# Using Program Behaviour to Exploit Heterogeneous Multi-Core Processors


Ross McIlroy

Department of Computing Science
Faculty of Information and Mathematical Sciences

University of Glasgow

A thesis submitted for the degree of

*Doctor of Philosophy*

April 2010

# Abstract

Multi-core CPU architectures have become prevalent in recent years. A number of multi-core CPUs consist of not only multiple processing cores, but multiple different types of processing cores, each with different capabilities and specialisations. These *heterogeneous multi-core architectures* (HMAs) can deliver exceptional performance; however, they are notoriously difficult to program effectively.

This dissertation investigates the feasibility of ameliorating many of the difficulties encountered in application development on HMA processors, by employing a behaviour-aware runtime system. This runtime system provides applications with the illusion of executing on a homogeneous architecture, by presenting a homogeneous virtual machine interface. The runtime system uses knowledge of a program's execution behaviour, gained through explicit code annotations, static analysis or runtime monitoring, to inform its resource allocation and scheduling decisions, such that the application makes best use of the HMA's heterogeneous processing cores. The goal of this runtime system is to enable non-specialist application developers to write applications that can exploit an HMA, without the developer requiring in-depth knowledge of the HMA's design.

This dissertation describes the development of a Java runtime system, called Hera-JVM, aimed at investigating this premise. Hera-JVM supports the execution of unmodified Java applications on both processing core types of the heterogeneous IBM Cell processor. An application's threads of execution can be transparently migrated between the Cell's different core types by Hera-JVM, without requiring the application's involvement. A number of real-world Java benchmarks are executed across both of the Cell's core types, to evaluate the efficacy of abstracting a heterogeneous architecture behind a homogeneous virtual machine.

By characterising the performance of each of the Cell processor's core types under different program behaviours, a set of influential program behaviour characteristics is

uncovered. A set of code annotations are presented, which enable program code to be tagged with these behaviour characteristics, enabling a runtime system to track a program's behaviour throughout its execution. This information is fed into a cost function, which Hera-JVM uses to automatically estimate whether the executing program's threads of execution would benefit from being migrated to a different core type, given their current behaviour characteristics. The use of history, hysteresis and trend tracking, by this cost function, is explored as a means of increasing its stability and limiting detrimental thread migrations. The effectiveness of a number of different migration strategies is also investigated under real-world Java benchmarks, with the most effective found to be a strategy that can target code, such that a thread is migrated whenever it executes this code.

This dissertation also investigates the use of runtime monitoring to enable a runtime system to automatically infer a program's behaviour characteristics, without the need for explicit code annotations. A lightweight runtime behaviour monitoring system is developed, and its effectiveness at choosing the most appropriate core type on which to execute a set of real-world Java benchmarks is examined. Combining explicit behaviour characteristic annotations with those characteristics which are monitored at runtime is also explored.

Finally, an initial investigation is performed into the use of behaviour characteristics to improve application performance under a different type of heterogeneous architecture, specifically, a non-uniform memory access (NUMA) architecture. *Thread teams* are proposed as a method of automatically clustering communicating threads onto the same NUMA node, thereby reducing data access overheads. Evaluation of this approach shows that it is effective at improving application performance, if the application's threads can be partitioned across the available NUMA nodes of a system.

The findings of this work demonstrate that a runtime system with a homogeneous virtual machine interface can reduce the challenge of application development for HMA processors, whilst still being able to exploit such a processor by taking program behaviour into account.

# Acknowledgements

I would like to thank my PhD supervisor, Professor Joseph Sventek, for all his enthusiasm, guidance and support over the years. He was a constant source of inspiration and I have benefited greatly from his experience and wisdom.

I would also like to thank the various second supervisors I have been lucky enough to work with over the course of this PhD: in chronological order, Dr Peter Dickman, Professor Nigel Topham and Dr Wim Vanderbauwhede. They were invaluable in providing guidance from a different perspective.

To my external and internal examiners, Dr Steven Hand and Dr Colin Perkins, for their interest in this work and for taking the time to study this thesis extensively.

I would like to acknowledge the Carnegie Trust for the Universities of Scotland, who funded this work. I would also like to thank Microsoft Research Cambridge for providing the NUMA server used in this work.

Many thanks to my friends, who helped keep me (relatively) sane through the last four years with encouragement and understanding. A special thanks to my fellow ENDS research group cohort - Alexandros Koliousis, Martin Ellis, Oliver Sharma and Stephen Strowes - for their daily conversations and shared coffee addiction. I am also indebted to Rachel Lo, Craig MacDonald, Stephen Strowes and my parents, Alan and Liz McIlroy, for proof-reading and commenting on various drafts of this dissertation.

Last, but by no means least, I am eternally grateful to my parents, Liz and Alan, and my *wee* sister, Kim, for their love, support and encouragement throughout my life. I cannot thank them enough for all they have done for me.

# Contents


**1 Introduction** **1**

1.1 Thesis Statement . . . 3
1.2 Contributions . . . 3
1.3 Publications . . . 4
1.4 Outline . . . 5

**2 Heterogeneous Multi-Core Processors** **8**

2.1 The Case for Heterogeneous Multi-Core Architectures . . . 9
2.2 The History of Heterogeneous Processors . . . 11
2.3 Heterogeneous Processors in Commodity Systems . . . 14
2.3.1 Graphics Processing Units as Heterogeneous Cores . . . 14
2.3.2 Many-Core CPUs . . . 16
2.4 Summary . . . 18

**3 Related Work** **19**

3.1 Parallel Programming . . . 20
3.2 Abstraction of Heterogeneous Programming Environments . . . 22
3.2.1 Programming Models and Compilers . . . 22
3.2.2 Runtime Systems and Operating Systems . . . 32
3.3 Thread Scheduling on HMAs . . . 36
3.4 Managing Non-Uniform Memory . . . 41
3.5 Summary . . . 46

**4 Abstracting Heterogeneity using Behaviour Characteristics** **47**

4.1 Aspects of Processor Heterogeneity . . . 49
4.1.1 Heterogeneous Processing Resources . . . 49

4.1.2 Heterogeneous Memory Hierarchy . . . . . 50
4.1.3 Heterogeneous Inter-Core Communication . . . . . 50
4.1.4 Summary . . . . . 51
4.2 Behaviour Characteristics . . . . . 51
4.2.1 Processing Requirement Characteristics . . . . . 52
4.2.2 Execution Behaviour Characteristics . . . . . 53
4.2.3 Thread Communication Characteristics . . . . . 55
4.3 Tagging Mechanisms . . . . . 57
4.3.1 Explicit Annotations . . . . . 57
4.3.2 Source Code Analysis Tools . . . . . 61
4.3.3 Runtime Monitoring . . . . . 62
4.4 Costing Behaviour . . . . . 63
4.5 Discussion . . . . . 65

**5 Hera-JVM: A Runtime System for Heterogeneous Architectures 67**
5.1 The Cell Processor . . . . . 68
5.2 Hera-JVM Design Decisions . . . . . 71
5.3 Executing Java Code on the SPE Cores . . . . . 73
5.3.1 Overview . . . . . 73
5.3.2 Local Variables and Stack Management . . . . . 76
5.3.3 Software Caching of Heap Objects . . . . . 78
5.3.4 Invocation and Caching of Methods . . . . . 86
5.3.5 Scheduling and Thread Switching . . . . . 91
5.3.6 System Calls and Native Methods . . . . . 95
5.4 Migration between Core Types . . . . . 96
5.4.1 Migration Mechanism . . . . . 96
5.4.2 Scanning a Migrated Thread's Stack . . . . . 97
5.5 Experimental Analysis . . . . . 98
5.5.1 Experimental Setup . . . . . 99
5.5.2 Micro-Benchmarks . . . . . 100
5.5.3 Real World Benchmarks . . . . . 106
5.6 Discussion . . . . . 116

**6 Migration Based Upon Behaviour Annotations 120**
6.1 Maintaining Per-Thread Behaviour Knowledge . . . 121
6.1.1 Set of Tracked Behaviour Annotations . . . 121
6.1.2 Tracking Thread Behaviour at Runtime . . . 123
6.2 A Cost Function-Based Migration Policy . . . 124
6.3 Implementing Behaviour Based Thread Migration . . . 128
6.3.1 Evaluating a Thread's Cost . . . 129
6.3.2 Triggering Thread Migration . . . 131
6.3.3 Combining Thread Costing and Migration Triggering . . . 135
6.4 Experimental Analysis . . . 135
6.4.1 Experimental Setup . . . 136
6.4.2 Two Phase Synthetic Benchmark . . . 136
6.4.3 XML Parsing Synthetic Benchmark . . . 141
6.4.4 Real World Benchmarks . . . 149
6.5 Summary . . . 154

**7 Monitoring Program Behaviour at Runtime 156**
7.1 Monitoring Execution of Different Bytecode Types . . . 157
7.1.1 Scoring Methods . . . 157
7.1.2 Monitoring a Thread's Behaviour . . . 160
7.2 Migration Decisions . . . 161
7.2.1 The Cost Function . . . 161
7.2.2 Combining Annotations with Runtime Monitoring . . . 163
7.2.3 Triggering Thread Migration . . . 163
7.3 Experimental Analysis . . . 164
7.3.1 XML Parsing Synthetic Benchmark . . . 165
7.3.2 Real World Benchmarks . . . 170
7.3.3 Combining Annotations and Runtime Monitoring . . . 173
7.4 Summary . . . 175

**8 Inter-Thread Communication on NUMA Architectures 177**
8.1 Non-Uniform Memory Access Architectures . . . 178
8.2 Abstracting NUMA Node Placement Decisions . . . 182
8.3 Scheduling based upon Thread Teams . . . 183
8.3.1 Making Hera-JVM NUMA Aware . . . 184
8.3.2 Applying a Cost to a Thread's Placement . . . 186
8.3.3 Scheduling Threads with Per-Node Costs . . . 189
8.4 Experimental Analysis . . . 192
8.4.1 Experimental Setup . . . 192
8.4.2 Scalability . . . 193
8.4.3 Multiple Teams per Thread . . . 198
8.5 Summary . . . 202

**9 Conclusion and Future Work 204**
9.1 Thesis Statement Revisited . . . 204
9.2 Contributions . . . 208
9.3 Future Work . . . 210
9.3.1 Other Heterogeneous Architectures . . . 211
9.3.2 Other Behaviour Characteristics . . . 211
9.3.3 Tagging Data with Behaviour Characteristics . . . 212
9.3.4 Inferring Behaviour Characteristics Through Static Analysis . . . 213
9.3.5 Low-Level Abstraction of HMAs . . . 214
9.3.6 Summary . . . 214

**References 216**

# List of Figures


2.1 A simplified example showing the performance that can be achieved from different core layouts on the same area of a processor's silicon die. . . . 10

4.1 A typical thread communication pattern. . . . . 57
4.2 Using thread teams to place threads on a NUMA system. . . . . 65

5.1 The architecture of the Cell processor. . . . . 69
5.2 An SPE core's memory subsystem. . . . . 70
5.3 The structure of Hera-JVM. . . . . 72
5.4 Outline of the SPE data cache. . . . . 79
5.5 The code cache data structures. . . . . 88
5.6 TIB layout for super-class methods. . . . . 89
5.7 Performance difference between SPE and PPE cores for fundamental Java operations in the Java Grande micro-benchmarks. . . . . 100
5.8 Heap data access performance. . . . . 102
5.9 The effect of a thread's data working set on performance. . . . . 104
5.10 The effect of a thread's code working set on performance. . . . . 105
5.11 Performance comparison between benchmarks running on a single SPE core, and running on the single PPE core. . . . . 108
5.12 Percentage of cycles spent executing different classes of machine instructions on SPE. . . . . 109
5.13 The effect of varying the proportion of local memory reserved for use by the data and code caches (Java Grande and SpecJVM:monte_carlo benchmarks). . . . . 111

5.14 The effect of varying the proportion of local memory reserved for use by the data and code caches (SpecJVM Scimark benchmarks). . . . . 112
5.15 The effect of varying the proportion of local memory reserved for use by the data and code caches (remaining SpecJVM and Dacapo benchmarks).113
5.16 Performance with a per-benchmark optimal code / data cache ratio. . . 115
5.17 Scalability of the Java Grande Parallel benchmarks. . . . . 117
5.18 Scalability of the SpecJVM scimark benchmarks. . . . . 117
5.19 Scalability of the remaining SpecJVM, Dacapo and mandelbrot benchmarks. . . . . 117
5.20 Performance comparison between benchmarks running on all 6 SPE cores and running on the single PPE core. . . . . 118

6.1 Migration with Hysteresis. . . . . 127
6.2 An example of thread behaviour sampling at timer ticks. . . . . 131
6.3 Targeted migration example. . . . . 134
6.4 Speedup of the *two phase* benchmark as annotation placement is varied. 139
6.5 Speedup of the *two phase* benchmark as the phase length is varied. . . . 141
6.6 Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 0$. . . . . 145
6.7 Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 0.2$. . . . . 145
6.8 Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 0.4$. . . . . 145
6.9 Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 0.6$. . . . . 146
6.10 Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 0.8$. . . . . 146
6.11 Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 1$. . . . . 146
6.12 Comparing the interaction between the $\beta$ and $\gamma$. . . . . 147
6.13 Comparing the interaction between the $\alpha$ and $\gamma$. . . . . 147
6.14 Results for the targeted migration strategy when history, hysteresis and trend tracking are enabled and disabled in the cost function. . . . . 149
6.15 Speedup for the XML Parsing benchmark under different migration strategies. . . . . 150
6.16 Performance of real world benchmarks, which have been annotated with their behaviour characteristics, when executed under Hera-JVM. . . . . 152
6.17 Overhead of tracking behaviour annotations. . . . . 153

7.1 The structure of a method's score, related to its code structure. . . . . 159

7.2 Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 0$ . . . . . 166
7.3 Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 0.2$ . . . . . 166
7.4 Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 0.4$ . . . . . 166
7.5 Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 0.6$ . . . . . 167
7.6 Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 0.8$ . . . . . 167
7.7 Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 1$ . . . . . 167
7.8 Comparing the interaction between the $\beta$ and $\gamma$. . . . . 169
7.9 Comparing the interaction between the $\alpha$ and $\gamma$. . . . . 169
7.10 Comparing runtime monitoring and behaviour annotations for the XML Parsing benchmark. . . . . 170
7.11 Performance of real world benchmarks running under Hera-JVM, when using behaviour runtime monitoring. . . . . 171
7.12 Overhead of runtime monitoring for real world benchmarks. . . . . 172
7.13 Performance of the Hera-JVM when runtime monitoring and annotation behaviour information are combined. . . . . 174
7.14 Overhead of runtime monitoring with annotations approach. . . . . 174

8.1 The NUMA node layout of a 4x4 core AMD Opteron system. . . . . 179
8.2 The effect of NUMA on inter-thread communication. . . . . 181
8.3 Using thread teams to place threads on a NUMA system. . . . . 188
8.4 Scaling performance of mandelbrot benchmark on a NUMA system . . . 195
8.5 HyperTransport inter-core data traffic and L3 cache misses for mandelbrot benchmark on NUMA system. . . . . 197
8.6 The performance of the mandelbrot benchmark on a NUMA architecture as the number of thread teams assigned to each thread is varied. . . . . 199
8.7 Example of reducing the clusterablity of thread team assignments by shuffling thread / team assignments. . . . . 200
8.8 The performance of the mandelbrot benchmark on a NUMA architecture as its thread team clusterability is varied. . . . . 201

# List of Tables

6.1 Costs associated with each core type. . . . . 125
6.2 Migration policy parameters. . . . . 128
6.3 Behaviour-based migration strategies implemented in Hera-JVM. . . . . 135
6.4 Average execution time of the workload methods on each core type. . . 137
6.5 Execution time and migration count for the *two phase* benchmark under different annotation placements. . . . . 138
6.6 Migration policy parameters. . . . . 143
6.7 Behaviour characteristic annotations added to each benchmark. . . . . 151

7.1 Costs associated with each core type. . . . . 162

# Listings

4.1 Example of Behaviour Characteristic Annotations. . . . . 60
6.1 *Two phase* synthetic benchmark psuedo-code. . . . . 137
8.1 NUMA inter-thread communication micro-benchmark pseudo-code. . . . 180
8.2 Pseudo-code of algorithm used to calculate the preferred NUMA node order of a thread. . . . . 187
8.3 Pseudo-code of Hera-JVM scheduling algorithm. The additions made to support NUMA aware scheduling are shown in red. . . . . 190

# Glossary

**ABI** Application Binary Interface - the low-level interface between an application program and the operating system.

**API** Application Programming Interface - a standard interface through which applications can make use of a service that is provided by another application or software library.

**CIL** Common Intermediate Language - the bytecode format that is employed by Microsoft's .NET framework.

**CMP** Chip Multi-Processor - a central processing unit which consists of multiple processing cores on a single processor chip. Also known as a multi-core processor.

**CPU** Central Processing Unit - the part of a computer system that executes a computer program's instructions.

**DMA** Direct Memory Access - a feature in modern computer systems that enables hardware subsystems to access memory independently from the central processing unit.

**GPU** Graphics Processing Unit - a specialised processor that is designed to enable 3D graphics operations to be offloaded from the CPU. Due to their high performance in data-parallel floating point computations, GPUs are being employed for non-graphics related computations. Such GPUs are known as general purpose GPUs (GPGPUs).

**HMA** Heterogeneous Multi-core Architecture - a CPU architecture that that consists of not only multiple processing cores, but also multiple processing core types, each with different capabilities and limitations.

**ISA** Instruction Set Architecture - the set of instructions that a given processing core type can execute, and the format of the machine code used to specify these instructions.

**JIT** Just In Time (compilation) - a compilation technique in which a program's code is distributed in a machine architecture independent form, and then compiled to native machine code immediately before it is executed.

**NUMA** Non-Uniform Memory Access - a computer memory architecture in which memory access latencies differ depending upon the location of the memory and the processing core that is accessing it.

**MFC** Memory Flow Controller - the component of the Cell processor's SPE core type that controls movement of data between main memory and the SPE core's local memory.

**OS** Operating System - the software that controls a given computer system, manages its hardware resources, and provides an interface between hardware and user level applications.

**PPE** Power Processing Element - one of the processing core type of the Cell Processor. The PPE core is the master processing core of the Cell processor, able to control the system overall and execute the operating system.

**SIMD** Single Instruction, Multiple Data - a technique to exploit data-level parallelism, by enabling a single processing instruction to be applied to multiple pieces of data simultaneously.

**SPE** Synergistic Processing Engine - one of the processing core types of the Cell Processor. The SPE core type is a slave processing core, i.e. one that cannot control the system overall, but provides very high arithmetic computation performance.

**VM** Virtual Machine - a machine abstraction that encapsulates software from the underlying computer system and operating system on which it is executing, thus providing a portable environment on which to build applications. This work deals with runtime system virtual machines, such as the Java virtual machine, as apposed to full system virtual machines, such as the Xen or VMware virtual machines.

# Chapter 1

# Introduction

One of the primary components of a computer system is its central processing unit (CPU). Originally, a CPU consisted of a single processing core, on which program code was executed as a single sequential sequence of instructions. A computer system's performance has traditionally been improved by increasing the speed at which these instructions are performed (e.g., by increasing the clock frequency of the processing core). However, this approach has stalled in recent years, due to challenges such as: increased processing core complexity; energy and heat budgets, limiting the rate at which clock speed can be increased; and diminishing gains in program performance from faster processing cores. Instead, multi-core CPU architectures are becoming more prevalent as a means of increasing computer system performance. These multi-core CPUs consist of multiple processing cores, each of which can concurrently execute its own sequence of instructions, thus increasing overall system performance.

The majority of multi-core CPU architectures are homogeneous[1], in that each of the processing cores in a given CPU is identical. This provides a uniform platform on which to build computer programs; an application's code can be executed by, and run equally well on, any of the CPU's processing cores. However, a symmetric multi-core architecture may not provide the best possible level of performance. Since each core is identical, it must have a general purpose design (i.e., *jack of all trades, master of none*). However, different applications, or even different parts of the same application, can have different processing requirements, and therefore might benefit from being executed on a more specialised core type.

[1] These are often known as symmetric chip multi-processors, or symmetric multi-core processors.

Heterogeneous multi-core architectures (HMAs) have multiple different types of processing cores, each of which is designed to perform a specific set of functions well. This enables each core type to be optimised for its particular set of functions, providing the potential to increase overall performance. By having different processing core types, an HMA processor design can also make more effective use of the given area of silicon from which it is built. For example, only one core type need be capable of supporting the system overall by, for example, performing virtual memory page table updates or I/O operations. Thus, the functionality required to perform these operations can be eliminated from the other core types, simplifying their design and enabling more cores to fit in the same area of silicon.

However, while HMA processors have the potential to increase performance and efficiency, they are notoriously difficult to program. Not only must programmers deal with concerns implicit in concurrent programming on multi-core architectures, such as synchronisation, scalability and deadlock prevention, they must also contend with different processing core instruction sets, thread scheduling on cores with asymmetric performance capabilities and, often, complex memory hierarchies. As such, the use of HMA processors is currently restricted to specialist fields, such as network packet processing, computer games consoles and high performance embedded devices.

The goal of this work is to reduce the burden involved in developing applications for HMA processors, such that they can be employed for non-specalist applications. This work focuses on addressing the challenges that are unique to development on an heterogeneous multi-core architecture (it deals with, but does not focus on, issues that are inherent to concurrent programming of any multi-core processor). The philosophy behind this work is to move the responsibility for dealing with the challenges of an HMA processor away from application developers, and into a runtime system that supports application execution. The intent is that applications can exploit the potential performance of an HMA processor, without their developers requiring in-depth knowledge of the processor's architecture. This burden can, instead, be placed upon the runtime system developers, who are likely to have more specialist knowledge of the capabilities of the hardware and be more willing to deal with an HMA's peculiarities. The provision of such a runtime system will enable a much broader range of applications to exploit the potential performance of HMA processors and will ease the likely transition toward greater processing core heterogeneity in commodity systems.

## 1.1 Thesis Statement

Given the difficulties involved in programming and managing heterogeneous multi-core architectures (HMAs), their use is currently limited to specialist applications. I assert that a homogeneous multi-core virtual machine abstraction can be employed to reduce the burden of developing applications for HMAs, while still enabling the disparate processing resources of an HMA to be exploited. By tracking a program's behaviour, a runtime system can make informed thread and data placement decisions, enabling the program to make effective use of heterogeneous processing resources. By employing program behaviour characteristics to guide the partitioning of a program's execution across heterogeneous processing cores, application developers do not require in-depth knowledge of an HMA's design in order to exploit it effectively.

This assertion will be demonstrated by:

- The creation of a Java virtual machine runtime system that provides a homogenous abstraction on which unmodified Java applications can be executed by the different core types provided by an HMA. Two HMA systems will be targeted by this runtime system — the IBM Cell processor and an x86 NUMA (non-uniform memory access) architecture — to explore this assertion under two, very different, of heterogeneous architectures;
- The augmentation of this runtime system to enable it to track a program's behaviour and use this information to inform its thread and data placement decisions with respect to the HMA on which it is executing. The effectiveness of this behaviour-aware runtime system will be evaluated by measuring the performance of a number of real-world Java benchmarks, that have either been augmented with code annotations to express their behaviour characteristics, or have their behaviour monitored at runtime by the runtime system.

## 1.2 Contributions

This work contributes to the abstraction of heterogeneous multi-core architectures in a number of ways:

- Demonstration of the feasibility of hiding a heterogeneous multi-core architecture behind a homogeneous virtual machine abstraction;

- Presentation of program behaviour characteristics as an abstraction that enables a behaviour-aware runtime system to make effective use of heterogeneous processing cores without unduly burdening the application developer;
- Development of a cost function that uses a program's behaviour characteristics to inform thread placement and migration across heterogeneous processing cores;
- Development of a *targeted migration* strategy, which enables future invocations of a method, that initiates a change of phase in a program's behaviour, to automatically trigger a thread's migration to another core type;
- Development of a lightweight runtime monitoring system that enables a runtime system to automatically infer a program's behaviour during its execution;
- Development of a thread and data placement strategy that can reduce inter-thread communication overheads on a NUMA architecture, based upon program behaviour characteristics; and
- Creation of a Java compiler for the SPE core type of the Cell processor, which involved the development of a novel software caching scheme that exploits high level type information to improve its performance.

## 1.3 Publications

The work reported in this dissertation led to the following publications:

> R. McIlroy & J. Sventek, *Hera-JVM: Abstracting Processor Heterogeneity Behind a Virtual Machine*, in Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'09), 2009. (McIlroy & Sventek, 2009*b*).

> R. McIlroy & J. Sventek, *Abstracting Heterogeneous Multi-Core Architectures using a Code Annotation Aware Runtime System* (Poster), in the EuroSys Conference (EuroSys'09), 2009. (McIlroy & Sventek, 2009*a*).

During work on this dissertation, the following papers were also published by the author on closely related topics:

> R. McIlroy & O. Hodson, *Subordinate Kernels: Application Offloading in Asymmetric Multi-Processor Systems*, in Proceedings of the Workshop on Operating System Support for Heterogeneous Multi-Core Architectures. 2007 (McIlroy & Hodson, 2007).
>
> R. McIlroy, P. Dickman & J. Sventek, *Efficient Dynamic Heap Allocation of Scratch-Pad Memory*, in Proceedings of the International Symposium on Memory Management, 2008. (McIlroy *et al.*, 2008).
>
> E. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel & G. Hunt, *Helios: Heterogeneous multiprocessing with satellite kernels*, in Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP'09), 2009. (Nightingale *et al.*, 2009).

The work reported in these related papers is not directly discussed by this dissertation; however, the Helios operating system, as presented in (McIlroy & Hodson, 2007; Nightingale *et al.*, 2009), is discussed in the related work chapter.

## 1.4 Outline

The remainder of this dissertation is organised as follows:

**Chapter 2** provides background information on heterogeneous multi-core architectures. The reasoning behind introducing heterogeneity into processor design is discussed and the history of heterogeneous multi-core architectures is presented. Finally, a case is presented as to why commodity processor architectures are likely to expose more heterogeneity in the near future.

**Chapter 3** surveys related work that is aimed at easing application development on heterogeneous architectures. Work that hides, abstracts or manages processing core heterogeneity in the fields of programming models, compilers, runtime systems and operating systems is examined.

**Chapter 4** presents an approach for abstracting the challenging aspects of application development on a heterogeneous multi-core architecture, based upon a program behaviour-aware runtime system. A set of behaviour characteristics is defined, that can be used by a runtime system to automatically partition an application across heterogeneous processing cores in an effective manner, without burdening application developers with details of the underlying architecture's heterogeneous nature.

**Chapter 5** describes the development of a Java runtime system, called Hera-JVM, that hides the heterogeneity of the IBM Cell processor behind a homogeneous virtual machine abstraction. This runtime system supports simultaneous execution of a single application across two different processing core architectures and enables transparent migration of an application's execution between these two architectures.

**Chapter 6** builds upon Hera-JVM to enable it to make use of knowledge of a program's behaviour characteristics, provided through code annotations, to choose the most appropriate core type on which to execute the different parts of a program. A cost function-based thread migration policy is introduced as a means of deciding whether a thread might benefit from being migrated onto a different core type. A number of migration mechanisms are presented and evaluated experimentally.

**Chapter 7** presents an extension to Hera-JVM that enables it to automatically infer a program's behaviour through runtime monitoring. This enables programs that have not been annotated with behaviour characteristics to exploit heterogeneous multi-core architectures without modification. The simultaneous use of both code annotations and runtime monitoring to inform the runtime system of a program's behaviour characteristics is also investigated.

**Chapter 8** explores the use of a behaviour-aware runtime system under a different heterogeneous multi-core architecture from the Cell processor. Specifically, the use of *thread team* characteristics, to express a program's inter-thread communication patterns, is investigated as a means of informing a runtime system's thread

and data placement decisions, so as to improve performance under a non-uniform memory access (NUMA) architecture.

**Chapter 9** concludes and explores opportunities for future work that build upon the work presented in this dissertation.

# Chapter 2

# Heterogeneous Multi-Core Processors

Commodity microprocessors have traditionally relied on increased clock frequency and instruction-level parallelism to improve their performance from one generation to the next. However, there are limits on the amount of instruction-level parallelism that can be extracted from sequential programs (Wall, 1991), and CPU clock frequency increases have also stalled, due to heat and energy issues (Ross, 2008). With the number of transistors available on a given sized processor die continuing to increase with each improvement in manufacturing process feature size[1], processor designers attempt to exploit these additional transistors to improve the performance of each processor generation. However, the diminishing returns of instruction-level parallelism, as well as the increase in processing core design and verification complexity from the increased transistor count, have instead led to processors incorporating multiple cores onto a single processor die, in order to exploit thread level parallelism (Olukotun *et al.*, 1996).

