) {
322       spe_status.spe = i;
323       spe_status.r3 = s_SPELS.ea_addr[i];
324       spe_status.status = SPE_NO_DEPENDENCY;
325
326       _SPEStatusList.push_back(spe_status);
327     }
328
329     pthread_attr_t detach;
330
331     if(pthread_attr_init(&detach) != 0) {
332       perror("pthread_attr_init");
333     }
334
335     if(pthread_attr_setdetachstate(&detach,
              PTHREAD_CREATE_DETACHED) != 0) {
336       perror ("pthread_attr_setdetachstate");
337     }
338
339     if(pthread_create(&t_SPEShutdownMonitor, &detach,
              ptf_SPEAllShutdownCheck, NULL)) {
340       perror("pthread_create");
341       exit(1);
342     }
```

```
343
344     for(i=0; i<1; i++) {
345       if(pthread_create(&pt_SPEResolveDual[i], &detach,
              ptf_SPERequestResolverScalar, NULL)) {
346         perror("pthread_create");
347         exit(1);
348       }
349     }
350
351     for(i=0; i<i_ActiveSPECount; i++) {
352       if(pthread_create(&pt_SPEMailboxMonitor[i], &detach,
              ptf_SPEMonitorMailbox, &pi_SPENumber[i])) {
353         perror("pthread_create");
354         exit(1);
355       }
356     }
357
358     pthread_attr_destroy(&detach);
359   }
360 }
361
362 void Kernel::InsertBenchmarkParameters(param_io_t ioParameters)
      {
363   context_params_t s_Context;
364
365   s_Context.bench =      ioParameters.bench_name;
366   s_Context.region =     ioParameters.bench_id;
367   s_Context.itr_total =    ioParameters.total_iterations;
368   s_Context.size =     ioParameters.size_1_dataset;
369   s_Context.size2 =      ioParameters.size_2_dataset;
370   s_Context.ea_in =      ioParameters.input;
371   s_Context.ea_out =     ioParameters.output;
372   s_Context.ea_aux =     ioParameters.aux;
373   s_Context.ea_aux2 =    ioParameters.aux2;
374   s_Context.ea_aux3 =    ioParameters.aux3;
375
376   kern.k_Partition(s_Context);
```

```
377 }
378
379 unsigned int Kernel::k_GetSPEContexts(void) {
380   return i_ActiveSPECount;
381 }
382
383 unsigned int Kernel::k_GetAvailableSPECount(unsigned int info,
      int cpu) {
384   return spe_cpu_info_get(info, cpu);
385 }
386
387 unsigned int Kernel::k_GetLastProcessedIPC(unsigned int id) {
388   static unsigned int ipc_data;
389   unsigned int *tag_status;
390
391   for(_SPEStatusListIterator = _SPEStatusList.begin();
        _SPEStatusListIterator != _SPEStatusList.end(); ++
        _SPEStatusListIterator) {
392     if(_SPEStatusListIterator->spe==id) {
393       spe_mfcio_put(ps_SPEContext[id].context,
            _SPEStatusListIterator->ipc, &ipc_data, sizeof(ipc_data
            ), id, 0, 0);
394       spe_mfcio_tag_status_read(ps_SPEContext[id].context, 1<<id
            , SPE_TAG_ANY, tag_status);
395       break;
396     }
397   }
398
399   return ipc_data;
400 }
401
402 unsigned int Kernel::k_Partition(context_params_t
      regionParameters) {
403   unsigned int *itarray, *sizearray, *sizearray2, total,
        totalsize, totalsize2, i, last, last2;
404
405   sizearray = new unsigned int [i_ActiveSPECount];
```

```
406   sizearray2 = new unsigned int [i_ActiveSPECount];
407   itarray = new unsigned int [i_ActiveSPECount];
408
409   totalsize = regionParameters.size;
410   totalsize2 = regionParameters.size2;
411
412   total = regionParameters.itr_total;
413
414   for(i=0; i<i_ActiveSPECount; i++) {
415     itarray[i] = 0;
416     sizearray[i] = 0;
417     sizearray2[i] = 0;
418   }
419
420   if(regionParameters.itr_total>1) {
421     ADD_AGAIN:
422     for(i=0; i<i_ActiveSPECount; i++) {
423       if(total!=0) {
424         itarray[i]=itarray[i]+1;
425         total--;
426         continue;
427       }
428       else
429         break;
430     }
431
432     if(total>0)
433       goto ADD_AGAIN;
434   }
435   else {
436     for(i=0; i<i_ActiveSPECount; i++) {
437       itarray[i] = regionParameters.itr_total;
438     }
439   }
440
441   if(regionParameters.size!=0) {
442     SIZE_ADD_AGAIN:
```

```
443     for(i=0; i<i_ActiveSPECount; i++) {
444       if(totalsize != 0) {
445         sizearray[i]=sizearray[i]+1;
446         totalsize--;
447 //        continue;
448       }
449       else
450         break;
451     }
452
453     if(totalsize != 0)
454       goto SIZE_ADD_AGAIN;
455   }
456
457   if(regionParameters.size2!=0) {
458     SIZE_ADD_AGAIN2:
459     for(i=0; i<i_ActiveSPECount; i++) {
460       if(totalsize2 != 0) {
461         sizearray2[i]=sizearray2[i]+1;
462         totalsize2--;
463 //        continue;
464       }
465       else
466         break;
467     }
468
469     if(totalsize2 != 0)
470       goto SIZE_ADD_AGAIN2;
471   }
472
473   for(i=0; i<i_ActiveSPECount; i++) {
474     if(i==(i_ActiveSPECount-1))
475       s_RegionParameters.final = 1;
476     else
477       s_RegionParameters.final = 0;
478
479     s_RegionParameters.ea_aux   = regionParameters.ea_aux;
```

```
480     s_RegionParameters.ea_aux2  = regionParameters.ea_aux2;
481     s_RegionParameters.ea_aux3  = regionParameters.ea_aux3;
482
483     s_RegionParameters.ea_in  = regionParameters.ea_in;
484     s_RegionParameters.ea_out   = regionParameters.ea_out;
485     s_RegionParameters.region   = regionParameters.region;
486     s_RegionParameters.spe    = i;
487     s_RegionParameters.assigned = -1;
488     s_RegionParameters.size   = sizearray[i];
489     s_RegionParameters.size2  = sizearray2[i];
490     s_RegionParameters.region_total_spe_count = i_ActiveSPECount
            ;
491
492     if(itarray[i]==1) {
493       s_RegionParameters.itr_begin = 0;
494       s_RegionParameters.itr_end = 1;
495     }
496     else {
497       if(i==(i_ActiveSPECount-1)) {
498         if(itarray[i]==1) {
499           s_RegionParameters.itr_begin = regionParameters.
                itr_total;
500           s_RegionParameters.itr_end = regionParameters.
                itr_total+1;
501         }
502       }
503       else {
504         s_RegionParameters.itr_begin = i*itarray[i];
505         s_RegionParameters.itr_end = i*itarray[i]+itarray[i];
506       }
507     }
508
509     if(s_RegionParameters.size > 0) {
510       if(i==0) {
511         s_RegionParameters.size_begin = i*sizearray[i];
512         s_RegionParameters.size_end = i*sizearray[i]+sizearray[i
              ];
```

```
513         last = s_RegionParameters.size_end;
514       }
515       else {
516         s_RegionParameters.size_begin = last;
517         s_RegionParameters.size_end = last+sizearray[i];
518         last = s_RegionParameters.size_end;
519       }
520     }
521
522     if(s_RegionParameters.size2 > 0) {
523       if(i==0) {
524         s_RegionParameters.size_begin_2 = i*sizearray2[i];
525         s_RegionParameters.size_end_2 = i*sizearray2[i]+
              sizearray2[i];
526         last2 = s_RegionParameters.size_end_2;
527       }
528       else {
529         s_RegionParameters.size_begin_2 = last2;
530         s_RegionParameters.size_end_2 = last2+sizearray2[i];
531         last2 = s_RegionParameters.size_end_2;
532       }
533     }
534
535     _WorkList.push_back(s_RegionParameters);
536   }
537
538   return 0;
539 }
540
541 unsigned int Kernel::k_WaitForRegionCompletion(unsigned int id)
      {
542   unsigned int _i_fc_counter;
543
544   while(1) {
545     _i_fc_counter = 0;
546     for(_RegionListIterator = _RegionList.begin();
          _RegionListIterator != _RegionList.end(); ++
```

```
        _RegionListIterator) {
547       if(_RegionListIterator->region == id) {
548         _RegionListIterator->complete = REGION_COMPLETED;
549         _i_fc_counter++;
550       }
551     }
552
553     if(_i_fc_counter==i_ActiveSPECount) {
554       break;
555     }
556
557     THREAD_USLEEP_DEFAULT
558   }
559   return TRUE;
560 }
561
562
563 int Kernel::IterationBelongsToSPE(element_t *obj) {
564   unsigned int buff = obj->io_array_type;
565
566   for(_WorkListIterator = _WorkList.begin(); _WorkListIterator
        != _WorkList.end(); ++_WorkListIterator) {
567
568     if(buff==0) {
569       for(unsigned int i = _WorkListIterator->size_begin; i <
            _WorkListIterator->size_end; i++) {
570         if(i==obj->index_a)
571           return (int)obj->spe;
572       }
573     }
574
575     if(buff==1) {
576       for(unsigned int i = _WorkListIterator->size_begin_2; i <
            _WorkListIterator->size_end_2; i++) {
577         if(i==obj->index_b)
578           return (int)obj->spe;
579       }
```

```
580       }
581     }
582
583     return -1;
584 }
585
586 int Kernel::IterationNumber(element_t *obj) {
587     unsigned int buff = obj->io_array_type;
588
589     for(_WorkListIterator = _WorkList.begin(); _WorkListIterator
          != _WorkList.end(); ++_WorkListIterator) {
590
591       if(buff==0) {
592         for(unsigned int i = _WorkListIterator->size_begin; i <
              _WorkListIterator->size_end; i++) {
593           if(i==obj->index_a)
594             return i;
595         }
596       }
597
598       if(buff==1) {
599         for(unsigned int i = _WorkListIterator->size_begin_2; i <
              _WorkListIterator->size_end_2; i++) {
600           if(i==obj->index_b)
601             return i;
602         }
603       }
604     }
605
606     return FUNCTION_FAILED;
607 }
608
609
610 void Kernel::k_StartAllSPE(void) {
611     unsigned int i;
612     for(i=0; i<i_ActiveSPECount; i++) {
613       if(pthread_create(&ps_SPEContext[i].thread, NULL,
```

```
          ptf_SPEContext, &ps_SPEContext[i])) {
614         perror("pthread_create");
615         exit(1);
616     }
617     }
618 }
619
620 void Kernel::k_RunSPE(int id) {
621   if(pthread_create(&ps_SPEContext[id].thread, NULL,
        ptf_SPEContext, &ps_SPEContext[id])) {
622         perror("pthread_create");
623         exit(1);
624     }
625 }
626
627 void Kernel::k_JoinSPEContexts(int id) {
628   for(_SPEStatusListIterator = _SPEStatusList.begin();
        _SPEStatusListIterator != _SPEStatusList.end(); ++
        _SPEStatusListIterator) {
629     if(_SPEStatusListIterator->status==SPE_STANDBY ||
          _SPEStatusListIterator->status==SPE_HALT) {
630       spe_signal_write(ps_SPEContext[_SPEStatusListIterator->spe
            ].context, SPE_SIG_NOTIFY_REG_1, SPE_SHUTDOWN);
631     }
632   }
633
634   pthread_join(ps_SPEContext[id].thread, NULL);
635 }
636
637 void Kernel::k_DestroySPE(int id) {
638   if(spe_context_destroy(ps_SPEContext[id].context) == -1) {
639       perror("spe_context_destroy");
640       exit(1);
641   }
642 }
643
644 void Kernel::k_LoadSPEProgram(int id) {
```

```
645   if((spe_program_load(ps_SPEContext[id].context, &D_SPU)) != 0)
          {
646     perror("spe_program_load");
647     exit(1);
648   }
649 }
650
651 void Kernel::k_CreateSPEContext(int id, unsigned int flags) {
652   if((ps_SPEContext[id].context = spe_context_create(flags, NULL
        )) == NULL) {
653       perror("spe_context_create");
654         exit(1);
655     }
656
657     spe_sig_notify_1_area_t *sig_area1;
658     spe_sig_notify_2_area_t *sig_area2;
659
660   sig_area1 = (spe_sig_notify_1_area_t *)spe_ps_area_get(
        ps_SPEContext[id].context, SPE_SIG_NOTIFY_1_AREA);
661     sig_area2 = (spe_sig_notify_2_area_t *)spe_ps_area_get(
          ps_SPEContext[id].context, SPE_SIG_NOTIFY_2_AREA);
662
663     s_SPEArea.reg1[id] = (unsigned long long)&(sig_area1->
          SPU_Sig_Notify_1);
664     s_SPEArea.reg2[id] = (unsigned long long)&(sig_area2->
          SPU_Sig_Notify_2);
665 }
666
667 void Kernel::k_SignalSPEStart(void) {
668   unsigned int i;
669   for(i=0; i<i_ActiveSPECount; i++) {
670       spe_signal_write(ps_SPEContext[i].context,
            SPE_SIG_NOTIFY_REG_1, 1);
671   }
672 }
673
674 void Kernel::k_AssignSPELS(int id) {
```

```
675   s_SPELS.ea_addr[id] = (unsigned long long)spe_ls_area_get(
          ps_SPEContext[id].context);
676 }
677
678
679
680
681
682
683
684
685
686
687
688
689
690 //
      ////////////////////////////////////////////////////////////////

691 ////////////////////// THREAD FUNCTIONS
      ////////////////////////
692 //
      ////////////////////////////////////////////////////////////////

693
694 void *ptf_SPEDataOutputViolationAnalyser(void *arg) {
695   unsigned long long exec_start, exec_stop;
696
697   /*******************happens only once*********************/
698   parameters_t *_s_Init = (parameters_t *)arg;
699   do {
700     THREAD_USLEEP_DEFAULT
701     pthread_testcancel();
702   } while(i_ElementAnalyserFlag[_s_Init->spe]==THREAD_PAUSE);
703   /*********************************************************/
704
705
```

```
706  __ANALYSER_THREAD_RESTART:
707  recovery_info_t s_Recovery;
708  parameters_t *_s_Range = (parameters_t *)arg;
709  unsigned __ir_IndexS, __ir_IndexE, _i_Pos;
710
711  pthread_testcancel();
712
713  while(1) {
714    exec_start = __mftb();
715
716    if(i_ElementAnalyserFlag[_s_Range->spe]==TRUE) {
717      pthread_testcancel();
718
719      __ir_IndexS = _s_Range->begin;
720      __ir_IndexE = _s_Range->end;
721
722      for(_i_Pos = __ir_IndexS; _i_Pos < __ir_IndexE; _i_Pos++)
           {
723        pthread_testcancel();
724        if(i_ElementAnalyserFlag[_s_Range->spe]==THREAD_PAUSE)
             goto ___ANALYSER_THREAD_HAS_PAUSED;
725        if(i_ElementAnalyserFlag[_s_Range->spe]==THREAD_RESTART)
             goto __ANALYSER_THREAD_RESTART;
726        if(i_ElementAnalyserFlag[_s_Range->spe]==THREAD_EXIT)
             goto __ANALYSER_THREAD_EXIT;
727
728        /* Check if the _i_Pos iteration exists in the EMAP. If
             it does not
729         * exist then create an element and set the following
              values to the
730         * member functions of the EMAP element
731         *
732         *  ELEMENT.owner =  _i_Pos;
733         *  ELEMENT.load = FALSE;
734         *  ELEMENT.store = TRUE;
735         *  ELEMENT.last_store = _i_Pos;
736         */
```

```
737           if(em.CheckByIndex(_i_Pos)==FALSE) {
738             element_t _e;
739             _e.iteration_owner = _i_Pos;
740             _e.load = FALSE;
741             _e.store = TRUE;
742             _e.last_store = _i_Pos;
743
744             em.InsertElement(_i_Pos, _e);
745           }
746           else {
747             element_t _e;
748
749             /* The element in the multiset exists. Therefore, we
                    need
750              * to firstly determine what flags have been set.
                    Remember,
751              * we need to lock the found element!
752              */
753             _e = em.GetElementByIndex(_i_Pos);
754
755             pthread_mutex_lock(&m_MonitorArrayMonitor);
756             if(MonitorArray[_i_Pos]==EXTERNAL_STORE) {
757               pthread_mutex_unlock(&m_MonitorArrayMonitor);
758 //              continue;
759             }
760             else if (MonitorArray[_i_Pos]==INTERNAL_STORE) {
761               pthread_mutex_unlock(&m_MonitorArrayMonitor);
762               em.WaitAndEnableExclusive(_e, _i_Pos);
763
764               if(_e.load == FALSE) {
765                 _e.store = TRUE;
766                 _e.last_store = _i_Pos;
767
768                 em.ReinsertElement(_i_Pos, _e);
769                 em.WaitAndDisableExclusive(_e, _i_Pos);
770               }
771
```

```
772             if(_e.load == TRUE) {
773
774               if(_i_Pos < _e.last_load) {
775                 em.WaitAndDisableExclusive(_e, _i_Pos);
776
777                 //Violation
778                 s_Recovery.element_data_from = _e;
779                 s_Recovery.element_to_update = _e;
780                 _RecoveryQueue.push_back(s_Recovery);
781               }
782               else {
783                 //_i_Pos > _e.last_load
784                 _e.store = TRUE;
785                 _e.last_store = _i_Pos;
786
787                 em.ReinsertElement(_i_Pos, _e);
788                 em.WaitAndDisableExclusive(_e, _i_Pos);
789               }
790
791             }//if(_e.load == TRUE)
792
793           }//else if (MonitorArray[_i_Pos]==INTERNAL_STORE)
794           else {
795             pthread_mutex_unlock(&m_MonitorArrayMonitor);
796           }
797         }//else
798
799         THREAD_USLEEP_DEFAULT
800         pthread_testcancel();
801       }//for(_i_Pos = __ir_IndexS; _i_Pos < __ir_IndexE; _i_Pos
            ++)
802     }//if(i_ElementAnalyserFlag[_s_Range->spe]==TRUE) {
803
804     pthread_testcancel();
805     if(i_ElementAnalyserFlag[_s_Range->spe]==THREAD_EXIT) {
806       goto __ANALYSER_THREAD_EXIT;
807     }
```

```
808
809     pthread_testcancel();
810     if(i_ElementAnalyserFlag[_s_Range->spe]==THREAD_RESTART) {
811       i_ElementAnalyserFlag = 0;
812       goto __ANALYSER_THREAD_RESTART;
813     }
814
815
816     ___ANALYSER_THREAD_HAS_PAUSED:
817     if(i_ElementAnalyserFlag[_s_Range->spe]==THREAD_PAUSE) {
818
819       do {
820         THREAD_USLEEP_DEFAULT
821         pthread_testcancel();
822       } while(i_ElementAnalyserFlag[_s_Range->spe]==THREAD_PAUSE
            );
823
824       switch(i_ElementAnalyserFlag[_s_Range->spe]) {
825
826         case THREAD_RESTART:
827           goto __ANALYSER_THREAD_RESTART;
828           break;
829
830         case THREAD_EXIT:
831           goto __ANALYSER_THREAD_EXIT;
832           break;
833
834         default:
835           break;
836       }
837     }
838
839     exec_stop = ___mftb();
840     kern.InsertMetricResult(THREAD_DOVA, exec_start, exec_stop);
841
842     THREAD_USLEEP_DEFAULT
843     pthread_testcancel();
```

```
844   }//while(1)
845
846   pthread_testcancel();
847
848   __ANALYSER_THREAD_EXIT:
849   pthread_exit(NULL);
850 }
851
852 void *ptf_SPERecovery(void *unused) {
853   element_t _e_src, _e_dst;
854   recovery_info_t s_Recovery;
855   unsigned i_Source, i_Destination, i_Total;
856   unsigned long long exec_start, exec_stop;
857
858   while(1) {
859     exec_start = __mftb();
860
861     while(_RecoveryQueue.empty()) {
862       pthread_testcancel();
863       THREAD_USLEEP_DEFAULT
864     }
865
866     i_ResolveScalarRequestFlag=FALSE;
867
868     pthread_testcancel();
869
870     pthread_mutex_lock(&m_Recovery);
871     s_Recovery = _RecoveryQueue.front();
872     _RecoveryQueue.pop_front();
873     pthread_mutex_unlock(&m_Recovery);
874
875     _e_src = s_Recovery.element_data_from;
876     _e_dst = s_Recovery.element_to_update;
877
878     i_Source = _e_src.iteration_owner;
879     i_Destination = _e_dst.iteration_owner;
880
```

```
881     /*Send a signal to all SPEs to HALT*/
882     for(unsigned i = 0; i < i_ActiveSPECount; i++) {
883       if(i!=_e_src.spe)
884         spe_signal_write(ps_SPEContext[i].context,
              SPE_SIG_NOTIFY_REG_1, SPE_HALT);
885     }
886
887     /*Update the SPE status list*/
888     for(_SPEStatusListIterator = _SPEStatusList.begin();
          _SPEStatusListIterator != _SPEStatusList.end(); ++
          _SPEStatusListIterator) {
889       _SPEStatusListIterator->status=SPE_HALT;
890     }
891
892     /*Clean up the element map*/
893     /*Element type = INNER/OUTER & Element identifier = [i:i-n)
          */
894     for(unsigned long int i = i_Destination+1; i <= em.GetSize()
          ; i++) {
895       em.RemoveElementByLoopRangeIdentifer(_e_src.loop_level, i)
            ;
896 //      em.RemoveByIndex(i);
897     }
898
899     for(_WorkListIterator = _WorkList.begin(); _WorkListIterator
          != _WorkList.end(); ++_WorkListIterator) {
900       if(_WorkListIterator->region==_e_dst.region) {
901         i_Total = _WorkListIterator->itr_total;
902         break;
903       }
904     }
905
906     /*Delete all pending requests*/
907     _RequestList.erase(_RequestList.begin(), _RequestList.end())
          ;
908
909     /*Start request listening thread(s)*/
```

```
910     i_ResolveScalarRequestFlag=TRUE;
911
912     /*Tell the SPE who made the request to go ahead and do its
            thing*/
913     spe_signal_write(ps_SPEContext[_e_src.spe].context,
            SPE_SIG_NOTIFY_REG_2, SPE_CONTINUE);
914
915     /*Signal all SPEs to start*/
916     for(unsigned i = 0; i < i_ActiveSPECount; i++) {
917       if(i!=_e_src.spe)
918         spe_signal_write(ps_SPEContext[i].context,
              SPE_SIG_NOTIFY_REG_2, SPE_RESTART);
919     }
920
921     /*Update the SPE status list*/
922     for(_SPEStatusListIterator = _SPEStatusList.begin();
            _SPEStatusListIterator != _SPEStatusList.end(); ++
            _SPEStatusListIterator) {
923       _SPEStatusListIterator->status=SPE_CONTINUE;
924     }
925
926     exec_stop = __mftb();
927     kern.InsertMetricResult(THREAD_R, exec_start, exec_stop);
928   }
929 }
930
931 void *ptf_SPEContext(void *arg) {
932   unsigned int entry = SPE_DEFAULT_ENTRY;
933   context_setup_t *self = (context_setup_t*)arg;
934
935   if(spe_context_run(self->context, &entry, 0, self->argp, self
          ->envp, &ps_StopInfo[self->sid]) < 0) {
936       perror("spe_context_run");
937       exit(1);
938     }
939
940     pthread_exit(NULL);
```

```
941 }
942
943 void *ptf_SPEAllShutdownCheck(void *unused) {
944   unsigned int _i_fn_counter = 0;
945   unsigned long long exec_start, exec_stop;
946
947   while(1) {
948     exec_start = __mftb();
949     THREAD_USLEEP_DEFAULT
950
951     for(_SPEStatusListIterator = _SPEStatusList.begin();
          _SPEStatusListIterator != _SPEStatusList.end(); ++
          _SPEStatusListIterator) {
952       if(_SPEStatusListIterator->status==SPE_SHUTDOWN)
953         _i_fn_counter++;
954     }
955
956     if(i_ActiveSPECount==_i_fn_counter) {
957       i_AllSPEShutdownFlag=TRUE;
958       exec_stop = __mftb();
959       kern.InsertMetricResult(THREAD_ASC, exec_start, exec_stop)
            ;
960       goto SHUTDOWN_IF_EXIT;
961     }
962
963     _i_fn_counter=0;
964     exec_stop = __mftb();
965     kern.InsertMetricResult(THREAD_ASC, exec_start, exec_stop);
966   }
967
968  SHUTDOWN_IF_EXIT:
969   pthread_exit(NULL);
970 }
971
972 void *ptf_SPEMonitorMailbox(void *arg) {
973   unsigned int *spe_number = (unsigned int*)arg;
974   int _i_Last_IPC = -1;
```

```
975   unsigned int _i_Return;
976   unsigned long long exec_start, exec_stop;
977
978   while(1) {
979     exec_start = __mftb();
980     THREAD_USLEEP_DEFAULT
981     pthread_testcancel();
982
983     if(i_MailboxMonitorFlag==TRUE) {
984       /*test for thread cancel*/
985       pthread_testcancel();
986
987       /*read the contents of the mailbox*/
988       _i_Return = spe_out_mbox_status(ps_SPEContext[spe_number
              [0]].context);
989       if(_i_Return==1) {
990         spe_out_mbox_read(ps_SPEContext[spe_number[0]].context,
              &_i_Return, 1);
991
992         /*if the last processed iteration == the latest read
              from the mailbox, then do nothing*/
993         if(_i_Last_IPC!=_i_Return) {
994           /*if the last processed iteration != the latest read
                from the mailbox, then update
995           * the processed value*/
996           pthread_mutex_lock(&m_SPEIPCUpdater);
997           pi_SPEIPC[spe_number[0]] = _i_Return;
998           pthread_mutex_unlock(&m_SPEIPCUpdater);
999         }
1000      }
1001    }
1002
1003    exec_stop = __mftb();
1004    kern.InsertMetricResult(THREAD_MM, exec_start, exec_stop);
1005  }
1006 }
1007
```