These *chip multi-processors* (CMPs) can be either symmetric, with each processing core being identical, or heterogeneous, with multiple different processing core types that each have different capabilities. Most current commodity CMPs are symmetric, since this provides a much simpler platform on which to build applications. However, a *heterogeneous multi-core architecture* (HMA) has the potential to provide much greater performance and efficiency than an equivalent (in terms of transistor count) symmetric CMP.

[1]The number of transistors on a CPU continues to roughly double every two years, a trend commonly known as Moore's law.

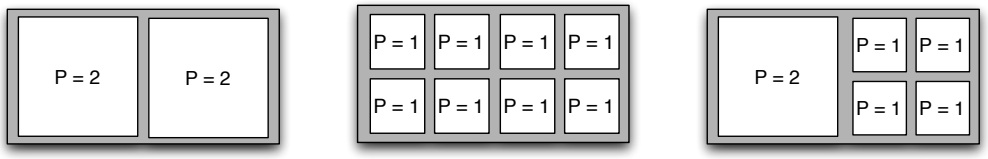
This chapter outlines the development of heterogeneous multi-core architectures. Section 2.1 discusses the reasons why HMA processors have the potential to provide better performance than symmetric CMPs. Section 2.2 presents the history of HMA processors. Finally, Section 2.3 describes current HMA processors and argues a case as to why commodity processor architectures are likely to expose more heterogeneity in the future.

## 2.1 The Case for Heterogeneous Multi-Core Architectures

A heterogeneous architecture can provide opportunities to improve performance and efficiency, both through increased specialisation of core types and having the potential to provide a better balance between sequential and parallel workload performance.

The different core types of an HMA are often designed such that each is specialised for a different type of workload. Core types can be specialised through various means, such as the provisioning of additional machine instructions targeted at the needs of a particular workload, or improving the performance of a particular functional unit (e.g., the floating point unit) with additional circuitry. Of course, such specialisation has inherent trade-offs — e.g. improving the performance of the floating point unit leaves less room on the silicon die for other circuitry, such as improved integer performance, branch predictors or larger caches. Therefore, traditional symmetric CMP processors take a *jack of all trades, master of none* approach, with their cores being competent in all areas but not specialising in any particular area. HMAs, on the other hand, have the option of providing a general purpose *master* core type, as well as one or more additional core types, each of which is specialised for a particular aspect of the expected workload of the system. This enables the system to have better performance for its expected workload, while still being capable of executing general purpose code on its master core type.

Another advantage of a heterogeneous multi-core architecture is that it can provide a better trade-off between the performance of sequential and parallel code. When developing a processor architecture, the design team can limit the number of processing cores in the architecture, but devote more silicon area to each core to improve its performance (i.e. through improved branch prediction, speculative execution or instruction-level parallelism). Alternatively, the same area of silicon can be used to




Sequential Performance = 2
Parallel Performance = 4

(a) Symmetric - 2 cores.

Sequential Performance = 1
Parallel Performance = 8

(b) Symmetric - 8 cores.

Sequential Performance = 2
Parallel Performance = 6

(c) Heterogeneous

Figure 2.1: A simplified example showing the performance that can be achieved from different core layouts on the same area of a processor's silicon die.

support a larger number of simpler cores which, while individually slower than the larger cores, combine to provide greater processing power for parallel workloads.

As more silicon area is allotted to a processing core design, the performance increase which can be expected from the increased transistor count is generally governed by Pollack's Rule (Borkar, 2007). Pollack's Rule states that a core's performance is roughly proportional to the square root of the increase in transistor count. In other words, doubling the size of a core will increase performance by about 40%. On the other hand, doubling the number of cores has the potential to provide a 100% increase in the performance of parallel workloads. This would seem to suggest that a processor with a large number of small cores will provide better performance than an equivalent processor containing a smaller number of more complex processing cores. However this is not always the case. Firstly, not all algorithms can be parallelised effectively, and thus can only be executed on a single processing core. Secondly, even if an algorithm can be parallelised, it typically has a fraction of code that must run sequentially. Amdahl's law (Amdahl, 1967; Gustafson, 1988) shows that even a relatively small fraction of sequential code can severely limit the overall scalability of an algorithm. Since this sequential code can only be executed on a single core, there is an inherent trade-off between the performance of each core (to execute sequential code) and the number of cores available (to scale-up parallelisable code).

Figure 2.1 shows a simplified example of the options available to a processor designer when developing a processor that fits on a given area of silicon. The cores of a symmetric multi-core architecture are identical, therefore, they must all be tailored to either provide high sequential performance, at the cost of having fewer cores overall

(Figure 2.1(a)), or be simplified to enable more cores on the same silicon die area (Figure 2.1(b)). However, a heterogeneous multi-core architecture (Figure 2.1(c)) has the option of providing a more complex core type for high sequential code performance, and many smaller, less complex cores, to provide much greater performance for code that can be parallelised. Hill & Marty (2008) re-evaluate Amdahl's law under a variety of potential processor designs and find that, under simplistic assumptions (e.g., no scheduling or migration overheads), heterogeneous multi-core architectures can provide better potential speedups compared with symmetric architectures. This was due to their ability to execute the sequential portion of an algorithm on a more complex core, while scaling up the parallel portion on many simple cores.

Of course, in order to exploit this potential for improved performance, an application must partition naturally between the core types, such that its threads and phases of execution are executed on the most appropriate core type. Enabling a runtime system to perform this partitioning effectively, without burdening a programmer with the details of core type specialisation, is one of the main research questions addressed in this dissertation.

## 2.2 The History of Heterogeneous Processors

Computer systems that incorporate heterogeneous processing resources in their design are not new. As early as 1957, the IBM 709 (Greenstadt, 1957) used *channel processors* to remove the burden of I/O operations from the main CPU. These channel processors could be programmed to perform I/O transfers asynchronously and independently from the main CPU, and can therefore be thought of as heterogeneous processing cores. These channel processors continued to evolve in subsequent systems such as the IBM System/360 and System/370 (Case & Padegs, 1978) and the Control Data CDC 6600 (Thornton, 1970), eventually leading to the Bus Master DMA (direct memory access) devices now found in commodity PCs. However, these I/O processors are generally limited to performing basic data transfer and signalling operations[1] and are not equal partners in the system's processing capabilities.

[1]The *peripheral processors* of the CDC 6600 are more capable than most channel processors, able to perform operating system tasks and manage the system overall. Modern day programmable I/O offloading engines can be thought of as their decendents.

I/O operations are not the only type of processing which have been offloaded to a heterogeneous coprocessor. Other coprocessors have been used to accelerate floating point (Chandra, 1988), string matching (Cho & Mangione-Smith, 2005), encryption (Yee, 1994) and many other applications. As with channel processors, these coprocessors are generally not capable of executing general purpose program code; they are limited to the subset of computation for which they are specialised.

A heterogeneous multi-core architecture (Kumar *et al.*, 2005*a*) consists of multiple processing core types on a single processor chip. Some of the processing cores of a heterogeneous multi-core architecture are specialised for a particular type of computation, much like coprocessors. However, unlike coprocessors, all the processing cores of an HMA are CPUs, capable of executing general purpose program code in their own right. Thus, rather than thinking of these cores as accelerators for a particular operation, they become first class processing resources in their own right. This changes the application development model from one where the program mostly runs on the main CPU, but accelerates various operations though calls to a coprocessor, to one where the application's execution is spread across various processing cores, each of which has different capabilities and specialisations. Heterogeneous multi-core architectures have recently found their way into a number of specialist markets, such as network processing, multimedia applications and low-power embedded hardware.

One of the first markets to employ HMAs was network packet processing equipment. A number of *network processors*, such as the Intel IXP (Adiletta *et al.*, 2002; George & Blume, 2003), the IBM PowerNP (Allen *et al.*, 2003) and the Motorola C-Port (Motorola, 2001), consist of multiple different processing core types. Network packet processing is a natural fit for an HMA processor, since there are two distinct types of network packet: control packets, which involve complex operations, such as routing table updates, but are received relatively infrequently; and data packets, which must be dealt with quickly, but have much simpler processing requirements. Network processors, therefore, generally consist of a complex control processor, which manages the system overall and processes control packets, as well as a number of relatively simple packet processing cores (variously known as microengines, picoprocessors or channel processors depending upon the network processor), used to processes data packets. Shah (2001) provides a comprehensive survey of the field of network processors.

Another recent HMA design is the IBM Cell processor (Chen *et al.*, 2007; Hofstee, 2005; Pham *et al.*, 2005). The Cell processor is targeted at both multimedia workloads, being the main processor of the Sony Playstation 3 games console, and scientific computation, as a component in a number of supercomputer designs (Barker *et al.*, 2008). Its design evolved from that of network processors, consisting of both a control processor and multiple, simple, but fast worker processors. These worker processors (called SPEs on the Cell processors) are specially designed for computationally intensive floating point and multimedia oriented workloads. The design of the IBM Cell processor will be presented in more detail in Section 5.1.

Another sector which can benefit from a heterogeneous processor design is the embedded market. The performance required by embedded devices, such as mobile phones, is rapidly increasing as new features are added to the devices (e.g. music playback, video, etc.). At the same time, these devices are often powered by batteries and, therefore, have significant energy consumption constraints. The use of multiple core types, each designed for a particular purpose, can provide significant power efficiency improvements, as each core type can perform its function more efficiently than could a more general purpose core (Kumar *et al.*, 2003). Many of these embedded architectures also employ System On Chip (SOC) techniques, where many different components are integrated into a single integrated chip (e.g. processor cores, memory and transmission circuitry) to further decrease power requirements. The Intel PXA800F (Krishnaswamy *et al.*, 2003) is one such HMA processor. It is aimed at the high performance mobile phone market and contains a general purpose XScale core, a Micro Signal Architecture core used for voice or video processing and integrated GSM / GPRS mobile phone communication circuitry.

The defining feature of these HMA processors is that they consist of multiple different processing core types on a single processor chip. In addition, these architectures often incorporate unusual memory and inter-core communication features. For example, the Intel IXP network processor has multiple levels of explicitly accessible memory (in order of increasing capacity and memory latency — local memory, scratchpad memory, SRAM and DRAM), with no hardware caching to automatically move data between these memories. The IBM Cell processor also employs different explicitly accessible memory levels, and requires direct memory access (DMA) operations to move data between these levels[1].

[1]The IBM Cell processor's memory architecture is described in more detail in Section 5.1.

If exploited effectively, these architectures can offer significant performance and efficiency advantages over an equivalent symmetric CMP. However, developing an application that makes full use of the available core types of an HMA processor is challenging. This, combined with the programming complications presented by heterogeneous instruction sets and their unusual memory architectures, has currently limited their uptake to specialist fields.

## 2.3 Heterogeneous Processors in Commodity Systems

If HMA processors continue to be limited to specialist domains, there will be little benefit in the provision of a runtime system aimed at non-specialist programmers. However, current trends suggest that future commodity computer systems will consist of heterogeneous processor architectures. While these commodity systems may not be as extreme as those currently found in specialist domains, they are likely to embody many of the concepts found in current HMAs. This section attempts to justify the argument that commodity systems are going to expose more processing heterogeneity in the near future, through a survey of upcoming processor designs.

There are two driving forces behind this move towards heterogeneous commodity processors: the availability and increased capabilities of graphics processor units in commodity systems, and the steadily increasing core count of commodity processors.

### 2.3.1 Graphics Processing Units as Heterogeneous Cores

Graphics processing units (GPUs) are now ubiquitous in commodity computer systems. These GPUs enable graphical operations, such as 3D graphics rendering and video decoding, to be offloaded from the main CPU. These graphics operations are typically highly floating point intensive and data parallel (e.g., the colour of a particular pixel is not generally dependent upon the colour of other pixels, and therefore the calculations used to compute each pixel's colour can be carried out in parallel). As such, GPUs are specifically designed to take advantage of data parallelism and have considerably better floating point performance than an equivalent CPU.

GPUs were originally fixed function in design; they accelerated a fixed portion of the graphics rendering pipeline, such as rasterisation (Kelley *et al.*, 1992), but could not

be programmed to perform any other function. They have gradually evolved to provide more programmability. Modern GPUs enable replacement of hard-coded stages of their graphics pipeline with small programs, known as *shaders*. Initially, vertex shaders (Lindholm *et al.*, 2001) were introduced to provide programmable geometry transformations and lighting calculations. This was followed by pixel or fragment shaders (Montrym & Moreton, 2005), to provide programmable per-pixel effects (e.g., shadow or bump mapping effects), and most recently, geometry shaders (Blythe, 2006), which can be used to procedurally generate additional geometry in the scene. A GPU consists of a large number (tens to hundreds) of processing cores, used for shader execution. Thus, shader operations can be executed on many graphical elements concurrently to provide a very high degree of data parallel performance.

With this highly data-parallel computation model, a GPU can be considered as a high performance streaming processor. The Brook project (Buck *et al.*, 2004) exploited this fact to enable a GPU to be treated as a streaming coprocessor for general purpose (non-graphics) computation. This was done by mapping computational kernels, written in an extended form of C, to graphics shaders, and storing the data to be manipulated as geometry and textures. This *general purpose GPU* (GPGPU) trend has continued, with a variety of systems, such as Nvidia CUDA (Ryoo *et al.*, 2008) and OpenCL (Munshi, 2009), having been developed to enable general purpose code to be offloaded to GPUs[1].

The type of code that could be offloaded to initial GPGPU systems was limited by a GPU's fixed graphics pipeline and the lack of processing core features, compared to that found on a CPU (e.g., lack of hardware caches and inefficient branch operations). This has improved as GPUs have become more programmable and flexible. The Nvidia Tesla (Lindholm *et al.*, 2008) was one of the first GPU architectures specifically designed with general purpose streaming computation, as well as graphics processing, in mind. It unified the previously separate vertex and pixel processing cores into a single set of shader processing cores, simplifying kernel development and enabling more flexible load balancing. The (as yet unreleased) Nvidia Fermi architecture (Nvidia, 2009*a*) takes GPU programmability even further, with a unified address space, advanced control flow mechanisms (such as indirect branches and exception handling) and the ability to run C++ (including virtual functions and function pointers).

[1]These GPGPU systems are described in greater detail in the next chapter.

Underlining this convergence of CPU and GPU capabilities is the Intel Larrabee project (Seiler *et al.*, 2008). Intel's entry into the high performance GPU market takes the opposite approach from Nvidia's — instead of taking a GPU and making it more like a general purpose CPU, Larrabee takes a general purpose CPU and tailors it for data-parallel workloads, such as graphics processing. Larrabee is a many-core architecture, consisting of multiple x86-based, in-order processing cores. Since these cores are based on the x86 architecture, a Larrabee GPU can execute any general purpose code that could be executed on an x86 CPU (other than OS system calls, which are proxied to the *master* CPU that runs the operating system).

As well as becoming more programmable, GPUs are being integrated with CPUs in a single chip. In the embedded market, the Nvidia Tegra platform (Nvidia, 2009*b*) already provides integrated CPU and GPU cores on a single chip. With the (as yet unreleased) AMD Fusion project (AMD, 2008) and Intel Clarkdale architecture (Vaughn-Nichols, 2009), the desktop market is also heading in this direction. This increased coupling will reduce the overhead of offloading computation to a GPU processor, as well as enable better memory coherence between core types. This will enable more flexible and simple work sharing between the CPU and GPU cores.

These improvements in programmability and the closer coupling to CPU cores mean that a GPU is now less of a coprocessor and more of a fully fledged heterogeneous processing core: a first class processing resource able to execute the majority of an application's code, but having very different performance characteristics from a conventional CPU. The choice of where application code should be executed is a question of expected performance (e.g., data-parallel floating point computations on a GPU type core, control code on a CPU type core), rather than being limited by a GPU's capabilities.

A commodity processor chip, consisting of a modern Pentium-based CPU core and many Larrabee-based GPU cores, would look much like a current HMA, such as the IBM Cell Processor. Indeed, such a processor design is already planned as a future commodity PC processor by Intel, and was one of the motivations behind this work.

### 2.3.2 Many-Core CPUs

Another trend driving commodity systems toward a more heterogeneous processing environment is the ever increasing core count of commodity processors. Multi-core

processor architectures are now commonplace, and a number of many-core architectures have been proposed, such as Intel's 80 core terascale processor (Mattson *et al.*, 2008; Vangal *et al.*, 2008), and Tilera's 64 core Tile processor (Wentzlaff *et al.*, 2007). These many-core architectures present opportunities for processor designers to introduce heterogeneous processing cores as well as introducing scaling challenges which may inevitably lead to some form of heterogeneity.

As the number of cores in commodity processors increases, the benefits of providing different types of processing cores, as discussed in Section 2.1, becomes more apparent. Modern multi-core processors already implement dynamic clock scaling of individual cores (Charles *et al.*, 2009) as a means of boosting system performance while remaining within the processor's thermal limits. This technique enables a particular core's frequency to be boosted above its base frequency, if the other cores are not being fully utilised. Thus, these processors have become heterogeneous multi-core architectures, with boosted cores having better performance than their non-boosted counterparts. Another approach which can be used to adapt to different workloads is *core fusion* (Ipek *et al.*, 2007). Core fusion enables a processor to dynamically fuse together multiple simple cores into one, more powerful, virtual core. This enables the processor to adapt to either a parallel workload (where all the simple cores can execute different threads simultaneously), or a sequential workload (where a fused core can execute the single thread more quickly).

As well as introducing opportunities to improve performance through heterogeneity, many-core architectures also introduce scalability challenges that can be overcome through processing core heterogeneity. An example of this type of processor heterogeneity is non-uniform memory access (NUMA) architectures. As the number of processing cores increases, accessing memory though a shared memory bus can become a scalability bottleneck (Archibald & Baer, 1986; Rettberg & Thomas, 1986). NUMA architectures (Cox & Fowler, 1989; Laudon *et al.*, 1997) address this scalability bottleneck by having memory attached locally to each processor. An inter-processor network enables access to data on non-local memory, but these accesses are slower than those to local memory. Thus, the processing environment of a NUMA machine is heterogeneous, in that the performance of a thread will depend upon whether it is placed on a core *close* to the data it accesses.

## 2.4 Summary

In summary, heterogeneous multi-core architectures can provide performance and efficiency improvements over symmetric counterparts. While currently occupying specialist domains, trends such as programmable GPUs and many-core architectures make it likely that commodity multi-core processors will soon consist of heterogeneous processing cores. The difficulties involved in programming these architectures mean that new tools and programming techniques are required if they are to be exploited by non-specialist programmers.

# Chapter 3

# Related Work

The heterogeneous nature of HMA processors complicates application development for these processors. Developers must not only deal with the problems inherent in multi-threaded programming, such as scalability and deadlock avoidance, but they must also take into account the capabilities of the different processing core types in order to effectively exploit an HMA processor. The impact this heterogeneity has on thread scheduling has been investigated by Balakrishnan *et al.* (2005) and Bower *et al.* (2008). Issues, such as different instruction sets, unusual programming models, complex memory architectures and asymmetric inter-core communication, further complicate development on these architectures (Penry, 2009).

The goal of this dissertation is to reduce the burden of dealing with heterogeneous processing resources for non-specialist programmers. This chapter reviews work related to the field of managing, abstracting and hiding processor heterogeneity from developers. Section 3.1 reviews parallel programming models and languages which have been developed to ease the creation of programs which can concurrently execute on multiple processing cores. Section 3.2 discusses work aimed at abstracting the heterogeneous programming environment presented by architectures that consist of different processing core types. Thread scheduling algorithms which take processing core heterogeneity into account are discussed in Section 3.3. Finally, Section 3.4 discusses operating system and runtime system designs aimed at managment of non-uniform memory.

## 3.1 Parallel Programming

To fully exploit a heterogeneous multi-core processor, an application must be able to distribute its computation across the multiple cores available on the platform on which it is executing. The application must therefore be designed such that it can be parallelised into a number of parts that can each be executed concurrently. A number of programming models, programming languages and compilers have been developed to ease the process of developing parallel applications.

The work in this dissertation is not aimed at directly abstracting the creation of parallel programs; instead, its goal is to enable programs, which are already parallel, to effectively exploit heterogeneous core types. The work discussed in this section is therefore complementary to the work in this dissertation, rather than being directly comparable; both concurrency and heterogeneity abstractions are important if developers are to exploit heterogeneous multi-core architectures.

Application programming interfaces (APIs), such as MPI (Foster & Karonis, 1998; Gropp *et al.*, 1994) and OpenMP (Dagum & Menon, 1998), provide a variety of mechanisms to simplify the process of parallelising a given workload. These APIs are used extensively by high performance and scientific applications.

OpenMP provides a set of preprocessor directives, or *pragmas*, that can be used to direct the compiler to parallelise sections of code, by dividing a given task amongst a number of different threads. OpenMP uses a fork-join threading model, in which the master thread parallelises a task by forking a number of worker threads, that share the work, and then waiting for each of these worker threads to complete (joining with them) before continuing. OpenMP presents a shared memory programming model to applications, with variables being visible to all threads of the application by default.

MPI takes a different approach, based upon message-passing. Communication between threads is performed using explicit messages. This enables developers to write parallel programs that span multiple computer systems (e.g., within a computing cluster) without having to be concerned with the details of inter-machine communication. Thus, programs built using MPI can scale across multiple processing cores in a single system, or across processors in multiple machines of a computing cluster.

Some programming languages have been built around the notion of concurrent programming itself, rather than relying upon libraries or APIs to abstract concurrency.

These concurrency-oriented languages feature constructs that enable a program's concurrency to be expressed in as straightforward a manner as possible.

The Erlang programming language (Armstrong, 2003), originally developed for telecommunication applications, follows the actor model (Hewitt *et al.*, 1973) to abstract the creation of concurrent programs. In this model actors are primitives of computation, that send and receive messages amongst themselves and make local decisions based upon received messages. Communication between actors in Erlang is performed through *shared-nothing* asynchronous message passing, with messages being sent to and retrieved from per-actor *mailboxes*. Messages may be consumed in a different order from that in which they were received using pattern matching-based message consumption. This model is inherently concurrent, with each actor being independent from the others. Erlang influenced other concurrency-oriented languages, such as the Java-based, Scala programming language (Odersky *et al.*, 2004), which also uses the actor model.

A similar approach is taken by languages based upon communicating sequential processes (CSP) (Hoare, 1978), such as Occam (Roscoe & Hoare, 1988). These languages have a similar concept of independent communicating entities (called processes in this case). However, these processes are anonymous (unlike actors which possess an identity), with communication occurring across named channels.

Functional languages, such as Haskell, provide another approach. Purely functional languages are side-effect free. This, in principle, enables the runtime system to extract parallelism by executing expressions in parallel, without changing the program's result at all. In practice, it is difficult for the runtime system to ensure that a given expression is large enough to warrant the overhead of forking a thread to compute its value in parallel with other expressions. Glasgow Parallel Haskell (GPH) (Trinder *et al.*, 1999), therefore, provides a `par` annotation, which programmers can use to identify promising expressions for parallel computation. Use of the `par` annotation does not change the semantics of the program, but rather enables exploitation of the already present potential parallelism. Concurrent Haskell (Peyton Jones *et al.*, 1996) provides additional operators to enable threads to be forked explicitly, thus enabling a developer to express a program in a concurrent manner, if this is appropriate. However, exploiting the potential parallelism provided by both concurrent and parallel Haskell efficiently continues to be a challenging problem (Harris *et al.*, 2005).

## 3.2 Abstraction of Heterogeneous Programming Environments

As well as having multiple cores, over which an application must map its work if it is to fully exploit the processor, the cores of a heterogeneous multi-core architecture are of different types, each with a potentially different programming environment. Dealing with the heterogeneous programming environments of an HMA's different core types is one of the most challenging aspects of developing an application for these architectures. Typically, to develop an application that can exploit multiple different processing core types, the program must be split between the core types at design time. The programming environment of each core type is likely to be different, either due to the different capabilities of the core type (e.g., a GPU is optimised for data parallel workloads and is therefore usually programmed using a data parallel model), or the lack of operating system support on *slave* cores (e.g., standard libraries may not be available on slave processing cores if they do not execute operating system code).

This section discusses approaches which have attempted to abstract aspects of this heterogeneity to various degrees, and through various means. Programming model and compiler-based approaches are discussed in Section 3.2.1. Runtime systems and operating systems aimed at abstracting processor heterogeneity are discussed in Section 3.2.2.

### 3.2.1 Programming Models and Compilers

This section investigates programming models and compilers which have been developed to ease the process of creating applications for heterogeneous processors. The three main areas which have exploited heterogeneous processing cores are: coprocessors and programmable devices (Section 3.2.1.1); network processors (Section 3.2.1.2); and the IBM Cell processor (Section 3.2.1.3).

#### 3.2.1.1 Coprocessors and Programmable Devices

The application development model for heterogeneous processors has evolved from that of coprocessor-based approaches. Originally, coprocessors were programmed using two distinct approaches: an ISA-based approach or a device driver-based approach. In the ISA-based approach, the master CPU's instruction set architecture (ISA) is augmented with custom instructions which enable execution of coprocessor operations. The master

CPU is charged with controlling the coprocessor during its execution and, therefore, is typically stalled for the duration of the coprocessor's execution. This, combined with the lack of flexibility afforded by having to modify the master CPU's ISA to support a given coprocessor, means that this approach was only used for the most closely coupled coprocessor designs, such as the Intel x87 floating point coprocessor and the *Cipher* coprocessor units of the Intel IXP network processor (Feghali *et al.*, 2002).

Coprocessors which evolved from more loosely coupled devices, such as I/O offload processors or GPUs, usually employ a device driver-based programming model. In this approach a custom device driver controls offloading of work to the coprocessor. To take advantage of these coprocessors, an application would originally have had to interface with the coprocessor driver in a device specific manner. However, as GPUs became more prevalent, device-independent, domain-specific programming APIs, such as OpenGL (Woo *et al.*, 1999) and DirectX (Microsoft, 2002), were built on top of these drivers to provide graphics acceleration portability.

Domain-specific languages, such as the OpenGL Shader Language (Peercy *et al.*, 2000) and Cg (Mark *et al.*, 2003), are provided by these APIs to enable applications to exploit the programability of modern GPUs. These languages hide the GPU's hardware complexity and device-specific ISAs from application developers, but are heavily tailored towards graphics processing. While it is possible to offload non-graphics related code to GPUs using these languages, doing so is cumbersome, requiring the developer to convert their algorithm into an equivalent graphical operation and map data into graphical textures or vertices. Restrictions in these languages also limit the algorithms that they can express. For example, while Cg is a C-like language, it does not provide support for pointers, recursion or many other standard C operations required for efficient general purpose computation.

As the performance and programability of GPUs increase, general purpose computation on GPUs (GPGPUs) has become a more attractive proposition. To achieve high performance from a GPGPU, offloaded code must be highly parallelisable so that it can concurrently use the large number of processing cores provided by these architectures. It must also minimise off-chip communication to avoid this becoming a bottleneck. A streaming programming model, in which GPUs are abstracted as general purpose stream processors (Kapasi *et al.*, 2003), has been used by a number of GPGPU sys-

tems, such as Brook (Buck *et al.*, 2004) and Cuda (Ryoo *et al.*, 2008), to enable general purpose code to exploit GPUs in an effective manner.

A streaming processing model expresses offloaded code in the form of *computational kernels*, and the data on which they operate as *streams*. A computational kernel pulls data from one or more input streams, performs a calculation on this data, and then pushes its output to one or more output streams. A streaming application can be built up from many computational kernels, connected by a network of data streams. This model suits the architecture of GPUs because it expresses parallel work in the form of computational kernels, and expresses inter-kernel data locality through streams. Therefore, different computational kernels can be mapped to each of the cores of a GPU, and off-chip communication can be minimised by mapping streams to direct inter-core data transfers.

The Brook and Cuda GPGPU systems both extend the C language to enable streams and computational kernels to be expressed. An application built using these systems consists of both normal C / C++ code, that runs on the master CPU, and computational kernels, which are offloaded to the GPU. Kernels are written as special C functions, specified by a kernel keyword. These kernel functions are restricted to a subset of C, with limited pointer support and no support for recursion or indirect branches. The Brook and Cuda GPGPU systems consist of a compiler and a runtime system. The compiler converts each kernel into a shader program so that it can be executed on the GPU. The runtime system controls the initialisation, set-up and management of kernels on GPU cores, as well as providing routines to transfer data between the CPU and GPU cores.

These systems enable general purpose computation on GPUs, but are relatively inflexible. Since the computational kernels are compiled into shader programs, they are restricted to execution on a GPU core, even if the GPU is already in-use or oversubscribed. OpenCL (Munshi, 2009) presents a similar streaming programming model, but provides more flexibility in where the computational kernels are executed. It does this by distributing the kernels in an architecture independent *intermediate form*, then *just in time* (JIT) compiling the kernels at runtime, for the core type on which they will be run. OpenCL uses the Low Level Virtual Machine (LLVM) compiler infrastructure (Lattner & Adve, 2004) to JIT compile computational kernels to optimised

machine code. Therefore, these kernels can be run in an efficient manner on any available core-type which is supported by the LLVM back-end. This currently includes general purpose CPU cores, GPU cores and embedded processing cores.