```
1008 void *ptf_SPERequestResolverScalar(void *arg) {
1009   pthread_testcancel();
1010
1011   /*temporary elements*/
1012   element_t _e_tmp, _e_src, _e_dst;
1013
1014   /*recovery struct*/
1015   recovery_info_t s_Recovery;
1016
1017   /*request*/
1018   request_message_t s_Request;
1019
1020   context_params_t s_Context;
1021
1022   unsigned long long exec_start, exec_stop;
1023
1024   while(1) {
1025     __RESOLVE_TRYAGAIN:
1026     exec_start = ___mftb();
1027
1028     if(i_ResolveScalarRequestFlag==TRUE) {
1029
1030       while(_RequestList.empty()) {
1031         pthread_testcancel();
1032         THREAD_USLEEP_DEFAULT
1033
1034         if(i_ResolveScalarRequestFlag==FALSE)
1035           goto __RESOLVE_TRYAGAIN;
1036       }
1037
1038       pthread_testcancel();
1039
1040       pthread_mutex_lock(&m_RequestQueueLock);
1041
1042       s_Request = _RequestList.front();
1043       _RequestList.pop_front();
1044
```

```
1045        pthread_mutex_unlock(&m_RequestQueueLock);
1046
1047
1048        //REQUEST STRUCT ---(PARSING)---> TEMPORARY VARIABLES ---->
                ELEMENT
1049
1050        int spe_sid =         s_Request.spe_sid;
1051        int region_number =     s_Request.region_number;
1052        int request_type =      s_Request.request_type;
1053        int iteration_owner =     s_Request.owner_itr;
1054        int index_a =         s_Request.index_a;
1055        int index_b =         s_Request.index_b;
1056
1057        _e_tmp.spe =          spe_sid;
1058        _e_tmp.region =         region_number;
1059        _e_tmp.request_type =       request_type;
1060        _e_tmp.iteration_owner =  iteration_owner;
1061        _e_tmp.aux =          s_Request.aux;
1062        _e_tmp.index_a =        index_a;
1063        _e_tmp.index_b =        index_b;
1064        _e_tmp.loop_level =       s_Request.loop_level;
1065        _e_tmp.level =          s_Request.level;
1066        _e_tmp.exclusive =        FALSE;
1067        _e_tmp.io_array_type =      s_Request.io_array_type;
1068        _e_tmp.outer_level =      s_Request.outer_level;
1069        _e_src = _e_tmp;
1070
1071
1072        // Insert OWNER element into the correct table
1073        if(s_Request.aux==-1) {
1074
1075          if(em.Find(_e_src, _e_src.io_array_type)==FALSE) {
1076            if(_e_src.iteration_owner <= pi_SPEIPC[_e_src.spe]) {
1077              _e_src.load = FALSE;
1078              _e_src.store = TRUE;
1079              _e_src.last_store = _e_src.iteration_owner;
1080            }
```

```
1081
1082            em.InsertElement(_e_src.iteration_owner, _e_src);
1083          }
1084
1085          _e_dst = _e_tmp;
1086          _e_dst.iteration_owner = em.GetElementOwner(_e_tmp,
                  _e_tmp.io_array_type);
1087
1088          if(em.Find(_e_dst, _e_dst.io_array_type)==FALSE) {
1089
1090            _e_dst.spe = kern.IterationBelongsToSPE(&_e_dst);
1091  //          _e_dst.iteration_owner = kern.IterationNumber(&
      _e_src);
1092
1093            if(_e_dst.iteration_owner <= pi_SPEIPC[_e_dst.spe]) {
1094              _e_dst.load = FALSE;
1095              _e_dst.store = TRUE;
1096              _e_dst.last_store = _e_src.iteration_owner;
1097            }
1098
1099            em.InsertElement(_e_dst.iteration_owner, _e_dst);
1100          }
1101
1102
1103          /*context params holds the parameters for the needed
                region*/
1104          for(_WorkListIterator = _WorkList.begin();
                _WorkListIterator != _WorkList.end(); ++
                _WorkListIterator) {
1105            if(_WorkListIterator->region==(unsigned int)
                  region_number) {
1106              s_Context = *_WorkListIterator;
1107              break;
1108            }
1109          }
1110
1111          /*lock both elements (source & destination)*/
```

```
1112          em.WaitAndEnableExclusive(_e_src, _e_src.io_array_type);
1113          em.WaitAndEnableExclusive(_e_dst, _e_dst.io_array_type);
1114        }
1115
1116
1117        _e_src = em.GetElement(_e_src, _e_src.io_array_type);
1118        _e_dst = em.GetElement(_e_dst, _e_dst.io_array_type);
1119
1120        if(request_type==LOAD_AUX) {
1121          //does the aux element exist? if yes, then lock it. if
                  no,
1122          //then create the element and then lock it!
1123          if(em.ExistElementByAuxIndex(_e_src.aux, _e_src.
                  iteration_owner)==FALSE) {
1124            em.InsertElement(_e_src.iteration_owner, _e_src);
1125          }
1126
1127          em.WaitAndEnableExclusive(_e_src, _e_src.io_array_type);
1128
1129          _e_dst.iteration_owner = em.GetElementOwner(_e_tmp,
                  _e_tmp.io_array_type);
1130          if(em.ExistElementByAuxIndex(_e_dst.aux, _e_dst.
                  iteration_owner)==FALSE) {
1131            em.InsertElement(_e_dst.iteration_owner, _e_dst);
1132          }
1133
1134          em.WaitAndEnableExclusive(_e_dst, _e_dst.io_array_type);
1135
1136          _e_dst.load = TRUE;
1137          _e_dst.last_load = _e_src.iteration_owner;
1138          em.ReinsertElement(_e_dst.iteration_owner, _e_dst);
1139
1140          /*the SPE must get the data from the primary buffer*/
1141          spe_signal_write(ps_SPEContext[spe_sid].context,
                  SPE_SIG_NOTIFY_REG_2, BUFFER1);
1142        }
1143
```

```
1144
1145        if(request_type==STORE_AUX) {
1146          //does the aux element exist? if yes, then lock it. if
                no,
1147          //then create the element and then lock it!
1148          if(em.ExistElementByAuxIndex(_e_src.aux, _e_src.
                iteration_owner)==FALSE) {
1149            em.InsertElement(_e_src.iteration_owner, _e_src);
1150          }
1151
1152          em.WaitAndEnableExclusive(_e_src, _e_src.io_array_type);
1153
1154          _e_dst.iteration_owner = em.GetElementOwner(_e_tmp,
                _e_tmp.io_array_type);
1155          if(em.ExistElementByAuxIndex(_e_dst.aux, _e_dst.
                iteration_owner)==FALSE) {
1156            em.InsertElement(_e_dst.iteration_owner, _e_dst);
1157          }
1158
1159          em.WaitAndEnableExclusive(_e_dst, _e_dst.io_array_type);
1160
1161          if(_e_dst.load==TRUE) {
1162            if(_e_src.iteration_owner < _e_dst.last_load) {
1163              //violation
1164              printf("Violation␣-␣(d)Iteration␣%i␣<----␣(s)
                    Iteration␣%i\n",(_e_dst.iteration_owner), _e_dst.
                    last_load);
1165
1166              em.WaitAndDisableExclusive(_e_src, _e_src.
                    io_array_type);
1167              em.WaitAndDisableExclusive(_e_dst, _e_dst.
                    io_array_type);
1168
1169              i_ResolveScalarRequestFlag=FALSE;
1170
1171              s_Recovery.element_data_from = _e_src;
1172              s_Recovery.element_to_update = _e_dst;
```

```
1173
1174              exec_stop = ___mftb();
1175              kern.InsertMetricResult(THREAD_RRS, exec_start,
                      exec_stop);
1176
1177              _RecoveryQueue.push_back(s_Recovery);
1178
1179
1180            }
1181            else {
1182              _e_dst.store = TRUE;
1183              _e_dst.last_store = _e_src.iteration_owner;
1184              em.ReinsertElement(_e_dst.iteration_owner, _e_dst);
1185
1186              spe_signal_write(ps_SPEContext[spe_sid].context,
                      SPE_SIG_NOTIFY_REG_2, BUFFER1);
1187            }
1188          }
1189
1190          if(_e_dst.load!=TRUE) {
1191            _e_dst.store = TRUE;
1192            _e_dst.last_store = _e_src.iteration_owner;
1193            em.ReinsertElement(_e_dst.iteration_owner, _e_dst);
1194          }
1195
1196          /*the SPE must get the data from the primary buffer*/
1197          spe_signal_write(ps_SPEContext[spe_sid].context,
                SPE_SIG_NOTIFY_REG_2, BUFFER1);
1198        }
1199
1200
1201        if(request_type==LOAD) {
1202          _e_dst.load = TRUE;
1203          _e_dst.last_load=_e_src.iteration_owner;
1204          em.ReinsertElement(_e_dst.iteration_owner, _e_dst);
1205
1206          /*tell the SPE who created this request to go and get
```

```
            the data.
1207        once the SPE has completed the transfer, it must report
                back
1208        and tell the PPE. the PPE can then set the exclusive
                property
1209        to false for both elements*/
1210
1211        if(_e_dst.store==TRUE) {
1212          /*the SPE must get the data from the secondary buffer
                  */
1213          spe_signal_write(ps_SPEContext[spe_sid].context,
                  SPE_SIG_NOTIFY_REG_2, BUFFER2);
1214        }
1215        else {
1216          /*the SPE must get the data from the primary buffer*/
1217          spe_signal_write(ps_SPEContext[spe_sid].context,
                  SPE_SIG_NOTIFY_REG_2, BUFFER1);
1218        }
1219      }
1220
1221
1222      if(request_type==STORE) {
1223        if(_e_dst.store!=TRUE) {
1224
1225          _e_dst.store = TRUE;
1226          _e_dst.last_store=_e_src.iteration_owner;
1227          em.ReinsertElement(_e_dst.iteration_owner, _e_dst);
1228
1229          if(_e_dst.store==TRUE) {
1230            /*the SPE must store the data from the secondary
                    buffer*/
1231            spe_signal_write(ps_SPEContext[spe_sid].context,
                    SPE_SIG_NOTIFY_REG_2, BUFFER2);
1232          }
1233          else {
1234            /*the SPE must store the data from the primary
                    buffer*/
```

```
1235            spe_signal_write(ps_SPEContext[spe_sid].context,
                    SPE_SIG_NOTIFY_REG_2, BUFFER1);
1236          }
1237        }
1238
1239
1240        /*determine if the owner is younger then the 'needed'
                element. if it is, then
1241        we have a problem, this is a violation*/
1242        if(_e_dst.load==TRUE) {
1243          if(_e_src.iteration_owner < _e_dst.last_load) {
1244            //violation
1245            printf("Violation : Iteration %i\n", (_e_dst.
                    iteration_owner));
1246
1247            em.WaitAndDisableExclusive(_e_src, _e_src.
                    io_array_type);
1248            em.WaitAndDisableExclusive(_e_dst, _e_dst.
                    io_array_type);
1249
1250            i_ResolveScalarRequestFlag=FALSE;
1251
1252            s_Recovery.element_data_from = _e_src;
1253            s_Recovery.element_to_update = _e_dst;
1254
1255            exec_stop = __mftb();
1256            kern.InsertMetricResult(THREAD_RRS, exec_start,
                    exec_stop);
1257            _RecoveryQueue.push_back(s_Recovery);
1258          }
1259
1260          if(_e_src.iteration_owner > _e_dst.last_load) {
1261            /*owner can store its data into needed element*/
1262            _e_dst.store=TRUE;
1263            _e_dst.last_store=_e_src.iteration_owner;
1264            em.ReinsertElement(_e_dst.iteration_owner, _e_dst);
1265
```

```
1266                if(_e_dst.load==TRUE) {
1267                  /*the SPE must get the data from the secondary
                         buffer*/
1268                  spe_signal_write(ps_SPEContext[spe_sid].context,
                         SPE_SIG_NOTIFY_REG_2, BUFFER2);
1269                }
1270                else {
1271                  /*the SPE must get the data from the primary
                         buffer*/
1272                  spe_signal_write(ps_SPEContext[spe_sid].context,
                         SPE_SIG_NOTIFY_REG_2, BUFFER1);
1273                }
1274              }
1275            }
1276          }
1277        }
1278
1279
1280        exec_stop = ___mftb();
1281        kern.InsertMetricResult(THREAD_RRS, exec_start, exec_stop);
1282        THREAD_USLEEP_DEFAULT
1283      }
1284
1285 //   pthread_exit(NULL);
1286 }
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
```

```
1299
1300
1301
1302
1303 //
       ////////////////////////////////////////////////////////////////

1304 ////////////////////// CALLBACK FUNCTIONS
       ///////////////////////
1305 //
       ////////////////////////////////////////////////////////////////

1306
1307 int cf_MeasurementReading(void *ls_base_tmp, unsigned int data)
       {
1308   THREAD_USLEEP_DEFAULT
1309   pthread_mutex_lock(&m_MeasureReading);
1310
1311   char *ls_base = (char *)ls_base_tmp;
1312   spe_offset_t params_offset = *((spe_offset_t *)(ls_base + data
         ));
1313
1314   measure_t *params = (measure_t *)(ls_base + params_offset);
1315
1316   mr.Insert(*params);
1317
1318   pthread_mutex_unlock(&m_MeasureReading);
1319   return 0;
1320 }
1321
1322
1323
1324
1325 int cf_SPEShutdownRequest(void *ls_base_tmp, unsigned int data)
       {
1326   THREAD_USLEEP_DEFAULT
1327   pthread_mutex_lock(&m_SPEShutdown);
```

```
1328
1329    unsigned long long exec_start, exec_stop;
1330    exec_start = __mftb();
1331
1332    char *ls_base = (char *)ls_base_tmp;
1333    spe_offset_t params_offset = *((spe_offset_t *)(ls_base + data
            ));
1334
1335    transmit_report_t *params = (transmit_report_t *)(ls_base +
            params_offset);
1336
1337    for(_SPEStatusListIterator = _SPEStatusList.begin();
            _SPEStatusListIterator != _SPEStatusList.end(); ++
            _SPEStatusListIterator) {
1338      if(_SPEStatusListIterator->spe==params->spe) {
1339        switch(_SPEStatusListIterator->status) {
1340          case SPE_DEPENDENCY:
1341            params->result = SPE_HALT;
1342            _SPEStatusListIterator->status = SPE_HALT;
1343            goto SWITCH_EXIT;
1344            break;
1345
1346          case SPE_NO_DEPENDENCY:
1347            params->result = SPE_SHUTDOWN;
1348            _SPEStatusListIterator->status = SPE_SHUTDOWN;
1349            goto SWITCH_EXIT;
1350            break;
1351
1352          case SPE_SHUTDOWN:
1353            params->result = SPE_SHUTDOWN;
1354            _SPEStatusListIterator->status = SPE_SHUTDOWN;
1355            goto SWITCH_EXIT;
1356            break;
1357
1358          default:
1359            goto SWITCH_EXIT;
1360            break;
```

```
1361         }
1362       }
1363     }
1364
1365     SWITCH_EXIT:
1366
1367     exec_stop = __mftb();
1368     kern.InsertMetricResult(CALLBACK_SR, exec_start, exec_stop);
1369
1370     pthread_mutex_unlock(&m_SPEShutdown);
1371
1372     return 0;
1373 }
1374
1375 int cf_SPERequest(void *ls_base_tmp, unsigned int data) {
1376     THREAD_USLEEP_DEFAULT
1377     pthread_mutex_lock(&m_Insert);
1378
1379     unsigned long long exec_start, exec_stop;
1380     exec_start = __mftb();
1381
1382     char *ls_base = (char *)ls_base_tmp;
1383     spe_offset_t params_offset = *((spe_offset_t *)(ls_base + data
          ));
1384
1385     request_message_t *params = (request_message_t *)(ls_base +
          params_offset);
1386
1387     _RequestList.push_back(*params);
1388
1389     exec_stop = __mftb();
1390     kern.InsertMetricResult(CALLBACK_R, exec_start, exec_stop);
1391
1392     pthread_mutex_unlock(&m_Insert);
1393     return 0;
1394 }
1395
```

```
1396 int cf_SPERequestComplete(void *ls_base_tmp, unsigned int data)
       {
1397   THREAD_USLEEP_DEFAULT
1398   pthread_mutex_lock(&m_RegionComplete);
1399
1400   unsigned long long exec_start, exec_stop;
1401   exec_start = __mftb();
1402
1403   char *ls_base = (char *)ls_base_tmp;
1404   spe_offset_t params_offset = *((spe_offset_t *)(ls_base + data
         ));
1405
1406   request_message_t *params = (request_message_t *)(ls_base +
         params_offset);
1407
1408   element_t _e_tmp, _e_src, _e_dst;
1409
1410   /*Disable Exclusiveness*/
1411   _e_tmp.spe =          params->spe_sid;
1412   _e_tmp.region =         params->region_number;
1413   _e_tmp.request_type =       params->request_type;
1414   _e_tmp.iteration_owner =   params->owner_itr;
1415   _e_tmp.index_a =        params->index_a;
1416   _e_tmp.index_b =        params->index_b;
1417   _e_tmp.loop_level =       params->loop_level;
1418   _e_tmp.level =          params->level;
1419   _e_tmp.io_array_type =      params->io_array_type;
1420
1421   _e_src = _e_tmp;
1422   em.WaitAndDisableExclusive(_e_src, _e_src.io_array_type);
1423
1424   _e_dst = _e_tmp;
1425   _e_dst.iteration_owner = em.GetElementOwner(_e_tmp, _e_tmp.
         io_array_type);
1426   em.WaitAndDisableExclusive(_e_dst, _e_dst.io_array_type);
1427
1428   for(_SPEStatusListIterator = _SPEStatusList.begin();
```

```
        _SPEStatusListIterator != _SPEStatusList.end(); ++
        _SPEStatusListIterator) {
1429      if(_SPEStatusListIterator->spe==params->spe_sid) {
1430
1431        _SPEStatusListIterator->counter-=1;
1432
1433        if(_SPEStatusListIterator->counter==0) {
1434          _SPEStatusListIterator->status=SPE_NO_DEPENDENCY;
1435        }
1436
1437        break;
1438      }
1439    }
1440
1441    exec_stop = __mftb();
1442    kern.InsertMetricResult(CALLBACK_RC, exec_start, exec_stop);
1443
1444    pthread_mutex_unlock(&m_RegionComplete);
1445    return 0;
1446  }
1447
1448  int cf_SPERegionRequest(void *ls_base_tmp, unsigned int data) {
1449    THREAD_USLEEP_DEFAULT
1450    pthread_mutex_lock(&m_RegionParameters);
1451
1452    unsigned int spe, _i_found = FALSE;
1453    unsigned int region;
1454
1455    unsigned long long exec_start, exec_stop;
1456    exec_start = __mftb();
1457
1458    char *ls_base = (char *)ls_base_tmp;
1459    spe_offset_t params_offset = *((spe_offset_t *)(ls_base + data
        ));
1460
1461    context_params_t *params = (context_params_t *)(ls_base +
        params_offset);
```

```
1462
1463    spe = params->spe;
1464    region = params->region;
1465
1466    if(params->spe >= i_ActiveSPECount) {
1467
1468      params->assigned=SPE_HALT;
1469      for(_SPEStatusListIterator = _SPEStatusList.begin();
            _SPEStatusListIterator != _SPEStatusList.end(); ++
            _SPEStatusListIterator) {
1470        if(_SPEStatusListIterator->spe==params->spe) {
1471          _SPEStatusListIterator->status=SPE_STANDBY;
1472          params->assigned=SPE_STANDBY;
1473
1474          break;
1475        }
1476      }
1477    }
1478    else {
1479      params->assigned=SPE_CONTINUE;
1480
1481      /*
1482       * We need to update the region value for the SPE. If the
              SPE
1483       * object has not been created, then create one with new
              data
1484       */
1485      for(_RegionListIterator = _RegionList.begin();
            _RegionListIterator != _RegionList.end(); ++
            _RegionListIterator) {
1486
1487        if(_RegionListIterator->spe == params->spe &&
              _RegionListIterator->connected==TRUE) {
1488
1489          _RegionListIterator->region = params->region;
1490          _RegionListIterator->complete = REGION_PROCESSING;
1491          _RegionListIterator->commit_back = FALSE;
```

```
1492
1493          _i_found = TRUE;
1494          break;
1495        }
1496      }
1497
1498      if(_i_found != TRUE) {
1499        region_complete_t s_RegionComplete;
1500
1501        s_RegionComplete.spe = spe;
1502        s_RegionComplete.region = params->region;
1503        s_RegionComplete.complete = REGION_PROCESSING;
1504
1505        _RegionList.push_back(s_RegionComplete);
1506      }
1507
1508
1509
1510      // Use the work_list STL LIST to locate the right work
             parameters for SPE
1511      for(_WorkListIterator = _WorkList.begin(); _WorkListIterator
           != _WorkList.end(); ++_WorkListIterator) {
1512        if((_WorkListIterator->spe==params->spe) && (
             _WorkListIterator->region==params->region)) {
1513          params->bench =      _WorkListIterator->bench;
1514
1515          params->ea_in =      _WorkListIterator->ea_in;
1516          params->ea_out =     _WorkListIterator->ea_out;
1517
1518          params->ipc_internal_counter = _WorkListIterator->
               ipc_internal_counter;
1519
1520          params->size =       _WorkListIterator->size;
1521          params->size2 =      _WorkListIterator->size2;
1522
1523          params->size_begin =  _WorkListIterator->size_begin;
1524          params->size_end =    _WorkListIterator->size_end;
```

```
1525
1526            params->size_begin_2 =  _WorkListIterator->size_begin_2;
1527            params->size_end_2 =  _WorkListIterator->size_end_2;
1528
1529            params->itr_begin =    _WorkListIterator->itr_begin;
1530            params->itr_end =      _WorkListIterator->itr_end;
1531
1532            params->itr_total =    _WorkListIterator->itr_total;
1533            params->final =      _WorkListIterator->final;
1534
1535            params->io_array_type = _WorkListIterator->io_array_type
                    ;
1536
1537            params->ea_aux =     _WorkListIterator->ea_aux;
1538            params->ea_aux2 =    _WorkListIterator->ea_aux2;
1539            params->ea_aux3 =    _WorkListIterator->ea_aux3;
1540
1541            params->store_array = (unsigned long long)&MonitorArray
                    [0];
1542            break;
1543          }
1544        }
1545    }
1546
1547
1548    exec_stop = __mftb();
1549    kern.InsertMetricResult(CALLBACK_RR, exec_start, exec_stop);
1550    pthread_mutex_unlock(&m_RegionParameters);
1551
1552    return 0;
1553 }
1554
1555 int cf_SPERegionComplete(void *ls_base_tmp, unsigned int data) {
1556    THREAD_USLEEP_DEFAULT
1557    pthread_mutex_lock(&m_RegionComplete);
1558
1559    unsigned long long exec_start, exec_stop;
```

```
1560  exec_start = __mftb();
1561
1562  char *ls_base = (char *)ls_base_tmp;
1563  spe_offset_t params_offset = *((spe_offset_t *)(ls_base + data
          ));
1564
1565  context_params_t *params = (context_params_t *)(ls_base +
          params_offset);
1566
1567  unsigned int _i_fn_counter = 0;
1568
1569  /*
1570   * Put this function contents into a thread. We need to return
            the message
1571   * back to the SPE so it can monitor for signals !
1572   *
1573   * ***CORRECTION***
1574   * This function you placed a pthread_exit(NULL). This is only
            used for a threaded
1575   * function. This is a callback SPE function, you need a '
            return 0;' statement. You
1576   * have done this now.
1577   *
1578   */
1579
1580
1581  /* Once REGION #n == SPE COUNT - It is time to destroy  hash
          table #n. Clean up routine! */
1582
1583
1584  /* Update the current state of SPE = TRUE */
1585  for(_WorkListIterator = _WorkList.begin(); _WorkListIterator
          != _WorkList.end(); ++_WorkListIterator) {
1586    if(_WorkListIterator->spe == params->spe)
1587      _WorkListIterator->completed = params->completed;
1588  }
1589
```

```
1590    for(_RegionListIterator = _RegionList.begin();
            _RegionListIterator != _RegionList.end(); ++
            _RegionListIterator) {
1591      if(_RegionListIterator->region==params->region) {
1592        _RegionListIterator->complete=params->region;
1593        break;
1594      }
1595    }
1596
1597    /* Now determine if REGION #n has been completed by all SPEs ?
           */
1598    for(_WorkListIterator = _WorkList.begin(); _WorkListIterator
           != _WorkList.end(); ++_WorkListIterator) {
1599      if((_WorkListIterator->spe==params->spe) && (
             _WorkListIterator->region==params->region) && (
             _WorkListIterator->completed==params->completed)) {
1600        _i_fn_counter++;
1601        break;
1602      }
1603    }
1604
1605    /*Now, if all regions have completed then we need to send a
           signal to all SPEs to continue*/
1606    _i_fn_counter=0;
1607
1608    for(_WorkListIterator = _WorkList.begin(); _WorkListIterator
           != _WorkList.end(); ++_WorkListIterator) {
1609      if(_WorkListIterator->completed==REGION_COMPLETED) {
1610        ++_i_fn_counter;
1611      }
1612    }
1613
1614    if(_i_fn_counter==i_ActiveSPECount) {
1615      params->assigned=SPE_CONTINUE;
1616
1617      i_ResolveScalarRequestFlag = FALSE;
1618      i_UpdaterFlag = FALSE;
```

```
1619      i_MailboxMonitorFlag = FALSE;
1620
1621      params->assigned=SPE_CONTINUE;
1622
1623      for(unsigned int i = 0; i < i_ActiveSPECount; i++) {
1624 //       printf("SPE %i has finished on region %i\n", i, params->
          region);
1625         if(i!=params->spe)
1626           spe_signal_write(ps_SPEContext[i].context,
                SPE_SIG_NOTIFY_REG_1, SPE_CONTINUE);
1627
1628      }
1629
1630      em.DeleteAllElements();
1631    }
1632    else {
1633      params->assigned=SPE_HALT;
1634    }
1635
1636
1637    exec_stop = __mftb();
1638    kern.InsertMetricResult(CALLBACK_RC1, exec_start, exec_stop);
1639
1640    pthread_mutex_unlock(&m_RegionComplete);
1641
1642    return 0;
1643 }
1644
1645 int cf_SPERegisterR3(void *ls_base_tmp, unsigned int data) {
1646    THREAD_USLEEP_DEFAULT
1647    pthread_mutex_lock(&m_SPESID);
1648
1649    char *ls_base = (char *)ls_base_tmp;
1650    spe_offset_t params_offset = *((spe_offset_t *)(ls_base + data
          ));
1651
1652    spe_info_init_t *params = (spe_info_init_t *)(ls_base +
```

```
        params_offset);
1653
1654    for(_WorkListIterator = _WorkList.begin(); _WorkListIterator
        != _WorkList.end(); ++_WorkListIterator) {
1655      if(_WorkListIterator->spe == params->spe) {
1656        _WorkListIterator->spe_r3 = params->ea_addr[0];
1657        _WorkListIterator->spe_r4 = params->ea_addr[1];
1658        _WorkListIterator->spe_r5 = params->ea_addr[2];
1659
1660        for(_SPEStatusListIterator = _SPEStatusList.begin();
            _SPEStatusListIterator != _SPEStatusList.end(); ++
            _SPEStatusListIterator) {
1661          if(_SPEStatusListIterator->spe == _WorkListIterator->spe
              ) {
1662            _SPEStatusListIterator->r3 = _WorkListIterator->spe_r3
                ;
1663            _SPEStatusListIterator->r4 = _WorkListIterator->spe_r4
                ;
1664            _SPEStatusListIterator->r5 = _WorkListIterator->spe_r5
                ;
1665            _SPEStatusListIterator->status = SPE_NO_DEPENDENCY;
1666
1667            break;
1668          }
1669        }
1670
1671        break;
1672      }
1673    }
1674
1675    pthread_mutex_unlock(&m_SPESID);
1676
1677    return 0;
1678 }
1679
1680 int cf_SPERegisterIPC(void *ls_base_tmp, unsigned int data) {
1681   THREAD_USLEEP_DEFAULT
```

```
1682    pthread_mutex_lock(&m_SPEIPC);
1683
1684    char *ls_base = (char *)ls_base_tmp;
1685    spe_offset_t params_offset = *((spe_offset_t *)(ls_base + data
            ));
1686
1687    ipc_info_t *params = (ipc_info_t *)(ls_base + params_offset);
1688
1689    for(_SPEStatusListIterator = _SPEStatusList.begin();
            _SPEStatusListIterator != _SPEStatusList.end(); ++
            _SPEStatusListIterator) {
1690      if(_SPEStatusListIterator->spe==params->spe) {
1691        _SPEStatusListIterator->ipc = params->ipc_addr;
1692        break;
1693      }
1694    }
1695
1696    pthread_mutex_unlock(&m_SPEIPC);
1697
1698    return 0;
1699 }
1700
1701 //int cf_SPERegionCompleteSignal(void *ls_base_tmp, unsigned int
        data) {
1702 //  THREAD_USLEEP_DEFAULT
1703 //  pthread_mutex_lock(&m_SPESignal);
1704 //
1705 //  char *ls_base = (char *)ls_base_tmp;
1706 //  spe_offset_t params_offset = *((spe_offset_t *)(ls_base +
        data));
1707 //
1708 //  spe_sig_ob_t *params = (spe_sig_ob_t *)(ls_base +
        params_offset);
1709 //
1710 //  if(params->send_all==FALSE){
1711 //    spe_signal_write(ps_SPEContext[params->spe].context,
        SPE_SIG_NOTIFY_REG_2, SPE_CONTINUE);
```

```
1712 //   }
1713 ////
1714 ////   if(params->send_all==TRUE){
1715 ////     for(unsigned i = 0; i < i_ActiveSPECount; i++) {
1716 ////       spe_signal_write(ps_SPEContext[i].context,
         SPE_SIG_NOTIFY_REG_2, SPE_CONTINUE);
1717 ////     }
1718 ////   }
1719 //
1720 //   pthread_mutex_unlock(&m_SPESignal);
1721 //
1722 //   return 0;
1723 //}
1724 int _i_LoopOuterCounter=0;
1725 int _i_LoopSyncInnerCounter=0;
1726
1727 int cf_SPERegionCompleteSignal(void *ls_base_tmp, unsigned int
       data) {
1728   THREAD_USLEEP_DEFAULT
1729   pthread_mutex_lock(&m_SPESignal);
1730
1731   unsigned long long exec_start, exec_stop;
1732   exec_start = __mftb();
1733
1734   char *ls_base = (char *)ls_base_tmp;
1735   spe_offset_t params_offset = *((spe_offset_t *)(ls_base + data
         ));
1736
1737   spe_sig_ob_t *params = (spe_sig_ob_t *)(ls_base +
         params_offset);
1738
1739   if(params->loop_level==OUTER) {
1740     ++_i_LoopOuterCounter;
1741     i_ElementAnalyserFlag[params->spe]=THREAD_PAUSE;
1742
1743     THREAD_USLEEP_DEFAULT
1744     i_ElementAnalyserFlag[params->spe]=THREAD_EXIT;
```

```
1745
1746      for(_WorkListIterator = _WorkList.begin(); _WorkListIterator
              != _WorkList.end(); ++_WorkListIterator) {
1747        if(_WorkListIterator->spe==params->spe) {
1748          _WorkListIterator->completed=REGION_PROCESSING;
1749        }
1750      }
1751
1752
1753
1754      if(_i_LoopOuterCounter==i_ActiveSPECount) {
1755        for(_WorkListIterator = _WorkList.begin();
               _WorkListIterator != _WorkList.end(); ++
               _WorkListIterator) {
1756          if(_WorkListIterator->spe==params->spe) {
1757            _WorkListIterator->completed=REGION_COMPLETED;
1758          }
1759        }
1760
1761        _i_LoopOuterCounter=0;
1762
1763        params->id = SPE_CONTINUE;
1764
1765        for(unsigned i = 0; i < i_ActiveSPECount; i++) {
1766          if(params->spe != i)
1767            spe_signal_write(ps_SPEContext[i].context,
                   SPE_SIG_NOTIFY_REG_2, SPE_CONTINUE);
1768        }
1769      }
1770    }
1771
1772    if(params->loop_level==INNER) {
1773      ++_i_LoopSyncInnerCounter;
1774
1775      if(_i_LoopSyncInnerCounter==i_ActiveSPECount) {
1776        _i_LoopSyncInnerCounter=0;
1777
```

```
1778        params->id = SPE_CONTINUE;
1779        for(unsigned i = 0; i < i_ActiveSPECount; i++) {
1780          if(params->spe != i)
1781            spe_signal_write(ps_SPEContext[i].context,
                  SPE_SIG_NOTIFY_REG_2, SPE_CONTINUE);
1782        }
1783      }
1784    }
1785
1786    exec_stop = __mftb();
1787    kern.InsertMetricResult(CALLBACK_RCS, exec_start, exec_stop);
1788
1789    pthread_mutex_unlock(&m_SPESignal);
1790
1791    return 0;
1792  }
1793
1794  /*This callback function will register, deregister and copy back
        loop data*/
1795  int cf_SPERegisterLoop(void *ls_base_tmp, unsigned int data) {
1796    THREAD_USLEEP_DEFAULT
1797    pthread_mutex_lock(&m_LoopRegion);
1798
1799    unsigned long long exec_start, exec_stop;
1800    exec_start = __mftb();
1801
1802    char *ls_base = (char *)ls_base_tmp;
1803    spe_offset_t params_offset = *((spe_offset_t *)(ls_base + data
        ));
1804
1805    loop_t *params = (loop_t *)(ls_base + params_offset);
1806
1807    int _found = FALSE;
1808    int _pos_start, _pos_end;
1809
1810    if(params->type==REGISTER) {
1811
```

```
1812      //If params already exists then update the current
              parameters, otherwise add params to list
1813      __CF_SPEREGISTERLOOP_FORCEREGISTER:
1814      for(_LoopRegionListIterator = _LoopRegionList.begin();
              _LoopRegionListIterator != _LoopRegionList.end(); ++
              _LoopRegionListIterator) {
1815
1816        if(_LoopRegionListIterator->spe == params->spe &&
               _LoopRegionListIterator->id == params->id) {
1817          _found = TRUE;
1818
1819          _LoopRegionListIterator->pos_start = params->pos_start;
1820          _LoopRegionListIterator->pos_end = params->pos_end;
1821
1822          ps_MonitorParameters[params->spe].spe = params->spe;
1823          ps_MonitorParameters[params->spe].begin = params->
                  pos_start;
1824          ps_MonitorParameters[params->spe].end = params->pos_end;
1825
1826          i_ElementAnalyserFlag[params->spe] = TRUE;
1827
1828          if(pthread_create(&pt_ElementAnalyser[params->spe], NULL
                  , ptf_SPEDataOutputViolationAnalyser, &
                  ps_MonitorParameters[params->spe])) {
1829            perror("pthread_create");
1830            exit(1);
1831          }
1832
1833          break;
1834        }
1835      }
1836
1837      if(_found==FALSE) {
1838        params->done=FALSE;
1839        _LoopRegionList.push_front(*params);
1840
1841  //        goto __CF_SPEREGISTERLOOP_FORCEREGISTER;
```

```
1842      }
1843
1844      if(params->spe==0) {
1845        s_Report.spe = params->spe;
1846        s_Report.loop = params->level;
1847        i_UpdaterFlag=TRUE;
1848      }
1849
1850      if(i_UpdaterFlag==THREAD_PAUSE)
1851        i_UpdaterFlag=THREAD_RESTART;
1852
1853      i_MailboxMonitorFlag=TRUE;
1854    }
1855
1856    if(params->type==DEREGISTER) {
1857      for(_LoopRegionListIterator = _LoopRegionList.begin();
              _LoopRegionListIterator != _LoopRegionList.end(); ++
              _LoopRegionListIterator) {
1858
1859        if(_LoopRegionListIterator->spe == params->spe &&
              _LoopRegionListIterator->id == params->id) {
1860
1861          _LoopRegionListIterator->done=TRUE;
1862
1863          _pos_start = _LoopRegionListIterator->pos_start;
1864          _pos_end = _LoopRegionListIterator->pos_end;
1865
1866          pthread_mutex_lock(&m_MonitorArrayMonitor);
1867          for(int i = _pos_start; i < _pos_end; i++) {
1868            MonitorArray[i] = NO_STORE;
1869          }
1870          pthread_mutex_unlock(&m_MonitorArrayMonitor);
1871
1872          _LoopRegionList.erase(_LoopRegionListIterator);
1873
1874          i_ElementAnalyserFlag[params->spe] = THREAD_PAUSE;
1875
```

```
1876 //          THREAD_USLEEP_DEFAULT
1877           i_ElementAnalyserFlag[params->spe] = THREAD_EXIT;
1878
1879 //          THREAD_USLEEP_DEFAULT
1880 //          pthread_cancel(pt_ElementAnalyser[params->spe]);
1881 //          pthread_join(pt_ElementAnalyser[params->spe], NULL);
1882           break;
1883         }
1884       }
1885
1886 //      i_UpdaterFlag=THREAD_PAUSE;
1887 //      i_MailboxMonitorFlag=FALSE;
1888
1889       unsigned int __i_counter = 0;
1890
1891       for(_LoopRegionListIterator = _LoopRegionList.begin();
              _LoopRegionListIterator != _LoopRegionList.end(); ++
              _LoopRegionListIterator) {
1892         if(_LoopRegionListIterator->level==OUTER) {
1893           ++__i_counter;
1894         }
1895       }
1896
1897       if(__i_counter==i_ActiveSPECount) {
1898         if(_LoopRegionListIterator->level==OUTER) {
1899           //Delete
1900           em.RemoveElementByLoopRange(params->outer_level);
1901         }
1902       }
1903     }
1904
1905     if(params->type==COPY_BACK) {
1906       for(_LoopRegionListIterator = _LoopRegionList.begin();
              _LoopRegionListIterator != _LoopRegionList.end(); ++
              _LoopRegionListIterator) {
1907         if(_LoopRegionListIterator->spe == params->spe &&
              _LoopRegionListIterator->id == params->id) {
```

```
1908            params->pos_start = _LoopRegionListIterator->pos_start;
1909            params->pos_end = _LoopRegionListIterator->pos_end;
1910
1911            break;
1912          }
1913        }
1914
1915        if(params->spe==0) {
1916          i_UpdaterFlag=THREAD_RESTART;
1917          i_MailboxMonitorFlag=TRUE;
1918        }
1919      }
1920
1921    ___THREAD_NOW_EXIT:
1922      exec_stop = __mftb();
1923      kern.InsertMetricResult(CALLBACK_RL, exec_start, exec_stop);
1924      pthread_mutex_unlock(&m_LoopRegion);
1925
1926      return 0;
1927  }
```

B.6 main.cpp

```
1 #include "kernel.hpp"
2
3 int main(int argc, char *argv[]) {
4
5   ////////////// BENCHMARK PARAMETERS ////////////
6 //     int FFT_size = FFT_SIZE;  //TINY_FFT_SIZE
7
8     int SOR_size =  SOR_SIZE;      //TINY_SOR_SIZE    SOR_SIZE
9
10    int Sparse_size_M = SPARSE_SIZE_M;     //TINY_SPARSE_SIZE_M
          SPARSE_SIZE_M
11    int Sparse_size_nz = SPARSE_SIZE_nz;  //TINY_SPARSE_SIZE_nz
          SPARSE_SIZE_nz
```

```
12
13      int LU_size = LG_LU_SIZE;              //TINY_LU_SIZE   LU_SIZE
14
15      Random R = new_Random_seed(RANDOM_SEED);
16    ////////////// ^ BENCHMARK PARAMETERS ^ //////////////
17
18
19
20
21
22
23
24
25
26
27
28    /*FFT benchmark*/
29 //   SingleArrayInput = (double *)RandomVector(2*FFT_size, R);
30 //   SingleArrayOutput = (double *)malloc(2*FFT_size*sizeof(float
       ));
31 //
32 //   MonitorSize(2*FFT_size);
33 //
34 //   BenchmarkAttribute.bench_name      = FFT;
35 //   BenchmarkAttribute.bench_id      = 0;
36 //   BenchmarkAttribute.total_iterations = fft_only_int_log2(2*
       FFT_size);
37 //   BenchmarkAttribute.size_1_dataset    = FFT_size;
38 //   BenchmarkAttribute.aux          = (unsigned long long)
       SingleArrayInput;
39 //   BenchmarkAttribute.aux2         = (unsigned long long)
       SingleArrayOutput;
40 //   BenchmarkAttribute.input        = (unsigned long long)
       SingleArrayInput;
41 //   BenchmarkAttribute.output         = (unsigned long long)
       SingleArrayOutput;
42 //   BenchmarkAttribute.io_array_type  = SINGLE_ARRAY;
```

```
43 //
44 //   InsertBenchmark(BenchmarkAttribute);
45
46
47
48
49
50
51
52
53
54
55   /*SOR benchmark*/
56 //   DoubleArrayInput = RandomMatrix(SOR_size, SOR_size, R);
57 //   DoubleArrayOutput = (double **) malloc(sizeof(double*)*
     SOR_size);
58 //
59 //   for(int i=0; i<SOR_size; i++) {
60 //     DoubleArrayOutput[i] = (double *) malloc(sizeof(double)*
     SOR_size);
61 //   }
62 //
63 //   MonitorSize(SOR_size);
64 //
65 //   BenchmarkAttribute.bench_name      = SOR;
66 //   BenchmarkAttribute.bench_id        = 1;
67 //   BenchmarkAttribute.total_iterations = 1;
68 //   BenchmarkAttribute.size_1_dataset   = SOR_size;
69 //   BenchmarkAttribute.size_2_dataset   = SOR_size;
70 //   BenchmarkAttribute.input       = (unsigned long long)&
     DoubleArrayInput;
71 //   BenchmarkAttribute.output        = (unsigned long long)&
     DoubleArrayOutput;
72 //   BenchmarkAttribute.io_array_type  = DOUBLE_ARRAY;
73 //
74 //   InsertBenchmark(BenchmarkAttribute);
75
```

```
76
77
78
79
80
81
82
83
84
85    /*Sparse benchmark*/
86    int N = Sparse_size_M;
87    int nz = Sparse_size_nz;
88
89    SingleArrayInput = RandomVector(N, R);
90      SingleArrayOutput = (double*) malloc(sizeof(double)*N);
91
92    MonitorSize(N);
93
94 #if 0
95     /*initialize square sparse matrix
96
97           for this test, we create a sparse matrix with M/nz
                nonzeros
98           per row, with spaced-out evenly between the begining of
                 the
99           row to the main diagonal.  Thus, the resulting pattern
                looks
100          like
101                      +--------------------+
102                      +*                   +
103                      +***                 +
104                      +* * *               +
105                      +** *  *             +
106                      +**  *   *           +
107                      +* *   *   *         +
108                      +*  *   *    *       +
109                      +*   *    *    *  +
```

```
110                         +-----------------+
111
112           (as best reproducible with integer artihmetic)
113           Note that the first nr rows will have elements past
114           the diagonal.*/
115 #endif
116   int nr = nz/N;       /* average number of nonzeros per row  */
117     int anz = nr*N;     /* _actual_ number of nonzeros          */
118
119
120     val = RandomVector(anz, R);
121     col = (int*) malloc(sizeof(int)*nz);
122     row = (int*) malloc(sizeof(int)*(N+1));
123     int r=0;
124
125     row[0] = 0;
126     for(r=0; r<N; r++) {
127         /* initialize elements for row r */
128         int rowr = row[r];
129         int step = r/ nr;
130         int i=0;
131
132         row[r+1] = rowr + nr;
133         if (step < 1) step = 1;   /* take at least unit steps */
134
135
136         for (i=0; i<nr; i++)
137             col[rowr+i] = i*step;
138     }
139
140
141   BenchmarkAttribute.bench_name    = SPARSE;
142   BenchmarkAttribute.bench_id      = 2;
143   BenchmarkAttribute.total_iterations = 1;
144   BenchmarkAttribute.size_1_dataset    = N;
145   BenchmarkAttribute.aux          = (unsigned long long)val;
146   BenchmarkAttribute.aux2         = (unsigned long long)row;
```

```
147   BenchmarkAttribute.aux3        = (unsigned long long)col;
148   BenchmarkAttribute.input       = (unsigned long long)
          SingleArrayInput;
149   BenchmarkAttribute.output        = (unsigned long long)
          SingleArrayOutput;
150   BenchmarkAttribute.io_array_type  = SINGLE_ARRAY;
151
152   InsertBenchmark(BenchmarkAttribute);
153
154
155
156
157
158
159
160
161
162   /*MeasureLU benchmark*/
163 //   int N = LU_size;
164 //   double **A = NULL;
165 //     double **lu = NULL;
166 //     int *pivot = NULL;
167 //
168 //   if((A = RandomMatrix(N, N,   R)) == NULL)  exit(1);
169 //     if((lu = new_Array2D_double(N, N)) == NULL)  exit(1);
170 //     pivot = (int *)malloc(N * sizeof(int));
171 //
172 //     if(pivot==NULL) {
173 //       printf("problem\n");
174 //       exit(1);
175 //     }
176 //
177 //
178 //   MonitorSize(N);
179 //
180 //
181 //   BenchmarkAttribute.bench_name    = LU;
```

```
182 //   BenchmarkAttribute.total_iterations = 1;
183 //   BenchmarkAttribute.size_1_dataset   = LU_size;
184 //   BenchmarkAttribute.aux          = (unsigned long long)&A
        [0][0];
185 //   BenchmarkAttribute.aux2         = (unsigned long long)&lu
        [0][0];
186 //   BenchmarkAttribute.aux3         = (unsigned long long)pivot;
187 ////   BenchmarkAttribute.input        = (unsigned long long)&
        DoubleArrayInput[0][0];
188 ////   BenchmarkAttribute.output         = (unsigned long long)&
        DoubleArrayOutput[0][0];
189 //   BenchmarkAttribute.input        = (unsigned long long)
        SingleArrayInput;
190 //   BenchmarkAttribute.output         = (unsigned long long)
        SingleArrayOutput;
191
192
193   //Array2D - Array2D_double_copy(N, N, lu, A);
194 //   BenchmarkAttribute.bench_id       = 3;
195 //   BenchmarkAttribute.io_array_type  = SINGLE_ARRAY;
196 //   InsertBenchmark(BenchmarkAttribute);
197
198   //LU - LU_factor(N, N, lu, pivot); - NOT WORKING
199 //   BenchmarkAttribute.bench_id       = 4;
200 //   BenchmarkAttribute.io_array_type  = DOUBLE_ARRAY;
201 //   InsertBenchmark(BenchmarkAttribute);
202
203   //LU - LU_copy_matrix()
204 //   BenchmarkAttribute.bench_id       = 5;
205 //   BenchmarkAttribute.io_array_type  = DOUBLE_ARRAY;
206 //   InsertBenchmark(BenchmarkAttribute);
207
208
209   /*Run*/
210   SystemRun;
211
212
```

```
213
214     BenchmarkReport(LOAD_E);
215     BenchmarkReport(STORE_E);
216     BenchmarkReport(MAILBOX_INTERRUPT);
217
218
219     BenchmarkReport(THREAD_DOVA);
220     BenchmarkReport(THREAD_R);
221     BenchmarkReport(THREAD_ASC);
222     BenchmarkReport(THREAD_MM);
223     BenchmarkReport(THREAD_RRS);
224
225
226     BenchmarkReport(CALLBACK_SR);
227     BenchmarkReport(CALLBACK_R);
228     BenchmarkReport(CALLBACK_RC);
229     BenchmarkReport(CALLBACK_RR);
230     BenchmarkReport(CALLBACK_RC1);
231     BenchmarkReport(CALLBACK_RCS);
232     BenchmarkReport(CALLBACK_RL);
233
234     BenchmarkReport(PROGRAM_COMPLETE_EXECUTION);
235     return 0;
236 }
```

B.7 random.cpp

```
1 #include <stdlib.h>
2
3 #include "random.hpp"
4
5 #ifndef NULL
6 #define NULL 0
7 #endif
8
9
```

```
10   /* static const int mdig = 32; */
11 #define MDIG 32
12
13   /* static const int one = 1; */
14 #define ONE 1
15
16   static const int m1 = (ONE << (MDIG-2)) + ((ONE << (MDIG-2) )-
       ONE);
17   static const int m2 = ONE << MDIG/2;
18
19   /* For mdig = 32 : m1 =          2147483647, m2 =       65536
20      For mdig = 64 : m1 = 9223372036854775807, m2 = 4294967296
21   */
22
23                                  /* move to initialize() because
                                       */
24                                  /* compiler could not resolve as
                                       */
25                                  /*   a constant.
                                                    */
26
27   static /*const*/ double dm1;  /*  = 1.0 / (double) m1; */
28
29
30 /* private methods (defined below, but not in Random.h */
31
32 static void initialize(Random R, int seed);
33
34 Random new_Random_seed(int seed)
35 {
36     Random R = (Random) malloc(sizeof(Random_struct));
37
38     initialize(R, seed);
39     R->left = 0.0;
40     R->right = 1.0;
41     R->width = 1.0;
42     R->haveRange = 0 /*false*/;
```

```
43
44      return R;
45  }
46
47  Random new_Random(int seed, double left, double right)
48  {
49      Random R = (Random) malloc(sizeof(Random_struct));
50
51      initialize(R, seed);
52      R->left = left;
53      R->right = right;
54      R->width = right - left;
55      R->haveRange = 1;           /* true */
56
57      return R;
58  }
59
60  void Random_delete(Random R)
61  {
62      free(R);
63  }
64
65
66
67  /* Returns the next random number in the sequence.  */
68
69  double Random_nextDouble(Random R)
70  {
71      int k;
72
73      int I = R->i;
74      int J = R->j;
75      int *m = R->m;
76
77      k = m[I] - m[J];
78      if (k < 0) k += m1;
79      R->m[J] = k;
```

```
    if (I == 0)
        I = 16;
    else I--;
    R->i = I;

    if (J == 0)
        J = 16 ;
    else J--;
    R->j = J;

    if (R->haveRange)
        return  R->left +  dm1 * (double) k * R->width;
    else
        return dm1 * (double) k;

}




/*
    ------------------------------------------------------------------------------
                                PRIVATE METHODS
    ------------------------------------------------------------------------------
        */

static void initialize(Random R, int seed)
{

    int jseed, k0, k1, j0, j1, iloop;

    dm1  = 1.0 / (double) m1;

    R->seed = seed;

```

```
114     if (seed < 0 ) seed = -seed;              /* seed = abs(seed)
            */
115     jseed = (seed < m1 ? seed : m1);          /* jseed = min(seed,
            m1) */
116     if (jseed % 2 == 0) --jseed;
117     k0 = 9069 % m2;
118     k1 = 9069 / m2;
119     j0 = jseed % m2;
120     j1 = jseed / m2;
121     for (iloop = 0; iloop < 17; ++iloop)
122     {
123         jseed = j0 * k0;
124         j1 = (jseed / m2 + j0 * k1 + j1 * k0) % (m2 / 2);
125         j0 = jseed % m2;
126         R->m[iloop] = j0 + m2 * j1;
127     }
128     R->i = 4;
129     R->j = 16;
130
131 }
132
133 double *RandomVector(int N, Random R)
134 {
135     int i;
136     double *x = (double *) malloc(sizeof(double)*N);
137
138     for (i=0; i<N; i++)
139         x[i] = Random_nextDouble(R);
140
141     return x;
142 }
143
144
145 double **RandomMatrix(int M, int N, Random R)
146 {
147     int i;
148     int j;
```

```
149
150     /* allocate matrix */
151
152     double **A = (double **) malloc(sizeof(double*)*M);
153
154     if (A == NULL) return NULL;
155
156     for (i=0; i<M; i++)
157     {
158         A[i] = (double *) malloc(sizeof(double)*N);
159         if (A[i] == NULL)
160         {
161             free(A);
162             return NULL;
163         }
164         for (j=0; j<N; j++)
165             A[i][j] = Random_nextDouble(R);
166     }
167     return A;
168 }
```

Appendix C

L-API SPE Code

C.1 main.c

```
1 #include "kernel.h"
2 #include "sor.h"
3 #include "sparse.h"
4 #include "lu.h"
5 #include "array.h"
6 #include "fft.h"
7
8
9
10 int main(unsigned long long r3, unsigned long long r4, unsigned
     long long r5) {
11   StartSystem(r3, r4, r5);
12
13 /*************************************/
14   /*Work in progress*/
15 //   Region(0);
16 //   FFT_transform_internal(-1);
17 //   RegionEnd;
18 /*************************************/
19
```

```
20
21
22
23     /*Working*/
24 //    Region(1)
25 //    SOR_execute();
26 //    RegionEnd
27
28     /*Working*/
29     Region(2)
30     SparseCompRow_matmult();
31     RegionEnd
32
33
34     /*Working*/
35 //    Region(3);
36 //    Array2D_double_copy();
37 //    RegionEnd;
38
39     /*Not  Working*/
40 //    Region(4);
41 //    LU_factor();
42 //    RegionEnd;
43
44     /*Working*/
45 //    Region(5);
46 //    LU_copy_matrix();
47 //    RegionEnd;
48
49     ShutdownRequest;
50     ShutdownReturn;
51 }
```

C.2 kernel.h