This gives the developer much more flexibility to exploit the available core types on any given system. However, the onus is still on the application developer to decide which core type is most suitable for a given computation: unlike the work in this dissertation, OpenCL does not attempt to automatically schedule code on the most appropriate core type. OpenCL also requires offloaded code to be written in a streaming programming model style, which limits the type of code which can be offloaded and prevents efficient interaction between offloaded and non-offloaded code.

A number of other systems, such as the Sh Programming Language (McCool & Du Toit, 2004), the RapidMind Streaming Execution Manager (McCool, 2006) and the PeakStream Virtual Machine (Papakipos, 2006) provide a similar streaming programming model. These three systems all extend C++ with additional constructs to enable developers to express computation kernels and declare data streams. They also manage the offloading of code and data to the GPU, removing this burden from the developer. A similar approach is taken by Microsoft Accelerator (Tarditi *et al.*, 2006), which extends C# with a data-parallel programming model. It provides a set of parallel array operations which can be offloaded to GPU cores automatically.

EXOCHI (Wang *et al.*, 2007) provides a more typical multithreaded programming model, by exposing heterogeneous cores as application-level resources, rather than a separate device managed through a device driver. EXOCHI enables an application to create user-level threads, calls *shreds*, which can target heterogeneous core types. Shreds are created by invoking an extended form of the OpenMP shared-memory parallel programming interface (Dagum & Menon, 1998). Standard OpenMP constructs – such as the fork-join or producer-consumer threading models – can be used to parallelise code. These parallel sections can be compiled for multiple machine architectures to produce a *fat binary*. The EXOCHI runtime system can then schedule shreds on heterogeneous core types.

EXOCHI hides some of the complexity of partitioning an application between heterogeneous core types, by providing similar execution environments for shreds running on master and slave processors through: (i) an architectural *wrapper*; (ii) shared virtual

memory translation; and (iii) collaborative exception handling (e.g., faulting to a master processor if a shred running on a slave processor requests an OS service). However, it also relies upon the developer to decide upon the most appropriate core type for the parallel portions of their code and does not enable shreds to be migrated between core types once they have been started.

The Merge framework (Linderman *et al.*, 2008) extends EXOCHI with a library-oriented language, based upon the MapReduce programming model. The MapReduce programming model (Dean & Ghemawat, 2004) can be used to decompose a large computation into a set of independent *map* and *reduce* operations. Merge translates the explicit parallelism, presented by the MapReduce programming model, into a series of tasks that can be dispatched to any of the available processing cores on the system. The tasks are expressed as a hierarchical set of functions with different degrees of parallelism granularity, as well as different target architecture variants. The runtime system can then select the most appropriate implementation for a given task, based upon the core types available on a given platform and the degree of parallelism available. Tasks are placed onto an architecture-agnostic work queue, with an appropriate implementation of a work item being dispatched to a specific processing core when it becomes idle, thus distributing work across the heterogeneous cores of a platform.

The Merge framework enables an application to exploit heterogeneous processing cores in a conceptually simple and relatively elegant manner. However, applications must be built using the MapReduce programming model if they are to exploit the heterogeneous cores using this framework. This limits the set of algorithms which can be offloaded to those which can be expressed in the MapReduce programming model. Merge also relies on the developer or a library to provide multiple versions of a particular abstract computation, for different target architectures or parallelism granularities, in order to effectively exploit a given heterogeneous platform.

#### 3.2.1.2 Network Processors

Network processors were one of the first domains in which true heterogeneous multi-core architectures were found. As such, a number of programming models and compilers were proposed to limit the complexity of building network processing applications on these HMA network processors.

NP-Click (Shah *et al.*, 2003) is a network processor programming model which is based upon the Click Modular Router (Morris *et al.*, 1999). A Click router consists of a number of routing components, each of which performs a simple operation, such as packet queueing or packet classification. The routing components are *plugged* together using a simple language to create a particular router configuration. A network packet traverses this pipeline of routing components, being partially processed by each component. NP-Click maps each of these routing components onto a particular core of a heterogeneous network processor, thus spreading the packet processing load across a heterogeneous architecture.

However, NP-Click is a very domain-specific programming model. It is based upon a dataflow paradigm, and is therefore only suitable for applications which process a very regular stream of data, such as network routers. Additionally, router components are statically allotted to processing engines at compile-time, reducing portability across different heterogeneous platforms and limiting flexibility in dealing with changing workloads.

The NETKIT (Coulson *et al.*, 2003) project provides a similar programming model to NP-Click, with different routing components each performing a part of a network packet's processing and then passing the packet to the next component. However, more flexibility is provided by *pluggable* loaders and binders, which enables dynamic loading and binding of components at runtime. Different loaders can be employed to load components onto different core types, with dynamic binding enabling components to continue to communicate, even if their communication mechanism must change, due to one being moved to a different core type.

NETKIT provides a dynamic method of linking routing components together, without having to know *a priori* the type of processing core to which a component is connecting. However, as a middleware-based solution, NETKIT does not abstract the process of writing a routing component for different core types, just the process of loading and connecting these components on a heterogeneous architecture. Additionally, its middleware design is a relatively heavyweight solution which is likely to have high overheads (no performance results are available).

The auto-partitioning compiler, developed by Intel for network processors (Dai *et al.*, 2005), takes a different approach. Instead of having the programmer partition the application manually, the compiler takes synchronous C code, and automatically

partitions it into a set of components. These components can be mapped to the heterogeneous processing cores of a network processor, parallelising the code by pipelining each packet's processing through a series of component stages. The sequential code is partitioned by cutting its control flow graph into a series of non-overlapping components. To enable the routing application to fully utilise a network processor's processing cores, the auto-partitioning compiler partitions the application such that there are no data dependencies from later stages to earlier ones, as these would stall the pipeline and prevent parallelism. It also attempts to choose cuts in the control flow graph which minimise the amount of data which must be transmitted between component stages and balance the amount of processing that each stage must perform.

Intel's auto-partitioning compiler enables programmers to develop applications for a heterogeneous multi-core architecture using the familiar sequential programming model. However, extracting parallelisation from this sequential code depends upon the algorithm having inherent data parallelism and a minimal number of inter-component dependencies. Network routing is one of the *best-case* applications for this approach, since it typically has inherent data parallelism, with a number of largely independent operations that can be performed in a pipelined fashion on successive packets. More general purpose applications with more complex structures and dependencies are much more difficult to parallelise in this manner. There has been a large body of work investigating automatically parallelising programs on homogeneous processors, especially in the Fortran language, due to its stronger pointer aliasing guarantees compared to C. Compilers such as the Fortran D compiler (Hiranandani *et al.*, 1992) and the SUIF Explorer (Liao *et al.*, 1999) have investigated auto-parallelisation of scientific applications in Fortran: however, achieving significant parallelism using these approaches is difficult, with these compilers relying upon additional *hints* from the developer (in the form of a data decomposition specification for Fortran D, and through user-driven interactive analysis in the SUIF Explorer). Furthermore, the sequential programming model does not guide the programmer to design applications in a manner that enables them to exploit the available cores of a given system, and hence an algorithm's potential parallelism may be hidden when it has been written in a sequential form.

#### 3.2.1.3 IBM Cell Processor

Thanks to its role as the main processor of the Sony Playstation 3 games console, the IBM Cell processor is one of the most widely available HMA processors. Its performance potential has also seen it incorporated into high performance scientific computing applications, either as a processor in an IBM server, a custom supercomputer (e.g., the RoadRunner machine (Barker *et al.*, 2008)) or a cluster of Playstation 3 devices.

There are two processing core types in the Cell processor, each with different capabilities and instruction set architectures (ISAs). Thus, different machine code must be generated for each core type. In fact, as well as having to be compiled for a particular core type, the very different programming environments of these two cores means that code is likely to have to be specifically designed for a particular core type. For example, of the two core types on the Cell processor, only one (the PPE core) runs the operating system. The other core type (the SPE core) does not run any operating system code, meaning standard libraries and operating system features are not directly available.

As discussed briefly in Section 2.2, the heterogeneous programming environment is further complicated by an unusual memory hierarchy. Each SPE only has direct access to a small (256KB) private memory store. An SPE core can initiate direct memory access (DMA) transfers between main memory and its private store in order to access shared data: however, this is a costly operation and must be performed sparingly if code is to perform well on the SPE cores.

The standard approach to application development on the Cell processor is a *split program* approach, in which the application is effectively partitioned in two at design time, with each of these parts being developed separately for its respective core type. A number of specialist applications, such as games and scientific applications, have been developed using this approach (Bader & Agarwal, 2007; Benthin *et al.*, 2006; Liu *et al.*, 2007). However this approach is not ideal, due to its lack of portability and the limit to load balancing and adaptability to changing workloads incurred by the design-time fixing of application components to a particular core type. Furthermore, developing each portion of the application under a different programming environment also complicates application development, and hence may introduce defects. To alleviate these challenges, a number of compilers and programming frameworks have been developed

to present the application developer with a less complex programming environment on the IBM Cell processor.

A compiler, called the Octopiler (Eichenberger *et al.*, 2006), was developed by IBM to provide a more unified programming environment on which to develop applications for the Cell processor. It extends the OpenMP programming model to enable programmers to specify regions of code that can be executed in parallel and would be suitable for execution on the SPE cores. The compiler duplicates compilation of these code sections for both core types and inserts code to automatically co-ordinate their execution on the heterogeneous core types. As well as the task-level parallelism provided by the OpenMP pragmas, the Octopiler also automates the extraction of data-level parallelism through auto-SIMDization of appropriate scalar loops[1].

The Octopiler hides the Cell processor's unusual memory heirarchy by providing a single shared-memory abstraction. It does this with a compiler-controlled software cache, which caches recently accessed data in the SPE's private memory to reduce the overheads of DMAing from main memory. This system is conceptually similar to the software caching system presented in Section 5.3.3 of this dissertation. However, it does not use high level type information to optimise data transfers by caching complete objects and large array blocks. Instead, it emulates a 4-way set associative hardware cache with 16 byte cache-lines.

The Octopiler provides a much simpler programming environment for the Cell processor than the standard split-program approach. However, its use of the OpenMP programming model limits its flexibility, in that only code blocks which have been annotated as being parallel regions will run on the SPE cores. Only programs which exhibit relatively regular task or data parallelism can use this model to exploit the potential performance of the Cell Processor.

The Cell Superscalar (CellSs) framework (Bellens *et al.*, 2006; Perez *et al.*, 2007) builds upon the Octopiler compiler to provide an alternative programming model, based upon task dependency graphs, that can provide the developer with more flexibility in program design. CellSs enables the developer to use code annotations to specify *task* functions, that can be run on the SPE core type, as well as data dependencies between these tasks. When a thread invokes a task function, it adds the task to a queue of

[1] Auto-SIMDization unrolls a loop so that multiple iterations of the loop's body can be executed in parallel using the SIMD (Single-Instruction, Multiple-Data) instructions available on the SPE cores.

pending work. It also adds this task as a node of a task dependency graph and inserts edges between tasks which have data-dependencies. Tasks which have no dependencies are scheduled on SPE cores, with the dependency graph being updated once they have completed.

This programming model presents the developer with a more flexible model for sharing work between the Cell's heterogeneous processors. Since the CellSs runtime system dynamically tracks inter-task dependencies, it can exploit irregular parallelism that would not be easily expressible with the OpenMP-based Octopiler approach. However, the developer is still responsible for identifying the code blocks that are likely to benefit from execution on the SPE core type. Also, if the granularity of tasks is too small, the overheads of updating the dependency graph and scheduling tasks for execution is likely to overwhelm any performance benefits this approach could provide.

The Sieve C++ language (Donaldson *et al.*, 2008), created by Codeplay Software Ltd., presents the developer with a different model, based around the concept of *sieve* blocks. Within these sieve blocks memory consistency is purposely relaxed, with writes to *global* data (data defined outside that block) being delayed until after the block completes. These sieve blocks can then be automatically parallelised by splitting the blocks into fragments of computation (a fragment may, for example, be a bundle of 20 loop iterations) and speculatively executing these fragments concurrently on multiple cores. The relaxed memory consistency, provided by delaying writes to global data, eases this auto-parallelisation process, since the dependency analysis must only account for variables within the sieve block, not global data. When all of a sieve block's fragments complete, their delayed writes are written to global memory in the order that would have occurred, had the blocks been executed sequentially. The delayed writes from any fragments which were executed speculatively and were subsequently found to be invalid (e.g., loop iterations which, in a sequential execution, would have occurred after a `break` statement) are simply discarded.

Codeplay have implemented the Sieve C++ system for the Cell processor, as well as for multi-core x86 systems. This programming model simplifies dependency analysis and enables a program written in sequential manner to exploit the heterogeneous core of the Cell processor. However, delaying writes to global memory until sieve blocks complete presents an unusual memory consistency model to developers and may cause unexpected bugs. It also limits the algorithms which can be expressed within a sieve

block, since a sieve block fragment is not able to see any of the writes to global variables performed by sieve block fragments which were conceptually executed earlier.

### 3.2.2 Runtime Systems and Operating Systems

A number of runtime systems and operating systems have been specifically developed to hide or abstract various aspects of heterogeneous multi-core architectures.

CellVM (Noll *et al.*, 2008) is a Java runtime system for the Cell processor. It aims to hide the heterogeneous nature of the Cell processor behind a homogeneous Java virtual machine. CellVM supports the execution of a standard Java application across the heterogeneous cores of the Cell processor. It consists of two co-operating virtual machines, one for each of the two core types on the Cell processor. The virtual machine running on the Cell's PPE core type manages the system overall and provides the full set of functionality, as specified by the Java Specification. The SPE cores run a stripped down virtual machine, which enables an application to offload a single Java thread to each of these cores. The SPE's virtual machine supports most common Java operations: however some more complex operations, such as object creation and thread synchronisation, are not directly supported. To perform these complex operations, the SPE virtual machine sends a blocking request to the PPE virtual machine, which performs the operation on its behalf and returns the result.

The approach taken by CellVM – hiding the heterogeneous architecture of the Cell processor behind a homogeneous Java virtual machine – is similar to that of the Hera-JVM runtime system, presented in Chapter 5. However, there are a number of significant differences in philosophy, design and implementation between these two runtime systems.

CellVM aims to provide a unified programming environment for both the PPE and SPE core types. However, it does not totally hide the Cell's heterogeneous architecture: the developer must still decide the core type upon which threads should be executed. Since the PPE and SPE core types have very different performance characteristics, the developer requires relatively in-depth knowledge of the Cell processor's architecture in order to exploit its potential performance. Hera-JVM attempts to hide the Cell's heterogeneous architecture from the developer entirely. Instead of specifying the core type on which a thread should run, hints about the program's likely behaviour are provided to the runtime system, either through code annotations (Chapter 6) or runtime

monitoring (Chapter 7). Hera-JVM uses these behaviour hints, and its knowledge of the architecture on which it is running, to optimise the program's use of the architecture's heterogeneous cores, by migrating threads between the available core types. This migration process is completely transparent to the application, meaning application developers need not even be aware that their code is executing on a heterogeneous environment.

The design of the two runtime systems also differs. CellVM is designed as two separate virtual machines, which co-ordinate through shared memory to support a single application's execution. In contrast, Hera-JVM's philosophy of abstracting heterogeneous core types runs right through the runtime system's design itself. The majority of the runtime system's code is shared by both core types; only a small number of low-level routines are core-type specific. While the runtime system's routines are, by necessity, compiled to different machine code for each core type, the same mechanisms, algorithms and data-structures are used by the runtime system, no matter the core type. This design limits the likelihood of interfacing bugs being introduced if updates to the runtime system are not applied to both core type runtime systems simultaneously and identically; updates made to Hera-JVM's runtime system are simultaneously shared by both core types.

The prototype implementation of CellVM has a number of limitations compared with Hera-JVM. Only a single thread can be bound to each SPE core in CellVM, with no thread switching provided by SPE cores and threads being fixed to a particular core type upon creation. Hera-JVM enables threads to be migrated between core types at any function call point, and supports the execution of multiple threads by a single SPE core with pre-emptable thread switching. Additionally, CellVM relies upon the PPE core to perform a number of common operations, such as object creation and thread synchronisation, on the behalf of the SPE cores, which limits its scalability. These operations are performed locally by the SPE cores in Hera-JVM. Finally, unlike Hera-JVM, the current CellVM prototype does not strictly follow the Java memory model. This is likely to introduce correctness issues for large scale concurrent applications.

Intel's many-core runtime McRT (Saha *et al.*, 2007) is a platform aimed at enabling application developers to exploit many-core architectures (tens to hundreds of cores). It provides a number of features, such as user-level thread scheduling and software

transactional memory (STM), to assist developers in creating highly scalable applications.

The current McRT prototype targets symmetric architectures, although it's design supports heterogeneous architectures. It provides a sequestered mode of execution, where one *master* processing core runs the operating system and the other cores execute application threads without operating system support (*bare metal*), being supported by a lightweight executive instead. This provides the application with complete control of thread scheduling on the sequestered processors. The runtime system also hides the lack of operating system support on the sequestered processors, by acting as a OS interface shim and forwarding all system call requests to the master processor for processing on the sequestered processor's behalf. While this provides a substrate for application execution on an HMA processor, there is no support for heterogeneous processor architectures currently provided by McRT. McRT does not, therefore, currently deal with issues such as scheduling an application's execution across an HMA processor, such that it makes best use of the capabilities of its heterogeneous cores.

The BarrelFish operating system (Baumann *et al.*, 2009; Schüpbach *et al.*, 2008) was specifically developed to tackle the increasing heterogeneity found in modern computing systems. It is a microkernel-based design, in the mould of L4 (Liedtke, 1995); however, it proposes a *multikernel* model, where each processing core runs its own instance of the kernel and all inter-core communication is performed through explicit messages. In essence, it treats modern heterogeneous computer systems like a distributed system, with cores as nodes and the system's inter-core interconnect as the network.

The multikernel model aims to expose an architecture's heterogeneity and provide tools and mechanisms to enable developers to deal with the complexities introduced by such architectures, rather than hiding or abstracting these complexities. Barrelfish provides a *system knowledge base* to expose details of the underlying machine to applications. An application can query this system knowledge base, using constraint logic programs, to gain knowledge of the topology of the hardware on which it is executing. The application can then adapt its execution by, for example, requesting to be scheduled on a core type with a specific set of features, or adapting its algorithm to suit the architecture on which it is being run.

This approach provides considerable flexibility for applications to deal with and adapt to heterogeneous systems. However, in doing so, it adds considerable complexity

to applications that wish to make use of the system knowledge base. It is likely that only specialist application developers would have the skill necessary to effectively make use of this exposed knowledge. Developers that do not have in-depth knowledge of the effects of system heterogeneity could easily misinterpret the implications of a given system topology and hinder performance, instead of improving it. It may be more appropriate to limit use of the system knowledge base to libraries and runtime systems, which then abstract these details from higher level applications. In this scenario, the Barrelfish operating system is complementary to the work of this dissertation: a runtime system such as Hera-JVM can be provided with information about the system's heterogeneity by the underlying operating system, but hide this heterogeneity from applications that it supports.

Work on the system knowledge base aspect of the Barrelfish operating system is currently at an early stage, with no published results as yet. Also, while Barrelfish was designed to support systems with heterogeneous processing cores, it currently supports only symmetric x86 multi-core architectures.

Another operating system aimed at heterogeneous systems is Helios (Nightingale *et al.*, 2009). It is based upon the Singularity operating system (Fähndrich *et al.*, 2006; Hunt & Larus, 2007) and, as such, is built in a variant of the C# programming language. Processes are isolated from each other (they cannot explicitly share memory), but can communicate through typed inter-process message channels.

Helios extends Singularity with the concept of *satellite kernels*. A satellite kernel is started on each programmable device or heterogeneous processing core in the system. Each satellite kernel manages the local resources of these heterogeneous cores (e.g., performing local memory management and thread scheduling), as well as providing a single set of abstractions across the heterogeneous system: all Helios satellite kernels export the same application binary interface (ABI), which, combined with application code being distributed in architecture independent CIL (common intermediate language) format, means that applications do not need to be designed or compiled for a particular core type. Inter-process communication between processes on different satellite kernels is also handled transparently, using the same message channel abstraction as local inter-process communication. These features enable Helios to transparently offload application processes and operating system services to the heterogeneous cores of programmable devices. The current prototype can offload processes from the x86 host

CPU to an ARM-based core on a programmable I/O device, without any modification of the applications' source code.

Helios provides an affinity metric for inter-process message channels to enable applications to express a desire to be *near* to the the process with which they are communicating to reduce communication overheads or to be located on a *different* core type, to isolate each process's computation. Processes can also explicitly specify an affinity for a particular core type. Helios uses these affinities to choose the satellite kernel (and therefore core type) on which the process should be started.

The ability of Helios to transparently offload application code to heterogeneous processing cores is similar to that of the Hera-JVM runtime system presented in this thesis. However, there are a number of significant differences. Helios offloads processes, while Hera-JVM enables offloading of individual threads within a single process. Also, once running, a Helios process is confined to the core type on which it started. Hera-JVM, by contrast, enables transparent migration of threads between core types at runtime. Finally, while process affinity abstracts the process of taking into account the communication overheads of offloaded processes, it does not abstract the performance asymmetry caused by the differing capabilities of heterogeneous core types.

## 3.3 Thread Scheduling on HMAs

As well as presenting a heterogeneous programming environment to developers, HMA processors also introduce asymmetries in performance, depending upon the core type on which application threads are executed. This complicates thread scheduling on these architectures. This section reviews work on thread scheduling algorithms that take processing core heterogeneity into account in an attempt to maximise application performance, energy efficiency or some other metric of interest.

Kumar *et al.* (2003) investigate the potential for energy savings using heterogeneous core types. The cores have identical instruction sets but have different levels of complexity, leading to differences in potential performance due to their different instruction level parallelism capabilities. The authors investigate switching an application's execution between these heterogeneous cores to improve the system's energy efficiency. They do this by sampling a thread's execution and switching to a less complex core, while

turning off the more complex core, if this would lead to a better energy / performance trade-off.

A number of oracle-based and more realistic, but simple, dynamic sampling heuristics were simulated to investigate different methods of selecting the most appropriate core type for different phases of a thread's execution. Their approach samples a thread's energy-delay product on the core types that are one step up and one step down in complexity, compared to the core upon which the thread is currently executing, and then switches to one of these core types if it has a lower energy-delay product than the current core. They achieve up to 80% of the performance of running on the most complex core type, while using only 30% of the energy.

This work only investigated the effect of moving a single thread across these heterogeneous cores as a means of reducing energy consumption. Subsequent work (Kumar *et al.*, 2004) built upon this approach to investigate using heterogeneous multi-core architectures to improve the performance of multi-threaded workloads. They propose a dynamic scheduling system in which a system's execution time is split into: (i) a sampling phase, where a number of thread-to-core assignments are evaluated to compare the system-wide performance they provide; and (ii) a steady phase, where the best assignment evaluated in the previous sampling phase is executed until the next sampling phase is triggered. Various triggering mechanisms are evaluated, including periodic triggering, and triggering based upon changes in behaviour, as measured by the processing core's *instructions per cycle* (IPC) metric.

Becchi & Crowley (2006) propose a similar approach: however, instead of having a system-wide sampling period to evaluate the most appropriate thread-to-core assignment, their scheduler profiles each thread individually. Each thread's IPC value is measured for both core types. This information is used to decide which threads benefit most from being executed on the more complex core type. The threads are then assigned a core type on which to execute in order to maximise system-wide performance, but are periodically migrated to the other core type to update the IPC measure for that core type and, if appropriate, are reassigned. In addition, they evaluate a round-robin approach that simply rotates threads across the available processing core types, with a preference towards fully utilising the more complex (and therefore faster) core types. This approach was found to perform almost as well as the sampling-based approach, while being much simpler to implement.

Both (Kumar *et al.*, 2004) and (Becchi & Crowley, 2006) claim significant speedups in system-wide performance on heterogeneous architectures using these sampling-based approaches. However, these results are arrived at through simulation, with simplified scheduling algorithms, and do not fully model the overheads incurred in a real system.

A two-stage core assignment strategy is proposed by Sondag *et al.* (Sondag & Rajan, 2009; Sondag *et al.*, 2007). In this work, a program is first analysed offline to cluster basic blocks into different *types*, based upon the ratio between different classes of instructions (e.g., integer, floating point, control flow, etc.) within that basic block. This information is used to introduce phase markers to the program's control flow graph, which are then compiled into the application's binary. During execution, the scheduler uses a representative set of basic blocks from each phase type to sample the IPC metric of each phase type on the different core types. This information is used to decide which core type is most appropriate for each phase type, with the scheduler automatically switching a thread's execution between core types when it moves onto a different phase type.

This approach was evaluated in a simulated heterogeneous processor system, consisting of two 2.4GHz x86 processing cores and two underclocked 1.6GHz processing cores, that were otherwise identical. While this approach introduced both memory and time overheads due to the inserted phase markers, it was able to improve the throughput of a heavily loaded system on this heterogeneous environment, compared with the stock Linux scheduler. However, the relatively contrived heterogeneous processing environment and limited workload employed (a continuous stream of SPEC CPU2000 benchmark tasks) limit the general applicability of this evaluation. Additionally, this approach assumes that all code sections which have been clustered into the same phase type by the offline analysis will perform similarly on the different core types. This may not be a valid assumption, if the metrics that are used to cluster the phase types do not take into account all aspects of the code's behaviour[1].

Mogul *et al.* (2008) use a similar model of heterogeneous multi-core architectures. However, instead of sampling a thread's IPC to select the core type on which it should run, a thread is migrated between core types based upon whether it is executing user

[1]For example, cache miss rates can have a very significant influence on per-core-type performance. However, this metric is hard or impossible to evaluate statically ahead of the program's execution and so cannot be used to cluster code blocks in this approach.

level or operating system code. Operating system code is bound to a simple, operating system friendly core, while user-level application code runs on more capable and complex application cores. The reasoning behind this approach is that operating system code, by its very nature, does not exploit the instruction level parallelism available in more complex core types. Therefore the extra energy required by these more complex cores is wasted when it is performing operating system functions. Offloading these OS operations to a simpler core type frees the more complex core to execute another application thread, or to power down if no application threads are available, improving energy efficiency.

The authors modified the Linux operating system so that device management and interrupt handling is bound to a simple (simulated) OS core. Long running system calls (such as `select` and `open`) were also modified, such that they trigger a core-switching function which moves the calling thread's execution onto the OS core for the duration of the system call. Linux's context switching code was optimised to reduce the overhead of switching cores on system calls (Strong *et al.*, 2009). Experimental evaluation of this approach showed that on benchmarks that frequently execute operating system code it can improve energy efficiency, albeit at the cost of some reduction in performance. This work is relatively preliminary and a number of open issues remain, such as preventing OS cores from becoming overloaded when application cores are idle and investigating how these changes interact with Linux's load balancer.

Li *et al.* (2007) present the Asymmetric Multi-Processor Scheduler (AMPS), that is aimed at balancing a system's workload across processing cores with asymmetric performance. Rather than simply attempting to balance the number of threads being executed on each processing core, as a typical load balancing algorithm would, AMPS scales the number of threads executed on each processing core, based upon the cores' relative computing power. This ensures that a processing core's load is proportional to its power, and prevents under-utilisation of fast cores. However, this approach assumes that the performance of each thread in the system scales proportionally to the power of the processing core on which it is executed. This is very unlikely, even in the relatively homogeneous HMA system[1] they use to evaluate this scheduler. A thread which is regularly stalled waiting for data due to cache misses will run slowly regardless of the

[1] In this paper's evaluation, otherwise identical processing cores were throttled to introduce performance asymmetries.

core upon which it is executed, whereas a thread in a tight computational loop will benefit greatly from being executed on a faster core.

The Heterogeneity-Aware Signature Supported (HASS) scheduler (Shelepov *et al.*, 2009) uses per-thread signatures to estimate the performance benefit that a thread is likely to gain from execution on processing cores with different clock frequencies or cache configurations. A signature consists of the expected number of cache misses for this application under a number of different cache configurations, which is approximated through offline profiling of the application. At runtime, the scheduler uses this signature to estimate a *completion time* metric for this thread if run on each core type. This metric approximates the time it will take a core to complete execution of a fixed number of hypothetical instructions, based upon the core's frequency, its cache miss latency and the thread's expected cache miss rate. It is used by the scheduler, alongside per-core load, to select a core type for this thread's execution that maximises system-wide performance. In general, this involves a preference for non-memory-bound threads on the faster cores, since their performance is affected more by core speed than memory-bound threads.

By using signatures that are generated offline, the HASS scheduler removes the need for dynamic profiling of thread behaviour, and thus, reduces runtime overheads. However, by describing a thread's behaviour using only one signature, this approach cannot take into account different behaviour phases of a thread's execution, or variation due to different inputs.