```
1 #ifndef _KERNEL_H_
```

```
 2 #define _KERNEL_H_
 3
 4 /*C header files*/
 5 #include <stdlib.h>
 6 #include <stdio.h>
 7 #include <math.h>
 8 #include <string.h>
 9 #include <stdint.h>
10 #include <sys/types.h>
11 #include "malloc.h"
12
13 /*SPU I/O header file*/
14 #include <spu_mfcio.h>
15
16 /*SPU math and logic header file*/
17 #include <spu_intrinsics.h>
18 #include <massv.h>
19 #include <simdmath.h>
20 #include <libmisc.h>
21
22 /*SPU Profile*/
23 #include "profile.h"
24
25 /*Custom header files*/
26 #include "../D_PPU/common.hpp"
27
28
29 /*--------------------------------------------------------
30                         CACHE STUFF
31  * --------------------------------------------------------*/
32
33
34 #define CACHE_NAME          ___CACHE_DOUBLE
35 #define CACHED_TYPE         double
36 #define CACHE_TYPE          1 /* r/w */
37 #define CACHELINE_LOG2SIZE  7 /* 128 bytes */
38 #define CACHE_LOG2NWAY      2 /* 4-way */
```

```
39 #define CACHE_LOG2NSETS    6 /* 32 lines */
40 #define CACHE_READ_X4
41 #include <cache-api.h>
42
43
44 #define CACHE_NAME         ___CACHE_INT
45 #define CACHED_TYPE        int
46 #define CACHE_TYPE         1 /* r/w */
47 #define CACHELINE_LOG2SIZE  7 /* 128 bytes */
48 #define CACHE_LOG2NWAY     2 /* 4-way */
49 #define CACHE_LOG2NSETS    6 /* 32 lines */
50 #define CACHE_READ_X4
51 #include <cache-api.h>
52
53 #define CACHE_NAME         ___CACHE_MONITOR
54 #define CACHED_TYPE        int
55 #define CACHE_TYPE         1 /* r/w */
56 #define CACHELINE_LOG2SIZE  7 /* 128 bytes */
57 #define CACHE_LOG2NWAY     2 /* 4-way */
58 #define CACHE_LOG2NSETS    6 /* 32 lines */
59 #define CACHE_READ_X4
60 #include <cache-api.h>
61
62
63
64
65 #define CACHE_RW_PTR(name, addr)                          (cache_rw(
       name, addr))
66 #define CACHE_LOAD(name, addr)                            (cache_rd(
       name, addr))
67 #define CACHE_STORE(name, addr, val)                        (cache_wr(
       name, addr, val))
68 #define CACHE_FLUSH(name)                                (cache_flush(
       name))
69
70 #define DATA_LOAD_PRIMARY(cache, idx)                       (
       CACHE_LOAD(cache, _io_InputAddress+(idx*sizeof(double))))
```

```
71 #define DATA_STORE_PRIMARY(cache, idx, data)                (
       CACHE_STORE(cache, _io_InputAddress+(idx*sizeof(double)),
       data))
72
73 #define DATA_LOAD_SECONDARY(cache, idx)                     (
       CACHE_LOAD(cache, _io_OutputAddress+(idx*sizeof(double))))
74 #define DATA_STORE_SECONDARY(cache, idx, data)              (
       CACHE_STORE(cache, _io_OutputAddress+(idx*sizeof(double)),
       data))
75
76 #define DATA_PTR_PRIMARY(cache, idx)                        (
       CACHE_RW_PTR(cache, _io_InputAddress+(idx*sizeof(double))))
77 #define DATA_PTR_SECONDARY(cache, idx)                      (
       CACHE_RW_PTR(cache, _io_OutputAddress+(idx*sizeof(double))))
78
79 #define DATA_PTR_DA_PRIMARY(cache, idxA, idxB)              (
       CACHE_RW_PTR(cache, _io_InputAddress+sizeof(double)*(idxA*2 +
         idxB)))
80 #define DATA_PTR_DA_SECONDARY(cache, idxA, idxB)            (
       CACHE_RW_PTR(cache, _io_InputAddress+sizeof(double)*(idxA*2 +
         idxB)))
81
82
83
84
85
86 //EXPERIMENTAL ////////////////////////////////
87 #define DATA_LOAD_SINGLE_AUX1(cache, idx, type)             (
       CACHE_LOAD(cache, _io_AuxAddress1+(idx*sizeof(type))))
88 #define DATA_LOAD_DOUBLE_AUX1(cache, idxA, idxB, type)      (
       CACHE_LOAD(cache, _io_AuxAddress1+sizeof(type)*(idxA*2 + idxB
       )))
89
90 #define DATA_LOAD_SINGLE_AUX2(cache, idx, type)             (
       CACHE_LOAD(cache, _io_AuxAddress2+(idx*sizeof(type))))
91 #define DATA_LOAD_DOUBLE_AUX2(cache, idxA, idxB, type)      (
       CACHE_LOAD(cache, _io_AuxAddress2+sizeof(type)*(idxA*2 + idxB
```

```
    )))
92
93 #define DATA_LOAD_SINGLE_AUX3(cache, idx, type)              (
    CACHE_LOAD(cache, _io_AuxAddress3+(idx*sizeof(type))))
94 #define DATA_LOAD_DOUBLE_AUX3(cache, idxA, idxB, type)       (
    CACHE_LOAD(cache, _io_AuxAddress3+sizeof(type)*(idxA*2 + idxB
    )))
95
96
97
98
99 #define DATA_LOAD_SINGLE_AUX(cache, idx, addr, type)         (
    CACHE_LOAD(cache, addr+(idx*sizeof(type))))
100 #define DATA_LOAD_DOUBLE_AUX(cache, idxA, idxB, addr, type)
        (CACHE_LOAD(cache, addr+sizeof(type)*(idxA*2 + idxB)))
101
102 #define DATA_STORE_SINGLE_AUX(cache, idx, addr, type, data)
        (CACHE_STORE(cache, addr+(idx*sizeof(type)), data))
103 #define DATA_STORE_DOUBLE_AUX(cache, idxA, idxB, addr, type,
    data)    (CACHE_STORE(cache, addr+sizeof(type)*(idxA*2 + idxB
    ), data))
104
105 #define DATA_PTR_SINGLE_AUX(cache, idx, addr, type)          (
    CACHE_RW_PTR(cache, addr+(idx*sizeof(type))))
106 #define DATA_PTR_DOUBLE_AUX(cache, idxA, idxB, addr, type)
        (CACHE_RW_PTR(cache, addr+sizeof(type)*(idxA*2 + idxB)))
107
108
109
110
111 #define DATA_MONITOR_STORE(cache, data, idx)                 (
    CACHE_STORE(cache, _io_MonitorArrayAddress+(idx*sizeof(int)),
     data))
112 #define DATA_MONITOR_LOAD(cache, idx)                        (
    CACHE_LOAD(cache, _io_MonitorArrayAddress+(idx*sizeof(int))))
113
114 #define UPDATE_STORE_VALUE(tag, val)                         (
```

```
      f_UpdateMonitorArray(tag, val))
115 #define UPDATE_PROCESSED_VALUE(val)                    (
      f_UpdateProcessedIPC(val))
116
117 /*Non-compiler optimised variables*/
118 volatile int _i_LoopIdentifier;
119 volatile int _i_RegionIdentifier;
120 volatile int _i_ActiveOuterLoopIdentifier;
121 volatile int _i_ActiveInnerLoopIdentifier;
122
123 unsigned _i_LoopStart;
124 unsigned _i_LoopEnd;
125 unsigned _i_ActiveLoopLevel;
126 unsigned _i_GlobalIOType;
127
128 int _i_ValueToUpdateInMonitorArray;
129
130 unsigned long exec_start, exec_end;
131
132 /*#define's*/
133 #define StartSystem(r3, r4, r5)   spu_write_decrementer(0);(
      _i_SPERegister = r3); (_i_ARGPRegister = r4); (
      _i_ENVPRegister = r5); (f_SystemInitialise())
134 #define ActiveRegion          (_i_RegionIdentifier)
135
136 #define Region(x)             (_i_RegionIdentifier = x); (
      f_GetRegionParameters(_i_RegionIdentifier));
137 #define RegionEnd             (f_CompleteRegion());
138
139 #define OuterLoop(id, rs, re, itr)  (_i_OuterStart =
      spu_read_decrementer()); (_i_ActiveLoopLevel=OUTER); (iter =
      &itr); (_i_ActiveOuterLoopIdentifier = id); (_i_LoopStart = (
      rs)); (_i_LoopEnd = (re)); (f_RegisterLoop((id), (rs), (re),
      OUTER))
140 #define OuterLoopEnd          (_i_OuterEnd = spu_read_decrementer
      ()); (f_LoopEnd(_i_ActiveOuterLoopIdentifier, OUTER)); (
      ReportMeasurement(PROGRAM_COMPLETE_EXECUTION, _i_OuterStart,
```

```
    _i_OuterEnd))
141
142 #define InnerLoop(id, rs, re, itr)  (_i_ActiveLoopLevel=INNER);
    (iter = &itr); (_i_ActiveInnerLoopIdentifier = id); (
    _i_LoopStart = (rs)); (_i_LoopEnd = (re)); (f_RegisterLoop((
    id), (rs), (re), INNER))
143 #define InnerLoopEnd          (f_LoopEnd(
    _i_ActiveInnerLoopIdentifier, INNER))
144
145 #define ShutdownRequest         (f_Shutdown())
146
147 #define ComCheck(x)           (f_QuickCheck(x))
148 #define COMListen(x)          (f_StopAndListen(x))
149 #define COMMailboxListen        (f_MailboxListen())
150 #define ShutdownReturn          return 0;
151
152 #define ReportMeasurement(name, start, end)
    f_SubmitMetricResult((name), (start), (end))
153
154 /*Interrupt routine prototype*/
155 void sys_mailbox_interrupt_routine(void) __attribute__((section(
    ".interrupt")));
156 volatile unsigned int _i_CheckValue;
157
158 unsigned int *iter;
159
160 /*SPU related variables*/
161 unsigned long long _i_SPERegister;
162 unsigned long long _i_ARGPRegister;
163 unsigned long long _i_ENVPRegister;
164
165 /*SPE short identifier*/
166 unsigned int _i_SPEIdentifier;
167
168 /*Vector for Load and Store function*/
169 int request_create = -1;
170 vector unsigned int vec_result_high, vec_result_low;
```

```
171 vector unsigned int vec_low;
172 vector signed int vec_request;
173
174 /*Restart*/
175 volatile short unsigned int _i_LoopRestart = FALSE;
176
177 /*Computation specific variables*/
178 unsigned int _i_SizeBeginPrimary;
179 unsigned int _i_SizeEndPrimary;
180 unsigned int _i_SizeBeginSecondary;
181 unsigned int _i_SizeEndSecondary;
182
183 unsigned int _i_Final = 0;
184
185 unsigned long long _io_InputAddress;
186 unsigned long long _io_OutputAddress;
187 unsigned long long _io_AuxAddress1;
188 unsigned long long _io_AuxAddress2;
189 unsigned long long _io_AuxAddress3;
190
191 unsigned long long *_p_io_AuxAddress1;
192 unsigned long long *_p_io_AuxAddress2;
193
194
195 unsigned long long _io_MonitorArrayAddress;
196
197 unsigned long long _i_OuterStart, _i_OuterEnd;
198
199 unsigned int size;
200 unsigned int size_prog;
201
202 unsigned int Lower_IPC;
203 unsigned int Upper_IPC;
204
205 ipc_info_t        s_IPCRegister;
206 spe_info_init_t    s_SPEInformation;
207 context_params_t   s_RegionParameters;
```

```
208 context_params_t  s_RegionComplete;
209 transmit_report_t   s_Report;
210 measure_t        s_Measure;
211
212 ////////////////////////////////////CALLBACK CODE
      ////////////////////////////////////////
213 int errno;
214 typedef unsigned int spe_offset_t;
215 void ___send_to_ppe(unsigned int signalcode, unsigned int opcode,
      void *data);
216 ////////////////////////////////// ^ CALLBACK CODE ^
      ////////////////////////////////////
217
218 /*System startup and shutdown func's*/
219 int f_SystemInitialise(void);
220 int f_Shutdown(void);
221
222 /*System mailbox listening func's*/
223 int f_MailboxListen(void);
224 void f_MailboxInterruptRoutine(void);
225
226 /*System monitoring and listening func's - mainly signals*/
227 unsigned f_StopAndListen(unsigned channel);
228 unsigned f_SignalStatus(unsigned int channel);
229 unsigned f_SignalMonitor(unsigned int channel);
230
231 /*Report back to the SPE when a region has been completed*/
232 int f_CompleteRegion(void);
233
234 /*Get regional parameters*/
235 void f_GetRegionParameters(unsigned int region);
236
237 /*Framework functions*/
238 void f_GetLastLoopParameters(int id, unsigned level);
239 void f_DeregisterLoop(unsigned id, unsigned level);
240 int f_LoopBarrier(int loop, int level);
241
```

```
242 /*Callback func*/
243 void __send_to_ppe(unsigned int signalcode, unsigned int opcode,
        void *data);
244
245 //unsigned int idx_send;
246 unsigned int _i_SPUMailbox;
247
248 //Report measurement back to the PPE
249 int f_SubmitMetricResult(unsigned int name, unsigned long start,
        unsigned long end);
250
251 ////////////////////////////////CALLBACK CODE
      ////////////////////////////////////////
252 void __send_to_ppe(unsigned int signalcode, unsigned int opcode,
        void *data) {
253
254   if(_i_LoopRestart==FALSE) {
255
256 //     exec_start = spu_read_decrementer();
257
258     unsigned int combined = ((opcode <<24) | ((unsigned int)data
          & 0x00FFFFFF));
259
260     vector unsigned int stopfunc = {
261       signalcode,
262       combined,
263       0x4020007F,
264       0x35000000
265     };
266
267     void (*f) (void) = (void *)&stopfunc;
268     asm("sync");
269     f ();
270     errno = ((unsigned int *) data)[3];
271
272 //     exec_end = spu_read_decrementer();
273 //     ReportMeasurement(CALLBACK, exec_start, exec_end);
```

```
274   }
275
276   _i_LoopRestart=FALSE;
277
278
279 }
280 ////////////////////////////// ^ CALLBACK CODE ^
      ////////////////////////////////////
281
282 int f_UpdateMonitorArray(int tag, unsigned index) {
283   DATA_MONITOR_STORE(___CACHE_MONITOR, tag, index);
284   CACHE_FLUSH(___CACHE_MONITOR);
285   return 0;
286 }
287
288 void f_UpdateProcessedIPC(unsigned val) {
289   /*check mailbox stat first !*/
290   _i_SPUMailbox = spu_stat_out_mbox();
291
292   if(_i_SPUMailbox==1) {
293     /*we can put something in the mailbox*/
294     spu_write_out_mbox(val);
295   }
296 }
297
298 void f_LoopEnd(unsigned id, unsigned level) {
299   f_GetLastLoopParameters(id, level);
300   f_LoopBarrier(id, level);
301   f_DeregisterLoop(id, level);
302 }
303
304 void f_GetLastLoopParameters(int id, unsigned level) {
305   loop_t s_LoopInformation;
306
307   s_LoopInformation.spe = _i_SPEIdentifier;
308   s_LoopInformation.level = level;
309   s_LoopInformation.type = COPY_BACK;
```

```
310   s_LoopInformation.id = (id-1);
311
312   //Send struct to PPE
313   __send_to_ppe(0x2126, 0, &s_LoopInformation);
314
315   if(id > 0) {
316     _i_LoopStart = s_LoopInformation.pos_start;
317     _i_LoopEnd = s_LoopInformation.pos_end;
318   }
319 }
320
321 void f_DeregisterLoop(unsigned id, unsigned level) {
322   loop_t s_LoopInformation;
323
324   s_LoopInformation.spe = _i_SPEIdentifier;
325   s_LoopInformation.level = level;
326   s_LoopInformation.type = DEREGISTER;
327   s_LoopInformation.id = id;
328   s_LoopInformation.outer_level = _i_ActiveOuterLoopIdentifier;
329
330   //Send struct to PPE
331   __send_to_ppe(0x2126, 0, &s_LoopInformation);
332
333   int _i_Temp = _i_ActiveInnerLoopIdentifier-1;
334
335   if(_i_Temp < 0)
336     _i_ActiveInnerLoopIdentifier = 0;
337   else
338       _i_ActiveInnerLoopIdentifier = _i_Temp;
339 }
340
341 void f_RegisterLoop(unsigned id, unsigned rs, unsigned re,
      unsigned level) {
342   loop_t s_LoopInformation;
343
344   s_LoopInformation.spe = _i_SPEIdentifier;
345   s_LoopInformation.level = level;
```

```
346   s_LoopInformation.type = REGISTER;
347   s_LoopInformation.id = id;
348   s_LoopInformation.pos_start = rs;
349   s_LoopInformation.pos_end = re;
350
351   s_LoopInformation.outer_level = _i_ActiveOuterLoopIdentifier;
352
353   _i_LoopStart = rs;
354   _i_LoopEnd = re;
355
356   //Send struct to PPE
357   __send_to_ppe(0x2126, 0, &s_LoopInformation);
358 }
359
360 /*
361  * When the SPE starts, the SPE must register its given r3
        address with the PPE. The
362  * PPE will then assign a simple short number. This short number
        is static and will
363  * remain attached with the r3 address.
364  */
365 int f_SystemInitialise(void) {
366   _io_InputAddress = 0;
367   _i_SPEIdentifier = _i_ARGPRegister;
368
369   /*Get SPE short number*/
370   s_SPEInformation.spe = _i_SPEIdentifier;
371   s_SPEInformation.ea_addr[0] = _i_SPERegister;
372   s_SPEInformation.ea_addr[1] = _i_ARGPRegister;
373   s_SPEInformation.ea_addr[2] = _i_ENVPRegister;
374   __send_to_ppe(0x2121, 0, &s_SPEInformation);
375
376
377   /*Register IPC variable*/
378 //  s_IPCRegister.spe = _i_SPEIdentifier;
379 //  s_IPCRegister.ipc_addr = (unsigned long long)&*iter;
380 //  __send_to_ppe(0x2116, 0, &s_IPCRegister);
```

```
381
382   return 0;
383 }
384
385 unsigned f_StopAndListen(unsigned channel) {
386   static unsigned _sui_SignalData;
387
388   unsigned int _i_SignalType;
389   if(channel==1) _i_SignalType = SIGNAL1_CL;
390   if(channel==2) _i_SignalType = SIGNAL2_CL;
391
392   exec_start = spu_read_decrementer();
393
394   f_SignalMonitor(channel); //will return 0 when done
395   _sui_SignalData = f_SignalStatus(channel);
396
397   switch(_sui_SignalData) {
398     case SPE_RESTART:
399       *iter = _i_LoopStart;
400       _i_LoopRestart=TRUE;
401       exec_end = spu_read_decrementer();
402       ReportMeasurement(_i_SignalType, exec_start ,exec_end);
403       return LOOP_RESTART;
404       break;
405
406     case SPE_SHUTDOWN:
407       f_Shutdown();
408       exec_end = spu_read_decrementer();
409       ReportMeasurement(_i_SignalType, exec_start ,exec_end);
410       return 0;
411       break;
412
413     case SPE_CONTINUE:
414       exec_end = spu_read_decrementer();
415       ReportMeasurement(_i_SignalType, exec_start ,exec_end);
416       return 0;
417       break;
```

```
418
419     case SPE_HALT:
420       exec_end = spu_read_decrementer();
421       ReportMeasurement(_i_SignalType, exec_start ,exec_end);
422       COMListen(channel);
423       return 0;
424       break;
425
426     case BUFFER1:
427       exec_end = spu_read_decrementer();
428       ReportMeasurement(_i_SignalType, exec_start ,exec_end);
429       return BUFFER1;
430       break;
431
432     case BUFFER2:
433       exec_end = spu_read_decrementer();
434       ReportMeasurement(_i_SignalType, exec_start ,exec_end);
435       return BUFFER2;
436       break;
437
438     default:
439       exec_end = spu_read_decrementer();
440       ReportMeasurement(_i_SignalType, exec_start ,exec_end);
441       return _sui_SignalData;
442       break;
443   }
444
445   exec_end = spu_read_decrementer();
446   ReportMeasurement(_i_SignalType, exec_start ,exec_end);
447   return FUNCTION_FAILED;
448 }
449
450
451 int f_QuickCheck(unsigned channel) {
452   int _sui_SignalData;
453   unsigned int _i_SignalType;
454   if(channel==1) _i_SignalType = SIGNAL1_QC;
```

```
455   if(channel==2) _i_SignalType = SIGNAL2_QC;
456
457   exec_start = spu_read_decrementer();
458   _sui_SignalData = f_SignalStatus(channel);
459
460   switch(_sui_SignalData) {
461     case SPE_RESTART:
462       *iter = _i_LoopStart;
463       _i_LoopRestart=TRUE;
464
465       exec_end = spu_read_decrementer();
466       ReportMeasurement(_i_SignalType, exec_start ,exec_end);
467
468       return LOOP_RESTART;
469       break;
470
471     case SPE_SHUTDOWN:
472       f_Shutdown();
473
474       exec_end = spu_read_decrementer();
475       ReportMeasurement(_i_SignalType, exec_start ,exec_end);
476
477       return 0;
478       break;
479
480     case SPE_CONTINUE:
481       exec_end = spu_read_decrementer();
482       ReportMeasurement(_i_SignalType, exec_start ,exec_end);
483       return 0;
484       break;
485
486     case SPE_HALT:
487       COMListen(channel);
488       exec_end = spu_read_decrementer();
489       ReportMeasurement(_i_SignalType, exec_start ,exec_end);
490       return 0;
491       break;
```

```
492
493       default:
494         exec_end = spu_read_decrementer();
495         ReportMeasurement(_i_SignalType, exec_start ,exec_end);
496         return 0;
497         break;
498     }
499
500     exec_end = spu_read_decrementer();
501     ReportMeasurement(_i_SignalType, exec_start ,exec_end);
502     return FUNCTION_FAILED;
503 }
504
505 /* Return the status of signal register i.e.
506  * if there are any consumed slots*/
507 unsigned f_SignalStatus(unsigned int channel) {
508     if(channel==1) { return spu_stat_signal1(); }
509     if(channel==2) { return spu_stat_signal2(); }
510     return FUNCTION_FAILED;
511 }
512
513 /*Wait on signal register numbered #*/
514 unsigned f_SignalMonitor(unsigned int channel) {
515     if(channel==1) { do {} while(!spu_stat_signal1()); }
516     if(channel==2) { do {} while(!spu_stat_signal2()); }
517     return 0;
518 }
519
520 /*SPE must get permission to shutdown*/
521 int f_Shutdown(void) {
522     s_Report.spe = _i_SPEIdentifier;
523     __send_to_ppe(0x2114, 0, &s_Report);
524
525     switch(s_Report.result) {
526       case SPE_SHUTDOWN:
527         return 0;
528         break;
```

```
529
530       default:
531         COMListen(1);
532         break;
533     }
534
535     return FUNCTION_FAILED;
536 }
537
538
539 /*Notify PPE that the SPE has completed region numbered #*/
540 int f_CompleteRegion(void) {
541     spe_sig_ob_t s_SPESignal;
542
543     if(_i_SPEIdentifier==0) {
544       s_SPESignal.id = _i_RegionIdentifier;
545       s_SPESignal.spe = _i_SPEIdentifier;
546       __send_to_ppe(0x2127, 0, &s_SPESignal);
547     }
548
549     /*Synchronise with all region_end, listening on signal 2*/
550     if(_i_SPEIdentifier!=0 && _i_SPEIdentifier!=5) {
551       s_SPESignal.id = _i_RegionIdentifier;
552       s_SPESignal.spe = _i_SPEIdentifier;
553       __send_to_ppe(0x2127, 0, &s_SPESignal);
554     }
555
556     if(_i_SPEIdentifier==5) {
557       s_SPESignal.id = _i_RegionIdentifier;
558       s_SPESignal.spe = _i_SPEIdentifier;
559       __send_to_ppe(0x2127, 0, &s_SPESignal);
560     }
561
562     s_RegionComplete.completed = REGION_COMPLETED;
563     s_RegionComplete.region = _i_RegionIdentifier;
564     s_RegionComplete.spe = _i_SPEIdentifier;
565
```

```
566   __send_to_ppe(0x2119, 0, &s_RegionComplete);
567
568   if(s_RegionComplete.assigned!=SPE_CONTINUE) {
569     COMListen(2);
570   }
571
572   return 0;
573 }
574
575 /*We need to ask the PPE for region parameters numbered #*/
576 void f_GetRegionParameters(unsigned int region) {
577   s_RegionParameters.region = region;
578   s_RegionParameters.spe = _i_SPEIdentifier;
579
580   __send_to_ppe(0x2118, 0, &s_RegionParameters);
581
582   if(s_RegionParameters.assigned==SPE_STANDBY) {
583     COMListen(1);
584   }
585
586   _i_GlobalIOType = s_RegionParameters.io_array_type;
587
588   size = s_RegionParameters.size;
589
590   _i_SizeBeginPrimary = s_RegionParameters.size_begin;
591   _i_SizeEndPrimary = s_RegionParameters.size_end;
592
593   _i_SizeBeginSecondary = s_RegionParameters.size_begin_2;
594   _i_SizeEndSecondary = s_RegionParameters.size_end_2;
595
596   Lower_IPC = s_RegionParameters.itr_begin;
597   Upper_IPC = s_RegionParameters.itr_end;
598
599   _io_InputAddress = s_RegionParameters.ea_in;
600   _io_OutputAddress = s_RegionParameters.ea_out;
601
602   _i_Final = s_RegionParameters.final;
```

```
603
604   _io_AuxAddress1 = s_RegionParameters.ea_aux;
605   _io_AuxAddress2 = s_RegionParameters.ea_aux2;
606   _io_AuxAddress3 = s_RegionParameters.ea_aux3;
607
608   _p_io_AuxAddress1 = (unsigned long long *)_io_AuxAddress1;
609   _p_io_AuxAddress2 = (unsigned long long*)_io_AuxAddress2;
610
611   _io_MonitorArrayAddress = (unsigned long long)
        s_RegionParameters.store_array;
612 }
613
614 int f_LoopBarrier(int loop, int level) {
615   spe_sig_ob_t s_SPESignal;
616
617   s_SPESignal.id = loop;
618   s_SPESignal.loop_level = level;
619   s_SPESignal.spe = _i_SPEIdentifier;
620   __send_to_ppe(0x2127, 0, &s_SPESignal);
621
622   if(s_SPESignal.id!=SPE_CONTINUE)
623     COMListen(2);
624
625   return 0;
626 }
627
628 int f_SubmitMetricResult(unsigned int name, unsigned long start,
       unsigned long end) {
629   s_Measure.name = name;
630   s_Measure.element = SPU_ELEMENT;
631   s_Measure.spe = _i_SPEIdentifier;
632   s_Measure.begin = start;
633   s_Measure.end = end;
634 //  s_Measure.value = (end-start);
635
636   __send_to_ppe(0x2125, 0, &s_Measure);
637
```