The work reviewed in this section is based upon heterogeneous multi-core architectures where the cores have asymmetric performance, either purely through differences in clock speed, or by having different instruction level parallelism capabilities. There is, therefore, a very clear ordering in processing core performance, from fast to slow. Different threads may benefit by different amounts if executed on a *fast* core, but they will execute no worse than if they were executed on a *slow* core. However, these assumptions do not hold for real world heterogeneous multi-core architectures, such as the Cell processor. The core types on these architectures have different capabilities and trade-offs; there is no clear ordering of core performance. For example, on the Cell processor, a thread that performs a high proportion of floating point operations is likely to perform better on the SPE core than the PPE core, whereas a thread that performs memory accesses frequently will achieve higher performance on the PPE core.

The heterogeneous programming environment of the different core types on HMA processors, such as the Cell processor, has limited research into dynamic scheduling on these platforms; without a mechanism to abstract these heterogeneous programming environments, an application must be statically partitioned between the core types at development-time, which prevents dynamic scheduling of threads between core types.

The only dynamic scheduling system for the Cell processor of which the author is aware is presented in (Blagojevic *et al.*, 2007*a*,*b*). In this system, an application is built such that computationally expensive functions can be offloaded from the PPE to the SPE cores, using an MPI approach. The scheduling system attempts to fully utilise the SPE cores with a combination of: task-level parallelism and loop-level parallelism. The scheduler then attempts to find the most efficient degree of parallelism granularity for a particular application by changing the ratio of task-level to loop-level parallelism. When an application enters a particular phase (these phases are currently manually hard-coded into the application), it is sampled under a variety of task-level and loop-level configurations, with the most effective then being used for the remainder of the application's execution.

This approach is limited for a number of reasons. Firstly, it is heavily application dependent: the application needs to be modified, both to enable functions to be offloaded to the SPE cores and to provide both task-level and loop-level parallelism using APIs such as MPI and OpenMP. Secondly, the scheduler does not take core heterogeneity into account at all: it is left to the application developer to decide, at application design-time, which core type is most appropriate for different sections of the application's execution. Finally, the sampling approach used to select the most appropriate parallel granularity is simply an exhaustive search of the possible configurations, which is not scalable and involves the application running in a large number of sub-optimal configurations before the most optimal configuration is chosen. The approach is only evaluated for a single application (an embarrassingly parallel gene matching algorithm), and therefore the evaluation also has limited applicability.

## 3.4 Managing Non-Uniform Memory

Differences in processing core performance and capabilities are not the only form of heterogeneity that developers can face. Another form of heterogeneity can arise from

non-uniform memory access (NUMA) architectures, where the access latency to a given memory location is not uniform amongst the processing cores. In a NUMA system, the processing cores are grouped into *NUMA nodes*, each of which is directly attached to its own local memory. Processing cores can access data in *remote* memory on other NUMA nodes: however, such accesses are slower than accesses to the core's local memory on its own NUMA node. This introduces a performance asymmetry to the system — if data is not placed near to the cores from which it will be accessed, system performance will suffer. NUMA architectures have been proposed as far back as the 1980's as a means of improving the scalability of multi-processor computer systems. Commodity multi-processor systems from both Intel and AMD are now regularly based upon NUMA architectures. As such, a large body of research has investigated techniques aimed at reducing the impact of this non-uniformity on system performance.

Whilst the underlying machine may have a NUMA architecture, most operating systems present a coherent and homogeneous view of memory to applications. Using an approach similar to that proposed in the Platinum operating system (Cox & Fowler, 1989), virtual memory page tables are kept consistent across NUMA node boundaries, meaning that a virtual address will map to the same physical address, and therefore the same location, from any NUMA node. If such a system is to achieve good performance, each new virtual page requested by an application must be allocated to a physical page on the NUMA node where the threads that are most likely to access this page are executing.

A basic technique, currently employed by most commodity operating systems, is to allocate physical memory on the NUMA node of the processing core upon which the thread requesting this memory is currently executing. This *prefer local* strategy works well if threads only access data that they have themselves allocated; however, threads often access data allocated by another thread. If communicating threads are not located on the same NUMA node, access to these shared data-structures will involve remote, and therefore longer latency, memory accesses. Additionally, if the operating system's load balancing algorithm moves a thread's execution to a core in a different NUMA node, the data which it has previously allocated will remain on the original NUMA node, and will therefore suffer from longer access latency times.

An improvement on this basic strategy is proposed by Bolosky *et al.* (1989). In this work, a virtual page of memory can be *cached* on the local memory of multiple different

NUMA nodes. This means that threads on different NUMA nodes can all access this shared data at local memory access speeds. However, since the data is replicated on a different physical page of memory in each NUMA node, these different copies will not be kept consistent by hardware. Therefore, caching is limited to read-only data. Pages which have been cached are protected as read-only, such that any attempt to write to them will trigger a page fault. Upon such a page fault, the operating system will remove this page's cached status, by updating the page tables on other NUMA nodes to point to this copy of the page, rather than any cached replicas they may have made. Thus, writable pages are restricted to a single physical location, preserving their consistency.

The system proposed by Bolosky *et al.* also attempts to improve system performance by migrating writable pages between NUMA nodes. When a thread performs a write to a page that is located on a different NUMA node, the page is first migrated to its own NUMA node by the operating system. This will reduce the overhead of subsequent writes to this page from this thread; however migrating this page introduces a high overhead, therefore this approach is only beneficial for pages which will be migrated infrequently. To avoid continually migrating pages, the operating system counts the number of times a page has been migrated, and fixes the page to a particular NUMA node if this count increases above a global threshold.

While this approach can provide performance improvements, migrating pages between NUMA nodes is an expensive operation and may degrade performance under certain unfavourable workloads. Additionally, this approach suffers from false sharing, where different data elements are located on a single page, but are accessed by threads located on different NUMA nodes. Pages which exhibit false sharing can never be placed optimally: either the threads accessing the data must be moved, or the application modified to remove the false sharing.

A number of operating systems have also been developed around the premise of partitioning the operating system along NUMA node boundaries in an effort to reduce contention and increase scalability on NUMA systems. The Hive (Chapin *et al.*, 1995) and Cellular Disco (Bugnion *et al.*, 1997; Govil *et al.*, 2000) operating systems are both structured as a set of independent kernels, each of which manages its own NUMA node. As well as providing stronger fault containment (a goal for both of these projects), this approach enables the operating system to scale across many NUMA nodes in a relatively

simple, yet effective, manner. With a single monolithic kernel, all processors share all resources and therefore contend for the same data structures, leading to bottlenecks. By having each independent kernel manage only the resources local to its NUMA node, contention can be reduced.

In the Hive operating system, these independent kernels co-ordinate through distributed system style techniques to provide applications with the illusion of running on a single instance of the IRIX operating system. Hive also enables these kernels to *borrow* memory pages from each other, to prevent a process from being limited to the memory present on a single NUMA node. Threads can migrate between nodes and can access shared memory across NUMA nodes in Hive; however, mechanisms to improve system-wide performance, by allotting data and threads to the most suitable NUMA nodes, were not investigated by the authors.

Cellular Disco takes a different approach. It uses a virtual machine monitor to segregate a computer system's resources between multiple independent kernels, in effect, creating a virtual cluster on a single machine. These virtual machines can be migrated across different physical CPUs and can lend memory to each other to enable load balancing of the system's CPUs and memory. While Cellular Disco can improve the scalability of large multi-core NUMA systems, it does not present a unified system to the application developer, instead presenting a virtual cluster. For an application to make full use of the system, it must be structured as a distributed system, with separate processes executed on each node of a virtual cluster, complicating its development.

The BarrelFish (Baumann *et al.*, 2009) and Helios (Nightingale *et al.*, 2009) operating systems, already discussed in Section 3.2.2, were developed to tackle the challenges of NUMA architectures, in addition to heterogeneous processing cores. Both BarrelFish and Helios also execute independent kernels on each NUMA node to reduce contention and increase scalability. They also provide mechanisms for applications to tune thread and data placement in order to improve system-wide performance on NUMA architectures.

Barrelfish provides a system knowledge base, as described in Section 3.2.2, that can be used by applications to express their resource requirements or behaviour characteristics under different configurations, thus providing information to the operating system with which it can make more informed NUMA node placement decisions. However, this work is still at a preliminary stage, with no real-world results as yet.

The Helios operating system attempts to abstract process placement decisions by providing an affinity metric that can be attached to message channels. This metric can be used by applications to express a preference for close coupling of the processes at either end of a channel (in which case the operating system may decide to place them on the same NUMA node), or a preference for non-interference between the processes (in which case they may be placed on different NUMA nodes). This affinity metric is relatively simple to understand, while still providing the developer with flexibility in process placement. Additionally, since this metric describes the characteristics of the application in an architecture independent fashion, the same affinity settings can be used to tune application placement under different NUMA configurations. However, Helios is limited in that processes cannot span multiple NUMA nodes: if an application wishes to scale across all the processing cores on a NUMA architecture, it must be built as a multi-process application that communicates over message channels.

The Tornado operating system (Gamsa *et al.*, 1999) also aims to provide scalability on NUMA architectures. Instead of structuring the operating system as a set of independent kernels to provide this scalability, Tornado uses an object-oriented design, where each virtual and physical resource is represented as a independent object. Representing resources as independent, localised objects, rather than in a unified shared data-structure, reduces contention and can enable the data-structures to be spread across a non-uniform memory hierarchy in a more efficient manner. Tornado also provides the concept of a *clustered object*, where a shared object can be replicated across multiple NUMA nodes, but access is provided through a unified object reference. A clustered object provides a unified interface, behind which different mechanisms can be used for reading and writing data consistently across these replicas, depending upon the machine's configuration and the likely access patterns to the data-structures.

While the object oriented approach, employed by Tornado, can increase scalability and improve data-structure locality in NUMA architectures, it is challenging to implement and becomes more complex as the number of processing cores and NUMA nodes increase. The use of an object oriented approach also incurs performance overheads, for example, in the form of virtual function invocation. Finally, as with most of the other operating systems described in this section, this work concentrates on improving the NUMA awareness of the operating system itself, but does not provide significant assistance to application code in this respect.

## 3.5 Summary

In order to build software that can exploit a heterogeneous multi-core architecture, a developer must deal with parallel programming, heterogeneous programming environments, scheduling under a heterogeneous processor environment, and management of non-uniform memory. Having to deal with these challenges places an additional burden on the application developer and limits the use of these architectures to specialist domains. This chapter has reviewed work which is aimed at reducing the burden placed on application development by each of these challenges.

This prior work has generally focused on a single aspect of processor heterogeneity. For example, the systems which deal with non-uniform memory, described in Section 3.4, do not take processing core heterogeneity into account, and the thread scheduling systems, described in Section 3.3, limit their approach to architectures in which the cores only differ in performance, not instruction sets or processing capabilities.

Even those systems which do deal with truly heterogeneous multi-core architectures, such as those described in Section 3.2, do so in the context of relatively specialised areas (e.g., stream processing or network packet processing). So far, there has been little research into a single approach towards abstracting all of these challenges, such that non-specialist programmers can easily exploit the potential performance of heterogeneous multi-core architectures.

# Chapter 4

# Abstracting Heterogeneity using Behaviour Characteristics

The processing capabilities of commodity computing systems are likely to become more heterogeneous in the future. Programmers of specialist applications might be willing to deal with this heterogeneity, in order to maximise an application's performance. However, most developers do not want to be burdened with issues relating to the heterogeneous resources available in these architectures: they may wish to take advantage of the performance gains which can be provided by exploiting these heterogeneous processing resources, but not at the cost of extra complexity.

The goal of this dissertation is to provide a runtime system that enables the different core types of a heterogeneous multi-core architecture (HMA) to be exploited without requiring non-specialist programmers to redesign their applications. The approach taken to achieve this aim is to completely hide the details of the heterogeneous architecture from user-level code behind a homogeneous virtual machine interface. The runtime system maps the execution of this homogeneous virtual machine to the heterogeneous processing cores, using its knowledge of the architecture on which it is running to inform its resource allocation decisions. In effect, this approach attempts to move the burden of exploiting heterogeneous architectures out of the hands of application developers and into the runtime system, where it can be managed by specialist programmers.

The heterogeneous core types are likely to have different performance characteristics, depending upon the type of code they execute. For example, a core with a large cache would be suited for the execution of code which has a large data working set,

whereas floating-point intensive code might perform better on a different core type with better floating point hardware.

By hiding an architecture's heterogeneity, developers do not need to manually partition an application's execution between these different core types. This has the advantage of reducing developer effort and application complexity. However, it means that the runtime system must perform this partitioning on behalf of the application if it wishes to make the best use of the different capabilities of the heterogeneous core types available to it. To do so effectively, it must have information about a program's expected execution requirements, so that it can infer the most appropriate core types for the different portions of a program's execution.

The approach taken by this work is to enable application code to be tagged with *behaviour characteristics* (e.g., floating point intensive or random memory access behaviours), either through explicit code annotations, automatically using compile-time tools or dynamically through runtime monitoring. A cost is assigned to each of these behaviour characteristics for each of the different core types supported by the runtime system. This cost is based upon the effect such a behaviour has on performance when performed on this core type. The runtime system then uses a cost function to select the *least costly* core type on which to execute code with a particular set of behaviour characteristics. Thus, these behaviour characteristics can be employed by the runtime system to influence code execution placement on an HMA system.

In the next section, the primary aspects of a processor's heterogeneity that are abstracted by this approach are described and the potential impact that each of these aspects has on an application's performance is discussed. A set of behaviour characteristics are then defined and their effects on heterogeneous architectures are outlined. Section 4.3 describes a variety of methods that can be used to characterise a program's behaviour and tag its code with appropriate behaviour characteristics. Section 4.4 outlines how per-core-type costs can be applied to an application's behaviour, enabling the runtime system to make informed resource allocation decisions. The chapter ends with a discussion of the relative merits of this approach, compared to other approaches.

## 4.1 Aspects of Processor Heterogeneity

There are three main aspects of processor heterogeneity which can have a significant effect on application performance: heterogeneous processing resources (i.e. different core types); a heterogeneous memory hierarchy; and heterogeneous inter-core communication. These features make resource allocation and thread placement decisions more complicated than on conventional architectures.

### 4.1.1 Heterogeneous Processing Resources

By its very nature, an HMA processor contains a number of processing cores types which each have different processing capabilities. For example, a certain core type may be tailored towards high performance execution of floating point code, but perform very poorly, relative to other core types in the system, when it has to successively dereference multiple pointers (e.g. traversing a tree data-structure).

These issues cause difficulties even when an application is manually partitioned, with code being statically assigned to a particular core type ahead of time. A runtime system which abstracts core type heterogeneity must dynamically assign the different threads and phases of execution of a particular application to the different core types at runtime. This makes scheduling and thread placement even more difficult, since the abilities of each processing core must be taken into account, as well as other scheduling parameters, such as which processing core has free cycles and which threads have higher priority.

The different types of processing cores available on an HMA processor typically have different instruction sets. If a thread is being migrated to a different type of processing core, that thread's code may need to be recompiled for the target core's instruction set (either ahead of time, or with a Just In Time (JIT) compiler in the runtime system). Since each core executes a different series of machine instructions and potentially has very different register and stack-frame layouts, migration of a thread between core types cannot occur arbitrarily during a thread's execution. Instead, thread migration must occur at well defined transition points, where the runtime system can ensure that execution of the thread will continue on the target core from the same point, and with the same data, as when it left the original core. These difficulties mean that careful consideration must be given as to whether the benefits of moving or re-scheduling a task outweigh the costs of moving its thread.

### 4.1.2 Heterogeneous Memory Hierarchy

Many HMAs do not have a globally uniform memory model. Instead, most have a heterogeneous memory hierarchy, consisting of multiple different types of memory with differing access times and sizes. Unlike conventional architectures, which hide their memory hierarchy using hardware caches, the different levels of a heterogeneous memory hierarchy are often explicitly visible and accessible. Some of the levels in this heterogeneous memory hierarchy are also non-uniform, in that different processing cores have different access times to the same area of memory. Some architectures even employ local *scratch-pad* memory, which can only be accessed by a single core.

This heterogeneous memory hierarchy complicates data placement. Data should be allocated close to those cores which are most likely to access it. If data is moved between different levels of the hierarchy (i.e. using a software cache to exploit temporal locality), then the software must also deal with issues such as coherency between different copies of the same data, and potentially distinct address spaces for different memory regions.

This heterogeneous memory hierarchy is an additional hindrance to thread migration between cores. If a thread has stored data in the local scratch-pad memory of the core on which it was executing, then this data will need to be moved, along with the thread, to the destination core.

### 4.1.3 Heterogeneous Inter-Core Communication

An HMA often provides multiple inter-core communication mechanisms. Potential options for inter-core communication can include: inter-core shared registers; mailbox registers; a ring-based data-channel; network on-chip communication; or simply shared memory. These communication mechanisms have different throughput, latency and connectivity capabilities. These trade-offs must be taken into account before the most appropriate communication mechanism for a particular situation can be chosen.

Thread placement also plays a significant role in the choice of inter-core communication. Certain communication mechanisms may only be available between neighbouring cores, or cores of the same type. Conversely, the communication patterns between different threads can also inform thread placement decisions. It is often more efficient for neighbouring cores to communicate with each other, both to shorten communication

delays and to reduce overall congestion. Thread placement decisions should take this into account, by placing threads which communicate frequently on neighbouring cores.

Other features of a processor's inter-connect may also influence thread placement. For example, the Cell processor uses a ring-based interconnection system (Ainsworth & Pinkston, 2007). This ring can accept up to three simultaneous memory transactions, as long as these transactions do not overlap. The runtime should therefore try to cluster communication threads in a manner which reduces the likelihood of overlapping transactions.

### 4.1.4 Summary

Different processor architectures will have different degrees of heterogeneity in each of these aspects. For example, a non-uniform memory access (NUMA) based server has a heterogeneous memory architecture, but its processing resources remain relatively homogeneous. A true HMA, such as the Cell processor, is likely to have heterogeneous features in each of these three aspects.

A concrete implementation of a runtime system which hides these heterogeneous features behind a homogenous Java virtual machine interface will be presented in Chapter 5. This runtime system targets the Cell processor and, therefore, must deal with each of these aspects of heterogeneity.

Another version of the runtime system was developed for an x86 server system. This system exhibits a non-uniform memory access (NUMA) architecture, which complicates thread and data placement in the presence of communicating threads. Chapter 8 describes a runtime system that uses knowledge of a program's behaviour to optimise its thread and data placement decisions in such a system.

## 4.2 Behaviour Characteristics

To enable a runtime system to abstract these heterogeneous features, it must track the behaviour of an application and use this information to inform its resource allocation decisions. This section describes the types of program behaviour which could influence an application's performance on an HMA system. Each behaviour characteristic is described in terms of how it should be applied and the way in which the knowledge of such a characteristic can inform the runtime system's allocation decisions, when taking processor heterogeneity into account.

The name given to each characteristic is indicative of an annotation that can tag code which has this characteristic. Code can be tagged through a variety of mechanisms, including these explicit code annotations, as described in Section 4.3.

There are three main categories of behaviour characteristics which are considered in this work: processing requirements, execution behaviour and inter-thread communication. These categories relate to each of the aspects of processor heterogeneity described in Section 4.1.

### 4.2.1 Processing Requirement Characteristics

Different applications will exercise the execution units and other subsystems of a processing core in different ways. For example, a scientific application is likely to heavily exercise the floating point execution unit of the core on which it is executing. A database application, on the other hand, is more likely to make heavy use of the core's memory subsystem. At the same time, the processing core types of an HMA could have differing capabilities. For example, an execution unit or subsystem could be more capable of performing a given type of computation, or the core may have more appropriate instructions for a given workload (e.g., support for vector-based operations or more powerful memory addressing modes). As a result, a particular core type may be more or less suitable for a given thread's execution, depending upon its particular processing requirements.

The set of processing requirement characteristics that the runtime system should track depends upon how significantly the different core types of the targeted HMA vary in processing capabilities. The major functional subsystems of a processing core are: the arithmetic logic unit, which performs integer-based and bit manipulation operations; the floating point unit, which performs floating point calculations; and the core's memory subsystem, which accesses data stored in system memory. Given that the performance of each of these functional subsystems could vary between different core types, the following set of behaviour characteristics were chosen to form the basis of the processing requirement behaviour characteristics:

`@IntegerCode` Tags code which makes heavy use of integer or fixed point operations.

`@FloatingPointCode` Tags code which makes heavy use of floating point operations.

`@DataAccessCode` Tags code which spends most of its time accessing data in memory.

While application code can be tagged with any of these behaviour characteristics, the runtime system is free to ignore this information if it is not relevant for the target HMA. For example, if the integer performance of two core types is identical, then there is no need for the runtime system to track the `@IntegerCode` behaviour characteristic.

This set could also be augmented with more specialised processing behaviour characteristics if core capabilities are similarly fragmented. For example, arithmetic code could be tagged as `@MultiplicativeCode` or `@DivisionalCode` if the performance of these operations varied between core types in a different manner from the more general `@IntegerCode` characteristic. These more specific characteristics could be ignored or interpreted as one of the more general characteristics if they are unnecessary on a particular architecture.

The influence of a program's processing requirements on its execution speed under different processor types is shown by the micro-benchmarks presented in Section 5.5.2. They show that integer and floating point operations are between three and five times faster on the Cell Processor's SPE core type than its PPE core type. On the other hand, data access operations can be up to five times slower on the SPE core than they are on the PPE core.

### 4.2.2 Execution Behaviour Characteristics

As well as the different processing requirements a program may have, other aspects of its execution behaviour can affect its performance on the different core types available in an HMA. For example, some processing core types (e.g. stream processors (Kapasi *et al.*, 2003)) are well suited to performing streaming accesses across a large sequential area of memory. However, these same core types often have long memory access latencies and, therefore, perform very poorly when accessing small areas of memory in a random fashion. Therefore, code which accesses memory sequentially, e.g. by iterating over an array or matrix would perform well on such a core, while code which accesses memory in a more random fashion, e.g. by traversing a tree, would perform poorly.

To exploit the full potential of a target HMA, the runtime system must be aware of the expected behaviour of an executing thread to make informed thread placement and migration decisions. The following behaviour characteristics enable an application to inform the runtime system of its expected execution behaviour.

`@SequentialAccessBehaviour` This characteristic tags code which accesses data sequentially, for example, iterating over an array. The runtime system will preferentially execute code with this annotation on cores which can stream or pre-fetch sequential memory accesses in the background.

`@RandomAccessBehaviour` This characteristic describes code, such as traversal of a tree or linked list, where memory addresses are being accessed in a non-sequential manner. Pre-fetching and caching are less effective under this type of memory access pattern, and therefore the runtime system will try to execute code with this annotation on cores which are closely coupled to memory. By being more closely coupled to memory, these cores have lower memory access latencies, and will therefore incur less overhead when performing random memory accesses.

`@LargeWorkingSet` A program that has a large working set of data that it accesses regularly is likely to perform better on core types which have more cache memory available to them and can, therefore, hold more of the program's working set in the cache (Smith, 1982). The `@LargeWorkingSet` behaviour characteristic informs the runtime system that the tagged block of code is expected to operate over a relatively large amount of data and may, therefore, benefit from being executed upon a core type with a large cache.

The working set size at which the runtime system should begin incorporating this `@LargeWorkingSet` behaviour characteristic into its thread placement decisions depends upon the size of the different core types' caches for the HMA it is targeting. To enable the runtime system to decide whether the program's working set is large enough to have an influence on its scheduling decisions, the `@LargeWorkingSet` characteristic can be augmented with the expected size of the code's working set. This size could be estimated or found by profiling the application ahead of time. In the Java annotation syntax, this can be expressed by parenthesising the annotation with an element-value pair, for example, `@LargeWorkingSet(SizeKB=256)`, specifies a working set size of 256KB. The runtime system can then selectively ignore this characteristic if the size of the working set is smaller than the size of the smallest cache of the core types it is managing.

`@IoAccessBehaviour` This characteristic specifies that a given region of code is expected to access I/O devices, for example, reading files on a disk drive or sending packets across the network. Typically, only one of the processing core types in an HMA can perform I/O operations directly. The runtime system will, therefore, try to execute code with this characteristic on a processing core which is capable of directly implementing the necessary I/O operations.

`@ExceptionsLikely` Handling exceptions can be very costly for some processing core types. For example, some cores do not support hardware trap instructions and must therefore emulate a trap instruction in software whenever an exception occurs. This characteristic notifies the runtime system that a given region of code is likely to generate many exceptions. The runtime system can then choose to place this thread on a core better suited to dealing with exceptions.

These behaviour characteristics can have a significant influence on the performance of programs on the different core types of an HMA processor. For instance, in the micro-benchmark presented in Section 5.5.2.2, simply increasing the data working set of a program reverses its relative performance on the two core types of the Cell Processor. As the program's working set is increased, its performance changes from running two times faster on the Cell's SPE core, compared to the PPE core, to running twice as fast on the PPE core than the SPE core.

As with the processing requirement characteristics, the runtime system is free to ignore those execution behaviour characteristics which do not influence performance on the core types of the architecture it is targeting.

### 4.2.3 Thread Communication Characteristics

Communication between different threads of the same process often occurs through shared data objects. In a heterogeneous architecture, the efficiency of inter-thread communication using shared data-structures will depend upon a number of factors, such as: the distance between the memory holding the shared data and the cores accessing this data; the presence of a shared data cache between the cores accessing the data; and the ability to use specialised inter-core communications mechanisms, such as direct scratchpad to scratchpad memory transfers or nearest neighbour registers.

For example, a micro-benchmark, presented in Section 8.1, aimed at stressing inter-thread communication on a non-uniform memory access (NUMA) architecture machine, shows a 60% decrease in performance if communicating threads and their shared data are placed on *far away* NUMA nodes. On NUMA systems, the performance of inter-thread communication is dependent upon the placement of the communicating threads and their shared data. To make informed decisions regarding these issues, the runtime system needs information about which threads communicate with each other, and through which data objects they are likely to communicate.

The approach taken in this work uses the concept of thread teams to define the scope of inter-thread communication. A thread can be tagged as being a member of one or more thread teams using a `@ThreadTeam(name="<name of team>")` characteristic. Threads which communicate with each other extensively are grouped into the same thread team. The runtime system will then try to optimise thread placement, such that members of the same team share efficient inter-core communication links (e.g., being placed on the same NUMA node).

While this thread team approach is conceptually simple, the ability to tag a thread as being a member of multiple teams endues it with the power to describe relatively complex communication patterns in a simple manner. For example, in the communication pattern shown in Figure 4.1, two sets of worker threads communicate amongst themselves. These threads can therefore be grouped into two teams, called Workers_1 and Workers_2. The runtime system may, therefore, decide to cluster these two thread teams on different NUMA nodes. A co-ordinator thread that communicates with both thread teams is tagged as being a member of both the Workers_1 and Workers_2 thread teams. The runtime system will try to minimise the communication cost between this thread and both worker thread teams. Thus, it may place this co-ordinator thread on a NUMA node which neighbours the two nodes on which the Workers_1 and Workers_2 thread teams were placed.

As well as tagging threads, it may also be beneficial to tag shared data with the thread teams which will access the data. By default, when a thread allocates data, it should be allocated on that thread's local node. This will provide this thread and other members of its team with the lowest latency access to this data. However, if this data is to be accessed by other thread teams, it may be better to allocate the data on a different node. For example, suppose the co-ordinator thread in the previous example

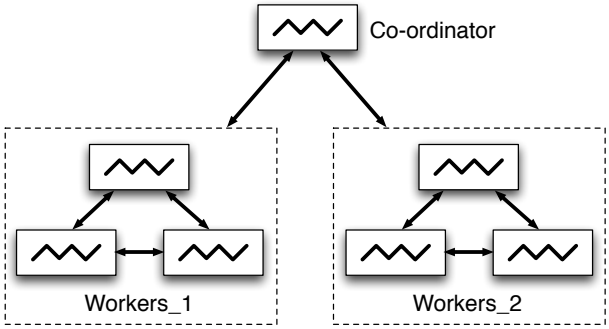



Figure 4.1: A typical thread communication pattern.

allocates data which is intended for use by a thread in the Worker_1 communication team. Instead of allocating this data on its local NUMA node, allocating the data on the node on which the Worker_1 team is located may lead to lower memory access overheads overall. By tagging data with the thread teams that access the data, the runtime system can choose the node which provides the least costly access to this data for these threads.

## 4.3 Tagging Mechanisms

A program can be tagged with these behaviour characteristics in a number of different ways. The most direct mechanism is to explicitly annotate source code with appropriate behaviour characteristic annotations. The addition of behaviour annotations to code could also be automated, by employing source code analysis tools or involving the compiler in the behaviour characteristic tagging process. Finally, the runtime system can monitor the behaviour characteristics of a program at runtime, in order to infer its behaviour characteristics in the absence of annotated code. This section outlines each of these options.

### 4.3.1 Explicit Annotations

A number of modern programming languages provide support for code annotations. These annotations can be used to provide meta-data about the code to the code's compiler, deployment tool or runtime system, without affecting the operations which

are performed by the code[1]. Code annotations are a particularly appropriate means of tagging the code with its behaviour characteristics, because the code itself need not be modified and annotations can often be inserted into code without having to recompile the code itself.

This dissertation focuses on Java as an application development language. Java was chosen for a number of reasons. One of these reasons was its intermediate bytecode representation, which provides a format for the distribution of machine architecture independent code to HMAs, which may employ different instruction sets in their different core types (see Chapter 5). Another reason for this choice, however, was Java's support, as of version 1.5, for code annotations. These annotations are retained through the source code compilation and are included in the compiled Java class files, alongside a program's bytecode. Thus, the runtime system can retrieve this information when loading a class's code.

The behaviour characteristics described in the previous section can be expressed as Java annotations. Java annotations can target a variety of program elements, such as classes, methods, object fields and method parameters. To define the behaviour characteristic annotations, it is necessary to identify the program elements which each annotation can target.

The behaviour characteristics all describe an aspect of a thread's behaviour during its execution. By tagging the class used to create a thread with the appropriate annotation, the annotations can be used to define the thread's behaviour throughout its execution. The thread will retain the characteristics defined by these annotations for the duration of its execution.

A thread may have different behaviour phases during its execution, or may act differently depending upon its input. Therefore, as well as targeting the threads themselves, these annotations can also target blocks of code to define their behaviour. Unfortunately, Java annotations cannot target arbitrary blocks of code. Therefore, this support is provided by enabling the targeting of method declarations by behaviour characteristic annotations. If a thread calls a method annotated with a particular behaviour characteristic annotation, it will be tagged with this characteristic for the duration of

[1]For example, Krintz & Calder (2001) propose a system in which code annotations are used to inform a runtime system of the potential to apply various compiler optimisations to an application's code, without changing the code's semantics.

the method call (i.e., it will retain the characteristic throughout this method's entire call stack, until the method returns).

As well as the behaviour characteristic annotations, a set of anti-behaviour characteristic annotations (e.g., `@NonFloatingPointCode`, etc.) can be used to remove a characteristic from a thread for a given period of its execution. These can target the same program elements as the behaviour characteristic annotations.

Some of the behaviour characteristics could usefully target data, as well as code. For example, the `@ThreadTeam` annotation could be used to indicate the team of threads which are likely to access a shared data object, and the `@SequentialAccessBehaviour` and `@RandomAccessBehaviour` annotations could tag data which is being accessed in a sequential or random manner. Annotating data in this manner would enable the runtime system to optimise data placement, in the case of the `@ThreadTeam` annotation, or data access, in the case of the `@SequentialAccessBehaviour` and `@RandomAccessBehaviour` annotations. In this dissertation, the focus is on the tagging of code, rather than data. However, data could be tagged with these characteristics by allowing the annotations to target the data's type declaration or the field or local variable used to hold the data.

Listing 4.1 shows an example of how these behaviour characteristics can be employed. In this example, the Worker class (which extends the Thread class) is annotated with `@IntegerCode`. Thus, any threads of this type will be tagged as making heavy use of integer operations. The `floatingCalcs()` method performs floating point operations, and therefore, is annotated with the `@FloatingPointCode` behaviour characteristic. If a thread calls this method, it will be tagged as having a floating point behaviour characteristic for the duration of this method call.

The main method creates two of these Worker threads. Since these threads are also tagged as being members of the same "workers" team, the runtime system will try to place these threads such that they can efficiently communicate with each other. The two threads are also passed different types of store objects; `worker1` is passed an ArrayStore object, which is annotated with the `@SequentialAccessBehaviour` characteristic, whereas `worker2` is passed a LinkedListStore object, annotated with the `@RandomAccessBehaviour` characteristic. Thus, when the `worker1` thread calls `store.add(value)`, on Line 20, it will be tagged as having sequential memory access

```
1  @SequentialAccessBehaviour
2  class ArrayStore extends StoreType { ... }
3
4  @RandomAccessBehaviour
5  class LinkedListStore extends StoreType { ... }
6
7  @IntegerCode
8  @ThreadTeam(name = "workers")
9  class Worker extends Thread {
10
11   private StoreType store;
12
13   Worker(StoreType s) {
14     store = s;
15   }
16
17   void run() {
18     // integer calculations ...
19     floatingCalcs();
20     store.add(value);
21   }
22
23   @FloatingPointCode
24   void floatingCalcs() {
25     // floating point calculations ...
26   }
27 }
28
29 class Coordinator {
30   void main (String args[]) {
31     Worker worker1 = new Worker(new ArrayStore());
32     Worker worker2 = new Worker(new LinkedListStore());
33     worker1.start();
34     worker2.start();
35   }
36 }
```

Listing 4.1: Example of Behaviour Characteristic Annotations.

behaviour, whereas, the `worker2` thread will be tagged as performing random memory accesses.

This example is purposefully contrived in order to demonstrate a varied selection of the behaviour characteristics in a small amount of code. It is not expected that a real world application would tag every element of a program; rather, the small minority which would be likely to have a significant impact on the program's performance.

### 4.3.2 Source Code Analysis Tools

Instead of having a developer manually annotate the behaviour of an application, a source code analysis tool could be employed to automatically infer a program's behaviour and tag its code appropriately.

The processing requirement behaviour characteristics are a potential candidate for inference using static code analysis techniques. The relative proportion of different types of processing operations could be calculated for each code block of an application during compilation. The tool then needs to approximate how often each of these code blocks are likely to be executed during the program execution, to infer the influence of the code block's operations on overall program behaviour. Data-flow analysis (Allen & Cocke, 1976) and loop bounding static analysis (Healy *et al.*, 1998) techniques could be adapted for this process.

The other characteristics would be more difficult to infer through static analysis. Instead, off-line profiling could be used to gather information about a program's behaviour and insert these characteristics into the code before the program is distributed. Profiling of the execution behaviour characteristics would be relatively straightforward. Inferring thread teams through profiling would be more challenging. A profiler which monitors access to objects on a per-thread basis could infer a program's inter-thread communication behaviour. However, monitoring every object access would have a very high overhead, even for an off-line profiler. This overhead could be reduced by employing escape analysis techniques (Choi *et al.*, 1999) to extrapolate the set of objects which can escape a thread's context, and are therefore useful to monitor for inferring inter-thread data sharing.

Once these tools have established the expected behaviour characteristics of a program, they can insert them into the program, by augmenting its code with the same annotations as used by developers to explicitly annotate their code. In the case of

Java code, this could be accomplished by using a bytecode manipulation framework, such as ASM[1]. These automated approaches can therefore be freely mixed with manual tagging, through explicit annotations, as necessary.

The creation of source code analysis tools for automated program behaviour tagging is beyond the scope of this dissertation, and is left as future work. However, the conclusions drawn by this work should be applicable, no matter the source of the behaviour characteristic annotations.

### 4.3.3 Runtime Monitoring

A runtime system can augment the behaviour characteristic information that has been inserted, ahead of time, into a program's code, or even eliminate the need for tagging of code entirely, by monitoring different aspects of a program's behaviour directly at program runtime. This would enable completely unmodified code to exploit heterogeneous architectures, without having to be annotated with its behaviour characteristics.

Chapter 7 describes an approach for monitoring a program's processing requirement behaviour characteristics at runtime. In this approach, blocks of code are *scored*, ahead of time, with regard to the proportion of different classes of processing operations performed by the block of code. The runtime system then uses these scores to update per-thread processing requirement characteristics, whenever a block of code is executed.

Monitoring of execution behaviour characteristics, such as `@IoAccessBehaviour` or `@ExceptionsLikely`, could be provided using software counters which are incremented on each I/O operation or exception. Given the relatively high cost of I/O operations and exceptions, this monitoring would add negligible additional overhead. The other execution characteristics (memory access behaviour and working set size) could be inferred using the hardware cache miss counters found in most modern processors.

Inferring inter-thread communication to build thread teams at runtime is a greater challenge. One possibility, for a Java runtime system, would be to monitor the set of objects locked by each thread using thread synchronisation operations. By comparing the overlap between each thread's set of lock objects, the runtime system can cluster threads into teams which are likely to be sharing data.

[1] `http://asm.ow2.org/`

This dissertation limits its focus to the runtime monitoring of performance requirement behaviour characteristics (Chapter 7). Runtime monitoring of execution behaviour characteristics and thread teams are left as future work.

## 4.4 Costing Behaviour

Once a program's behaviour characteristics have been identified, the runtime system can use this information to inform its thread placement and resource allocation decisions. To do so, the runtime system must account for the influence of each behaviour characteristic on the decision it is making. In this work, a cost is assigned to each of the behaviour characteristics for each heterogeneous resource (e.g., processing core type). A cost function is then used to calculate the total cost for each of the heterogeneous resource types from which the system can choose to allocate to a particular request. These costs can then be used to inform its decision, by either directly choosing the *least-cost* resource, or by using these costs as a guide alongside other factors (e.g., processing core utilisation).

To describe this more concretely, take the example of a thread placement decision on the Cell Processor. The Cell processor has two processing core types: an SPE core type with very good floating point performance; and a PPE core type which can access memory more efficiently. Each of these core types are therefore assigned a different cost value with regard to the floating point and data access behaviour characteristics; the SPE core will have a lower cost for the floating point behaviour characteristic than the PPE core, while the PPE costs will have a lower cost for the data access behaviour characteristic than the SPE core.

A per-core-type cost function calculates the cost of executing a thread on a particular core type. This cost function sums the costs of those behaviour characteristics with which the thread has been tagged. For this example, the following cost functions can be used for each core type:

$$C_{spe} = 1 \cdot B_F + 2 \cdot B_D$$

$$C_{ppe} = 4 \cdot B_F + 1 \cdot B_D$$

where $B_F$ and $B_D$ are one if the thread is tagged with the floating point or data access characteristics respectively, or are otherwise set to zero. In this case, the floating point

behaviour is four times more expensive on the PPE core than the SPE core, while data access is half as costly. Appropriate cost values for each behaviour characteristic can be found by profiling the effect each behaviour characteristic has on per-core-type performance (these costs for the Cell processor are based on the results of benchmarking experiments presented in Section 5.5).

In order to select the most appropriate core type on which to execute a thread, the runtime system will calculate its cost for both core types. The most appropriate core type on which to execute this thread, based upon these behaviour characteristics, is the one which provides the *least cost* using this process.

The cost function presented here is simplified to outline the overall approach. Section 6.2 details the creation of a more appropriate cost function for use in thread placement and migration decisions on the Cell processor.

The same cost function approach can be used to inform thread and data placement decisions on non-uniform memory access architectures, based upon thread team characteristics. In this case, the aim is to cluster threads that communicate onto the same NUMA node, but partition non-communicating threads onto different NUMA nodes[1].

To enable this support, each thread team is assigned a *preferred node*, based upon the node on which the first thread of this team is placed. When a new thread is created, the runtime system examines the teams of which it is a member, and creates a cost function for each node of the system, based upon these teams. The cost of placing this thread on a particular node is calculated as the sum of the number of hops between this node and the preferred node of each of the thread teams of which the thread is a member. To partition non-communicating threads onto different nodes, an extra cost is added for each team, located on this node, of which the newly created thread is not a member. The runtime system can then choose a core on the least costly node for the execution of this thread.

Figure 4.2 shows an example of this approach to thread placement. In this example, teams A to D already have preferred nodes, as shown in the figure. The runtime system must decide upon the optimal placement of a newly created thread, which has been tagged as a member of teams A, B and D. The runtime system calculates the cost of

[1]Pushing non-communicating threads onto cores which are on different NUMA nodes will help to reduce pollution of a processor's L3 cache, which is typically shared between all the cores on the same NUMA node.

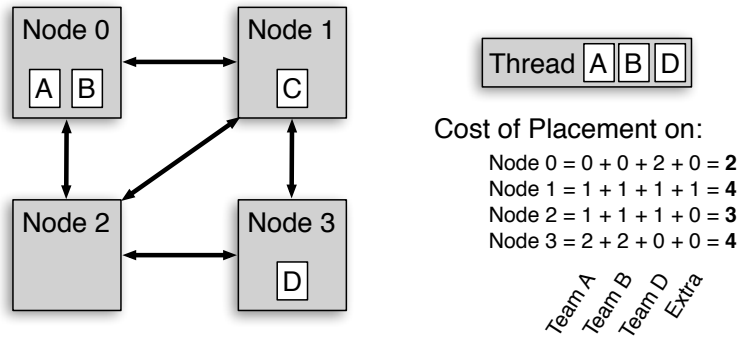



Figure 4.2: Using thread teams to place threads on a NUMA system.

placing the thread on each Node. Node 0 is zero hops away from team A and B, but two hops away from team D, leading to a total cost of two. Nodes 1 and 2 are both one hop away from all of this thread's teams; however, Node 1 has an extra cost of one, since it is the preferred node of an unrelated team (team C). Finally, Node 3 is two hops away from both team A and B, but is zero hops away from Node D.

In this case, Node 0 is the least costly node on which to place this thread. However, other factors may cause the runtime system to select an alternative node. For example, if all the processing cores of Node 0 are already heavily loaded with the threads of team A and B, a core on the second least costly node (Node 2) may be selected to execute this thread. This approach to thread placement on a NUMA system is investigated in more detail in Chapter 8.

## 4.5 Discussion

Abstracting a system's heterogeneity using a program's behaviour characteristics is advantageous for a number of reasons. The main advantage is that by abstracting the heterogeneous architecture from application developers, they do not require detailed knowledge of the capabilities of each heterogeneous core type and the system interconnect to exploit an HMA. The burden of dealing with these features is shifted from the application developer to the runtime system developer, where greater knowledge of the target architecture can be expected.

In the case of source code analysis or runtime monitoring of behaviour characteristics, the burden of dealing with system heterogeneity can be removed from the developer

entirely. When explicit annotations are used to tag code with behaviour characteristics, the application developer's task is transformed from having to understand and effectively exploit the heterogeneity of an architecture, to simply understanding and describing the heterogeneity of their own application, using behaviour characteristics.

Another important advantage of this approach is that it is portable between systems. Since behaviour characteristics are not explicitly related to a particular heterogeneous architecture, the behaviour characteristics used to tag an application are valid, no matter the architecture of the system on which it is run. The runtime system can choose what effect each behaviour characteristic has on its placement decisions, depending upon the architecture it is targeting.

Of course, this approach is not the most appropriate for every situation. By adding another layer of abstraction, application performance could suffer. Abstracting processor heterogeneity also removes the ability of a developer to explicitly tailor an application for a particular architecture. This approach is, therefore, complementary to approaches that provide more visibility of an architecture's heterogeneity (e.g., (Schüpbach *et al.*, 2008), (Munshi, 2009) and (Dagum & Menon, 1998)). Developers who wish to target a particular architecture and require the maximum performance from this architecture for their applications are likely to be willing to deal with the architecture's heterogeneity to do so.

However, the vast majority of developers want their applications to be portable to different architectures and do not want to deal with the added difficulties of targeting specific heterogeneous architectures. The remainder of this dissertation demonstrates that by abstracting a processor's heterogeneity using behaviour characteristics, the needs of these non-specialist developers can be supported.

# Chapter 5

# Hera-JVM: A Runtime System for Heterogeneous Architectures

To investigate the feasibility of abstracting the heterogeneity of these heterogeneous multi-core architectures (HMAs) from application developers, a prototype runtime system has been created. This runtime system, called *Hera-JVM* (*Heterogeneous Resource Aware — Java Virtual Machine*)[1], supports the execution of Java applications on a particular HMA - the Cell processor. Hera-JVM hides the heterogeneity of the Cell processor's architecture by presenting the application developer with the illusion of running on a homogeneous, multi-core processor, through a virtual machine abstraction. Unmodified Java applications can be executed by Hera-JVM on either of the two processor core types available on the Cell processor. Threads can be migrated between core types transparently from the point of view of the application developer, and without requiring modification of application source code.

This chapter describes the challenges involved in designing a runtime system which can support the execution of an application on two very different architectures concurrently. Later chapters build on this abstraction of processor heterogeneity to investigate automatic thread placement based upon application code behaviour characteristics.

The features of the Cell processor which make it challenging for application development, as well as making it an interesting target for this research, are described in Section 5.1. The main design decisions and overall approach taken in the creation of Hera-JVM are then presented in Section 5.2. To fully support the Cell processor, a

[1]The name Hera-JVM was also chosen due to Hera being the Greek goddess of marriage, and Hera-JVM attempting to marry the heterogeneous core types of an HMA processor behind a homogeneous interface.

Java compiler and low-level runtime support for the secondary core type of the Cell processor (the SPE cores) had to be added to Hera-JVM. These cores have a number of traits which make efficient execution of Java code challenging. Section 5.3 describes the implementation of the compiler and runtime system used by Hera-JVM to support efficient execution of Java code on the SPE cores. The implementation of a transparent migration mechanism, for moving threads between the two core types supported by Hera-JVM, is then presented. Finally, experimental analysis of Hera-JVM is performed in Section 5.5 to investigate the efficacy of this virtual machine approach for hiding processor heterogeneity, and to discover the performance characteristics of each core type under different Java workloads.

## 5.1 The Cell Processor

The Cell processor (Chen *et al.*, 2007; Hofstee, 2005; Pham *et al.*, 2005) was developed primarily for multimedia applications, specifically the game market, where it is the main processor used by the Sony Playstation 3. It is also being actively employed in a variety of other areas, such as: scientific and high performance computing, being incorporated into IBM Bladecenter servers and supercomputers (e.g., the Roadrunner machine (Barker *et al.*, 2008)); and in high performance media devices, for example, the Toshiba SpursEngine co-processor, employed in their Qosmio laptop range to accelerate video encoding and decoding operations.

It was selected as the target architecture for the Hera-JVM runtime system for a number of reasons. Firstly, the Cell processor is one of the few HMAs which is currently widely deployed, therefore, the results gained on this platform should be applicable in a real world setting. Secondly, it is a highly heterogeneous architecture, having an asymmetric memory hierarchy and processing cores which not only differ in performance, but also in capabilities and instruction sets. This enables investigation of techniques for abstracting all these different types of heterogeneity. Finally, unlike some other HMAs, such as the Intel IXP network processor (Adiletta *et al.*, 2002), both core types on the Cell processor have the capability to support the majority of Java code. Therefore, thread placement decisions can be made based upon the expected performance of the thread on the different core types, rather than being limited to a

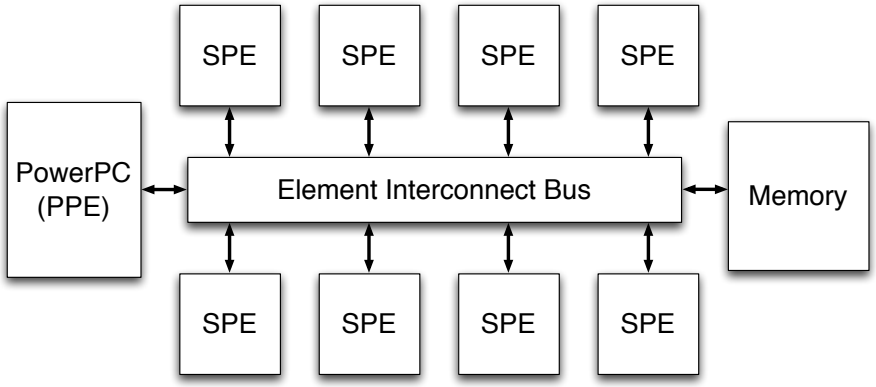



Figure 5.1: The architecture of the Cell processor.

particular core type due to the thread requiring features which are unsupported on the other core type.

The Cell processor contains two different processing core types: a single *Power Processing Element* (PPE) core; and eight *Synergistic Processing Engine* (SPE) cores (Figure 5.1). The PPE is intended to manage the system overall and co-ordinate the SPEs. As a PowerPC-based core, it can support the Linux operating system and run any applications compiled for the PowerPC architecture. The SPEs are designed to perform the bulk of the computation on the Cell processor. They have a unique instruction-set, highly tuned for floating point, data-parallel workloads. The SPEs do not run any operating system code, relying on the PPE to perform operations such as page table updates or file I/O.

The processing cores share access to external DRAM memory through a circular ring-based Element Interconnect Bus (Ainsworth & Pinkston, 2007). The PPE core has a two-level cache to reduce data access latencies, with a 64KB L1 cache (split evenly between data and instruction caches) and a 512KB L2 cache.

Unlike the PPE, the SPE cores do not have transparent hardware caches for accessing main memory; each SPE contains 256KB of non-coherent local memory. The processing elements of the SPEs can access only this local memory directly. To access data in main memory, an SPE must initiate a Direct Memory Access (DMA) transfer to copy the data from main memory to its local memory. It can then modify this data in local memory, but must initiate another DMA transfer to write the results back

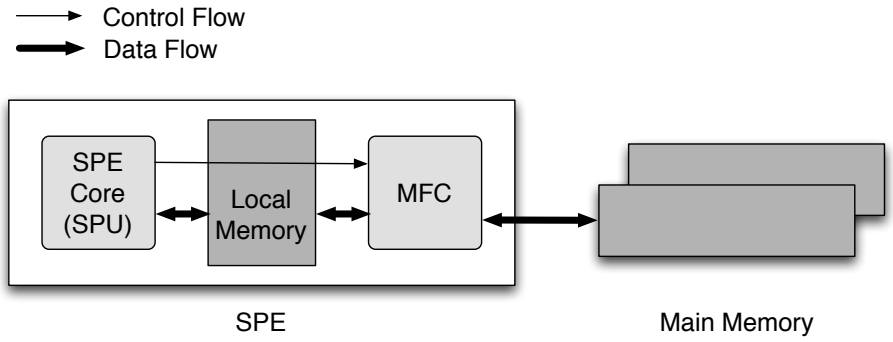



Figure 5.2: An SPE core's memory subsystem.

into main memory. These DMA transfers are supported by a Memory Flow Controller (MFC) unit associated with each SPE (see Figure 5.2). The MFCs have virtual memory support, translating virtual memory addresses to physical addresses using the page tables that are maintained by the operating system running on the PPE core. Thus, different threads of a single process share a consistent view of virtual memory, whether running on the SPE or PPE cores. However, data which has been copied to an SPE's local memory is not automatically kept consistent with the original copy in main memory, or copies made by other SPE cores, meaning cores do not automatically share a coherent view of data.

By offloading memory reads and writes to the MFC, the SPE cores can perform large block transfers very efficiently, however, small transfers are much less efficient, due to the overhead involved in setting up a DMA transfer. This approach suits the intended target applications of the Cell processor — large blocks of data (e.g. image fragments) being loaded, processed and then streamed out to main memory. However, it is much less suited to general purpose computation, where memory is seldom accessed in large chunks, and developers expect threads on different cores to share a coherent view of memory. Developing an efficient and coherent memory access mechanism for the SPE cores was therefore an important consideration in the creation of Hera-JVM (see Section 5.3).

These features, of heterogeneous core types and an unusual memory architecture, make developing efficient, or even correct, applications for the Cell processor challenging. The aim of Hera-JVM is to hide these challenging architectural features behind a more typical symmetric multi-core virtual machine abstraction, whilst still trying to preserve the performance benefits provided by the Cell processor's heterogeneity.

## 5.2 Hera-JVM Design Decisions

The philosophy behind Hera-JVM is to hide the Cell's challenging heterogeneous architecture behind a seamless homogeneous Java virtual machine abstraction, such that it can run unmodified, multithreaded Java applications. To enable applications to exploit the heterogeneous cores of the Cell processor under this homogeneous virtual machine abstraction, Java threads should be able to migrate transparently between core types.

This philosophy influenced a number of Hera-JVM's design decisions. Firstly, the runtime system must provide support for the full Java specification on both core types, to enable seamless migration of threads between them. If an operation cannot be supported by a particular core type, a stub must be created to transparently hide this limitation, by executing the operation on the other core type. Since trapping to another core type is an expensive operation, this should only be performed for non-performance critical operations, with the majority of the Java specification being supported natively by both core types. Secondly, threads should share a single, consistent view of the heap, with an object reference pointing to the same data when accessed from any core. By employing these two design decisions, application developers need not worry, nor indeed know, which core type their code is being executed on. Running a thread on a different core type may affect performance, but will not affect program correctness.

Version 3.0 of the JikesRVM (Alpern *et al.*, 2005) Java Virtual Machine was used to form the basis of Hera-JVM. JikesRVM is a fully capable JVM with performance comparable to production JVMs. It supports both x86 and PowerPC processor architectures under Linux, and can thus be run on the PowerPC-based PPE core of the Cell processor without any modification. The main enhancements made to JikesRVM to create Hera-JVM were: runtime and compiler support for the SPE cores; changes to the overall runtime system to support simultaneous execution of a Java application across two different architectures; and support for migration between the different core types of the Cell processor.

JikesRVM (and thus Hera-JVM) is designed as a *Java-in-Java* virtual machine. This means that the majority of the runtime system is written in Java. To build the runtime system, this runtime Java code is bootstrapped, using a previous build of the JVM or another JVM entirely, to create a machine executable boot-image.

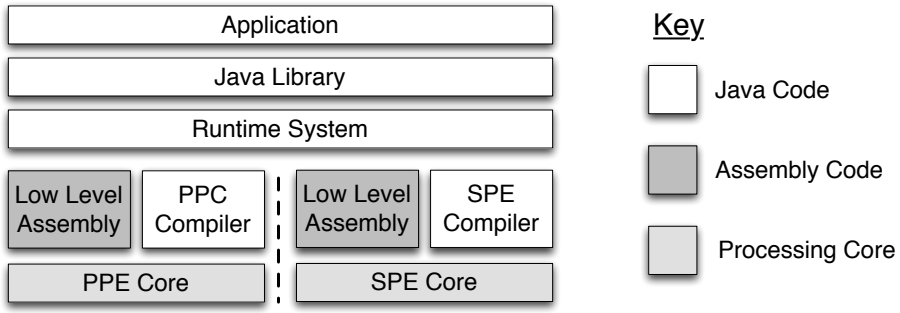



Figure 5.3: The structure of Hera-JVM. Much of Hera-JVM's runtime can be shared by both cores, given its Java-in-Java design.

The fact that much of the runtime system is itself written in Java confers a number of advantages in the design of Hera-JVM. Given Java's *write once, run anywhere* philosophy, this code is largely portable. Thus, other than a small number of architecture-specific routines[1], the same runtime code is shared by both core types (Figure 5.3). This approach extends the philosophy of hiding the architecture's heterogeneity right through application code, the Java Library code and the majority of the runtime system's code, simplifying the runtime's design. This also improves the runtime's maintainability. Using separate runtime systems on each core type — an approach employed by CellVM (Noll *et al.*, 2008) — introduces the risk of integration bugs and inconsistencies in data structures shared between the two runtime systems. Having a single runtime system for both core types limits the potential for such incompatibilities, and means that bug fixes and improvements to the runtime system are immediately applicable to both core types.

Hera-JVM is a non-interpreting JVM; all application, library and runtime Java methods are compiled to machine code before being executed. A Java bytecode to machine code compiler is therefore required for each processor architecture supported by Hera-JVM. A non-optimising, bytecode to SPE machine code compiler, described in Section 5.3, was built for Hera-JVM to enable execution of Java code on this core type. This compiler directly translates fundamental bytecodes, such as arithmetic and

[1]Some low-level routines (e.g. reference collecting code in the garbage collector) depend upon stack-frame layout, which varies between core types, and so have per-core type versions. Low-level routines that set up a core for code execution (e.g. the final stages of exception delivery and thread context switching support code) are also core type specific.

method invocation operations. More complex bytecodes, such as the `new` bytecode for object allocation and the `athrow` bytecode used to throw exceptions, trap into runtime system code, where they are implemented in Java. Since this runtime code is shared by both the PPE and SPE cores, these more complex bytecode operations can essentially be leveraged from the existing JikesRVM implementation, after the addition of some glue code. Similarly, complex runtime system components, such as file handling, class loading or thread scheduling, can be supported on either core type with little modification.

Other than the subset of the runtime system methods which are pre-compiled into the boot-image, all Java methods are compiled *just in time*. Thus Java code is distributed in architecturally-neutral Java Bytecode, which will only be compiled for a particular core architecture if it is to be executed by a thread running on that core type. Since it is expected that most applications will exhibit a partitioning between code which is best run on the PPE or the SPEs, most methods will only ever be compiled for one of the two core architectures. Thus, the compilation overhead (both in time and memory requirements) of running an application on an HMA, such as the Cell, need be little more than running on a single architecture processor.

## 5.3 Executing Java Code on the SPE Cores

To support execution of Java on the SPE cores of the Cell processor, a Java bytecode to SPE machine code compiler and some low-level assembly runtime support code is required by Hera-JVM. Supporting execution of Java threads on the SPE cores presents a number of challenges, not found in more typical architectures. These challenges include the lack of operating system support for the SPE cores, and difficulties presented by the cores having no direct access to main memory. This section details the design and implementation of a compiler, and associated runtime support, which overcome these challenges to provide efficient execution of Java code on the SPE cores.