```
638   return 0;
639 }
640
641
642 /*Mailbox listening.... (cannot be interrupted by any signal)*/
643 int f_MailboxListen(void) {
644   exec_start = spu_read_decrementer();
645
646   spu_write_event_mask(MFC_IN_MBOX_AVAILABLE_EVENT);
647   spu_ienable();
648   while(!_i_CheckValue);
649   spu_idisable();
650   _i_CheckValue = 0;
651
652   exec_end = spu_read_decrementer();
653   ReportMeasurement(MAILBOX_INTERRUPT, exec_start ,exec_end);
654   return 0;
655 }
656
657 void f_MailboxInterruptRoutine(void) {
658   spu_write_event_ack(MFC_IN_MBOX_AVAILABLE_EVENT);
659   _i_CheckValue++;
660   asm("iret");
661 }
662
663 ////////////////////////////////// SCALAR COMPARISON
      //////////////////////////////////
664
665 double* Load_DPTR(unsigned idx) {
666   ComCheck(2);
667   exec_start = spu_read_decrementer();
668   if(idx >= _i_LoopStart && idx < _i_LoopEnd) {
669     UPDATE_PROCESSED_VALUE(*iter);
670
671     exec_end = spu_read_decrementer();
672     ReportMeasurement(LOAD_E, exec_start, exec_end);
673     return DATA_PTR_PRIMARY(___CACHE_DOUBLE, idx);
```

```
674   }
675   else {
676
677     request_message_t request;
678     request.outer_level = _i_ActiveOuterLoopIdentifier;
679     request.request_type = LOAD;
680     request.region_number = _i_RegionIdentifier;
681     request.spe_sid = _i_SPEIdentifier;
682     request.owner_itr = *iter;
683     request.aux = -1;
684     request.index_a = idx;
685     request.loop_level = _i_ActiveLoopLevel;
686     request.io_array_type = _i_GlobalIOType;
687
688     if(_i_ActiveLoopLevel==INNER)
689       request.level = _i_ActiveInnerLoopIdentifier;
690
691     if(_i_ActiveLoopLevel==OUTER)
692       request.level = _i_ActiveOuterLoopIdentifier;
693
694     __send_to_ppe(0x2115, 0, &request);
695
696     unsigned short int buffer;
697     buffer = COMListen(2);
698
699     double *data;
700
701     if(buffer==BUFFER1) {
702       data = DATA_PTR_PRIMARY(___CACHE_DOUBLE, idx);
703       UPDATE_PROCESSED_VALUE(*iter);
704     }
705
706     if(buffer==BUFFER2) {
707       data = DATA_PTR_SECONDARY(___CACHE_DOUBLE, idx);
708       UPDATE_PROCESSED_VALUE(*iter);
709     }
710
```

```
711     /*tell PPE that we are now done with this request*/
712     __send_to_ppe(0x2123, 0, &request);
713
714     exec_end = spu_read_decrementer();
715     ReportMeasurement(LOAD_E, exec_start, exec_end);
716
717     return data;
718   }
719 }
720
721 double LoadD(double *src, unsigned idx) {
722   ComCheck(2);
723
724   exec_start = spu_read_decrementer();
725   if(idx >= _i_LoopStart && idx < _i_LoopEnd) {
726     UPDATE_PROCESSED_VALUE(*iter);
727
728     exec_end = spu_read_decrementer();
729     ReportMeasurement(LOAD_E, exec_start, exec_end);
730
731     return src[idx];
732   }
733   else {
734     request_message_t request;
735     request.outer_level = _i_ActiveOuterLoopIdentifier;
736     request.request_type = LOAD;
737     request.region_number = _i_RegionIdentifier;
738     request.spe_sid = _i_SPEIdentifier;
739     request.owner_itr = *iter;
740     request.aux = -1;
741     request.index_a = idx;
742     request.loop_level = _i_ActiveLoopLevel;
743     request.io_array_type = _i_GlobalIOType;
744
745     if(_i_ActiveLoopLevel==INNER)
746       request.level = _i_ActiveInnerLoopIdentifier;
747
```

```
748       if(_i_ActiveLoopLevel==OUTER)
749         request.level = _i_ActiveOuterLoopIdentifier;
750
751       __send_to_ppe(0x2115, 0, &request);
752
753       unsigned int buffer;
754
755       buffer = COMListen(2);
756
757
758       double data;
759
760       if(buffer==BUFFER1) {
761         data = *DATA_PTR_PRIMARY(___CACHE_DOUBLE, idx);
762         UPDATE_PROCESSED_VALUE(*iter);
763       }
764
765       if(buffer==BUFFER2) {
766         data = *DATA_PTR_SECONDARY(___CACHE_DOUBLE, idx);
767         UPDATE_PROCESSED_VALUE(*iter);
768       }
769
770       /*tell PPE that we are now done with this request*/
771       __send_to_ppe(0x2123, 0, &request);
772
773       exec_end = spu_read_decrementer();
774       ReportMeasurement(LOAD_E, exec_start, exec_end);
775       return data;
776     }
777 }
778
779 void StoreD(double *src, unsigned idx, double data) {
780     ComCheck(2);
781
782     exec_start = spu_read_decrementer();
783     if(idx >= _i_LoopStart && idx < _i_LoopEnd) {
784       UPDATE_PROCESSED_VALUE(*iter);
```

```
785     UPDATE_STORE_VALUE(INTERNAL_STORE, *iter);
786
787     src[idx]=data;
788   }
789   else {
790     request_message_t request;
791     request.outer_level = _i_ActiveOuterLoopIdentifier;
792     request.request_type = STORE;
793     request.region_number = _i_RegionIdentifier;
794     request.spe_sid = _i_SPEIdentifier;
795     request.owner_itr = *iter;
796     request.aux = -1;
797     request.index_a = idx;
798     request.loop_level = _i_ActiveLoopLevel;
799     request.data = data;
800     request.io_array_type = _i_GlobalIOType;
801
802     if(_i_ActiveLoopLevel==INNER)
803       request.level = _i_ActiveInnerLoopIdentifier;
804
805     if(_i_ActiveLoopLevel==OUTER)
806       request.level = _i_ActiveOuterLoopIdentifier;
807
808     __send_to_ppe(0x2115, 0, &request);
809
810
811     unsigned int buffer;
812     buffer = COMListen(2);
813
814     double *send;
815
816     if(buffer==BUFFER1) {
817       send = DATA_PTR_PRIMARY(___CACHE_DOUBLE, idx);
818       send[idx] = data;
819       UPDATE_PROCESSED_VALUE(*iter);
820       UPDATE_STORE_VALUE(EXTERNAL_STORE, *iter);
821     }
```

```
822
823       if(buffer==BUFFER2) {
824         send = DATA_PTR_SECONDARY(___CACHE_DOUBLE, idx);
825         send[idx] = data;
826         UPDATE_PROCESSED_VALUE(*iter);
827         UPDATE_STORE_VALUE(EXTERNAL_STORE, *iter);
828       }
829
830
831       /*tell PPE that we are now done with this request*/
832       ___send_to_ppe(0x2123, 0, &request);
833     }
834
835     exec_end = spu_read_decrementer();
836     ReportMeasurement(STORE_E, exec_start, exec_end);
837 }
838
839
840
841
842
843
844
845
846
847
848
849 double* Load_DPTR_DA(unsigned idxA, unsigned idxB) {
850     ComCheck(2);
851
852     exec_start = spu_read_decrementer();
853     if((idxA >= _i_SizeBeginPrimary && idxA < _i_SizeEndPrimary)
          && (idxB >= _i_LoopStart && idxB < _i_LoopEnd)) {
854       UPDATE_PROCESSED_VALUE(*iter);
855
856       exec_end = spu_read_decrementer();
857       ReportMeasurement(LOAD_E, exec_start, exec_end);
```

```
858
859       return DATA_PTR_DA_PRIMARY(___CACHE_DOUBLE, idxA, idxB);
860     }
861     else {
862       request_message_t request;
863       request.outer_level = _i_ActiveOuterLoopIdentifier;
864       request.request_type = LOAD;
865       request.region_number = _i_RegionIdentifier;
866       request.spe_sid = _i_SPEIdentifier;
867       request.owner_itr = *iter;
868       request.aux = -1;
869       request.index_a = idxA;
870       request.index_b = idxB;
871       request.io_array_type = _i_GlobalIOType;
872       request.loop_level = _i_ActiveLoopLevel;
873
874       if(_i_ActiveLoopLevel==INNER)
875         request.level = _i_ActiveInnerLoopIdentifier;
876
877       if(_i_ActiveLoopLevel==OUTER)
878         request.level = _i_ActiveOuterLoopIdentifier;
879
880       __send_to_ppe(0x2115, 0, &request);
881
882       unsigned int buffer;
883       buffer = COMListen(2);
884
885       double *data;
886
887       if(buffer==BUFFER1) {
888         data = DATA_PTR_DA_PRIMARY(___CACHE_DOUBLE, idxA, idxB);
889         UPDATE_PROCESSED_VALUE(*iter);
890       }
891
892       if(buffer==BUFFER2) {
893         data = DATA_PTR_DA_SECONDARY(___CACHE_DOUBLE, idxA, idxB);
894         UPDATE_PROCESSED_VALUE(*iter);
```

```
895       }
896
897       /*tell PPE that we are now done with this request*/
898       __send_to_ppe(0x2123, 0, &request);
899
900       exec_end = spu_read_decrementer();
901       ReportMeasurement(LOAD_E, exec_start, exec_end);
902       return data;
903     }
904 }
905
906
907 double Load_DDA(double **src, unsigned idxA, unsigned idxB) {
908     ComCheck(2);
909
910     exec_start = spu_read_decrementer();
911     if((idxA >= _i_SizeBeginPrimary && idxA < _i_SizeEndPrimary)
            && (idxB >= _i_LoopStart && idxB < _i_LoopEnd)) {
912       UPDATE_PROCESSED_VALUE(*iter);
913
914       exec_end = spu_read_decrementer();
915       ReportMeasurement(LOAD_E, exec_start, exec_end);
916       return src[idxA][idxB];
917     }
918     else {
919       request_message_t request;
920       request.outer_level = _i_ActiveOuterLoopIdentifier;
921       request.request_type = LOAD;
922       request.region_number = _i_RegionIdentifier;
923       request.spe_sid = _i_SPEIdentifier;
924       request.owner_itr = *iter;
925       request.aux = -1;
926       request.index_a = idxA;
927       request.index_b = idxB;
928       request.loop_level = _i_ActiveLoopLevel;
929       request.io_array_type = _i_GlobalIOType;
930
```

```
931       if(_i_ActiveLoopLevel==INNER)
932         request.level = _i_ActiveInnerLoopIdentifier;
933
934       if(_i_ActiveLoopLevel==OUTER)
935         request.level = _i_ActiveOuterLoopIdentifier;
936
937
938       __send_to_ppe(0x2115, 0, &request);
939
940       int buffer;
941
942       buffer = COMListen(2);
943
944       double* data;
945
946       if(buffer==BUFFER1) {
947         data = DATA_PTR_DA_PRIMARY(___CACHE_DOUBLE, idxA, idxB);
948         UPDATE_PROCESSED_VALUE(*iter);
949       }
950
951       if(buffer==BUFFER2) {
952         data = DATA_PTR_DA_SECONDARY(___CACHE_DOUBLE, idxA, idxB);
953         UPDATE_PROCESSED_VALUE(*iter);
954       }
955
956       /*tell PPE that we are now done with this request*/
957       __send_to_ppe(0x2123, 0, &request);
958
959       exec_end = spu_read_decrementer();
960       ReportMeasurement(LOAD_E, exec_start, exec_end);
961       return *data;
962     }
963 }
964
965 void Store_DDA(double **src, unsigned idxA, unsigned idxB,
        double data) {
966     ComCheck(2);
```

```
967
968   exec_start = spu_read_decrementer();
969   if((idxA >= _i_SizeBeginPrimary && idxA < _i_SizeEndPrimary)
          && (idxB >= _i_LoopStart && idxB < _i_LoopEnd)) {
970     UPDATE_PROCESSED_VALUE(*iter);
971     UPDATE_STORE_VALUE(INTERNAL_STORE, *iter);
972
973     src[idxA][idxB]=data;
974   }
975   else {
976     request_message_t request;
977     request.outer_level = _i_ActiveOuterLoopIdentifier;
978     request.request_type = STORE;
979     request.region_number = _i_RegionIdentifier;
980     request.spe_sid = _i_SPEIdentifier;
981     request.owner_itr = *iter;
982     request.aux = -1;
983     request.index_a = idxA;
984     request.index_b = idxB;
985     request.loop_level = _i_ActiveLoopLevel;
986     request.data = data;
987     request.io_array_type = _i_GlobalIOType;
988
989     if(_i_ActiveLoopLevel==INNER)
990       request.level = _i_ActiveInnerLoopIdentifier;
991
992     if(_i_ActiveLoopLevel==OUTER)
993       request.level = _i_ActiveOuterLoopIdentifier;
994
995     __send_to_ppe(0x2115, 0, &request);
996
997     unsigned int buffer;
998     buffer = COMListen(2);
999
1000    double** bsource;
1001
1002    if(buffer==BUFFER1) {
```

```
1003        DATA_STORE_DOUBLE_AUX(___CACHE_DOUBLE, idxA, idxB,
                _io_AuxAddress1, double, data);
1004 //        bsource = (double **)DATA_PTR_DA_PRIMARY(___CACHE_DOUBLE
        , idxA, idxB);
1005 //        bsource[idxA][idxB] = data;
1006        UPDATE_PROCESSED_VALUE(*iter);
1007        UPDATE_STORE_VALUE(EXTERNAL_STORE, *iter);
1008      }
1009
1010      if(buffer==BUFFER2) {
1011        DATA_STORE_DOUBLE_AUX(___CACHE_DOUBLE, idxA, idxB,
                _io_AuxAddress1, double, data);
1012 //        bsource = (double **)DATA_PTR_DA_SECONDARY(
       ___CACHE_DOUBLE, idxA, idxB);
1013 //        bsource[idxA][idxB] = data;
1014        UPDATE_PROCESSED_VALUE(*iter);
1015        UPDATE_STORE_VALUE(EXTERNAL_STORE, *iter);
1016      }
1017
1018      /*tell PPE that we are now done with this request*/
1019      ___send_to_ppe(0x2123, 0, &request);
1020    }
1021
1022    exec_end = spu_read_decrementer();
1023    ReportMeasurement(STORE_E, exec_start, exec_end);
1024 }
1025
1026 ////////////////////////////////// ^ SCALAR COMPARISON ^
        //////////////////////////////////
1027
1028
1029
1030
1031 //INTEGER
1032
1033 int* Load_IPTR(unsigned idx) {
1034    ComCheck(2);
```

```
1035
1036    exec_start = spu_read_decrementer();
1037    if(idx >= _i_LoopStart && idx < _i_LoopEnd) {
1038      UPDATE_PROCESSED_VALUE(*iter);
1039
1040      exec_end = spu_read_decrementer();
1041      ReportMeasurement(LOAD_E, exec_start, exec_end);
1042      return DATA_PTR_PRIMARY(___CACHE_INT, idx);
1043    }
1044    else {
1045      request_message_t request;
1046      request.outer_level = _i_ActiveOuterLoopIdentifier;
1047      request.request_type = LOAD;
1048      request.region_number = _i_RegionIdentifier;
1049      request.spe_sid = _i_SPEIdentifier;
1050      request.owner_itr = *iter;
1051      request.aux = -1;
1052      request.index_a = idx;
1053      request.loop_level = _i_ActiveLoopLevel;
1054      request.io_array_type = _i_GlobalIOType;
1055
1056      if(_i_ActiveLoopLevel==INNER)
1057        request.level = _i_ActiveInnerLoopIdentifier;
1058
1059      if(_i_ActiveLoopLevel==OUTER)
1060        request.level = _i_ActiveOuterLoopIdentifier;
1061
1062      __send_to_ppe(0x2115, 0, &request);
1063
1064      COMListen(2);
1065
1066      unsigned short int buffer = spu_read_signal2();
1067      static int *data;
1068
1069      if(buffer==BUFFER1) {
1070        data = DATA_PTR_PRIMARY(___CACHE_INT, idx);
1071        UPDATE_PROCESSED_VALUE(*iter);
```

```
1072      }
1073
1074      if(buffer==BUFFER2) {
1075        data = DATA_PTR_SECONDARY(___CACHE_INT, idx);
1076        UPDATE_PROCESSED_VALUE(*iter);
1077      }
1078
1079      /*tell PPE that we are now done with this request*/
1080      ___send_to_ppe(0x2123, 0, &request);
1081
1082      exec_end = spu_read_decrementer();
1083      ReportMeasurement(LOAD_E, exec_start, exec_end);
1084      return data;
1085    }
1086 }
1087
1088 int LoadI(int *src, unsigned idx) {
1089   ComCheck(2);
1090
1091   exec_start = spu_read_decrementer();
1092   if(idx >= _i_LoopStart && idx < _i_LoopEnd) {
1093     UPDATE_PROCESSED_VALUE(*iter);
1094
1095     exec_end = spu_read_decrementer();
1096     ReportMeasurement(LOAD_E, exec_start, exec_end);
1097     return src[idx];
1098   }
1099   else {
1100     request_message_t request;
1101     request.outer_level = _i_ActiveOuterLoopIdentifier;
1102     request.request_type = LOAD;
1103     request.region_number = _i_RegionIdentifier;
1104     request.spe_sid = _i_SPEIdentifier;
1105     request.owner_itr = *iter;
1106     request.aux = -1;
1107     request.index_a = idx;
1108     request.loop_level = _i_ActiveLoopLevel;
```

```
1109      request.io_array_type = _i_GlobalIOType;
1110
1111      if(_i_ActiveLoopLevel==INNER)
1112        request.level = _i_ActiveInnerLoopIdentifier;
1113
1114      if(_i_ActiveLoopLevel==OUTER)
1115        request.level = _i_ActiveOuterLoopIdentifier;
1116
1117      __send_to_ppe(0x2115, 0, &request);
1118
1119      unsigned int buffer;
1120      buffer = COMListen(2);
1121
1122      double data;
1123
1124      if(buffer==BUFFER1) {
1125        data = DATA_LOAD_PRIMARY(___CACHE_INT, idx);
1126        UPDATE_PROCESSED_VALUE(*iter);
1127      }
1128
1129      if(buffer==BUFFER2) {
1130        data = DATA_LOAD_SECONDARY(___CACHE_INT, idx);
1131        UPDATE_PROCESSED_VALUE(*iter);
1132      }
1133
1134      /*tell PPE that we are now done with this request*/
1135      __send_to_ppe(0x2123, 0, &request);
1136
1137      exec_end = spu_read_decrementer();
1138      ReportMeasurement(LOAD_E, exec_start, exec_end);
1139      return data;
1140    }
1141 }
1142
1143 void StoreI(int *src, unsigned idx, int data) {
1144    ComCheck(2);
1145
```

```
1146   exec_start = spu_read_decrementer();
1147   if(idx >= _i_LoopStart && idx < _i_LoopEnd) {
1148     UPDATE_PROCESSED_VALUE(*iter);
1149     UPDATE_STORE_VALUE(INTERNAL_STORE, *iter);
1150
1151     src[idx]=data;
1152   }
1153   else {
1154     request_message_t request;
1155     request.outer_level = _i_ActiveOuterLoopIdentifier;
1156     request.request_type = STORE;
1157     request.region_number = _i_RegionIdentifier;
1158     request.spe_sid = _i_SPEIdentifier;
1159     request.owner_itr = *iter;
1160     request.aux = -1;
1161     request.index_a = idx;
1162     request.loop_level = _i_ActiveLoopLevel;
1163     request.data = data;
1164     request.io_array_type = _i_GlobalIOType;
1165
1166     if(_i_ActiveLoopLevel==INNER)
1167       request.level = _i_ActiveInnerLoopIdentifier;
1168
1169     if(_i_ActiveLoopLevel==OUTER)
1170       request.level = _i_ActiveOuterLoopIdentifier;
1171
1172     __send_to_ppe(0x2115, 0, &request);
1173
1174     unsigned int buffer;
1175     buffer = COMListen(2);
1176
1177     int* bsource;
1178
1179     if(buffer==BUFFER1) {
1180       bsource = DATA_PTR_PRIMARY(___CACHE_INT, idx);
1181       bsource[idx] = data;
1182       UPDATE_PROCESSED_VALUE(*iter);
```

```
1183          UPDATE_STORE_VALUE(EXTERNAL_STORE, *iter);
1184        }
1185
1186        if(buffer==BUFFER2) {
1187          bsource = DATA_PTR_SECONDARY(___CACHE_INT, idx);
1188          bsource[idx] = data;
1189          UPDATE_PROCESSED_VALUE(*iter);
1190          UPDATE_STORE_VALUE(EXTERNAL_STORE, *iter);
1191        }
1192
1193        /*tell PPE that we are now done with this request*/
1194        __send_to_ppe(0x2123, 0, &request);
1195      }
1196
1197      exec_end = spu_read_decrementer();
1198      ReportMeasurement(STORE_E, exec_start, exec_end);
1199    }
1200
1201
1202
1203
1204    int* Load_IPTR_DA(unsigned idxA, unsigned idxB) {
1205      ComCheck(2);
1206
1207      exec_start = spu_read_decrementer();
1208      if((idxA >= _i_SizeBeginPrimary && idxA < _i_SizeEndPrimary)
            && (idxB >= _i_LoopStart && idxB < _i_LoopEnd)) {
1209        UPDATE_PROCESSED_VALUE(*iter);
1210
1211        exec_end = spu_read_decrementer();
1212        ReportMeasurement(LOAD_E, exec_start, exec_end);
1213
1214        return DATA_PTR_DA_PRIMARY(___CACHE_INT, idxA, idxB);
1215      }
1216      else {
1217        request_message_t request;
1218        request.outer_level = _i_ActiveOuterLoopIdentifier;
```

```
1219      request.request_type = LOAD;
1220      request.region_number = _i_RegionIdentifier;
1221      request.spe_sid = _i_SPEIdentifier;
1222      request.owner_itr = *iter;
1223      request.aux = -1;
1224      request.index_a = idxA;
1225      request.index_b = idxB;
1226      request.io_array_type = _i_GlobalIOType;
1227      request.loop_level = _i_ActiveLoopLevel;
1228
1229      if(_i_ActiveLoopLevel==INNER)
1230        request.level = _i_ActiveInnerLoopIdentifier;
1231
1232      if(_i_ActiveLoopLevel==OUTER)
1233        request.level = _i_ActiveOuterLoopIdentifier;
1234
1235     __send_to_ppe(0x2115, 0, &request);
1236
1237      unsigned int buffer;
1238      buffer = COMListen(2);
1239
1240      int *data;
1241
1242      if(buffer==BUFFER1) {
1243        data = DATA_PTR_DA_PRIMARY(___CACHE_INT, idxA, idxB);
1244        UPDATE_PROCESSED_VALUE(*iter);
1245      }
1246
1247      if(buffer==BUFFER2) {
1248        data = DATA_PTR_DA_SECONDARY(___CACHE_INT, idxA, idxB);
1249        UPDATE_PROCESSED_VALUE(*iter);
1250      }
1251
1252      /*tell PPE that we are now done with this request*/
1253      __send_to_ppe(0x2123, 0, &request);
1254
1255      exec_end = spu_read_decrementer();
```

```
1256      ReportMeasurement(LOAD_E, exec_start, exec_end);
1257      return data;
1258    }
1259 }
1260
1261
1262 int Load_IDA(int **src, unsigned idxA, unsigned idxB) {
1263   ComCheck(2);
1264
1265    exec_start = spu_read_decrementer();
1266    if((idxA >= _i_SizeBeginPrimary && idxA < _i_SizeEndPrimary)
          && (idxB >= _i_LoopStart && idxB < _i_LoopEnd)) {
1267      UPDATE_PROCESSED_VALUE(*iter);
1268
1269      exec_end = spu_read_decrementer();
1270      ReportMeasurement(LOAD_E, exec_start, exec_end);
1271
1272      return src[idxA][idxB];
1273    }
1274    else {
1275      request_message_t request;
1276      request.outer_level = _i_ActiveOuterLoopIdentifier;
1277      request.request_type = LOAD;
1278      request.region_number = _i_RegionIdentifier;
1279      request.spe_sid = _i_SPEIdentifier;
1280      request.owner_itr = *iter;
1281      request.aux = -1;
1282      request.index_a = idxA;
1283      request.index_b = idxB;
1284      request.loop_level = _i_ActiveLoopLevel;
1285      request.io_array_type = _i_GlobalIOType;
1286
1287      if(_i_ActiveLoopLevel==INNER)
1288        request.level = _i_ActiveInnerLoopIdentifier;
1289
1290      if(_i_ActiveLoopLevel==OUTER)
1291        request.level = _i_ActiveOuterLoopIdentifier;
```

```
1292
1293      __send_to_ppe(0x2115, 0, &request);
1294
1295      unsigned int buffer;
1296      buffer = COMListen(2);
1297
1298      int* data;
1299
1300      if(buffer==BUFFER1) {
1301        data = DATA_PTR_DA_PRIMARY(___CACHE_INT, idxA, idxB);
1302        UPDATE_PROCESSED_VALUE(*iter);
1303      }
1304
1305      if(buffer==BUFFER2) {
1306        data = DATA_PTR_DA_SECONDARY(___CACHE_INT, idxA, idxB);
1307        UPDATE_PROCESSED_VALUE(*iter);
1308      }
1309
1310      /*tell PPE that we are now done with this request*/
1311      __send_to_ppe(0x2123, 0, &request);
1312
1313      exec_end = spu_read_decrementer();
1314      ReportMeasurement(LOAD_E, exec_start, exec_end);
1315      return *data;
1316    }
1317 }
1318
1319 void Store_IDA(int **src, unsigned idxA, unsigned idxB, int data
       ) {
1320    ComCheck(2);
1321
1322    exec_start = spu_read_decrementer();
1323    if((idxA >= _i_SizeBeginPrimary && idxA < _i_SizeEndPrimary)
         && (idxB >= _i_LoopStart && idxB < _i_LoopEnd)) {
1324      UPDATE_PROCESSED_VALUE(*iter);
1325      UPDATE_STORE_VALUE(INTERNAL_STORE, *iter);
1326
```