### 5.3.1 Overview

The design of the SPE Java compiler in Hera-JVM follows that of the existing PowerPC compiler, provided by JikesRVM, which is used to support the PPE core. JikesRVM has two compiler types: a baseline compiler, where no inter-bytecode optimisations

are performed; and a much more complex optimising compiler, which recompiles *hot* methods to decrease their execution time. Building an optimising compiler for the SPE cores is beyond the scope of this dissertation and, in any case, would not significantly enhance the argument supporting its hypothesis. Therefore, Hera-JVM only supports the baseline compiler for both core types.

As with most of the rest of the runtime system, the PPE and SPE compilers are themselves written in Java. A common compiler infrastructure is shared by both architectures, with concrete implementations of machine specific classes extending a common set of abstract super-classes. There were three main machine specific classes which needed to be implemented to support the new SPE architecture: an assembler class (`Assembler`), a bytecode compiler class (`BaselineCompilerImpl`), and a class which generates low-level machine code, required to support the runtime system (`OutOfLineCodeGenerator`).

The `Assembler` class provides a set of methods, each of which represents a particular machine opcode (e.g. the instruction to add two registers, load a memory address, etc.). When one of these methods is called, with the opcode's operands passed as arguments, the associated machine code is generated and appended to an in-memory code array. An instruction's location in this code array can be associated with a label, so that it can be easily referred to in branch operations, improving readability and simplifying maintenance. This class can be used to generate executable code in much the same way as a stand-alone assembler, but is integrated into the runtime system in a fully programmable manner. The source code for a method which generates executable code (e.g. those in `BaselineCompilerImpl` or `OutOfLineCodeGenerator`) looks much like an assembly listing, with the opcode mnemonics replaced by calls to this `Assembler` class.

The `BaselineCompilerImpl` class uses this `Assembler` class to generate executable code from Java bytecodes. The `BaselineCompilerImpl` class consists of a set of *emitter* methods, each of which generate the machine code necessary to execute a particular bytecode operation. A Java method is compiled by calling the appropriate emitter for each operation in the method's bytecode stream. Thus, this class forms a single-pass compiler, which directly converts the stack-oriented Java bytecode to equivalent SPE machine code. It does **not** do any optimisation to reduce unnecessary stack operations by, for example, holding intermediate, inter-bytecode values in registers.

The majority of the bytecode operations can be directly converted into a short sequence of equivalent SPE machine instructions. However, some operations require additional runtime support. As described in Section 5.2, the complex bytecodes are implemented as Java code by a method in the runtime system. The emitter for each of these bytecodes simply generates a trap to the method which implements the bytecode. The bytecodes which are implemented in this manner are those dealing with: object creation (`new`, `newarray`, `anewarray` and `multianewarray`); type comparison (`instanceof`); exception throwing (`athrow`); and thread synchronisation (`monitorenter` and `monitorexit`).

The runtime system must sometimes perform an operation which cannot be implemented by standard Java code. For example, raw access to memory is not allowed by the Java language, but is required for the runtime system to perform memory management and garbage collection. To overcome these limitations, JikesRVM (and thus Hera-JVM) provides a set of *Magic* methods. These methods are treated like intrinsic functions, with their implementation provided by the compiler. The `BaselineCompilerImpl` class treats these Magic methods like special bytecodes, by replacing a call to these methods with inline assembly which directly performs the required operation.

Finally, the `OutOfLineCodeGenerator` class provides low-level support which is required by the SPE cores to operate. This class is, in effect, an assembly listing used to generate ∼4KB of low level runtime support code. This support code is permanently resident in each SPE's local memory, unlike the rest of the runtime system, which is cached on-demand. It provides the low-level operations which are fundamental to the execution of Java code on the SPE cores, such as: caching of objects into the SPE local memory (Section 5.3.3); caching of method code (Section 5.3.4); management of the SPE's local memory (Section 5.3.3.2); low-level thread synchronisation operations (Section 5.3.3.4); and interrupt handling and processor initialisation (Section 5.3.5). This support code can be thought of as a tiny OS kernel, which supports the *bare-metal* execution of Java code on the SPE cores.

The following sections detail the techniques which were employed by this compiler and low-level runtime support code in order to overcome the challenges presented by the unusual SPE architecture.

### 5.3.2 Local Variables and Stack Management

As a stack-oriented language, Java bytecodes implicitly operate on variables located on an *operand stack*. For example, the `iadd` bytecode pops two integer values off the operand stack, adds them together, and pushes the result back onto the operand stack. Since almost every bytecode pushes or pops values from the stack, it is important that these operations are efficient.

A thread's stack resides in main memory (so that it can be accessed by any core upon which it is scheduled). However, operating directly on this stack in main memory would be incredibly inefficient on the SPE cores, due to their DMA-based access to this memory. Therefore, the top portion of the currently executing thread's stack is held in the SPE's local memory to provide efficient stack access. Upon a thread switch, a 16KB block at the top of the thread's stack is copied into a reserved portion of the SPE's local memory. Stack updates are performed on this local copy, which is then written back to main memory when the thread is context switched from this core.

The Java stack consists of multiple frames, each of which contains the state of one method invocation. A frame pointer, held in a reserved register, points to the frame of the currently executing method. Each method has a fixed frame size, depending upon the set of bytecode operations performed by that method. When a method is invoked, it *buys* its stack frame by decrementing[1] the frame pointer by the method's required frame size. Push and pop operations then become loads and stores at a particular offset from the address now pointed to by the frame pointer. The offset used for each stack operation is calculated statically when the method is compiled. When the method returns, it will release its stack frame by incrementing the frame pointer by its frame size, such that it now points at the caller method's frame.

Stack overflow checks are required when a method buys a stack frame, to ensure that the stack does not grow beyond the 16KB limit held in SPE local memory. If a stack larger than 16KB is required, these overflow routines could be modified to *page* stack blocks on demand. However, this was not required for any of the Java benchmarks which were investigated in this dissertation, and so stack block paging is not currently supported by Hera-JVM.

[1] For historical reasons, stacks usually grow downwards in memory.

Whilst accessing local memory on the SPE cores is much more efficient than accessing main memory, it is still complicated by the SPE's unusual instruction set. The SPE cores have a Single Instruction, Multiple Data (SIMD) (Flynn, 1972) based instruction set. Each register is 128 bits wide, with instructions treating these 128 bits as a vector of sixteen 8-bit, eight 16-bit, four 32-bit or two 64-bit values, depending upon the operation. Hera-JVM does not currently exploit these SIMD operations for data parallelism, since Java has no in-built vectorization support (exploiting the SPE's SIMD capabilities in Hera-JVM is left as future work). Instead, the first vector slot is used exclusively by Hera-JVM for all arithmetic operations. The complication for stack management is caused by the fact that the SPE instruction set only allows 128-bit aligned load and store operations to local memory. Therefore, the choice in stack layout has considerable trade-offs:

- The stack can be laid out in the standard fashion, with variables placed in word-sized stack slots. However, this means that stack variables will not be aligned to 128-bit and must be shuffled into the first vector slot of a register before being used as an instruction operand. Similarly, the result must be shuffled before it can be stored in its required stack slot. These overheads make stack access inefficient, requiring two machine instructions for a pop operation, and four for a push operation.

- A full 128-bits can be used for each stack slot. Each stack variable can thus be aligned to 128-bits and located such that it is in the first vector slot when loaded into a register. This makes stack operations simple and efficient. However, given that most stack values will be 32-bit words, this approach wastes a significant proportion of the valuable local memory reserved for stack use.

Hera-JVM uses the first approach, since aligning every stack slot to 128-bits wastes considerable local memory. However, an optimisation is employed to reduce the overhead of variable shuffling. One of the SPE's registers is reserved to hold the 128-bits currently at the top of the stack. Variables are shuffled in and out of this register as required, but the values are only written to the local memory stack when the stack pointer passes a 128-bit boundary. This optimisation reduces stack operation overheads to one machine instruction for a pop operation, and two instructions for a push operation.

As well as the operand stack, a Java method can store intermediate values in an array of variables, known as *local variables*. Hera-JVM exploits the large register file provided by the SPEs (each SPE core has 128 registers) to hold each local variable in its own register. The value of a local variable is local to each method invocation, and so the registers holding them must be non-volatile across method invocations. The number of local variables required by a method is known at compile time. Thus, when a method is compiled, code is included in its prologue to save the previous values held in any local variable registers it may overwrite. When the method returns, the local variables can be restored with these values.

### 5.3.3 Software Caching of Heap Objects

In addition to variables on its stack, a Java thread has access to a data heap, which holds object instances and arrays. This heap is shared amongst all threads in the system, and is therefore located in main memory. In order to access data in this heap, an SPE core must first DMA the data into its local memory. To reduce heap data access latencies and limit the overhead of DMA transfers, Hera-JVM employs a software cache for SPE heap access. When a heap access results in a cache miss, the SPE core must perform a DMA transfer to copy the required data into the SPE's local memory. On a cache hit, the thread can access this local memory copy directly.

Section 5.3.3.1 outlines the strategy employed by Hera-JVM to enable it to cache heap data-elements in the SPE core's local memory in an efficient manner. This caching scheme dynamically allocates space for cached data-elements in the SPE's local memory using a process described in Section 5.3.3.2. Section 5.3.3.3 discusses the cache's write policy, and finally, Section 5.3.3.4 discusses synchronisation and coherency issues related to ensuring that Hera-JVM conforms to the Java memory model (Manson *et al.*, 2005) under this caching scheme.

#### 5.3.3.1 Caching Strategy

Setting up a DMA operation to transfer data to and from local memory is an expensive operation (about 30-50 cycles, not including the data transfer itself). Therefore, an early design decision of the software cache was to transfer large blocks of memory wherever possible. However, to limit cache pollution, only data which is likely to be used in the future should be cached. The high-level type information preserved in

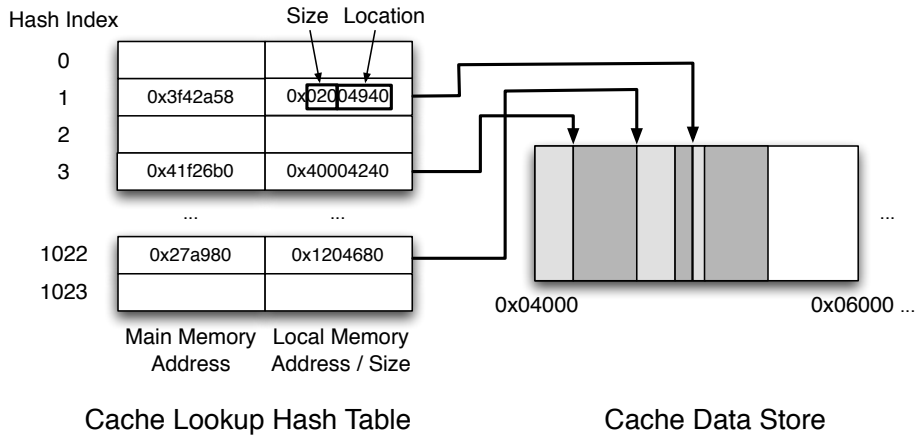



Figure 5.4: Outline of the SPE data cache.

Java bytecode provides the opportunity to meet these two conflicting demands. The software cache can specialise its caching mechanism depending upon the data type being accessed.

Java code can read data from, or write data to, an object instance in the heap using the `getField` and `setField` bytecodes, respectively. These operations access a single field of an object. Rather than caching just the field being accessed, or a fixed size block around that field, Hera-JVM exploits the flexibility of a software caching scheme that has access to high-level type information, to cache the full object instance[1]. This approach exploits *object-based* locality of reference — i.e., the thread is likely to access other fields in the same object instance.

Arrays are accessed using a different set of bytecodes (`iaload`, `iastore`, etc). Array accesses can therefore be cached in a different manner to object accesses. Array instances are generally much larger than object instances, and may be too large to fit in their entirety into the local memory cache. Therefore, rather than attempting to cache entire array instances, Hera-JVM caches arrays in 1KB blocks. Spacial locality of reference is exploited by this scheme, with neighbouring elements cached alongside the element being accessed, on the assumption that they are likely to be accessed shortly.

[1]Arrays or object instances that are held in fields of a particular type are not actually stored in instance objects of this type. Instead, these fields are pointers (or references in Java terminology) to this data, and thus require only one word of memory each. Therefore, unless the object's class type is badly designed with a huge number of fields, object instances are always less than a couple of hundred bytes, and can easily fit in local memory.

A small (1024 entry) local memory resident hash-table is used by each SPE to support this software cache (see Figure 5.4). Each entry is either blank, or holds the main memory address of an object instance or array block as a key, and the local memory address of its cached copy as a value. When emitting code for a heap access bytecode, the SPE compiler generates code which performs a lookup in this hash-table, using the main memory address requested as a key, to check if the data has been cached. Object access bytecodes look up the object's address, whereas array access bytecodes use the starting address of the required array block, which is calculated based upon the array's starting address and the index requested (both of which are passed on the operand stack). This key is hashed using a simple XOR folding hash[1] to provide an index into this hash table. If the entry at this index is the same as the main memory address requested, this access has *hit* the cache and the bytecode operation is performed directly on the cached copy pointed to by this entry. Otherwise, a cache miss has occurred and the data must first be pulled into local memory.

On a cache miss, space is reserved for this object instance in local memory, a DMA operation is set up to copy its data into local memory, the hash table is updated with this cached entry, and the thread blocks until the DMA copy completes. No collision resolution is performed by this software cache hash-table. A cache miss simply overwrites the hash-table entry to reflect this newly cached element, thereby evicting the previous element from the cache.

A cache miss is significantly more costly than a cache hit, both in terms of execution time and the amount of machine code required to perform these operations. Therefore, rather than the compiler generating *inline* code to deal with cache misses for every heap operation, cache misses *trap* to an out-of-line procedure generated by the `OutOfLineCodeGenerator` class. Thus, the fast path cache hit code is performed inline to reduce performance overheads, whilst the more complex, but less used cache miss code is kept out-of-line to reduce code bloat.

#### 5.3.3.2 Cache Element Space Allocation

This caching scheme is unusual in that the elements being cached are not of a fixed size. Therefore, a memory allocation scheme must be employed to manage the local

[1] Bits 4 to 13 of the object's main memory address are XORed with bits 14 to 23, then masked to provide an index into the hash table. An XOR hash was chosen due to its simplicity, to making cache look ups as lightweight as possible.

memory reserved for heap data caching. Hera-JVM uses a simple bump-pointer allocation scheme, where a newly cached object is appended to the end of the previously cached object, by incrementing a bump pointer with the size of the new object. The cache is simply flushed completely if it becomes full. A more effective, lightweight, free-list based memory management algorithm (McIlroy *et al.*, 2008) was considered as an alternative to this simple bump-pointer based approach. However, the requirement to flush this cache for synchronisation operations (see Section 5.3.3.4) would limit the benefits such an approach could provide over a much simpler approach, while incurring additional complexity in the cache's design. Improvement of the cache's memory management is therefore left as future work.

The size of memory which must be allocated depends upon what is being accessed. For object instance accesses, type information embedded in the `getField` and `setField` bytecodes enables the runtime system to infer, at compile time, the type, and thus the size of the object instance being accessed. Array accesses cache either a full block, of 1KB in size, or, if accessing the last block in the array, a block sized to fit the remainder of the array. The length of an array is held in its header. The cache miss handling code checks this length when caching a block to discover if this is the last block of the array and, if so, what size this last block will be.

One problem with this approach relates to Java's subtyping inheritance abilities. An object access bytecode (e.g. `getField` or `setField`) has a particular type associated with the operation. However, the actual instance object accessed at runtime may be a subtype of the type associated with the bytecode. This subtype may have more fields than the supertype referred to by the bytecode, resulting in its instance objects being larger. Since the caching system uses the type associated with the bytecode to infer the size of the object being cached, it will not cache the full subtype object instance, only those fields associated with its supertype. This is not a problem for this bytecode, since it is accessing one of the fields associated with the supertype. However, subsequent accesses to this object instance will *hit* this cached copy directly. If they are trying to access one of the subtype fields, invalid data will result from reading from this cached copy, since it does not contain any of the subclass data. To avoid this, Hera-JVM stores the size of the data it has cached alongside the local memory address

of the cached object in the hash table[1]. When a cache hit occurs, the size of the cached data is compared to the expected size of the object type being accessed. If the cached data is not large enough, the object is re-cached.

#### 5.3.3.3 Write Policy

Operations which write to the heap (`setField`, `iaStore`, etc.) must update both the cached data in local memory and the original copy in main memory. Two write policies were considered for this software cache: a write-through policy, where every write operation is performed on both the local copy and its main memory backing store; and a write-back policy, where writes are only performed directly on the local memory copy, but are propagated to main memory when the local copy is evicted from the cache. By default, Hera-JVM uses a write-through policy for all write operations due to its simplicity.

Write operations update the data in the local memory cache directly. They then immediately initiate a DMA transfer to copy the object field or array element which was modified to its original main memory location. Unlike the cache read operations, this DMA transfer is non-blocking; the thread can continue executing while the DMA engine performs this write to main memory concurrently. Threads do, however, block on these write operations before reading data from main memory to service a cache miss and thread synchronisation operations, to maintain memory consistency.

Whilst non-blocking DMA write operations greatly improve performance, compared to blocking transfers, the overhead of setting up a DMA transfer on every write operation is still non-negligible. Therefore, the use of a write-back policy was also investigated. Since SPE DMA operations are most efficient when transferring large blocks of data, a write-back policy was investigated as a means of coalescing multiple write operations on elements in the same array into a single main memory write-back DMA transfer, instead of each write being performed individually.

One approach for implementing a write-back policy for array updates would be to simply mark a cached array block as being dirty if one or more of its elements are written to, and then transfer the block back to main memory in its entirety when it is evicted from the cache. However, this will overwrite all the elements of this array

[1]Since the local memory has a small address space (18 bit address width), the object's size and cached local memory address can fit in a single 32 bit word entry of the hash table.

block, including those that were never updated by the thread running on this SPE. The Java Memory Model (Manson *et al.*, 2005) does not permit a thread to overwrite array elements which it did not update, since this will cause updates, performed on these elements by other threads, to be lost. This is clearly unacceptable, therefore, the write-back approach must log the elements that have been written to, and only write these back to main memory.

To limit the state required for array element write logging, only sequential writes to an array block are coalesced in this manner. A small (16 element), local memory resident hash-table is employed for this purpose. Each entry holds a start and end address, specifying a sequential range of elements of an array block that have been written to by the current thread, but have not yet been written-back to the main memory backing store. An array write operation looks up the hash-table entry, corresponding to the array block being operated upon[1]. It then checks if the address being written to is adjacent to the start or end addresses in this entry. If so, the write can be subsumed into the logged updates by modifying this entry's start or end address to include this write. If not, either because the hash-table entry refers to another array block, or the block is not being written to in a sequential manner, then the logged operations in this hash-table entry are written-back to main memory, and the entry is updated to log the current operation. As with the caching code, this slow path is trapped to and performed in out-of-line code, while the fast path, of simply updating the log to subsume this write operation, is processed inline.

Unfortunately, implementation of this write-back approach introduced errors into the Hera-JVM runtime which prevented execution of the more complex real world benchmarks used in Section 5.5.3. Therefore, by default, Hera-JVM writes-through all array write operations; this write-back approach is only used in the experiments in Section 5.5.2.1.

#### 5.3.3.4 Synchronisation and Coherency

In a multi-threaded application, the same object could be accessed by multiple threads simultaneously: thus as well as residing in the main memory heap, multiple copies of this object could reside in different SPE local memory caches. Keeping these copies

[1]The index for this table is generated by simply masking the index generated as part of the cache lookup operation on this array block (Section 5.3.3.1), such that it refers to an element in this smaller hash-table.

consistent is usually the job of a hardware cache coherency system. However, the Cell processor does not provide hardware cache coherency for SPE local memory, and providing a software coherency protocol that broadcasts every object update to all copies of the object's data would cripple the SPEs' performance. However, without some form of consistency model, a thread running on an SPE core will never see any updates made to an object by other threads, after it has cached this object in its local memory (unless the object has to be re-cached, due to it having been evicted).

To correctly execute Java programs, Hera-JVM must provide a consistency model, for code running on the SPE cores, that is allowed by the Java Memory Model (Manson *et al.*, 2005). Java's memory model is based upon the happens-before memory model. Synchronisation operations, such as lock operations and volatile field accesses, impose a *happens-before* order on program execution. A data read is not allowed to observe a write which happens after it in this *happens-before* partial order (i.e., it should not see any writes which happen after a subsequent synchronisation point). Similarly, the read can observe a write $w$, as long as there was no intervening write $w'$, where $w$ happened before $w'$ in the *happens-before* partial order (i.e., the thread does not see a value which was overwritten before the previous synchronisation point).

In virtual machine implementation terms, the effect of this model is to allow heap data to be cached by a thread (i.e., be inconsistent with the globally accessible original copy in main memory), as long as these cached copies are re-synchronised with main memory at thread synchronisation points. After performing a locking operation, a thread must see all heap updates which *happened-before* this lock. Before releasing this lock, the thread must ensure that any updates it has made to heap variables are fully propagated to main memory, thus ensuring that they will be visible to any thread which later synchronises on this same lock object.

Hera-JVM ensures this by completely purging the SPE local memory cache whenever the thread it is executing locks an object or reads a volatile field. Before unlocking an object or writing to a volatile field, the thread is blocked until all of its DMA write transfers have been completed. If the write-back caching policy is being used for array access, then the runtime system iterates through the write-back log and performs any outstanding write-back operations. Once these transfers are complete, the unlock operation is performed.

The Java memory model also requires synchronisation order consistency, where the order of synchronisation operations and volatile variable accesses is sequentially consistent across threads. These operations must, therefore, be performed atomically.

Volatile field accesses are restricted to reading or writing from a single field, which can have a maximum size of 8-bytes. DMA transfers, performed by an SPE core's memory flow controller (MFC), that are less than 16-bytes operate atomically on the Cell processor. Therefore, volatile field accesses can be treated like normal field accesses by Hera-JVM, with the additional constraints that: (i) the SPE local memory cache is flushed before a volatile read is performed (which was also required for *happens-before* consistency above); and (ii) the thread blocks on volatile writes to ensure they have been written to memory before continuing (unlike non-volatile field write operations, which are non-blocking).

To perform lock and unlock operations atomically, an atomic compare-and-swap type of operation must be used. The SPE MFCs provide two blocking DMA operations, called `GETLLAR` and `PUTLLC`, which can be used to build an atomic compare-and-swap operation. The `GETLLAR` operation performs a blocking read from a memory address, whilst simultaneously setting a reservation on this address. The `PUTLLC` operation conditionally writes to a memory address, if the processor still holds a reservation on this address, and returns a success or failure notification. If another core writes to this memory address between the `GETLLAR` and `PUTLLC` operations, the reservation will be lost and the `PUTLLC` operation will fail. Thus, an SPE core can perform an atomic (from the point of view of other cores) compare-and-swap operation to lock or unlock objects, based upon these operations.

By conforming to the Java Memory Model, any correctly synchronised, data-race-free, multi-threaded application will exhibit sequentially consistent behaviour and run correctly under Hera-JVM. To gain reasonable performance under the unusual SPE memory hierarchy, Hera-JVM exploits the fact that the Java memory model is relatively weak, by not guaranteeing sequential consistency in the presence of data-races. However, there may exist programs that have benign data-races, which, none the less, function correctly when executed on a typical cache coherent CPU. The limited coherency provided by Hera-JVM on SPE cores could cause such programs to fail. However, this has not been a problem for the range of real world benchmarks used to

investigate Hera-JVM, suggesting that most bug-free Java applications are also data-race-free.

### 5.3.4 Invocation and Caching of Methods

Code must also reside on the SPE's local memory before it can be executed. Since a Java thread is likely to execute more code than can fit in an SPE's local memory, a software-based code caching scheme is required.

Section 5.3.4.1 describes the strategy taken by Hera-JVM to cache method code in an SPE's local memory so that it can be executed by a thread executing on the SPE core. Section 5.3.4.2 describes the approach taken to ensure that execution resumes at the correct point when an invoked method returns. Since the cache has a limited size, it must be emptied when no space remains to cache a newly invoked method. The code cache purging approach, employed by Hera-JVM, is discussed in Section 5.3.4.3. Finally, Section 5.3.4.4 describes how this approach enables Hera-JVM to support the SPE cores using a runtime system that has code memory requirements larger than the size of the SPE cores' local memory.

#### 5.3.4.1 Caching Strategy

In keeping with Hera-JVM's approach of DMAing large blocks of data wherever possible, Java methods are cached in their entirety. As with the object cache, a bump pointer allocation scheme is used to manage this cache, with the cache being completely purged whenever it becomes full. Unlike the object cache, this code cache does not use a hash-table to perform look-ups. A hash-table was considered unsuitable as a means of looking up whether a method has already been cached, due to the need to support virtual method invocation for Java instance methods.

When an instance (as opposed to a static) method is called, the actual method which is invoked depends upon the type of the instance object upon which this method was called. This object instance could have a type which is a sub-class of that described by the `invokevirtual` bytecode. If this method has been overridden by the object instance's sub-class, then the runtime system must invoke the sub-class version of the method, not the super-class method described by the `invokevirtual` bytecode. Therefore, the actual code that should be invoked by the `invokevirtual` bytecode

is unknown at compile time; it must be inferred at runtime-based upon the object instance's type.

Typically, in order to support virtual method invocation, the header of every instance object includes a pointer to a *type information block* (TIB), which describes the class of the object instance. This TIB contains an entry for every method declared by a class, each of which points to the code that implements the method. The TIB is laid out such that inherited methods are located at the same index in the sub-class's TIB as in the super-class's TIB[1]. By looking up the index of the virtual method being invoked in the TIB of the object upon which it is being invoked, the runtime system can find the actual instance method which it should run for this virtual method invocation.

Since each TIB entry points to the machine code implementing a method, Hera-JVM requires two TIBs for every class — one which points to the PPE machine code and one which points to the SPE machine code. To limit memory overheads, Hera-JVM uses a two stage class-loading system. A class is initially *resolved* for the PPE core, with only the PPE TIB being created. If the class is referenced by code running on the SPE core, it will then be resolved for the SPE core, which will create the SPE TIB.

The SPE TIBs reside in main memory, being cached into an SPE's local memory only when required (exploiting *class locality* — if a method in a particular class is called, it is likely that other methods in that class will also be called). When invoking a virtual method, an SPE core must: (i) find the location of the SPE TIB for the current object instance; (ii) cache this TIB in local memory if necessary; (iii) look up the location of the requested method in this TIB; and (iv) if necessary, cache this method in local memory before invoking it.

Hera-JVM exploits the fact that only a limited number of classes will be resolved for the SPE core to simplify TIB and method caching. A small (4KB) class *table of contents* (TOC) resides in SPE local memory, with an entry for each class that has been resolved for the SPE (Figure 5.5). Each entry initially points to the location of the class's SPE TIB in main memory. Object instances have an index in their header which points to their class's entry in this class TOC, rather than a direct pointer to the SPE TIB's main memory address. When a method is invoked, the appropriate class's TOC entry is read to locate the class's TIB, which is cached if necessary. When

[1]This is possible because Java does not support multiple inheritance, and therefore each class can only inherit methods from its single super-class.

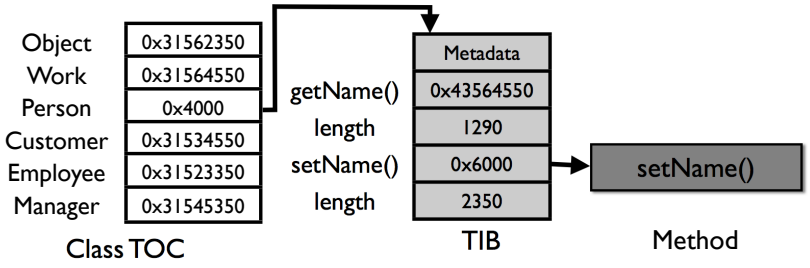



Figure 5.5: The code cache data structures.

a TIB is cached, its TOC entry is updated to point to this local memory copy, and so subsequent look-ups immediately know the location of the cached copy. The required method is then looked up in the TIB and, if necessary, is cached in local memory, with the TIB entry being updated to point to the cached method's address.

This differs from the data heap cache, in that references in the TOC and TIB data structures are directly updated with the local memory address when the data they refer to is cached, whereas, references in the data cache always point to the main memory original. Therefore, a cache hit in the code cache only requires two local memory de-references, instead of multiple hash-table lookups to translate from main memory addresses to the local memory cache addresses. These local memory references are never propagated to the original data structures in main memory; only the cached copies in the local memory of an SPE core are updated to point to data which has been cached by that core.