```
1327      src[idxA][idxB]=data;
1328    }
1329    else {
1330      request_message_t request;
1331      request.outer_level = _i_ActiveOuterLoopIdentifier;
1332      request.request_type = STORE;
1333      request.region_number = _i_RegionIdentifier;
1334      request.spe_sid = _i_SPEIdentifier;
1335      request.owner_itr = *iter;
1336      request.aux = -1;
1337      request.index_a = idxA;
1338      request.index_b = idxB;
1339      request.loop_level = _i_ActiveLoopLevel;
1340      request.data = data;
1341      request.io_array_type = _i_GlobalIOType;
1342
1343      if(_i_ActiveLoopLevel==INNER)
1344        request.level = _i_ActiveInnerLoopIdentifier;
1345
1346      if(_i_ActiveLoopLevel==OUTER)
1347        request.level = _i_ActiveOuterLoopIdentifier;
1348
1349      __send_to_ppe(0x2115, 0, &request);
1350
1351      unsigned int buffer;
1352      buffer = COMListen(2);
1353
1354
1355      int** bsource;
1356
1357      if(buffer==BUFFER1) {
1358        bsource = (int **)DATA_PTR_DA_PRIMARY(___CACHE_DOUBLE,
              idxA, idxB);
1359        bsource[idxA][idxB] = data;
1360        UPDATE_PROCESSED_VALUE(*iter);
1361        UPDATE_STORE_VALUE(EXTERNAL_STORE, *iter);
1362      }
```

```
1363
1364      if(buffer==BUFFER2) {
1365        bsource = (int **)DATA_PTR_DA_SECONDARY(___CACHE_DOUBLE,
              idxA, idxB);
1366        bsource[idxA][idxB] = data;
1367        UPDATE_PROCESSED_VALUE(*iter);
1368        UPDATE_STORE_VALUE(EXTERNAL_STORE, *iter);
1369      }
1370
1371      /*tell PPE that we are now done with this request*/
1372      __send_to_ppe(0x2123, 0, &request);
1373    }
1374
1375    exec_end = spu_read_decrementer();
1376    ReportMeasurement(STORE_E, exec_start, exec_end);
1377  }
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390  //AUX
        ////////////////////////////////////////////////////////////////

1391  double* Load_AUX_PTR(int aux, unsigned idx) {
1392    ComCheck(2);
1393
1394    exec_start = spu_read_decrementer();
1395    if(idx >= _i_SizeBeginPrimary && idx < _i_SizeEndPrimary) {
1396      UPDATE_PROCESSED_VALUE(*iter);
```

```
1397
1398      if(aux==1) {
1399        exec_end = spu_read_decrementer();
1400        ReportMeasurement(LOAD_E, exec_start, exec_end);
1401        return DATA_PTR_SINGLE_AUX(___CACHE_DOUBLE, idx,
              _io_AuxAddress1, double);
1402      }
1403
1404      if(aux==2) {
1405        exec_end = spu_read_decrementer();
1406        ReportMeasurement(LOAD_E, exec_start, exec_end);
1407        return (double *)DATA_PTR_SINGLE_AUX(___CACHE_INT, idx,
              _io_AuxAddress2, int);
1408      }
1409
1410      if(aux==3) {
1411        exec_end = spu_read_decrementer();
1412        ReportMeasurement(LOAD_E, exec_start, exec_end);
1413        return (double *)DATA_PTR_SINGLE_AUX(___CACHE_INT, idx,
              _io_AuxAddress3, int);
1414      }
1415    }
1416    else {
1417      request_message_t request;
1418      request.outer_level = _i_ActiveOuterLoopIdentifier;
1419      request.request_type = LOAD_AUX;
1420      request.region_number = _i_RegionIdentifier;
1421      request.spe_sid = _i_SPEIdentifier;
1422      request.owner_itr = *iter;
1423      request.aux = aux;
1424      request.index_a = idx;
1425      request.loop_level = _i_ActiveLoopLevel;
1426      request.io_array_type = _i_GlobalIOType;
1427
1428      if(_i_ActiveLoopLevel==INNER)
1429        request.level = _i_ActiveInnerLoopIdentifier;
1430
```

```
1431      if(_i_ActiveLoopLevel==OUTER)
1432        request.level = _i_ActiveOuterLoopIdentifier;
1433
1434      __send_to_ppe(0x2115, 0, &request);
1435
1436      unsigned int buffer;
1437      buffer = COMListen(2);
1438
1439      double *data;
1440
1441      if(buffer==BUFFER1) {
1442        if(aux==1)
1443          data = DATA_PTR_SINGLE_AUX(___CACHE_DOUBLE, idx,
                  _io_AuxAddress1, double);
1444
1445        if(aux==2) {
1446          data = (double *)DATA_PTR_SINGLE_AUX(___CACHE_INT, idx,
                  _io_AuxAddress2, int);
1447        }
1448
1449        if(aux==3)
1450          data = (double *)DATA_PTR_SINGLE_AUX(___CACHE_INT, idx,
                  _io_AuxAddress3, int);
1451
1452        UPDATE_PROCESSED_VALUE(*iter);
1453      }
1454
1455      if(buffer==BUFFER2) {
1456        if(aux==1)
1457          data = DATA_PTR_SINGLE_AUX(___CACHE_DOUBLE, idx,
                  _io_AuxAddress1, double);
1458
1459        if(aux==2)
1460          data = (double *)DATA_PTR_SINGLE_AUX(___CACHE_INT, idx,
                  _io_AuxAddress2, int);
1461
1462        if(aux==3)
```

```
1463          data = (double *)DATA_PTR_SINGLE_AUX(___CACHE_INT, idx,
                  _io_AuxAddress3, int);
1464
1465        UPDATE_PROCESSED_VALUE(*iter);
1466      }
1467
1468      /*tell PPE that we are now done with this request*/
1469      __send_to_ppe(0x2123, 0, &request);
1470
1471      exec_end = spu_read_decrementer();
1472      ReportMeasurement(LOAD_E, exec_start, exec_end);
1473      return data;
1474    }
1475 }
1476
1477 double Load_AUX(int aux, unsigned idx) {
1478    ComCheck(2);
1479
1480    exec_start = spu_read_decrementer();
1481    if(idx >= _i_LoopStart && idx < _i_LoopEnd) {
1482      UPDATE_PROCESSED_VALUE(*iter);
1483
1484      if(aux==1) {
1485        exec_end = spu_read_decrementer();
1486        ReportMeasurement(LOAD_E, exec_start, exec_end);
1487        return DATA_LOAD_SINGLE_AUX(___CACHE_DOUBLE, idx,
                    _io_AuxAddress1, double);
1488      }
1489
1490      if(aux==2) {
1491        exec_end = spu_read_decrementer();
1492        ReportMeasurement(LOAD_E, exec_start, exec_end);
1493        return (double)DATA_LOAD_SINGLE_AUX(___CACHE_INT, idx,
                    _io_AuxAddress2, int);
1494      }
1495
1496      if(aux==3) {
```

```
1497        exec_end = spu_read_decrementer();
1498        ReportMeasurement(LOAD_E, exec_start, exec_end);
1499        return (double)DATA_LOAD_SINGLE_AUX(___CACHE_INT, idx,
                _io_AuxAddress3, int);
1500      }
1501    }
1502    else {
1503      request_message_t request;
1504      request.outer_level = _i_ActiveOuterLoopIdentifier;
1505      request.request_type = LOAD_AUX;
1506      request.region_number = _i_RegionIdentifier;
1507      request.spe_sid = _i_SPEIdentifier;
1508      request.owner_itr = *iter;
1509      request.aux = aux;
1510      request.index_a = idx;
1511      request.loop_level = _i_ActiveLoopLevel;
1512      request.io_array_type = _i_GlobalIOType;
1513
1514      if(_i_ActiveLoopLevel==INNER)
1515        request.level = _i_ActiveInnerLoopIdentifier;
1516
1517      if(_i_ActiveLoopLevel==OUTER)
1518        request.level = _i_ActiveOuterLoopIdentifier;
1519
1520
1521      __send_to_ppe(0x2115, 0, &request);
1522
1523      unsigned int buffer;
1524      buffer = COMListen(2);
1525
1526      double data;
1527
1528      if(buffer==BUFFER1) {
1529        if(aux==1)
1530        data = DATA_LOAD_SINGLE_AUX(___CACHE_DOUBLE, idx,
                _io_AuxAddress1, double);
1531
```

```
1532        if(aux==2)
1533          data = (double)DATA_LOAD_SINGLE_AUX(___CACHE_INT, idx,
                  _io_AuxAddress2, int);
1534
1535        if(aux==3)
1536          data = (double)DATA_LOAD_SINGLE_AUX(___CACHE_INT, idx,
                  _io_AuxAddress3, int);
1537
1538        UPDATE_PROCESSED_VALUE(*iter);
1539      }
1540
1541      if(buffer==BUFFER1) {
1542        if(aux==1)
1543          data = DATA_LOAD_SINGLE_AUX(___CACHE_DOUBLE, idx,
                  _io_AuxAddress1, double);
1544
1545        if(aux==2)
1546          data = (double)DATA_LOAD_SINGLE_AUX(___CACHE_INT, idx,
                  _io_AuxAddress2, int);
1547
1548        if(aux==3)
1549          data = (double)DATA_LOAD_SINGLE_AUX(___CACHE_INT, idx,
                  _io_AuxAddress3, int);
1550
1551        UPDATE_PROCESSED_VALUE(*iter);
1552      }
1553
1554      /*tell PPE that we are now done with this request*/
1555      ___send_to_ppe(0x2123, 0, &request);
1556
1557      exec_end = spu_read_decrementer();
1558      ReportMeasurement(LOAD_E, exec_start, exec_end);
1559      return data;
1560    }
1561 }
1562
1563 void Store_AUX(int aux, unsigned idx, double data) {
```

```
1564  ComCheck(2);
1565
1566  exec_start = spu_read_decrementer();
1567  if(idx >= _i_LoopStart && idx < _i_LoopEnd) {
1568    UPDATE_PROCESSED_VALUE(*iter);
1569    UPDATE_STORE_VALUE(INTERNAL_STORE, *iter);
1570
1571    if(aux==1)
1572      DATA_STORE_SINGLE_AUX(___CACHE_DOUBLE, idx,
           _io_AuxAddress1, double, data);
1573
1574    if(aux==2)
1575      DATA_STORE_SINGLE_AUX(___CACHE_INT, idx, _io_AuxAddress2,
           int, (int)data);
1576
1577    if(aux==3)
1578      DATA_STORE_SINGLE_AUX(___CACHE_INT, idx, _io_AuxAddress3,
           int, (int)data);
1579  }
1580  else {
1581    request_message_t request;
1582    request.outer_level = _i_ActiveOuterLoopIdentifier;
1583    request.request_type = STORE_AUX;
1584    request.region_number = _i_RegionIdentifier;
1585    request.spe_sid = _i_SPEIdentifier;
1586    request.owner_itr = *iter;
1587    request.aux = aux;
1588    request.index_a = idx;
1589    request.loop_level = _i_ActiveLoopLevel;
1590    request.data = data;
1591    request.io_array_type = _i_GlobalIOType;
1592
1593    if(_i_ActiveLoopLevel==INNER)
1594      request.level = _i_ActiveInnerLoopIdentifier;
1595
1596    if(_i_ActiveLoopLevel==OUTER)
1597      request.level = _i_ActiveOuterLoopIdentifier;
```

```
1598
1599      ___send_to_ppe(0x2115, 0, &request);
1600
1601      unsigned int buffer;
1602      buffer = COMListen(2);
1603
1604
1605      if(buffer==BUFFER1) {
1606        if(aux==1)
1607          DATA_STORE_SINGLE_AUX(___CACHE_DOUBLE, idx,
                _io_AuxAddress1, double, data);
1608
1609        if(aux==2)
1610          DATA_STORE_SINGLE_AUX(___CACHE_INT, idx, _io_AuxAddress2
                , int, (int)data);
1611
1612        if(aux==3)
1613          DATA_STORE_SINGLE_AUX(___CACHE_INT, idx, _io_AuxAddress3
                , int, (int)data);
1614
1615        UPDATE_PROCESSED_VALUE(*iter);
1616        UPDATE_STORE_VALUE(EXTERNAL_STORE, *iter);
1617      }
1618
1619      if(buffer==BUFFER2) {
1620        if(aux==1)
1621          DATA_STORE_SINGLE_AUX(___CACHE_DOUBLE, idx,
                _io_AuxAddress1, double, data);
1622
1623        if(aux==2)
1624          DATA_STORE_SINGLE_AUX(___CACHE_INT, idx, _io_AuxAddress2
                , int, (int)data);
1625
1626        if(aux==3)
1627          DATA_STORE_SINGLE_AUX(___CACHE_INT, idx, _io_AuxAddress3
                , int, (int)data);
1628
```

```
1629         UPDATE_PROCESSED_VALUE(*iter);
1630         UPDATE_STORE_VALUE(EXTERNAL_STORE, *iter);
1631       }
1632
1633       /*tell PPE that we are now done with this request*/
1634       __send_to_ppe(0x2123, 0, &request);
1635     }
1636
1637     exec_end = spu_read_decrementer();
1638     ReportMeasurement(STORE_E, exec_start, exec_end);
1639 }
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656 //AUX (2-D ARRAY)
       ////////////////////////////////////////////////
1657 double* Load_AUX_PTR2(int aux, unsigned idxA, unsigned idxB) {
1658     ComCheck(2);
1659     return DATA_PTR_DOUBLE_AUX(___CACHE_DOUBLE, idxA, idxB,
          _io_AuxAddress1, double);
1660
1661     exec_start = spu_read_decrementer();
1662     if((idxA >= _i_SizeBeginPrimary && idxA < _i_SizeEndPrimary)
          && (idxB >= _i_SizeBeginPrimary && idxB < _i_SizeEndPrimary
```

```
        )) {
1663
1664      UPDATE_PROCESSED_VALUE(*iter);
1665
1666      if(aux==1) {
1667        exec_end = spu_read_decrementer();
1668        ReportMeasurement(LOAD_E, exec_start, exec_end);
1669        return DATA_PTR_DOUBLE_AUX(___CACHE_DOUBLE, idxA, idxB,
              _io_AuxAddress1, double);
1670      }
1671
1672
1673      if(aux==2) {
1674        exec_end = spu_read_decrementer();
1675        ReportMeasurement(LOAD_E, exec_start, exec_end);
1676        return DATA_PTR_DOUBLE_AUX(___CACHE_DOUBLE, idxA, idxB,
              _io_AuxAddress2, double);
1677      }
1678
1679      if(aux==3) {
1680        exec_end = spu_read_decrementer();
1681        ReportMeasurement(LOAD_E, exec_start, exec_end);
1682        return (double *)DATA_PTR_DOUBLE_AUX(___CACHE_INT, idxA,
              idxB, _io_AuxAddress3, int);
1683      }
1684    }
1685    else {
1686      request_message_t request;
1687      request.outer_level = _i_ActiveOuterLoopIdentifier;
1688      request.request_type = LOAD_AUX;
1689      request.region_number = _i_RegionIdentifier;
1690      request.spe_sid = _i_SPEIdentifier;
1691      request.owner_itr = *iter;
1692      request.aux = aux;
1693      request.index_a = idxA;
1694      request.index_b = idxB;
1695      request.loop_level = _i_ActiveLoopLevel;
```

```
1696      request.io_array_type = _i_GlobalIOType;
1697
1698      if(_i_ActiveLoopLevel==INNER)
1699        request.level = _i_ActiveInnerLoopIdentifier;
1700
1701      if(_i_ActiveLoopLevel==OUTER)
1702        request.level = _i_ActiveOuterLoopIdentifier;
1703
1704      __send_to_ppe(0x2115, 0, &request);
1705
1706      unsigned int buffer;
1707      buffer = COMListen(2);
1708
1709      double *data;
1710
1711      if(buffer==BUFFER1) {
1712        if(aux==1)
1713          data = DATA_PTR_DOUBLE_AUX(___CACHE_DOUBLE, idxA, idxB,
                _io_AuxAddress1, double);
1714
1715        if(aux==2)
1716          return DATA_PTR_DOUBLE_AUX(___CACHE_DOUBLE, idxA, idxB,
                _io_AuxAddress2, double);
1717
1718        if(aux==3)
1719          data = (double *)DATA_PTR_DOUBLE_AUX(___CACHE_INT, idxA,
                idxB, _io_AuxAddress3, int);
1720
1721        UPDATE_PROCESSED_VALUE(*iter);
1722      }
1723
1724      if(buffer==BUFFER2) {
1725        if(aux==1)
1726          data = DATA_PTR_DOUBLE_AUX(___CACHE_DOUBLE, idxA, idxB,
                _io_AuxAddress1, double);
1727
1728        if(aux==2)
```

```
1729            data = DATA_PTR_DOUBLE_AUX(___CACHE_DOUBLE, idxA, idxB,
                    _io_AuxAddress2, double);
1730
1731          if(aux==3)
1732            data = (double *)DATA_PTR_DOUBLE_AUX(___CACHE_INT, idxA,
                     idxB, _io_AuxAddress3, int);
1733
1734          UPDATE_PROCESSED_VALUE(*iter);
1735        }
1736
1737        /*tell PPE that we are now done with this request*/
1738        __send_to_ppe(0x2123, 0, &request);
1739
1740        exec_end = spu_read_decrementer();
1741        ReportMeasurement(LOAD_E, exec_start, exec_end);
1742        return data;
1743      }
1744 }
1745
1746 double Load_AUX2(int aux, unsigned idxA, unsigned idxB) {
1747    ComCheck(2);
1748
1749    exec_start = spu_read_decrementer();
1750    if((idxA >= _i_SizeBeginPrimary && idxA < _i_SizeEndPrimary)
          && (idxB >= _i_LoopStart && idxB < _i_LoopEnd)) {
1751      UPDATE_PROCESSED_VALUE(*iter);
1752
1753      if(aux==1) {
1754        exec_end = spu_read_decrementer();
1755        ReportMeasurement(LOAD_E, exec_start, exec_end);
1756        return DATA_LOAD_DOUBLE_AUX(___CACHE_DOUBLE, idxA, idxB,
                    _io_AuxAddress1, double);
1757      }
1758
1759      if(aux==2) {
1760        exec_end = spu_read_decrementer();
1761        ReportMeasurement(LOAD_E, exec_start, exec_end);
```

```
1762        return DATA_LOAD_DOUBLE_AUX(___CACHE_DOUBLE, idxA, idxB,
                _io_AuxAddress2, double);
1763      }
1764 //     if(aux==2)
1765 //       return (double)DATA_LOAD_DOUBLE_AUX(___CACHE_INT, idxA,
       idxB, _io_AuxAddress2, int);
1766
1767      if(aux==3) {
1768        exec_end = spu_read_decrementer();
1769        ReportMeasurement(LOAD_E, exec_start, exec_end);
1770        return (double)DATA_LOAD_DOUBLE_AUX(___CACHE_INT, idxA,
                idxB, _io_AuxAddress3, int);
1771      }
1772    }
1773    else {
1774      request_message_t request;
1775      request.outer_level = _i_ActiveOuterLoopIdentifier;
1776      request.request_type = LOAD_AUX;
1777      request.region_number = _i_RegionIdentifier;
1778      request.spe_sid = _i_SPEIdentifier;
1779      request.owner_itr = *iter;
1780      request.aux = aux;
1781      request.index_a = idxA;
1782      request.index_b = idxB;
1783      request.loop_level = _i_ActiveLoopLevel;
1784      request.io_array_type = _i_GlobalIOType;
1785
1786      if(_i_ActiveLoopLevel==INNER)
1787        request.level = _i_ActiveInnerLoopIdentifier;
1788
1789      if(_i_ActiveLoopLevel==OUTER)
1790        request.level = _i_ActiveOuterLoopIdentifier;
1791
1792      __send_to_ppe(0x2115, 0, &request);
1793
1794      unsigned int buffer;
1795      buffer = COMListen(2);
```

```
1796
1797      double data;
1798
1799      if(buffer==BUFFER1) {
1800        if(aux==1)
1801          data = DATA_LOAD_DOUBLE_AUX(___CACHE_DOUBLE, idxA, idxB,
                  _io_AuxAddress1, double);
1802
1803        if(aux==2)
1804          data = DATA_LOAD_DOUBLE_AUX(___CACHE_DOUBLE, idxA, idxB,
                  _io_AuxAddress2, double);
1805
1806 //       if(aux==2)
1807 //         data = (double)DATA_LOAD_DOUBLE_AUX(___CACHE_INT, idxA
        , idxB, _io_AuxAddress2, int);
1808
1809        if(aux==3)
1810          data = (double)DATA_LOAD_DOUBLE_AUX(___CACHE_INT, idxA,
                  idxB, _io_AuxAddress3, int);
1811
1812        UPDATE_PROCESSED_VALUE(*iter);
1813      }
1814
1815      if(buffer==BUFFER1) {
1816        if(aux==1)
1817          data = DATA_LOAD_DOUBLE_AUX(___CACHE_DOUBLE, idxA, idxB,
                  _io_AuxAddress1, double);
1818
1819        if(aux==2)
1820          data = DATA_LOAD_DOUBLE_AUX(___CACHE_DOUBLE, idxA, idxB,
                  _io_AuxAddress2, double);
1821
1822 //       if(aux==2)
1823 //         data = (double)DATA_LOAD_DOUBLE_AUX(___CACHE_INT, idxA
        , idxB, _io_AuxAddress2, int);
1824
1825        if(aux==3)
```

```
1826            data = (double)DATA_LOAD_DOUBLE_AUX(___CACHE_INT, idxA,
                    idxB, _io_AuxAddress3, int);
1827
1828          UPDATE_PROCESSED_VALUE(*iter);
1829        }
1830
1831        /*tell PPE that we are now done with this request*/
1832        __send_to_ppe(0x2123, 0, &request);
1833
1834        exec_end = spu_read_decrementer();
1835        ReportMeasurement(LOAD_E, exec_start, exec_end);
1836        return data;
1837      }
1838 }
1839
1840 void Store_AUX2(int aux, unsigned idxA, unsigned idxB, double
        data) {
1841      ComCheck(2);
1842
1843      exec_start = spu_read_decrementer();
1844      if((idxA >= _i_SizeBeginPrimary && idxA < _i_SizeEndPrimary)
           && (idxB >= _i_LoopStart && idxB < _i_LoopEnd)) {
1845        UPDATE_PROCESSED_VALUE(*iter);
1846        UPDATE_STORE_VALUE(INTERNAL_STORE, *iter);
1847
1848        if(aux==1)
1849          DATA_STORE_DOUBLE_AUX(___CACHE_DOUBLE, idxA, idxB,
                  _p_io_AuxAddress1, double, data);
1850
1851        if(aux==2)
1852          DATA_STORE_DOUBLE_AUX(___CACHE_DOUBLE, idxA, idxB,
                  _p_io_AuxAddress2, double, data);
1853
1854 //       if(aux==2)
1855 //         DATA_STORE_DOUBLE_AUX(___CACHE_INT, idxA, idxB,
        _io_AuxAddress2, int, (int)data);
1856
```

```
1857      if(aux==3)
1858        DATA_STORE_DOUBLE_AUX(___CACHE_INT, idxA, idxB,
              _io_AuxAddress2, int, (int)data);
1859    }
1860    else {
1861      request_message_t request;
1862      request.outer_level = _i_ActiveOuterLoopIdentifier;
1863      request.request_type = STORE_AUX;
1864      request.region_number = _i_RegionIdentifier;
1865      request.spe_sid = _i_SPEIdentifier;
1866      request.owner_itr = *iter;
1867      request.aux = aux;
1868      request.index_a = idxA;
1869      request.index_b = idxB;
1870      request.loop_level = _i_ActiveLoopLevel;
1871      request.data = data;
1872      request.io_array_type = _i_GlobalIOType;
1873
1874      if(_i_ActiveLoopLevel==INNER)
1875        request.level = _i_ActiveInnerLoopIdentifier;
1876
1877      if(_i_ActiveLoopLevel==OUTER)
1878        request.level = _i_ActiveOuterLoopIdentifier;
1879
1880      __send_to_ppe(0x2115, 0, &request);
1881
1882      unsigned int buffer;
1883      buffer = COMListen(2);
1884
1885      if(buffer==BUFFER1) {
1886        if(aux==1)
1887          DATA_STORE_DOUBLE_AUX(___CACHE_DOUBLE, idxA, idxB,
                _p_io_AuxAddress1, double, data);
1888
1889
1890        if(aux==2)
1891          DATA_STORE_DOUBLE_AUX(___CACHE_DOUBLE, idxA, idxB,
```

```
                _p_io_AuxAddress2, double, data);
1892
1893 //        if(aux==2)
1894 //          DATA_STORE_DOUBLE_AUX(___CACHE_INT, idxA, idxB,
      _io_AuxAddress2, int, (int)data);
1895
1896          if(aux==3)
1897            DATA_STORE_DOUBLE_AUX(___CACHE_INT, idxA, idxB,
                  _io_AuxAddress3, int, (int)data);
1898
1899          UPDATE_PROCESSED_VALUE(*iter);
1900          UPDATE_STORE_VALUE(EXTERNAL_STORE, *iter);
1901       }
1902
1903       if(buffer==BUFFER2) {
1904          if(aux==1)
1905            DATA_STORE_DOUBLE_AUX(___CACHE_DOUBLE, idxA, idxB,
                  _p_io_AuxAddress2, double, data);
1906
1907          if(aux==2)
1908            DATA_STORE_DOUBLE_AUX(___CACHE_DOUBLE, idxA, idxB,
                  _p_io_AuxAddress2, double, data);
1909
1910 //        if(aux==2)
1911 //          DATA_STORE_DOUBLE_AUX(___CACHE_INT, idxA, idxB,
      _io_AuxAddress2, int, (int)data);
1912
1913          if(aux==3)
1914            DATA_STORE_DOUBLE_AUX(___CACHE_INT, idxA, idxB,
                  _io_AuxAddress3, int, (int)data);
1915
1916          UPDATE_PROCESSED_VALUE(*iter);
1917          UPDATE_STORE_VALUE(EXTERNAL_STORE, *iter);
1918       }
1919
1920       /*tell PPE that we are now done with this request*/
1921       ___send_to_ppe(0x2123, 0, &request);
```

```
1922    }
1923
1924    exec_end = spu_read_decrementer();
1925    ReportMeasurement(STORE_E, exec_start, exec_end);
1926 }
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950 //AUX(n) with specifier ////////////////////////////////////
1951
1952 double Load_SA_AUX(int aux, double* arr, unsigned idx) {
1953    ComCheck(2);
1954
1955    exec_start = spu_read_decrementer();
1956    if(idx >= _i_LoopStart && idx < _i_LoopEnd) {
1957      UPDATE_PROCESSED_VALUE(*iter);
1958
```

```
1959      exec_end = spu_read_decrementer();
1960      ReportMeasurement(LOAD_E, exec_start, exec_end);
1961      return arr[idx];
1962    }
1963    else {
1964      request_message_t request;
1965      request.outer_level = _i_ActiveOuterLoopIdentifier;
1966      request.request_type = LOAD_AUX;
1967      request.region_number = _i_RegionIdentifier;
1968      request.spe_sid = _i_SPEIdentifier;
1969      request.owner_itr = *iter;
1970      request.aux = aux;
1971      request.index_a = idx;
1972      request.loop_level = _i_ActiveLoopLevel;
1973      request.io_array_type = _i_GlobalIOType;
1974
1975      if(_i_ActiveLoopLevel==INNER)
1976        request.level = _i_ActiveInnerLoopIdentifier;
1977
1978      if(_i_ActiveLoopLevel==OUTER)
1979        request.level = _i_ActiveOuterLoopIdentifier;
1980
1981      __send_to_ppe(0x2115, 0, &request);
1982
1983      unsigned int buffer;
1984      buffer = COMListen(2);
1985
1986      double data;
1987
1988      if(buffer==BUFFER1) {
1989        data = arr[idx];
1990 //       if(aux==1)
1991 //       data = DATA_LOAD_SINGLE_AUX(___CACHE_DOUBLE, idx,
       _io_AuxAddress1, double);
1992 //
1993 //       if(aux==2)
1994 //         data = (double)DATA_LOAD_SINGLE_AUX(___CACHE_INT, idx,
```

```
        _io_AuxAddress2, int);
1995 //
1996 //        if(aux==3)
1997 //          data = (double)DATA_LOAD_SINGLE_AUX(___CACHE_INT, idx,
        _io_AuxAddress3, int);
1998        UPDATE_PROCESSED_VALUE(*iter);
1999      }
2000
2001 //     if(buffer==BUFFER1) {
2002 //        if(aux==1)
2003 //          data = DATA_LOAD_SINGLE_AUX(___CACHE_DOUBLE, idx,
       _io_AuxAddress1, double);
2004 //
2005 //        if(aux==2)
2006 //          data = (double)DATA_LOAD_SINGLE_AUX(___CACHE_INT, idx,
        _io_AuxAddress2, int);
2007 //
2008 //        if(aux==3)
2009 //          data = (double)DATA_LOAD_SINGLE_AUX(___CACHE_INT, idx,
        _io_AuxAddress3, int);
2010 //     }
2011
2012      /*tell PPE that we are now done with this request*/
2013      __send_to_ppe(0x2123, 0, &request);
2014
2015      exec_end = spu_read_decrementer();
2016      ReportMeasurement(LOAD_E, exec_start, exec_end);
2017      return data;
2018    }
2019 }
2020
2021 void Store_SA_AUX(int aux,  double* arr, unsigned idx, double
       data) {
2022    ComCheck(2);
2023    exec_start = spu_read_decrementer();
2024    if(idx >= _i_LoopStart && idx < _i_LoopEnd) {
2025      UPDATE_PROCESSED_VALUE(*iter);
```

```
2026      UPDATE_STORE_VALUE(INTERNAL_STORE, *iter);
2027
2028      arr[idx]=data;
2029    }
2030    else {
2031      request_message_t request;
2032      request.outer_level = _i_ActiveOuterLoopIdentifier;
2033      request.request_type = STORE_AUX;
2034      request.region_number = _i_RegionIdentifier;
2035      request.spe_sid = _i_SPEIdentifier;
2036      request.owner_itr = *iter;
2037      request.aux = aux;
2038      request.index_a = idx;
2039      request.loop_level = _i_ActiveLoopLevel;
2040      request.data = data;
2041      request.io_array_type = _i_GlobalIOType;
2042
2043      if(_i_ActiveLoopLevel==INNER)
2044        request.level = _i_ActiveInnerLoopIdentifier;
2045
2046      if(_i_ActiveLoopLevel==OUTER)
2047        request.level = _i_ActiveOuterLoopIdentifier;
2048
2049      __send_to_ppe(0x2115, 0, &request);
2050
2051      unsigned int buffer;
2052      buffer = COMListen(2);
2053
2054
2055      if(buffer==BUFFER1) {
2056        arr[idx]=data;
2057 //       if(aux==1)
2058 //         DATA_STORE_SINGLE_AUX(___CACHE_DOUBLE, idx,
       _io_AuxAddress1, double, data);
2059 //
2060 //       if(aux==2)
2061 //         DATA_STORE_SINGLE_AUX(___CACHE_INT, idx,
```

```
        _io_AuxAddress2, int, (int)data);
2062 //
2063 //        if(aux==3)
2064 //          DATA_STORE_SINGLE_AUX(___CACHE_INT, idx,
        _io_AuxAddress3, int, (int)data);
2065
2066        UPDATE_PROCESSED_VALUE(*iter);
2067        UPDATE_STORE_VALUE(EXTERNAL_STORE, *iter);
2068      }
2069
2070 //     if(buffer==BUFFER2) {
2071 //        if(aux==1)
2072 //          DATA_STORE_SINGLE_AUX(___CACHE_DOUBLE, idx,
        _io_AuxAddress1, double, data);
2073 //
2074 //        if(aux==2)
2075 //          DATA_STORE_SINGLE_AUX(___CACHE_INT, idx,
        _io_AuxAddress2, int, (int)data);
2076 //
2077 //        if(aux==3)
2078 //          DATA_STORE_SINGLE_AUX(___CACHE_INT, idx,
        _io_AuxAddress3, int, (int)data);
2079 //     }
2080
2081      /*tell PPE that we are now done with this request*/
2082      __send_to_ppe(0x2123, 0, &request);
2083    }
2084    exec_end = spu_read_decrementer();
2085    ReportMeasurement(STORE_E, exec_start, exec_end);
2086 }
2087
2088
2089
2090
2091 #endif
```

C.3 array.h

```
1 #include "kernel.h"
2
3 void Array2D_double_copy() {
4   unsigned int remainder = _i_SizeEndSecondary & 3;        /* N
      mod 4; */
5     unsigned int i=0;
6     unsigned int j=0;
7   double *Bi;
8   double *Ai;
9
10  OuterLoop(0, _i_SizeBeginPrimary, _i_SizeEndPrimary, i);
11    for (i=_i_SizeBeginPrimary; i<_i_SizeEndPrimary; i++) {
12
13        Bi = Load_AUX_PTR(0, i);  //  double **B
14        Ai = Load_AUX_PTR(1, i);  //  double **A
15
16
17
18 //         InnerLoop(0, _i_SizeBeginPrimary, remainder, j);
19 //         for (j=_i_SizeBeginPrimary; j<remainder; j++) {
20 //             Store_SA_AUX(0, Bi, j, Load_SA_AUX(1, Ai, j));
21 //
22 //         }
23 //         InnerLoopEnd;
24
25        InnerLoop(1, remainder, _i_SizeEndSecondary, j);
26        for (j=remainder; j<_i_SizeEndSecondary; j+=4) {
27          Store_SA_AUX(1, Ai, j, Load_SA_AUX(0, Bi, j));
28          Store_SA_AUX(1, Ai, j+1, Load_SA_AUX(0, Bi, j+1));
29          Store_SA_AUX(1, Ai, j+2, Load_SA_AUX(0, Bi, j+2));
30          Store_SA_AUX(1, Ai, j+3, Load_SA_AUX(0, Bi, j+3));
31        }
32        InnerLoopEnd;
33
```

```
34      }
35      OuterLoopEnd;
36 }
```

C.4 fft.h

```
1 #include "kernel.h"
2 #include <simdmath.h>
3
4 #define PI  3.1415926535897932
5
6 /*
      ------------------------------------------------------------------------
      */
7
8 //static int int_log2(int n);
9 //
10 //double FFT_num_flops(int N)
11 //{
12 //
13 //      double Nd = (double) N;
14 //      double logN = (double) int_log2(N);
15 //
16 //      return (5.0*Nd-2)*logN + 2*(Nd+1);
17 //}
18 //
19 //static int int_log2 (int n)
20 //{
21 //    int k = 1;
22 //    int log = 0;
23 //    for(/*k=1*/; k < n; k *= 2, log++);
24 //    if (n != (1 << log))
25 //    {
26 //      printf("FFT: Data length is not a power of 2!: %d ",n);
27 //      exit(1);
28 //    }
```

```
29 //    return log;
30 //}
31
32 static void FFT_transform_internal(int direction) {
33     /* bit reverse the input data for decimation in time
          algorithm */
34
35     double *data = Load_DPTR(0);
36
37     unsigned int bit;
38     unsigned int dual = 1;
39
40     double w_real;
41     double w_imag;
42     unsigned int a;
43     unsigned int b;
44
45     double theta;
46     double s;
47     double t;
48     double s2;
49
50     int i;
51   int j;
52
53   double wd_real;
54   double wd_imag;
55
56   double tmp_real;
57     double tmp_imag;
58
59     double z1_real;
60   double z1_imag;
61
62
63     FFT_bitreverse();
64    printf("FFT_bitreverse␣done␣%i␣\n", _i_SPEIdentifier);
```

```
65
66      /* apply fft recursion */
67      /* this loop executed int_log2(N) times */
68      OuterLoop(0, Lower_IPC, Upper_IPC, bit);
69    for(bit = Lower_IPC; bit < Upper_IPC; bit++, dual *= 2) {
70      w_real = 1.0;
71          w_imag = 0.0;
72
73          theta = 2.0 * direction * PI / (2.0 * (double) dual);
74          s = sin(theta);
75          t = sin(theta / 2.0);
76          s2 = 2.0 * t * t;
77
78
79      InnerLoop(0, _i_SizeBeginPrimary, _i_SizeEndPrimary, bit);
80          for(a=_i_SizeBeginPrimary, b = _i_SizeBeginPrimary; b <
                _i_SizeEndPrimary; b += 2 * dual) {
81
82            i = b;
83            j = (b + dual);
84
85            wd_real = LoadD(data, j);
86            wd_imag = LoadD(data, j+1);
87
88            StoreD(data, j,    LoadD(data, i+1) - wd_real);
89            StoreD(data, j+1,    LoadD(data, i+1) - wd_imag);
90            StoreD(data, i,    LoadD(data, i) + wd_real);
91            StoreD(data, i+1,    LoadD(data, i+1) + wd_imag);
92
93          }
94          InnerLoopEnd;
95
96
97        /* a = 1 .. (dual-1) */
98        InnerLoop(1, 1, dual, a);
99          for(a = 1; a < dual; a++) {
100           /* trignometric recurrence for w-> exp(i theta) w */
```

```
101             tmp_real = w_real - s * w_imag - s2 * w_real;
102             tmp_imag = w_imag + s * w_real - s2 * w_imag;
103             w_real = tmp_real;
104             w_imag = tmp_imag;
105           }
106           InnerLoopEnd;
107
108
109           InnerLoop(2, _i_SizeBeginPrimary, _i_SizeEndPrimary, b);
110           for(b = _i_SizeBeginPrimary; b < _i_SizeEndPrimary; b +=
                  2 * dual) {
111         i = 2*(b + a);
112             j = 2*(b + a + dual);
113
114             z1_real = LoadD(data,j);
115           z1_imag = LoadD(data,j+1);
116
117             wd_real = w_real * z1_real - w_imag * z1_imag;
118             wd_imag = w_real * z1_imag + w_imag * z1_real;
119
120         StoreD(data, j,   LoadD(data,i) - wd_real);
121         StoreD(data, j+1,   LoadD(data,i+1) - wd_imag);
122         StoreD(data, i,   LoadD(data, i) + wd_real);
123         StoreD(data, i+1,   LoadD(data, i+1) + wd_imag);
124           }
125           InnerLoopEnd;
126
127   }
128   OuterLoopEnd;
129 }
130
131
132 int FFT_bitreverse() {
133   int N = _i_SizeEndPrimary; //_i_SizeEndPrimary;
134   double *data = Load_AUX_PTR(1, 0);// Load_DPTR(0);
135
136     /* This is the Goldrader bit-reversal algorithm */
```

```
137   unsigned int n=N/2;
138     unsigned int nm1 = n-1;
139     unsigned int i=0;
140     unsigned int j=0;
141     int ii;
142     int jj;
143     unsigned int k;
144     double tmp_real;
145     double tmp_imag;
146
147
148     OuterLoop(0, 0, nm1, i);
149     for(i=0; i < nm1; i++) {
150
151         /*int ii = 2*i; */
152       ii = i << 1;
153
154         /*int jj = 2*j; */
155         jj = j << 1;
156
157         /* int k = n / 2 ; */
158         k = n >> 1;
159
160         if (i < j) {
161           tmp_real = Load_AUX(1, ii); //LoadD(data, ii);
162           tmp_imag = Load_AUX(1, ii+1); //LoadD(data, ii+1);
163           Store_AUX(1, ii,  Load_AUX(1, jj));
164           Store_AUX(1, ii+1,  Load_AUX(1, jj+1));
165           Store_AUX(1, jj,  tmp_real);
166           Store_AUX(1, jj+1,  tmp_imag);
167         }
168
169         while (k <= j && (k!=0 && j!=0)) {
170           /*j = j - k ; */
171           j -= k;
172
173           /*k = k / 2 ;  */
```

```
174             k >>= 1 ;
175           }
176
177       j += k ;
178       }
179       OuterLoopEnd;
180
181       return 0;
182 }
183
184 void FFT_inverse() {
185   int N = _i_SizeEndPrimary;
186   double *data = DATA_PTR_PRIMARY(___CACHE_DOUBLE, 0);
187
188     int n = N/2;
189     double norm = 0.0;
190     unsigned int i;
191
192     FFT_transform_internal(+1);
193
194     /* Normalize */
195     norm=1/((double) n);
196
197   OuterLoop(0, _i_SizeBeginPrimary, _i_SizeEndPrimary, i);
198   for(i=_i_SizeBeginPrimary; i<_i_SizeEndPrimary; i++) {
199     StoreD(data, i, LoadD(data, i) * norm);
200   }
201   OuterLoopEnd;
202 }
```

C.5 lu.h

```
1 #include "kernel.h"
2
3 void LU_copy_matrix() {
4   double **A = (double **)Load_AUX_PTR2(1, 0, 0);
```

```
5   double **lu = (double **)Load_AUX_PTR2(2, 0, 0);
6
7     unsigned int i;
8     unsigned int j;
9   double data;
10
11  OuterLoop(0, Lower_IPC, Upper_IPC, i);
12    for(i=Lower_IPC; i<Upper_IPC; i++) {
13        InnerLoop(0, _i_SizeBeginPrimary, _i_SizeEndPrimary, i);
14        for(j=_i_SizeBeginPrimary; j<_i_SizeEndPrimary; j++) {
15            Store_DDA(lu, i, j, Load_DDA(A, i, j));
16        }
17        InnerLoopEnd;
18    }
19    OuterLoopEnd;
20 }
21
22
23 int LU_factor() {
24   double **A = (double **)Load_AUX_PTR2(1, 0, 0);
25   printf("ok\n");
26   int *pivot = (int *)Load_AUX_PTR(3, 0);
27
28     unsigned int minMN; // =  M < N ? M : N;
29     unsigned int j = 0;
30
31   double *Aii;
32     double *Aj;
33     double AiiJ;
34
35     double ab;
36
37     int jj;
38     double recp;
39     double t;
40
41     unsigned int i;
```

```
42      unsigned int jp;
43    unsigned int k;
44
45    unsigned int ii;
46
47    OuterLoop(0, Lower_IPC, Upper_IPC, j);
48      for(j=Lower_IPC; j<Upper_IPC; j++) {
49
50          /* find pivot in column j and  test for singularity. */
51          jp=j;
52          t = fabs(Load_AUX2(1, j, j));
53
54          InnerLoop(0, j+1, _i_SizeEndPrimary, i);
55          for(i=j+1; i<_i_SizeEndPrimary; i++) {
56              ab = fabs(Load_AUX2(1, i, j));
57
58              if(ab > t) {
59                  jp = i;
60                  t = ab;
61              }
62          }
63          InnerLoopEnd;
64
65          StoreI(pivot, j, jp);
66
67          if(jp != j) {
68              /* swap rows j and jp */
69              double *tA = Load_AUX_PTR2(1, j, 0); //A
70              Store_DDA(A, j, 0, Load_DDA(A, jp, 0));
71              Store_DDA(A, jp, 0, tA[0]);
72          }
73
74          if(j<_i_SizeEndPrimary-1) {                 /* compute
                elements j+1:M of jth column  */
75              /* note A(j,j), was A(jp,p) previously which was */
76              /* guarranteed not to be zero (Label #1)         */
77
```

```
78              recp =  1.0 / Load_DDA(A, j, j);
79
80
81              InnerLoop(1, j+1, _i_SizeEndPrimary, k);
82              for (k=j+1; k<_i_SizeEndPrimary; k++) {
83                  Store_DDA(A, k, j, Load_DDA(A, j, j) * recp);
84              }
85          InnerLoopEnd;
86
87          }
88
89
90          if (j < minMN-1) {
91              /* rank-1 update to trailing submatrix:   E = E - x*
                    y; */
92              /* E is the region A(j+1:M, j+1:N) */
93              /* x is the column vector A(j+1:M,j) */
94              /* y is row vector A(j,j+1:N)        */
95
96              InnerLoop(3, j+1, _i_SizeEndPrimary, ii);
97              for (ii=j+1; ii<_i_SizeEndPrimary; ii++) {
98                  Aii = Load_AUX_PTR(1, ii);// Load_DPTR(ii); //A
99                  Aj = Load_AUX_PTR(1, j); //Load_DPTR(j); //A
100                 AiiJ = LoadD(Aii, j);
101
102                 InnerLoop(4, j+1, _i_SizeEndSecondary, ii);
103                 for (jj=j+1; jj<_i_SizeEndSecondary; jj++) {
104           StoreD(Aii, jj, LoadD(Aii, jj) - AiiJ * LoadD(Aj, jj))
                  ;
105                 }
106         InnerLoopEnd;
107
108             }
109             InnerLoopEnd;
110         }
111     }
112 OuterLoopEnd;
```

```
113
114     return 0;
115 }
```

C.6 sor.h

```
 1 #include "kernel.h"
 2
 3 void SOR_execute() {
 4   unsigned int M_size_start = _i_SizeBeginPrimary;
 5   unsigned int M_size_end = _i_SizeEndPrimary;
 6
 7
 8   unsigned int N_size_start = _i_SizeBeginSecondary;
 9   unsigned int N_size_end = _i_SizeEndSecondary;
10
11   if(N_size_start==0) {
12     N_size_start++;
13   }
14
15   if(M_size_start==0) {
16     M_size_start++;
17   }
18
19   double omega = 1.25;
20
21     double omega_over_four = omega * 0.25;
22     double one_minus_omega = 1.0 - omega;
23
24     unsigned int p;
25     unsigned int i;
26     unsigned int j;
27
28     double *Gi;
29     double *Gim1;
30     double *Gip1;
```

```
31
32
33   OuterLoop(0, Lower_IPC, Upper_IPC, p);
34     for(p=Lower_IPC; p<Upper_IPC; p++) {
35
36       InnerLoop(0, M_size_start, M_size_end, i);
37         for(i=M_size_start; i<M_size_end; i++) {
38
39             Gi = Load_DPTR(i);
40             Gim1 = Load_DPTR(i-1);
41             Gip1 = Load_DPTR(i+1);
42
43             InnerLoop(1, N_size_start, N_size_end, j);
44             for(j=N_size_start; j<N_size_end; j++) {
45               StoreD(Gi, j, omega_over_four * (LoadD(Gim1, j) +
46                                 LoadD(Gip1, j) +
47                                 LoadD(Gi, j-1) +
48                                 LoadD(Gi, j+1)
49                                 +
50                                 one_minus_omega *
51                                 LoadD(Gi, j)));
52             }
53             InnerLoopEnd;
54
55         }
56         InnerLoopEnd;
57
58     }
59     OuterLoopEnd;
60 }
```

C.7 sparse.h

```
1 #include "kernel.h"
2
3 /* multiple iterations used to make kernel have roughly
```

```
4       same granulairty as other Scimark kernels. */
5   //
6   //double SparseCompRow_num_flops(int N, int nz, int
        num_iterations)
7   //{
8   //    /* Note that if nz does not divide N evenly, then the
9   //       actual number of nonzeros used is adjusted slightly.
10  //    */
11  //    int actual_nz = (nz/N) * N;
12  //    return ((double)actual_nz) * 2.0 * ((double)
        num_iterations);
13  //}
14
15
16  /* computes  a matrix-vector multiply with a sparse matrix
17      held in compress-row format.  If the size of the matrix
18      in MxN with nz nonzeros, then the val[] is the nz nonzeros,
19      with its ith entry in column col[i].  The integer vector row
            []
20      is of size M+1 and row[i] points to the begining of the
21      ith row in col[].
22  */
23
24  void SparseCompRow_matmult() {
25      unsigned int p;
26      unsigned int r;
27      unsigned int i;
28
29    double sum;
30
31
32  //  double *val;     //AUX1
33  //  int *row;        //AUX2
34  //  int *col;      //AUX3
35
36    double *y = DATA_PTR_SECONDARY(___CACHE_DOUBLE, 0);     //OUTPUT
37
```