An added benefit of this approach is that, while a direct pointer to a class's SPE TIB would require a full word to specify, a class's TOC index only requires 10 bits in Hera-JVM. Therefore, it can be accommodated in spare bits of the object instance's header, rather than having to reserve an additional word for the SPE TIB pointer in every object instance's header. Note, the object instance headers do still contain a pointer to their PPE TIB, so that the PPE code can perform virtual method invocation in the usual manner. However, since the TOC index is hidden in spare bits of the header, object instances are the same size in Hera-JVM as they are in JikesRVM.

Static method invocations always invoke the same class's method (they are statically associated with the class, not a particular object instance). These method invocations are cached in the same manner as instance methods, the only difference being that

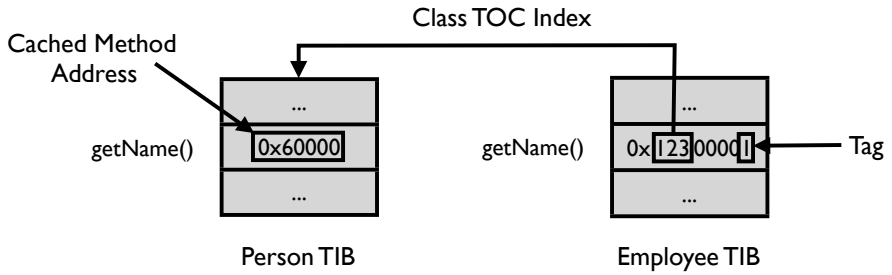



Figure 5.6: TIB layout for super-class methods.

the class TOC index is supplied statically by the compiler, rather than being read dynamically from an object instance's header at runtime.

One issue with this caching scheme, which is not immediately obvious, relates to methods that are inherited from a super-class, but are not overridden by the sub-class. A non-overriden method is shared by multiple classes, with its entry in each of these class's TIB pointing to the same method code. Since a method is recorded as being cached by updating its TIB entry, the fact that the same method can be pointed to by multiple class TIBs can lead to a method being cached multiple times.

Take, as an example, a program with three classes: a common super-class, called Person; and two sub-classes of Person, called Customer and Employee. Person contains a method, `getName`, which is inherited (but not overriden) by both the Customer and Employee classes. If the `getName` method is invoked on a Customer object instance, its entry in the Customer class TIB will be updated. However, its entry in the Person and Employee TIBs will not be updated (these TIBs may not even be cached themselves). Therefore, if the `getName` method is subsequently called on a Person or Employee instance object, the method will be needlessly re-cached.

To avoid this, only the class which actually implements the method includes a pointer to the method in its TIB. For classes which inherit this method, the TIB entry is, instead, an index pointing to the implementing class's TOC entry, with a tag to indicate this (see Figure 5.6). When caching an inherited method (e.g. invoking `getName` on a Customer object instance, as above), the cache system uses this TOC index to look up the implementing class's TIB (i.e., the Person TIB in the above example). The implementing class's TIB is checked to see if the method has been previously cached. If so, the method's entry in the invoking class's TIB (i.e. Customer)

is updated to point to the same cached copy (overwriting the TOC index it previously held). If not, the method is cached, and both TIBs are updated to reflect this. Since this approach updates the implementing class's TIB, as well as the invoking class's TIB, whenever an inherited method is cached, subsequent invocations of the same method on an instance object of a different class (e.g. Employee) will not re-cache this method.

#### 5.3.4.2 Returning from a Method

When a method returns, execution should return to the point in the caller method immediately after the callee method was called. Typically, this is supported by placing a return address on the stack; a return statement will branch to this return address to resume execution of the caller method. However, code executed by an SPE core is dynamically cached in local memory. Therefore, a simple return address is not sufficient, since the caller method may no longer be at the same location in local memory when execution returns to it (it could have been evicted from the cache or re-cached at a different location).

Instead, Hera-JVM places a *return offset* on the stack when code running on an SPE core invokes another method. The value of this return offset is the distance of the invoking instruction from the start of the caller method. Alongside this return offset, the caller's stack-frame also contains a method ID, which specifies both the TOC and TIB indexes necessary to look up this method in the code cache. When the callee method returns, it ensures the caller method is cached by performing the same look up process as if it were invoking the method, using the indices specified in the method ID in its caller's stack-frame. Adding the return offset on the caller's stack-frame to the start address of this cached method provides the callee method with an absolute return address, to which it can jump in order to resume execution of the caller method.

#### 5.3.4.3 Purging the Cache

If the caching of a method's code would overflow the the remaining space in the code cache, the cache is purged to make room for this new method. To do this, the class TOC is reloaded from main memory. This will replace any entries that had been updated to point to cached TIBs with the original main memory address of the TIB, thus effectively evicting all TIBs from the cache. Since cached methods are pointed to by these cached TIBs, evicting them also evicts every method from the cache. Finally,

the bump allocation pointer, used to allocate space in this cache, is reset to the start of the local memory area that is reserved for code caching.

Once the cache has been purged, execution should resume where it left off. However, the code of the executing method has now been evicted from the cache. Therefore, the cache purging routine must re-cache this method before resuming its execution. It does this in a similar manner to that described for returning from a method (Section 5.3.4.2), using the method ID on the current stack-frame to work out what code should be cached.

As well as being purged when it becomes full, the code cache is also purged whenever new methods are compiled for the SPE. This insures that the relevant updates to the TOC and TIB data-structures are propagated to all SPE cores, ensuring that these new methods will be cached correctly when invoked.

#### 5.3.4.4 Caching of Runtime System Code

Since the majority of the Hera-JVM runtime system is written in Java, it can be cached into local memory as required by this mechanism in the same manner as application code. Thus, there is no need to specialise the SPE runtime to enable it to fit in its entirety in the SPE cores' 256KB of local memory. This enables the same runtime code to be used on both the PPE core as the SPE cores, unlike CellVM (Noll *et al.*, 2008), which used two different runtime systems to support the two different core type.

### 5.3.5 Scheduling and Thread Switching

Java is a multi-threaded programming language. The runtime system must therefore schedule the execution of Java threads onto the processing cores it has available. The version of JikesRVM upon which Hera-JVM is based (version 3.0) uses a *green thread model* to schedule Java Threads. This model maps multiple Java threads to a single OS thread, with the runtime system performing user level scheduling of the Java threads, rather than the underlying operating system[1].

JikesRVM uses an m-to-n threading model, with the runtime system starting an OS thread for each processing core and pinning its execution to this core. Thus only

[1] JikesRVM has since moved to a native thread model (as of version 3.1), where each Java thread maps to a native OS thread. In the native model, the underlying operating system schedules the threads (although the runtime system retains some control).

a single OS thread (known as a *virtual processor* in JikesRVM) executes Java code on a particular processing core, on behalf of different Java threads. When a Java thread performs a blocking operation or a timer tick event occurs, the virtual processor's execution *traps* to runtime scheduling code. The runtime scheduler selects another Java thread from the virtual processor's run-queue and *morphs* its identity from the Java thread which it was previously executing to this new Java thread. Thus, the operating system is not involved in the scheduling of Java threads at all; it is only involved in sharing the processing core's execution between the JikesRVM virtual processor and any other processes running on this processing core.

Since the SPE cores do not run an operating system, code executes *bare-metal*, with no OS level support for multi-threading[1]. Thus, this green thread model is a natural fit for the creation of Java threading on SPE cores in Hera-JVM. A single virtual processor thread of execution is run *bare-metal* on the SPE core. This SPE virtual processor employs the same runtime scheduling code as the PPE core to schedule Java threads. No OS level scheduling support need be created to support threading on the SPEs.

#### 5.3.5.1 SPE Virtual Processor Initialisation

Hera-JVM initialises each SPE core by having the SPE execute a specially written boot-loader program, using the libspe2 library provided by IBM. This boot-loader is written in C so that it can be supported by libspe2. It initialises some reserved registers used by the SPE runtime system, then copies the low level, out-of-line, runtime code (used to provide code and object caching, as well as other services such as interrupt handling) into its local memory. The boot-loader then traps to this out-of-line code to cache and invoke the Java entry function of the SPE runtime system (this overwrites the C boot-loader code in the process).

The Java entry function performs some additional initialisation to set-up the SPE core's virtual processor data-structures. It then invokes the scheduling code to find a Java thread which it can execute. Initially, only a pre-built idle thread will be runnable on this SPE virtual processor. The idle thread does nothing other than yield, to enable

[1]The Linux kernel which runs on the PPE core has some support for the multiplexing of multiple virtual SPE processors onto a single physical SPE core. However, the swapping process is very heavyweight and must be performed by the PPE core, making it unsuitable for the scheduling of Java threads on SPEs.

the scheduling of a useful thread. After a given number of yields, it puts the core to sleep, by performing a blocking read on an inter-core signalling channel. To wake this core, another thread will send a signal through this channel to wake the core, after placing a thread on its run-queue or (in the case of the PPE core) migrating a thread to the SPE core.

#### 5.3.5.2 Scheduling Mechanism

Each virtual processor (whether PPE or SPE) has its own run-queue of Java threads. It schedules these threads for execution in a round-robin manner, with each thread running for a full scheduling quantum or until it blocks (e.g., on an I/O operation).

When a virtual processor is making a scheduling decision, it checks whether it has more threads in its run queue than the other virtual processors. If so, it will perform load balancing by transferring some threads to another virtual processor's ready queue[1]. This load balancing is only performed by virtual processors running on the same core type (i.e., SPE to SPE or PPE to PPE, but not SPE to PPE or PPE to SPE). The thread migration mechanism, described in Section 5.4, must be used to transfer a thread to a different core type.

Since the virtual processor is running on a single OS thread, a Java thread cannot perform blocking system calls, or it would block all threads scheduled on that virtual processor. Instead, the runtime system makes a corresponding non-blocking system call, then blocks the Java thread by placing it on a per-core type blocked queue. A virtual processor will periodically check if any of the threads in the blocked queue of its core type have become runnable. It does this by performing a non-blocking *select* system call on the file descriptor upon which the thread is waiting for data. If the select call indicates that data is available on this file descriptor, it will move the thread back onto its run queue.

#### 5.3.5.3 Context Switching Mechanism

The process of context switching a virtual processor's execution to a different Java thread is highly architecture dependent. The scheduling code calls a *magic* method to

[1]In fact, it uses a per-virtual processor transfer queue to transfer threads, rather than directly accessing another virtual processor's ready queue. This is because transfers between virtual processors must be thread safe, whereas, if the ready queue is only ever accessed by its own virtual processor, its access need not be thread safe, thus speeding up the normal scheduling path.

perform a context switch. This magic method is compiled directly into inline context switching machine code, specific to the core type for which it is being compiled.

To perform a context switch on an SPE core, the currently executing thread's state must be saved and the new thread's state restored onto the core. This involves the context switch code saving all non-volatile registers to an array associated with the executing thread. Reserved registers, such as the frame pointer and the top of stack register (Section 5.3.2) are also saved in this array. The thread's current method ID and offset is saved onto the stack as if a method was being invoked. The stack block currently cached in the SPE's local memory is then written back to the thread's stack in main memory.

To restore the new thread's context onto this core, the process is reversed. The reserved and non-volitile registers are set to those values which were saved in this new thread's register array when it was last swapped out. A block at the top of this new thread's stack is loaded onto the SPE's local memory stack area. The context switch code then performs a process similar to a method return, using the method ID and offset saved on this new thread's stack when it was swapped out. This ensures that the method which was being executed by the thread when it was last executing is cached in local memory, and execution of the thread resumes at the correct point within this method.

#### 5.3.5.4 Timer Interrupts

To implement pre-emptive scheduling, the scheduler must be able to interrupt the execution of a Java thread. SPE cores have a simple hardware interrupt mechanism which can be employed to provide timer interrupts and enable pre-emptive scheduling.

An SPE core can be set up to asynchronously transition to interrupt handling code whenever a particular set of hardware events occurs. One of the hardware events which can cause an SPE interrupt is an incoming signal on the SPE's inter-core signalling channel. Therefore, to provide SPE timer interrupts, a thread, running on the PPE core, signals each SPE core every 10ms.

The SPE interrupt handler saves the core's context and processes the hardware signal which caused the interrupt. As well as handling timer interrupts, the interrupt handler maintains hardware controlled data-structures, such as a hardware decrementer used for low-level timing information. If a scheduling operation should be performed

during this interrupt, the interrupt handler then invokes the scheduler's entry-point method.

A number of runtime operations must be completed in their entirety, without being pre-empted by another thread. Many low-level operations, such as updating a thread's stack frame pointer or transferring data from main memory, cannot be completed atomically under the SPE's unusual instruction set. Disabling and then re-enabling interrupts around all these low-level operations would be a considerable overhead, as well as being difficult to maintain. Instead, Hera-JVM explicitly checks for an interrupt event at specific points in a method's execution. The SPE compiler inserts a *branch on external condition* instruction into method prologues and loop branches. If an interrupt event is pending, this instruction triggers the interrupt handling code, otherwise it does nothing. Checking for interrupt events on loop branches, as well as method prologues, ensures that only a limited amount of time will pass between a timer interrupt being fired, and the interrupt handler running. This is actually a similar approach to that taken by the PPE compiler, but for different reasons.

Some higher level operations must also be non-preemptible. For example, runtime methods that deal with thread scheduling or heap allocation should not be pre-empted. Such methods have been annotated with an `@Uninterruptable` annotation by JikesRVM to enable them to be treated specially. To ensure that these methods are not pre-empted, the SPE compiler simply does not include explicit interrupt check instructions when compiling methods which are tagged with `@Uninterruptable` annotations.

### 5.3.6 System Calls and Native Methods

Occasionally, a method in the runtime system, the Java Library or a Java application requires access to native code (e.g. to write to a file or start an external process). JikesRVM / Hera-JVM provides this support with the JNI (Java Native Interface) for Java Library and Java applications, whilst methods in the runtime system can use a fast system call mechanism.

However, if a thread is running on an SPE core, there is no underlying OS to support native methods. SPE cores must rely on the PPE core to perform native code. In the case of a JNI method, the thread is migrated to the PPE core for the duration of the native method, using the process described in Section 5.4. For fast system call methods, the SPE core uses an inter-core mailbox channel to signal a dedicated thread

on the PPE core with an appropriate message. This dedicated thread performs the required system call on the SPE thread's behalf, then signals the SPE with the result.

There is one set of native methods which is treated specially by Hera-JVM. The Classpath Java library, used by HeraJVM, implements the Math class natively. This is done purely for performance reasons; these methods do not require OS support. Thus, they do not need to be offloaded to the PPE, when invoked by a thread on an SPE core. Indeed, since these methods perform complex floating point operations, they are likely to perform much better on the SPE core, than on the PPE core. Therefore, the SPE compiler treats these methods like intrinsic functions — directly generating the machine code required to perform the required operation — rather than offloading them.

## 5.4 Migration between Core Types

With the SPE Java support, described in the previous section, Hera-JVM can execute the same Java code on either an SPE or a PPE core and produce the same results. Hera-JVM supports migration of a Java thread between the PPE and SPE cores to enable it to exploit the core types available on the Cell processor. This migration process is transparent from the point of view of the application; no changes in application code are required to enable the application to be migrated between core types.

A thread can be migrated when it invokes a method that has either been tagged by an annotation or has been selected by the scheduler. The experiments in this chapter focuses on migration of threads using explicit annotations (`@RunOnSubArch` and `@RunOnMainArch`); Chapters 6 and 7 investigate dynamic migration triggered by the scheduler. However, the mechanism used for thread migration is the same in both cases.

### 5.4.1 Migration Mechanism

A method which triggers a migration is invoked as usual, but then code in the method's prologue will cause a trap to migration support code. In the case of methods tagged with the explicit `@RunOnSubArch` and `@RunOnMainArch` annotations, the method will unconditionally trap if it is being executed on the wrong core type (i.e., a method tagged with `@RunOnSubArch` will trigger a migration if it is run on the PPE core, but

not if it is run on the SPE core). Other conditional triggering mechanisms are discussed in Section 6.3.2.

The migration support code (executing on the original core type) will package the parameters of the migrating method and, if necessary, JIT compile this method for the core type to which the thread is being migrated. It then performs a scheduling operation to find another thread which this core can execute, placing the migrating thread on a per-core type migration queue, rather than returning it to the core's own run queue.

During scheduling operations, each core will periodically check the migration queue associated with its core type. Any threads it finds will be removed from the migration queue to be added to its own run queue. However, the current stack-frame of a migrated thread is laid out for the other core type. Therefore, before this thread is added to the run queue, a stack-frame for this core type is added to the end of the thread's stack. This synthetic stack-frame causes the thread to start executing at a migration entry-point method when it is scheduled. The migration entry-point method will unpack the parameters which were passed to the migrating method, then invoke the method using Java's reflection mechanism.

The thread continues to execute on this new core for the duration of this migrated method and the whole tree of methods which it calls[1]. Of course, a subsequent method invoked by this thread could cause it to migrate back to the previous core type using the same mechanism.

Once a thread returns from a migrated method it must return to its original core type. This is required by Hera-JVM because the frames below this point on the stack are formatted for the other core type. To return to the original core type, the migration entry-point method performs a return migration once the migrating method it invoked has returned.

### 5.4.2 Scanning a Migrated Thread's Stack

A number of runtime processes must scan a thread's stack for information. For example, the garbage collector must scan every thread's stack for references to act as roots in its tracing algorithm. Similarly, exception handling code must also scan the stack to find

[1]To migrate a thread for the entire duration of its execution, the thread's run method can be migrated.

the location of an appropriate catch block to handle a thrown exception. If a thread has been migrated between core types, its stack will consist of a mix of PPE and SPE stack-frames, which could confuse such stack scanning code. For example, while the vast majority of the garbage collector stack scanning code is architecture neutral, the actual code which retrieves an object reference from a stack-frame is necessarily architecture dependent, since stack-frame layout varies between the core types.

The synthetic stack-frame, placed on a thread's stack as part of the migration process, acts as a marker to signal the transition from stack-frames of one core type to those of another. This enables these stack scanning algorithms to transition between PPE and SPE stack-frame scanning code as required. The Garbage collector stack scanning code uses these markers to switch between PPE and SPE stack-frame walkers. The exception handling code, on the other hand, is scanning the stack to find a suitable catch block in which to resume the thread's execution. It therefore migrates the thread to the other core type if it encounters a migration marker, such that it is on the correct core type on which to resume the thread's execution when it finds a suitable cache control block.

## 5.5 Experimental Analysis

This section presents an experimental evaluation of Hera-JVM with the following aims:

- Ensure that the SPE Java compiler and runtime in Hera-JVM supports real world Java code, and produces identical results whether code is executed on the PPE or SPE core type.
- Investigate the effectiveness of the software caching mechanism used by the SPE runtime system to hide the Cell processor's unusual memory hierarchy.
- Verify the hypothesis that a heterogeneous multi-core architecture can be abstracted behind a homogeneous virtual machine interface.
- Characterise the performance of each core type under different application behaviours. This will be used to uncover the behaviour characteristics which the runtime system should track to make effective core placement decisions.

To achieve these aims, Hera-JVM's performance is investigated in this section, under a range of both synthetic and real world benchmarks. Synthetic micro-benchmarks are used in Section 5.5.2 to investigate the differences in performance of fundamental Java operations, when executed on either the PPE or SPE cores. Section 5.5.3 uses real world benchmarks from three different Java benchmark suites to: (i) ensure real world Java code can be executed correctly on either core type; (ii) uncover the most appropriate core type for different types of applications; and (iii) measure the software cache's performance under real world load.

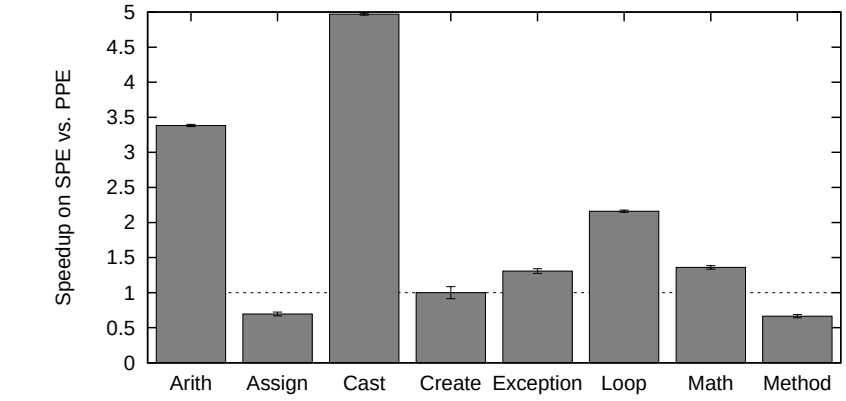
### 5.5.1 Experimental Setup

All the experiments in this section are performed on a Playstation 3 (PS3), with 256MB of RAM, running Fedora Linux 9. A 256MB swap space is located on the PS3's relatively fast video RAM, to minimise the paging overhead incurred, due to the small amount of RAM available on the PS3. The Cell processor contains 8 SPE cores; however, only 6 of these SPE cores are available on the PS3 used in this evaluation. All experiments compare single threaded performance of code executed on a single SPE core to that on a single PPE core, unless otherwise stated.

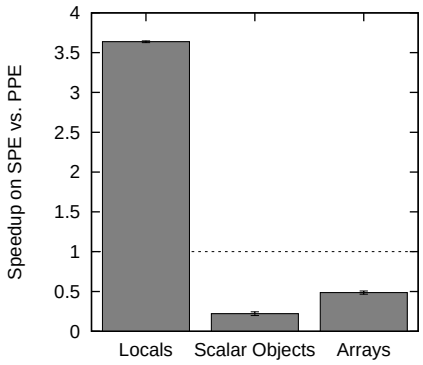
The baseline, non-optimising compiler was used to compile both PPE and SPE machine code. Hera-JVM was built with a stop-the-world, mark and sweep garbage collector. This collector only runs on the PPE core and thus becomes a scalability limitation if it runs for a considerable proportion of a benchmark. There is no fundamental reason the garbage collector cannot also execute on the SPE cores (it is written in Java like the rest of the runtime system); however, this support was not implemented in Hera-JVM for time reasons.

Each experiment was repeated ten times[1], with the average being reported and the standard deviation, between these runs, shown using error bars. The execution times of these benchmarks were calculated using the `System.currentTimeMillis()` method in the Java library.

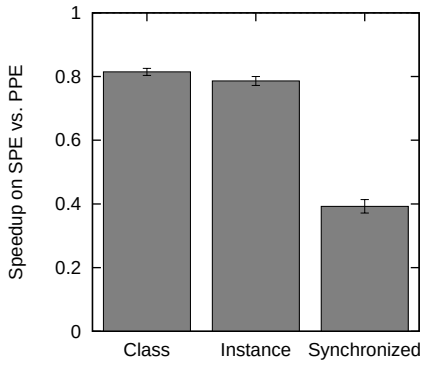
[1]After investigation, it was found that less than ten repeats were required before the standard deviation stabilised in these experiments.




(a) Overall benchmark results




(b) Assign tests




(c) Method tests

Figure 5.7: Performance difference between SPE and PPE cores for fundamental Java operations in the Java Grande micro-benchmarks.

### 5.5.2 Micro-Benchmarks

In this section, a series of micro-benchmarks are used to investigate different aspects of Hera-JVM's performance, under controlled conditions.

The first stage of this investigation is to characterise the performance of the various fundamental Java operations on both core types under Hera-JVM. The micro-benchmarks provided by the Java Grande benchmark suite (Mathew *et al.*, 1999) were employed to investigate the different Java operations individually.

Figure 5.7(a) shows the difference in performance between the core types for each of the micro-benchmarks included in section one of the Java Grande Suite[1]. There is

[1]These experiments use version 2.0 of the sequential Java Grande suite, available at `http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/sequential.html`

clearly a wide variation in capability between the PPE and SPE cores depending upon the type of Java operation being performed.

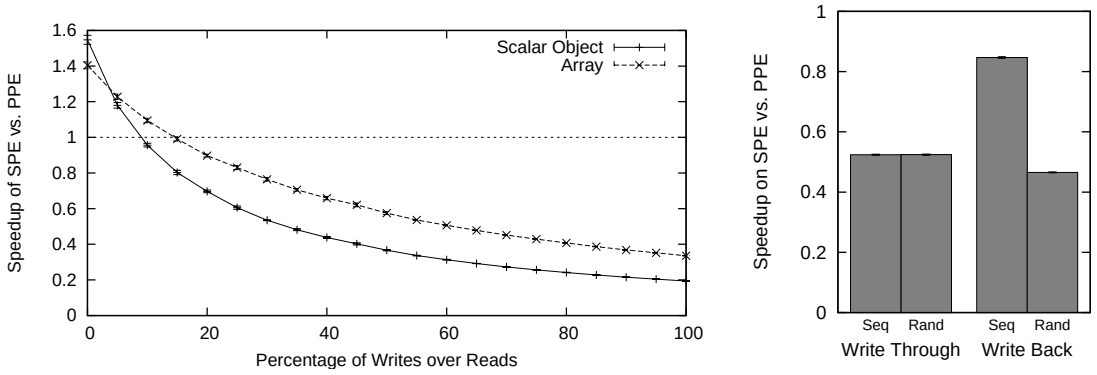
Basic operations, such as arithmetic, primitive casting and looping code, perform much better on the SPE core than on the PPE core. Some of these operations, such as floating point arithmetic and casting operations, are more than five times faster on the SPE core. This was expected, given that the SPE is highly tuned for floating point performance. However, even integer operations are significantly faster on the SPE core. The fact that looping code performs better on the SPE core, compared to the PPE core, was surprising. The PPE core has branch prediction hardware that is not found in the SPE cores[1]. This should reduce pipeline stalls on the PPE, thus increasing the performance of looping code. The fact that the loop benchmark performs worse on the PPE core may be explained by the shorter pipeline in the SPE core, which will reduce the impact on performance incurred by pipeline stalls.

More complex operations, such as object creation, exception handling and mathematical calculations, have roughly equivalent performance on both core types. The remaining two benchmarks (Assign and Method) both perform worse on the SPE than the PPE core. These benchmarks both directly test the software caching code for either heap data or method code access. To investigate which Java operations were responsible for the drop in performance, these benchmarks were broken out into their constituent tests (Figures 5.7(b) and 5.7(c) respectively).

Figure 5.7(b) shows that access to local variables (e.g. method parameters or variables on the thread's stack) is very fast on the SPE cores. However, accessing scalar objects or arrays on the heap is considerably slower. Sections 5.5.2.1 and 5.5.2.2 investigate the overheads involved in accessing heap data.

Figure 5.7(c) shows that most method invocations, whether static class methods or instance methods, are almost 80% as fast on the SPE core as on the PPE core. However, synchronised methods have a large overhead on the SPE core, due to the SPE core's software cache having to be purged before entering the synchronised method. This has a high cost on the SPE core for two reasons: it will cause cache misses for future heap accesses, which are much more expensive on the SPE core than the PPE core; and the software cache on the SPE must manually purge the cache by overwriting all

[1]The SPE cores do have explicit branch hint instructions, which can be used to reduce pipeline stalls. However, these are not yet used by Hera-JVM.




(a) Performance as ratio of reads to writes is varied.

(b) Effect of array write policy.

Figure 5.8: Heap data access performance.

cache hash-table entries, whereas the PPE does this with hardware. The code for the benchmarks presented in this figure fits easily in the SPE's code cache, and hence these method invocation tests do not cause any code cache misses. Section 5.5.2.3 investigates the effect of larger code working sets on SPE method invocation performance.

#### 5.5.2.1 Writing to the Heap

The tests in the assign benchmark of the Java Grande Suite read from, and write to, memory in equal measure. To investigate whether reading from objects and arrays is equally as costly as writing to them on the SPE, the benchmark was modified so that the ratio of reads to writes could be varied. Figure 5.8(a) shows the difference in performance between the SPE and PPE cores, as this ratio is varied. At small write ratios, the SPE core actually outperforms the PPE. The SPE is almost 55% faster than the PPE when reading from scalar objects and 40% faster when reading from array elements. Thus, even though the SPE must perform a software cache look-up operation for each object or array access[1], its simple design and the software cache's lightweight implementation make these accesses faster than the hardware cache on the PPE core.

However, the SPE's performance falls significantly as the write ratio increases. Above a write ratio of between 10% and 15%, the PPE core's performance outstrips

[1]This benchmark has a small enough working set such that reads always hit the cache, thus it only exercises the fast-path of the software cache.

that of the SPE. Writes are expensive on the SPE core because a write-through policy is used, meaning every write must be propagated to main memory. This is costly since each write to main memory requires a DMA transfer, which is relatively expensive to set up.