```
38   double *x = DATA_PTR_PRIMARY(___CACHE_DOUBLE, 0);     //INPUT
39
40   unsigned int rowR;
41   unsigned int rowRp1;
42
43   OuterLoop(0, Lower_IPC, Upper_IPC, p);
44     for(p=Lower_IPC; p<Upper_IPC; p++) {
45
46       InnerLoop(0, _i_SizeBeginPrimary, _i_SizeEndPrimary, r);
47         for (r=_i_SizeBeginPrimary; r<_i_SizeEndPrimary; r++) {
48             sum = 0.0;
49
50       rowR = Load_AUX(2, r);
51       rowRp1 = Load_AUX(2, r+1);
52
53             InnerLoop(1, rowR, rowRp1, i);
54             for(i=rowR; i<rowRp1; i++) {
55         sum = sum + LoadD(x, Load_AUX(3, i)) * Load_AUX(1, i);
56             }
57             InnerLoopEnd;
58
59       StoreD(y, r, sum);
60         }
61     InnerLoopEnd;
62
63     }
64     OuterLoopEnd;
65 }
```

Appendix D

Additional Processor Results

Figure D.1: CPI results for `SPR_SM` application with L-API transformations.

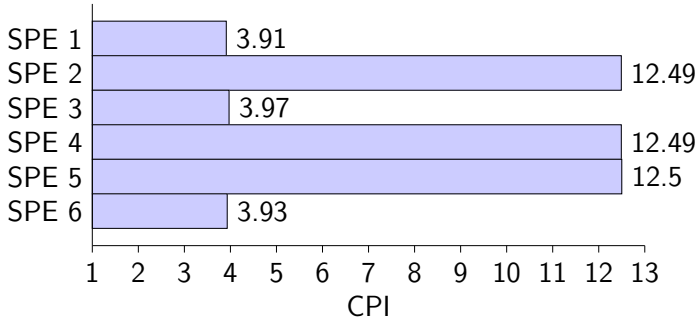


Figure D.2: CPI results for `SPR_LG` application with L-API transformations.

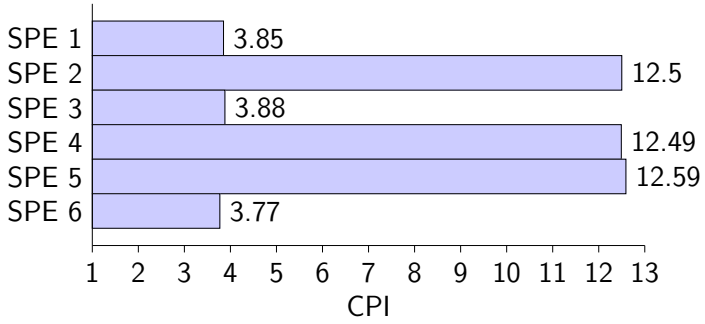


Figure D.3: CPI results for `SOR_SM` application with L-API transformations.

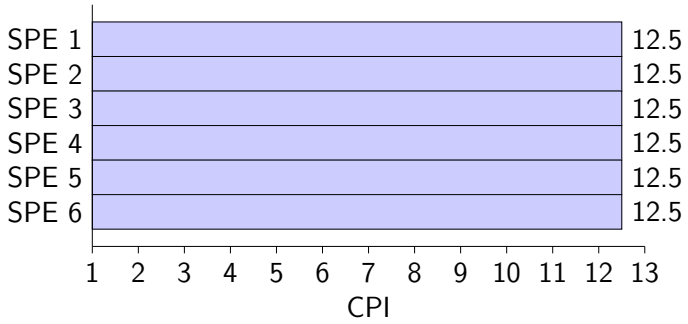


Figure D.4: CPI results for `SOR_LG` application with L-API transformations.

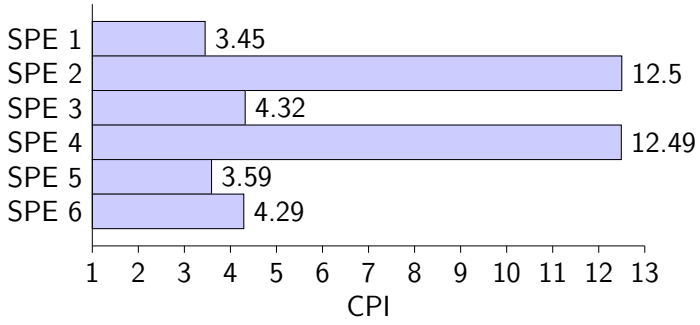


Figure D.5: CPI results for `LUCPYMX_SM` application with L-API transformations.

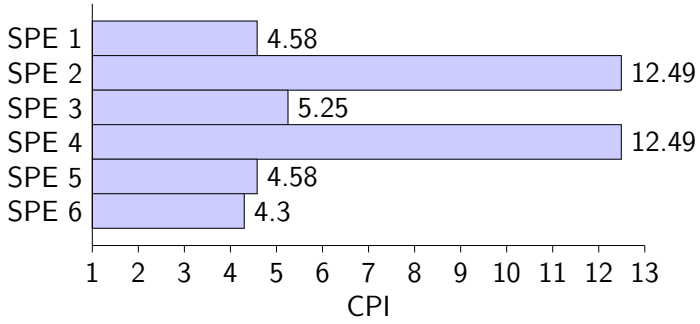


Figure D.6: CPI results for `LUCPYMX_LG` application with L-API transformations.

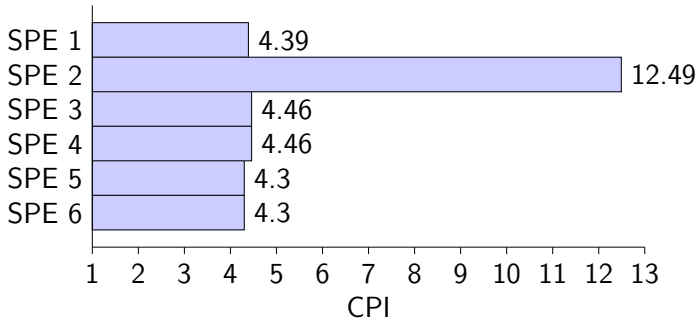


Figure D.7: CPI results for `ARR_SM` application with L-API transformations.

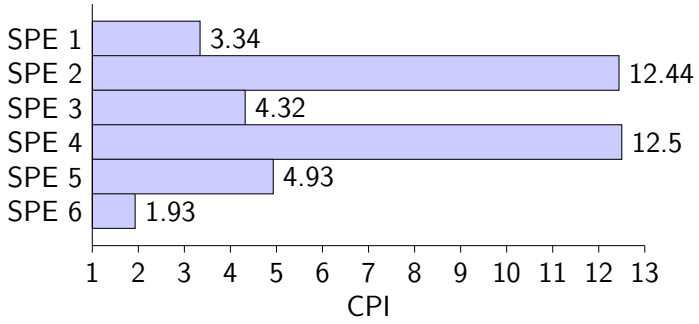


Figure D.8: CPI results for `ARR_MM` application with L-API transformations

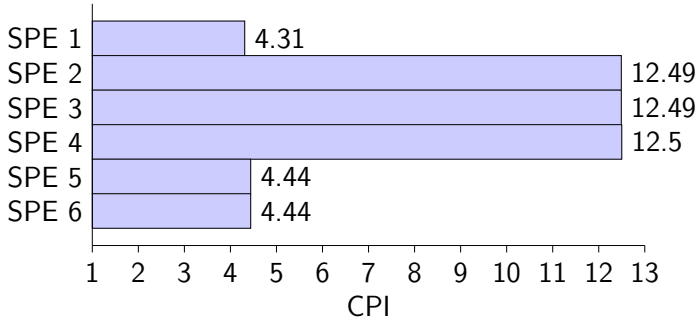


Figure D.9: CPI results for `ARR_LG` application with L-API transformations.

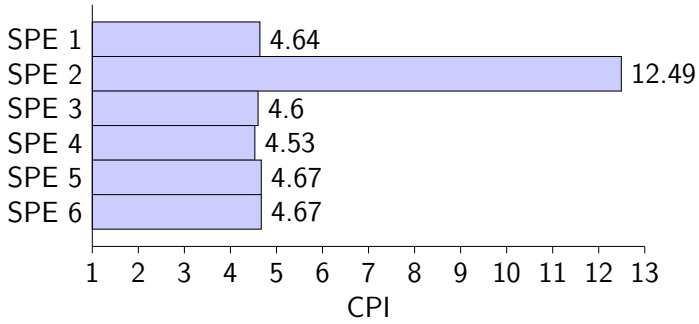


Figure D.10: CPI results for `FFT_SM` application with L-API transformations.

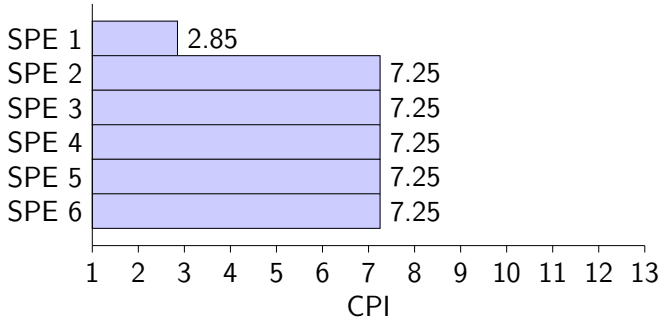
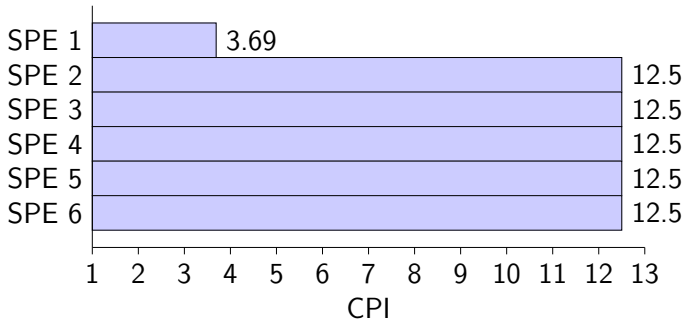


Figure D.11: CPI results for `FFT_LG` application with L-API transformations.



Figures D.1, D.2, D.3, D.4, D.5, D.6, D.7, D.8, D.9, D.10 and D.11 shows the CPI results that are composed from the SPU pipeline components which are FX2 (EVEN): Logical and integer arithmetic; SHUF (ODD): Shuffle, quad rotate/shift, mask; FX3 (EVEN): Element rotate/shuffle; LS (ODD): Load/store, hint; BR (ODD): Branch; SPR (ODD): Channel and SPR moves; LNOP (ODD): NOP; NOP (EVEN): NOP; FXB (EVEN): Special byte ops; FP6 (EVEN): SP floating point; FP7 (EVEN): Integer mult, float conversion and FPD (EVEN): DP floating point, see Section 4.5.2.

The SPE's architecture is designed for fast vector processing and does not particularly accelerate traditional code streams such as branching and if-statements. There are distinct architectural differences between PPE and SPE such as no caches or virtual memory, no scalar unit (every SPU operating on 128-bit vectors) and two distinct pipelines on the SPE (see Chapter 4) correspond to instructions with mathematical operations to utilise the even pipeline and the remaining instruction types operate on the odd pipeline.

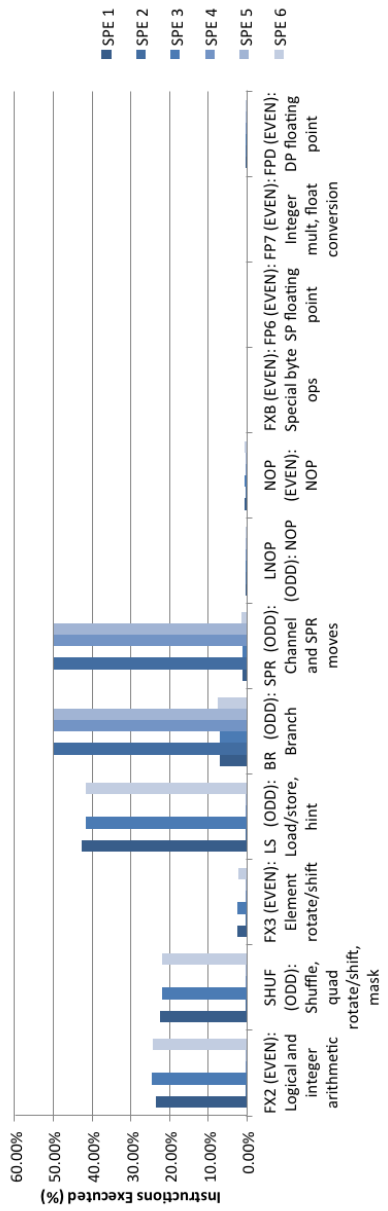


Figure D.12: `SPR_SM` Instructions Usage Percentage per SPE pipeline unit.

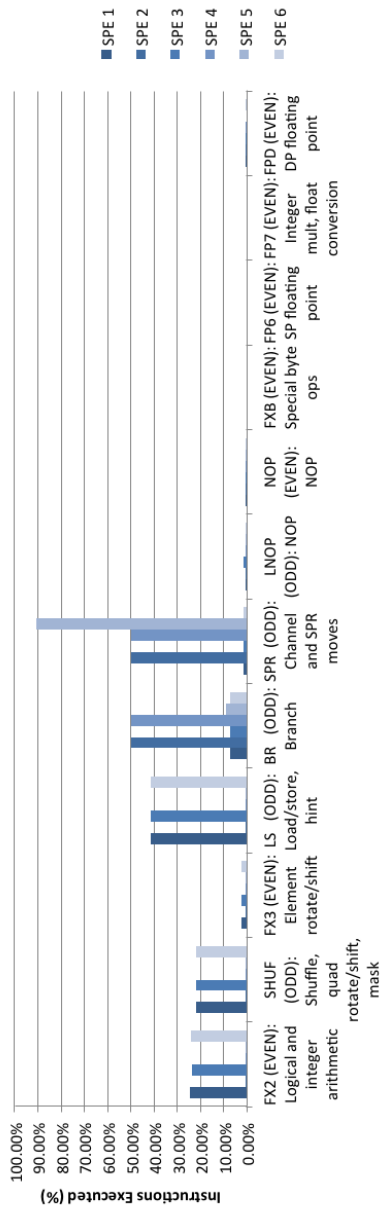


Figure D.13: `SPR_LG` Instructions Usage Percentage per SPE pipeline unit.

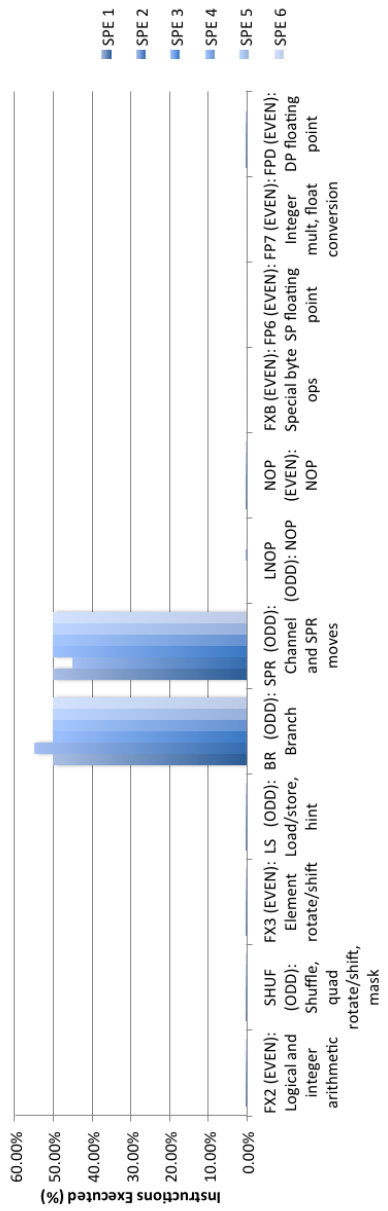


Figure D.14: `SOR_SM` Instructions Usage Percentage per SPE pipeline unit.

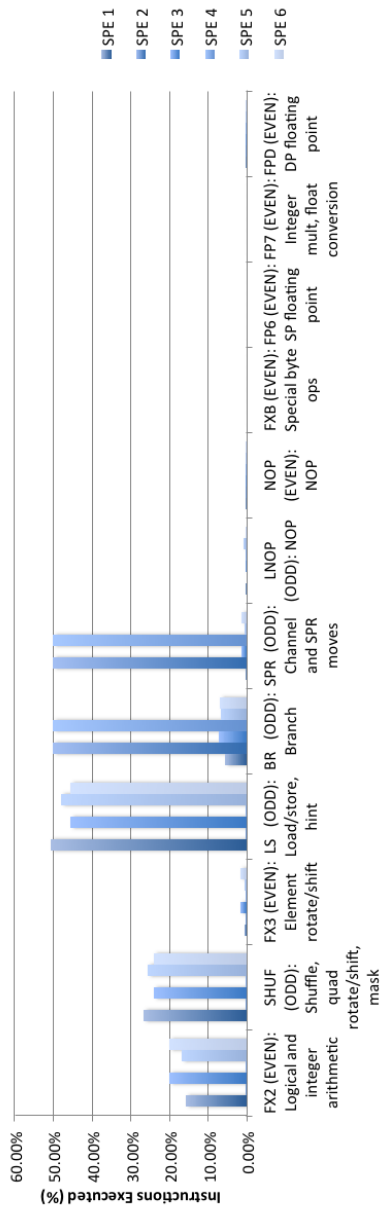


Figure D.15: `SOR_LG` Instructions Usage Percentage per SPE pipeline unit.

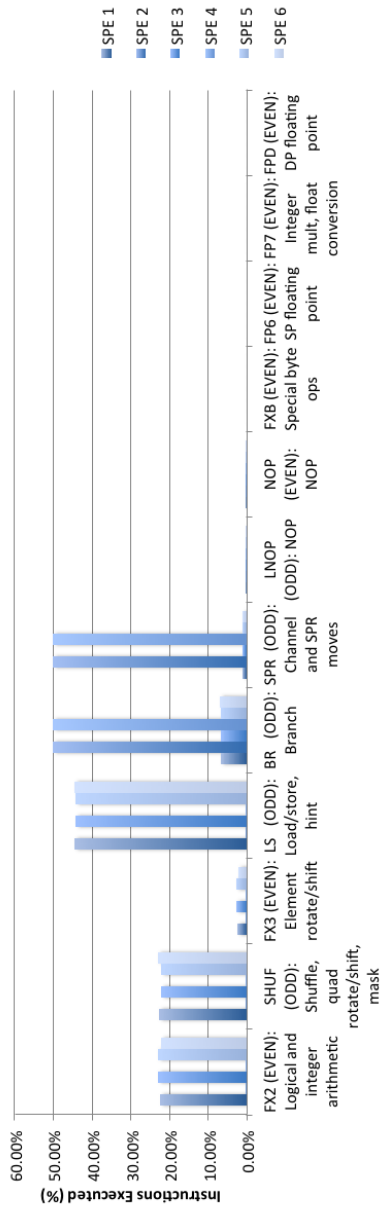


Figure D.16: `LUCPYMX_SM` Instructions Usage Percentage per SPE pipeline unit.

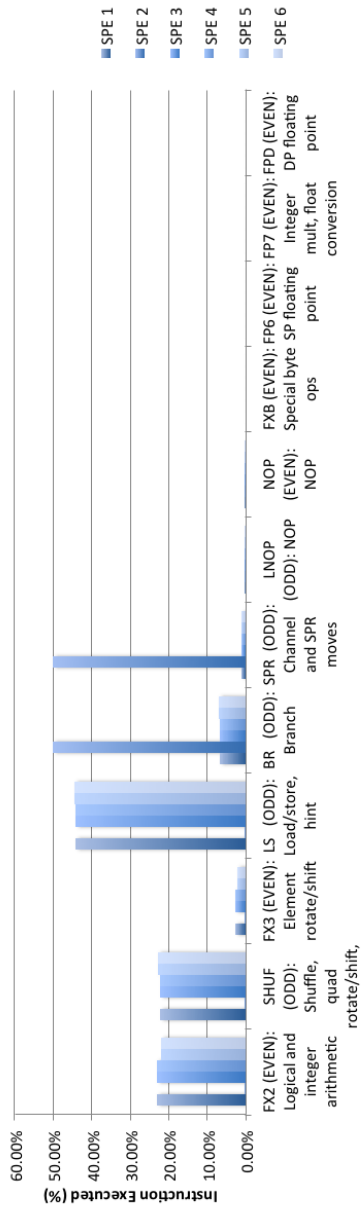


Figure D.17: `LUCPYMX_SM` Instructions Usage Percentage per SPE pipeline unit.

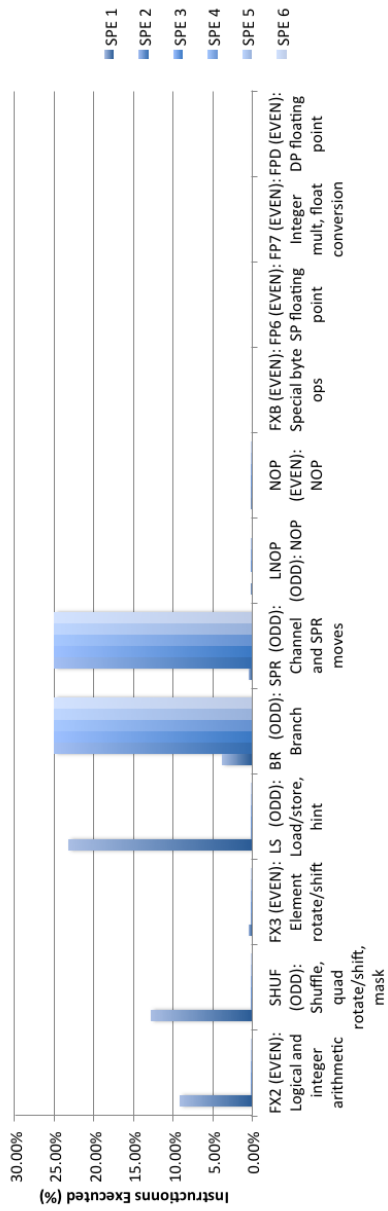


Figure D.18: `FFT_SM` Instructions Usage Percentage per SPE pipeline unit.

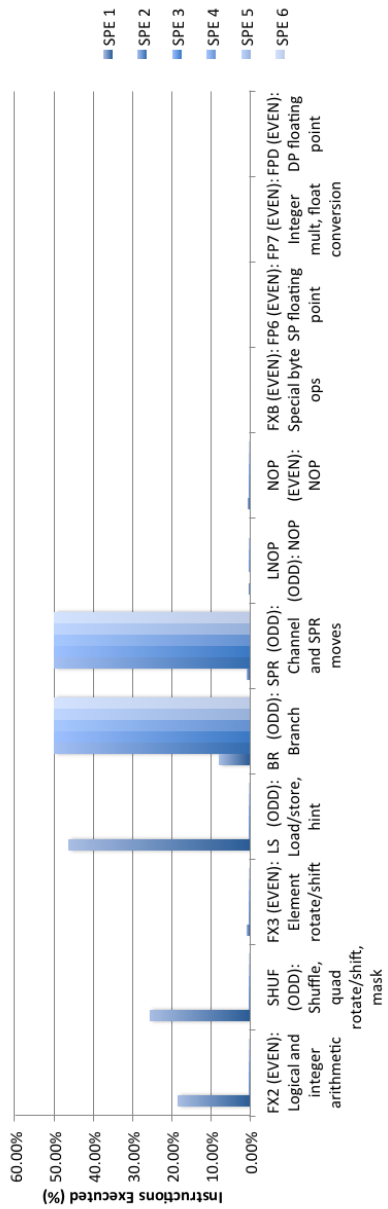


Figure D.19: `FFT_LG` Instructions Usage Percentage per SPE pipeline unit.

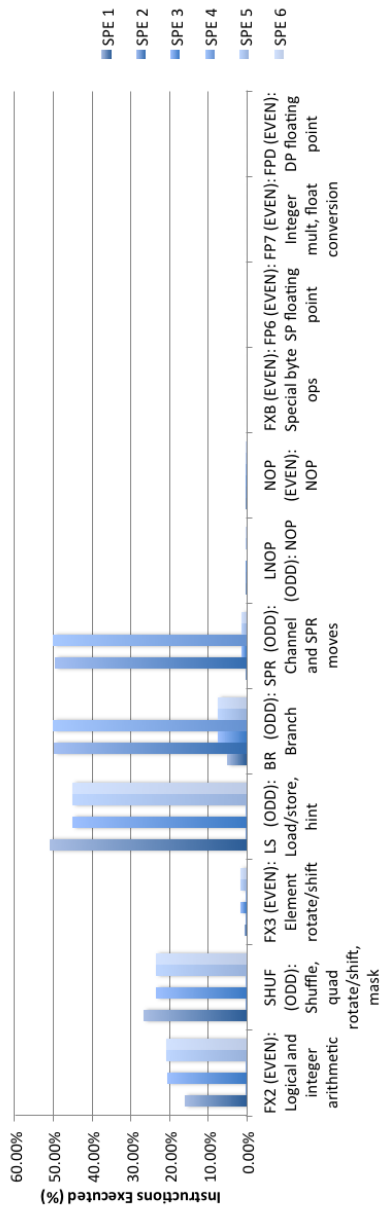


Figure D.20: `ARR_SM` Instructions Usage Percentage per SPE pipeline unit.

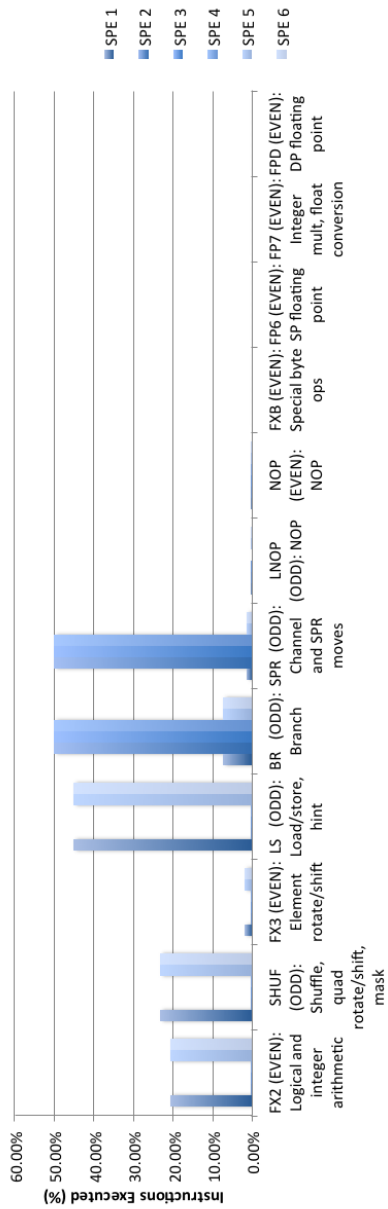


Figure D.21: `ARR_MM` Instructions Usage Percentage per SPE pipeline unit.

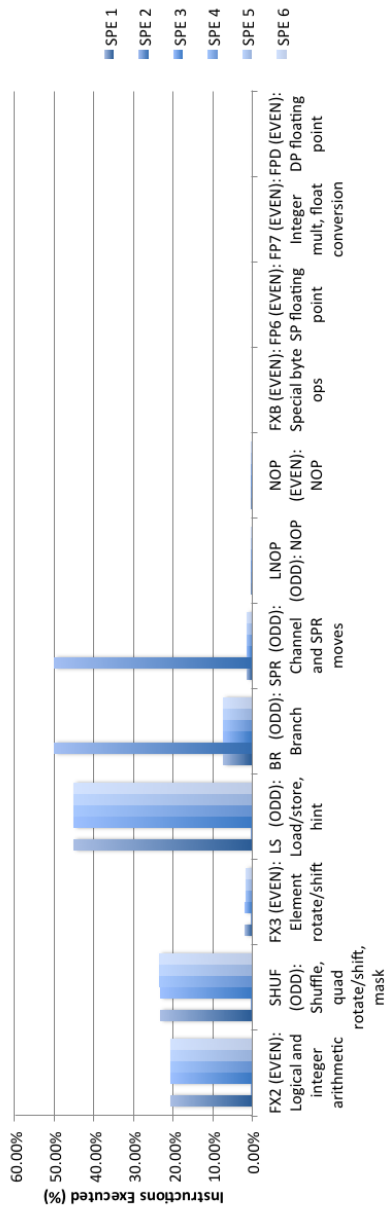


Figure D.22: `ARR_LG` Instructions Usage Percentage per SPE pipeline unit.

The results shown in the above figures show the impact of the SPE L-API kernels on the SPUs pipeline. Both branching and load-store functional units consume a large number of executed instructions which equate for a greater percentage of CPU usage that simultaneously resulted in abundant use of if-statements. No branching optimisation was applied throughout the framework. A performance issue caused by an increase of load-store instructions would infer the LS being accessed by both SPUs and PPU (external access). By allowing external SPU and PPU access would result commands to execute in specific order. Moreover, each SPU does guarantee in-order program order but with a distributed memory access results in a weakly consistent paradigm. This weakly consistent is prevalent as the L-API forces datum checking (load and store constructs) on external SPUs. Hence, the increased use of the LS functional unit on all SPUs.

The remaining functional units seem to be under-utilised or not required for the complete execution for each application. Comparing Figure D.12 and D.13 show a similar trend whereby, FX2 and BR functional units are relatively similar in execution percentage but interestingly, a higher consumption of SPR (channel operations for data moving to or from) instructions for SPE 5 (`SPR_LG`). An indicator for increased execution time is due to SPEs waiting for datum to consume or submit.

Results from Figure D.14 and D.15 highlight a significant difference such that, BR and SPR functional units exhibit a very high usage however, increasing the input dataset show a similar pattern with an increased instruction set usage from FX2 and SHUF units. Again, the odd-pipeline is significantly in demand from the L-API kernel. Figure D.16, D.17, D.18, D.19, D.20, D.21 and D.22 show a similar outcome with LS, BR and SPR units on the odd-pipeline utilised more aggressively than the even-pipeline.

CPI figure that are shown at the start of this appendix encapsulates the complete instruction class timings from the results shown in pipeline figures (see above) and present an overall performance result. Applications with

a reduced input dataset have significant overheads that affect the overall performance. Whilst increasing the dataset, the performance is somewhat equal to a lower input dataset with a similar overhead. The results also show, increasing the dataset, the overall overhead is similar or lower than a lower input dataset.

Appendix E

Single SPE Processor Results without L-API

Figure E.1: Single SPE Results for all applications without L-API transformations.

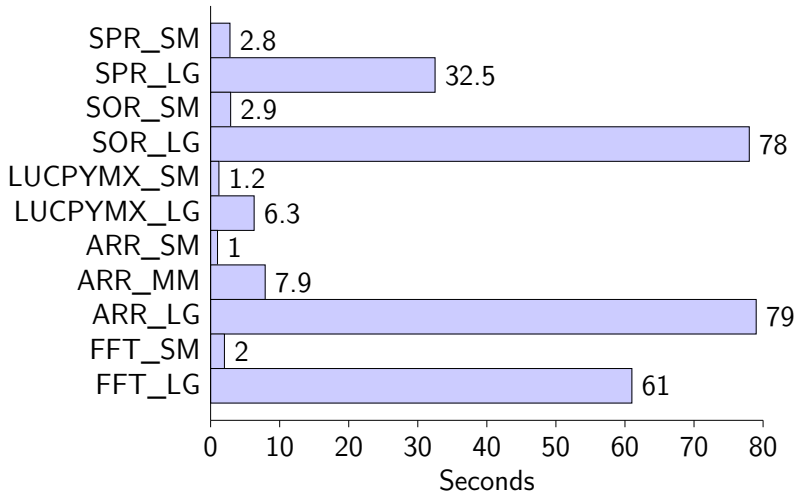


Figure E.1 illustrates the results for single SPE execution without L-API transformations. Each bar in the above figure includes the complete execution time by an SPE per application.