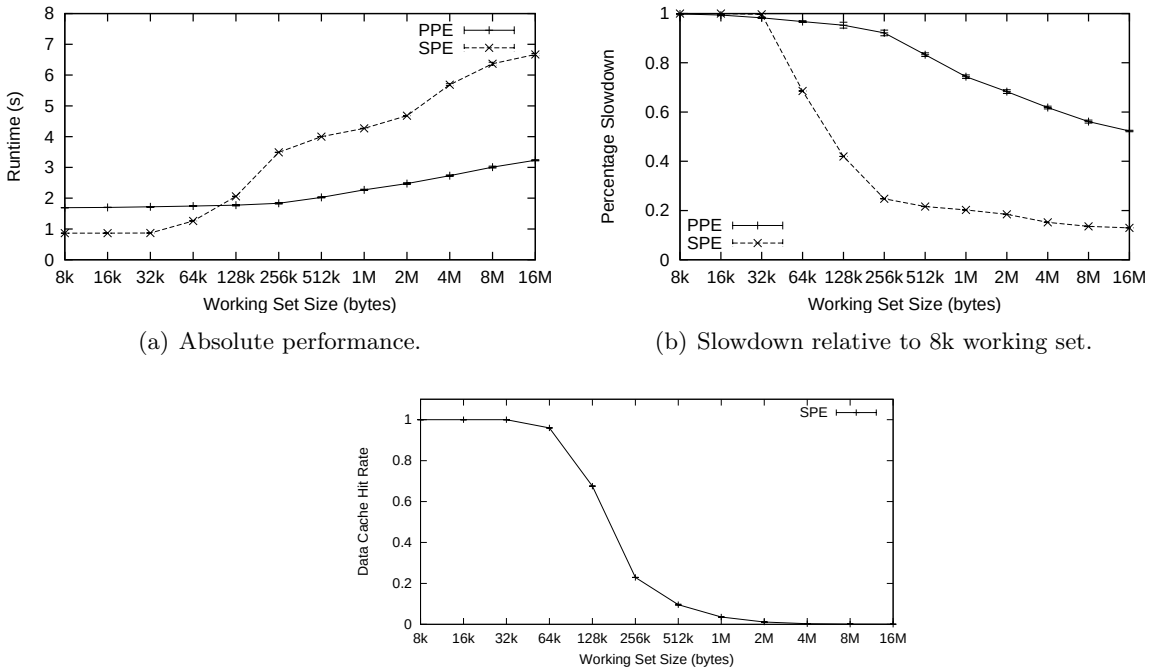
A write-back policy could significantly increase heap write performance by batching multiple write operations into a single DMA transfer. To investigate whether such a policy can improve performance, the array write-back policy, described in Section 5.3.3.3, was implemented. This approach batches multiple sequential writes to the same array into a single DMA transfer, thus it only benefits sequential access to an array.

Figure 5.8(b) shows the performance of the array assign benchmark (with 50% write ratio) under both caching schemes. The benchmark was modified to either write to array elements sequentiality or randomly. The write-back policy can batch up to 32 writes in a single DMA transfer, achieving a 60% performance increase over the write-through approach, when the array is accessed sequentially. For random array accesses, the write-back policy cannot batch transfers, and consequently does not improve performance. In fact, in this case, the added logging overhead decreases performance by 10% over the write-through policy. However, since arrays are often accessed sequentially, this would seem a worthwhile trade-off. In fact, the magnitude of performance increase provided by the write-back policy suggests that exploring this policy for object accesses would also be worthwhile.

Unfortunately, this write-back implementation introduced errors when executing more complex applications. These could not be corrected due to time constraints and therefore all subsequent experiments use the write-through policy. However, informal experiments were performed using the mandelbrot and compress benchmarks, introduced in Section 5.5.3. These showed a performance improvement of 4% for the mandlebrot benchmark, and 26% for the compress benchmark, when the write-back policy was used.

#### 5.5.2.2 Data Caching Overheads

Each micro-benchmark presented above has a small enough working set that the data it accesses always fits in the cache. To investigate the overhead of data caching, a micro-benchmark was devised in which the size of the program's data working set could be varied. This benchmark reads from, and writes to, randomly selected elements of an




(a) Absolute performance.

(b) Slowdown relative to 8k working set.

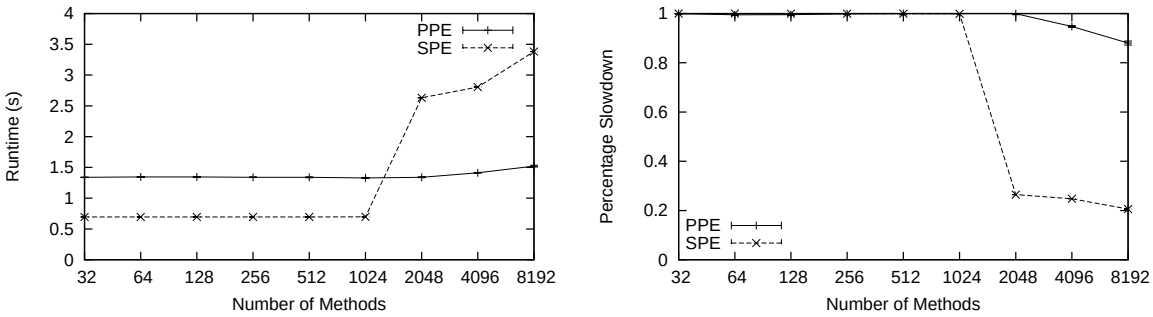
(c) SPE Cache Hit Rate.

Figure 5.9: The effect of a thread's data working set on performance.

array. The size of the array can be varied to alter the program's working set size and affect its cache hit rate. This benchmark represents the worst case in performance for a particular working set size, since access is entirely random and no real work is done between heap accesses.

Figure 5.9 shows the performance of the SPE and PPE cores for this benchmark, both as the absolute runtime required (a), and the slowdown, relative to the smallest working set size (b). Figure 5.9(c) shows how the hit rate of the SPE core's software data cache varies with this benchmark's working set size.

The SPE's performance initially surpasses that of the PPE. However, as expected, cache misses on the SPE core are more expensive than on the PPE core, due to the caching being performed in software, rather than under hardware control. Once the size of the working set grows larger than the amount of local memory reserved for the SPE's data cache (96KB), its performance degrades severely. The PPE core has a larger data cache (256KB in its L2 cache). Its performance does suffer after the working




(a) Absolute performance.

(b) Slowdown relative to 32 methods.

Figure 5.10: The effect of a thread's code working set on performance.

set size increases above this cache size, but not as severely as on the SPE core. For the maximum working set size of 16MB, the overhead due to cache misses reduces the SPE core's performance to about an eighth of its original value, while the PPE core's performance only drops by a half.

#### 5.5.2.3 Code Caching Overheads

Method invocation also involves caching of code from main memory. To investigate the performance of the software caching scheme used by Hera-JVM, a micro-benchmark was developed, in which the amount of code executed can be varied, while the same amount of real work is done. This benchmark performs three million method invocations, randomly selecting which method to invoke from a set of available methods. Every method in this set performs the same operation (incrementing a local variable). However, each is compiled into separate machine code, and so its code is cached separately when run. By varying the number of methods in the set, the amount of code which the benchmark executes can be varied, without altering the amount of "real" work that it performs.

Figure 5.10 shows the performance of this benchmark on both the SPE and PPE cores. Once the working set of methods that this benchmark invokes grows beyond 1024, it can no longer fit in the SPE's local memory cache. Performance on the SPE core drops to about one fifth of its original value, since almost every method invocation will need to re-cache the method's code, as it is likely to have been evicted since the method was last called. The benchmark's performance does not suffer so severely on the PPE core, again due to its dedicated caching hardware.

### 5.5.3 Real World Benchmarks

In this section, a selection of benchmarks from three real world benchmark suites are used to evaluate Hera-JVM in a realistic setting. To provide a range of applications, with different types of behaviour, benchmarks were selected from: (i) SpecJVM 2008 (Shiv *et al.*, 2009), a suite which mimics a variety of general purpose applications; (ii) the Java Grande Parallel benchmark suite (Smith *et al.*, 2001), which aims to replicate high performance computing workloads, such as scientific, engineering or financial applications; and (iii) the Dacapo 2006 benchmark suite (Blackburn *et al.*, 2006), which focuses on memory hungry benchmarks. The following benchmarks were run under Hera-JVM:

**mandelbrot** generates an 800x600 pixel image of the mandelbrot set, using an escape time algorithm with a maximum of 200 iterations. This benchmark was developed independently and does not belong to any of the benchmark suites.

**JavaGrande: mol_dyn** performs a molecular dynamics particle simulation, using the Lennard-Jones potential.

**JavaGrande: monte_carlo** performs a financial simulation, using Monte Carlo techniques to price products derived from the price of an underlying asset.

**JavaGrande: ray_trace** renders a scene containing 64 spheres, using a 3D ray tracer with a resolution of 150x150 pixels.

**Spec: fft** performs Fast Fourier Transformation, using a one-dimensional, in-place algorithm with bit-reversal and Nlog(N) complexity.

**Spec: lu** computes the LU factorization of a dense, in-place matrix using partial pivoting. It uses a linear algebra kernel and dense matrix operations on a 100x100 matrix.

**Spec: monte_carlo** approximates the value of Pi by computing the integral of the quarter circle y = $\sqrt{1 - x^2}$. It chooses random points within the unit square and computes the ratio of those within the circle, versus those outside the circle.

**Spec: sor** simulates the Jacobi successive over-relaxation algorithm for a 250x250 grid data set. The algorithm performs basic grid averaging, where each element is assigned an average weighting using the weights of its four nearest neighbours.

**Spec: sparse** performs matrix multiplication on an unstructured sparse matrix in compressed-row format with a prescribed sparsity structure. The data set is a compressed $25,000 \times 25,000$ matrix with 62,500 non-zero elements.

**Spec: compress** compresses and decompresses 3.36MB of data, using a modified Lempel-Ziv method.

**Spec: mpegaudio** decodes six MP3 files which range in size from 20KB to 3MB. It uses an MP3 library called JLayer.

**Dacapo: antlr** parses multiple grammar files, and generates a parser and lexical analyzer for each.

**Dacapo: hsqldb** uses JDBC to invoke an in-memory, SQL relational database, modelling the transactions of a banking application with 20 client threads, performing 64 transactions each.

Hera-JVM can support execution of these benchmarks on either core type. By annotating each benchmark's main method, Hera-JVM can migrate the whole benchmark to the SPE core. Other than adding this annotation, the only modification required to execute these benchmarks on the SPE cores was to split a small number of exceptionally long methods into multiple smaller methods, so that they could fit in the SPE's code cache in their entirety. Developing a code caching scheme which splits a method into multiple cacheable blocks would remove the need for these modifications.

The same Java code for each benchmark is run on either the PPE or SPE cores. These benchmarks execute correctly and consistently on either core type under Hera-JVM. This validates the correctness of the SPE compiler and runtime for real world Java applications, and verifies that a heterogeneous multi-core architecture can be abstracted behind a homogeneous virtual machine interface.

There is an inconsistency between the PPE and SPE floating point instructions which can lead to subtle inconsistencies in output, depending upon the core type on which a thread is executed. The PPE and SPE cores use different rounding modes for

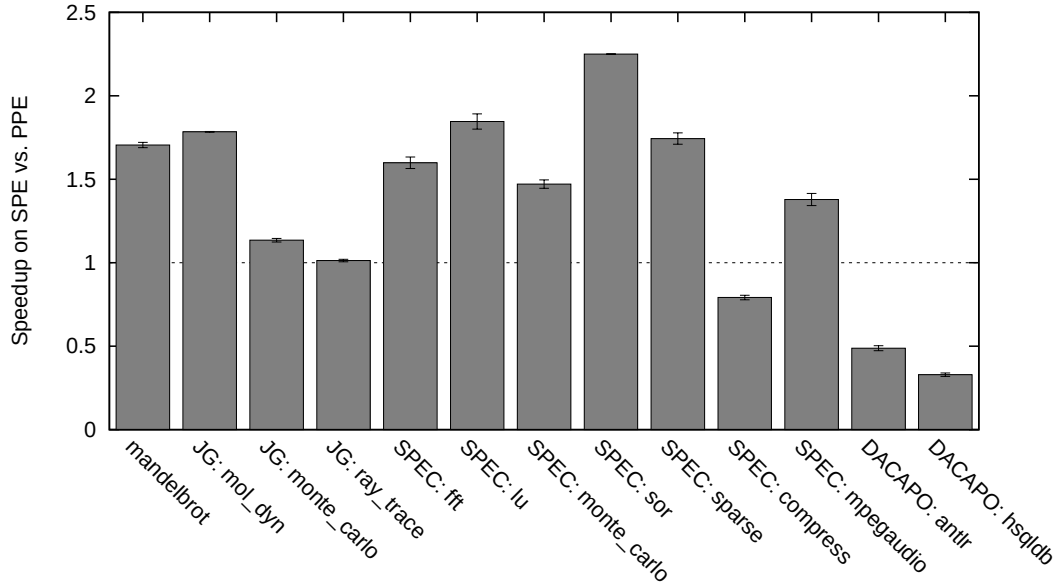



Figure 5.11: Performance comparison between benchmarks running on a single SPE core, and running on the single PPE core.

single-precision floating point calculations[1]. This can lead to minor (least significant bit) differences in the result of the same floating point operation under both core types. However, only single-precision floating point operations are affected: both the SPE and PPE cores support the *round to nearest* rounding mode for the more commonly used (in Java applications) double-precision floating point operations. The only benchmark affected by this inconsistency is the SPEC:fft benchmark; however, this results in a deviation of less than $10^{-11}\%$ in its final result when run on the SPE core, as compared to the PPE core.

#### 5.5.3.1 Single Threaded Performance

Figure 5.11 shows the difference in the performance of these benchmarks when they are run on the SPE, versus the PPE core. The error bars represent the standard deviation between ten runs on each core type. There is a wide variation in the performance of

[1]Java requires floating point calculations to use the IEEE *round to nearest* rounding mode, which is supported by the PPE core; however the SPE only supports the *rounding towards zero* rounding mode for single-precision operations.

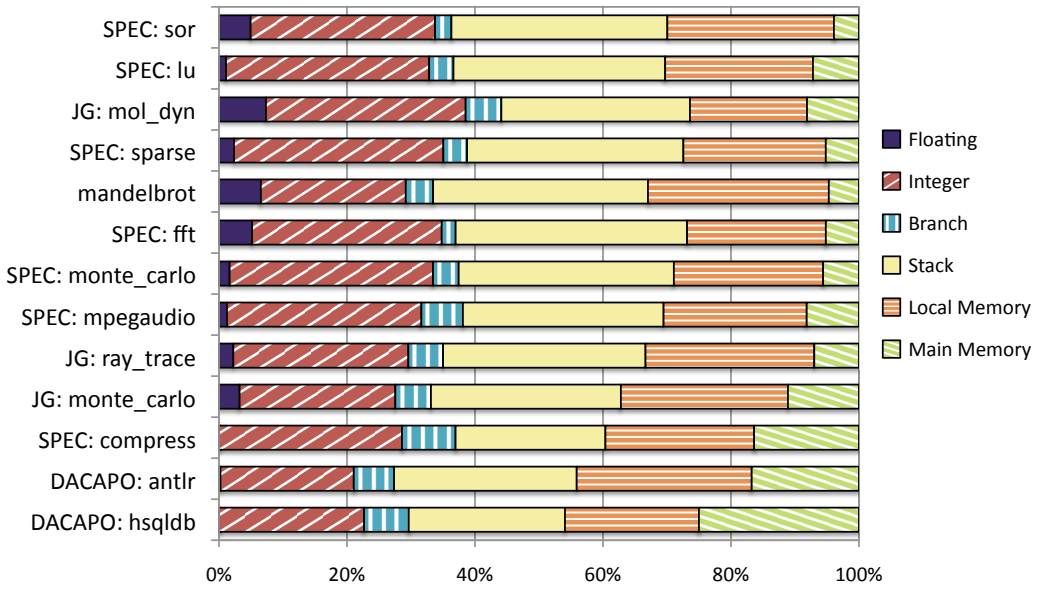



Figure 5.12: Percentage of cycles spent executing different classes of machine instructions on SPE. Benchmarks are ordered upwards by increasing SPE performance.

the benchmarks between core types, from a 2.25x increase in SPEC:sor on the SPE core, to a 3x slowdown for DACAPO:hsqldb.

The mandelbrot benchmark, SpecJVM 2008 suite (other than SPEC:compress) and Java Grande suite all perform well on the SPE core. These benchmarks are of a similar workload to that which the SPE was designed to support: computationally intensive scientific or multimedia centric workloads. The SPEC:compress and Dacapo benchmarks do not perform as well on the SPE core. The common trait linking these benchmarks is that they access large amounts of data, thus exercising the software cache on the SPE.

To further investigate how a program's behaviour affects its performance on the different core types, a simulator was used to calculate the percentage of time the SPE core spends executing different classes of machine instructions. Figure 5.12 shows this breakout by instruction type for each benchmark. The benchmarks are ordered by their performance on the SPE core, relative to the PPE core, with the best performing benchmark at the top.

Benchmarks which perform well on the SPE core also generally spend more of their

time executing floating point or integer-based calculations. While benchmarks with floating point code generally perform better on the SPE core, the best performing benchmarks do not always have a high proportion of floating point operations (e.g. SPEC:lu and SPEC:sparse). The higher performance of integer operations on the SPE core, as compared with the PPE core, seems to be as important as its impressive floating point performance.

There is also a clear trend towards a benchmark's performance decreasing on the SPE core when it spends a greater proportion of time accessing data elements in the heap (the local memory and main memory categories). This is especially prominent when those accesses result in cache misses or write operations which require DMA operations to main memory.

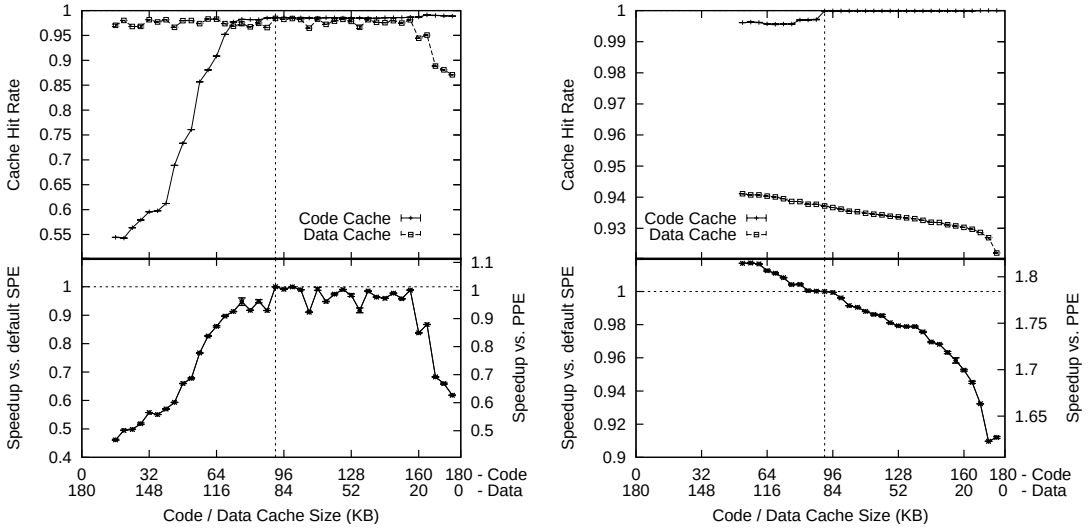
#### 5.5.3.2 The Effect of Cache Size

By default, the size of software data and code caches on the SPE core are fixed at 92KB and 88KB respectively. These sizes were chosen to give roughly equal weighting to caching of code and data by default. However, applications do not necessarily access the same amounts of heap data as code. Therefore, these applications may benefit from having a different proportion of local memory reserved for each cache.

To investigate the relationship between each benchmark's performance and the proportion of local memory reserved for code and data caching, Hera-JVM was modified, so that the ratio of data cache size to code cache size could be altered. Figures 5.13 to 5.15 show how the performance of each benchmark is affected, as this ratio is altered. The effect of cache size on hit rate for both data and code accesses is also shown in these figures. The cross formed by the dotted lines in these figures shows the performance at the default cache sizes.

The relationship between a benchmark's performance and this ratio falls into four main categories:

**Peaked:** The ray_trace and mpegaudio benchmarks show a peak in performance, when roughly an equal percentage of memory is reserved for each cache. These benchmarks are equally affected by small code or data caches. However, the plateau shape of these two graphs suggests that the working set of both code and data for both benchmarks fits comfortably in the local memory provided by the SPE core.



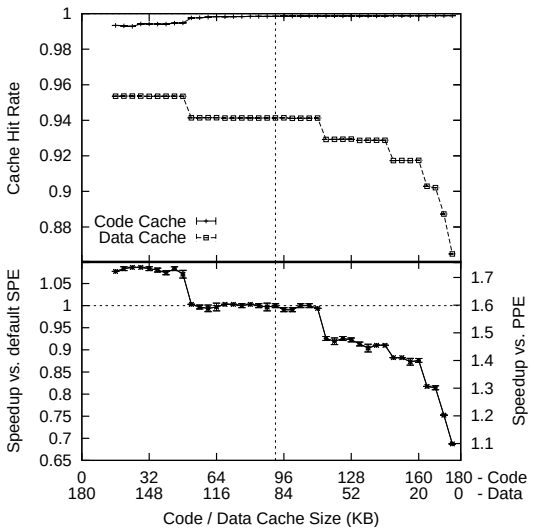

(a) JavaGrande: ray_trace

(b) JavaGrande: mol_dyn

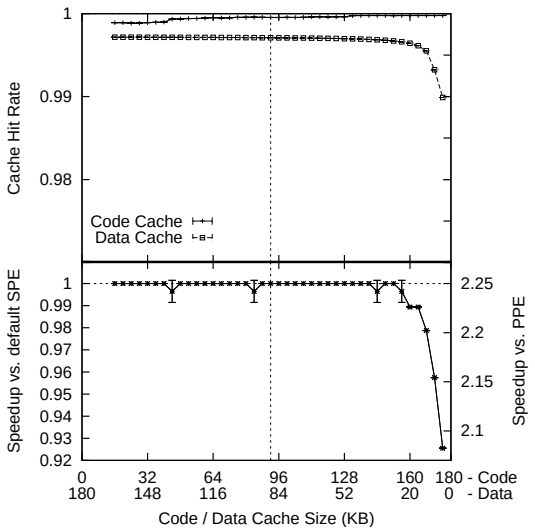
(c) JavaGrande: monte_carlo

(d) SpecJVM: monte_carlo

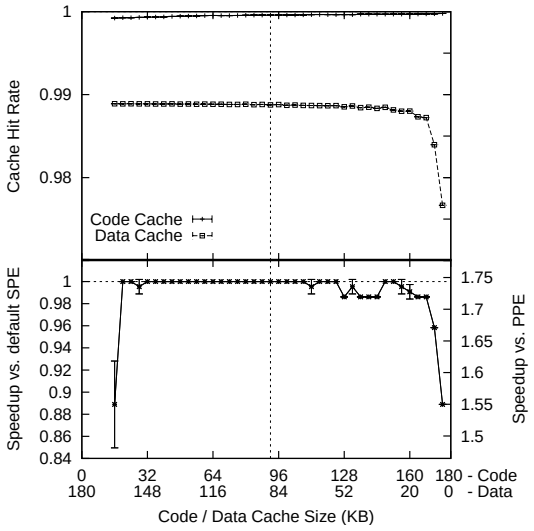
Figure 5.13: The effect of varying the proportion of local memory reserved for use by the data and code caches (Java Grande and SpecJVM:monte_carlo benchmarks).



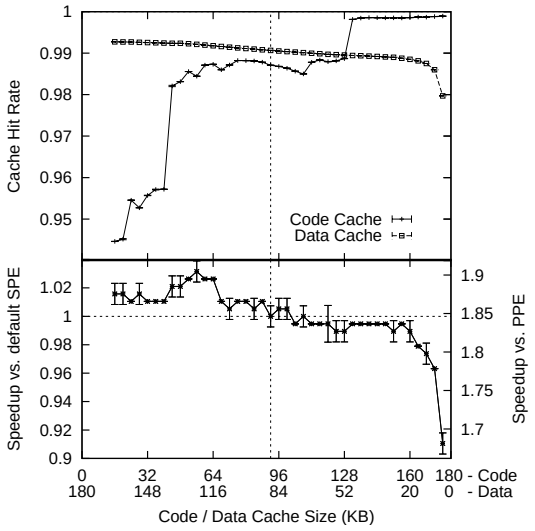

(a) SpecJVM: fft




(b) SpecJVM: sor

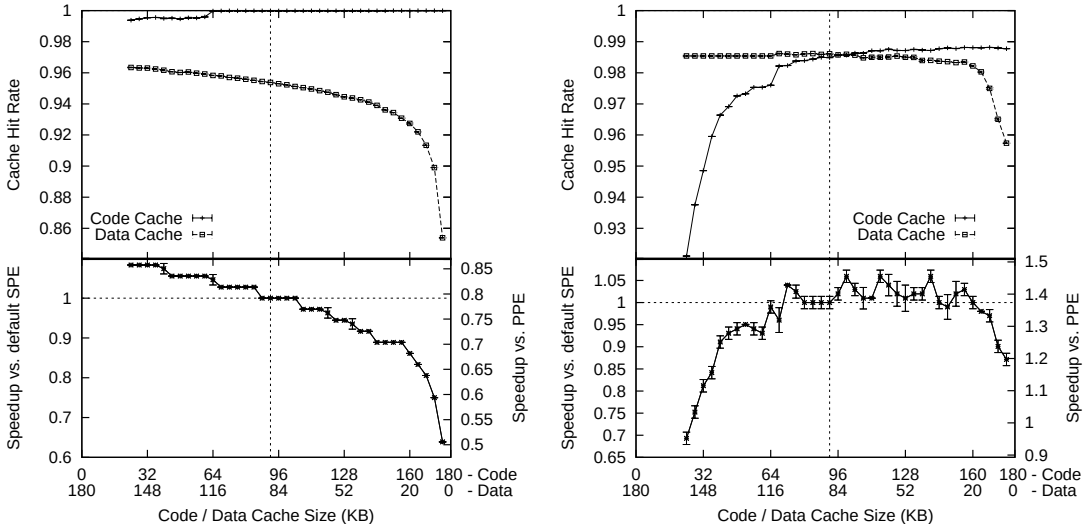



(c) SpecJVM: sparse



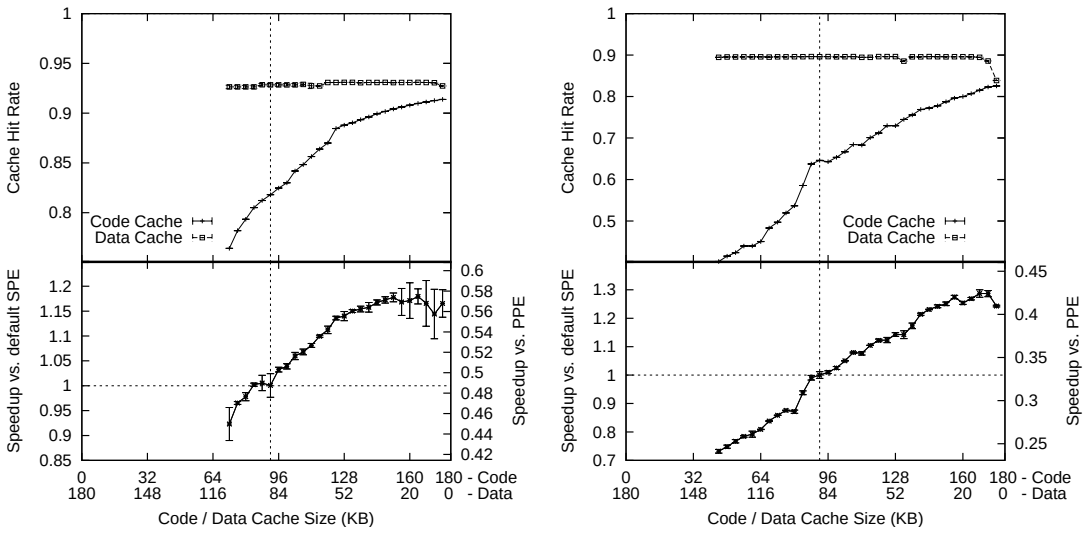

(d) SpecJVM: lu

Figure 5.14: The effect of varying the proportion of local memory reserved for use by the data and code caches (SpecJVM Scimark benchmarks).




(a) SpecJVM: compress

(b) SpecJVM: mpegaudio




(c) Dacapo: antlr

(d) Dacapo: hsqldb

Figure 5.15: The effect of varying the proportion of local memory reserved for use by the data and code caches (remaining SpecJVM and Dacapo benchmarks).

**Rising:** The performance of the JavaGrande:monte_carlo and Dacapo benchmarks rises as the proportion of memory provided to cached code increases. The working set of code required by these benchmarks is clearly too large to completely fit into the SPE's local memory, leading to this behaviour. It is also noticeable that the data cache hit rate for the Dacapo benchmarks is quite low (93% and 89%), compared to other benchmarks, and does not improve as the size of the data cache is increased. It is likely that this is caused by these benchmarks performing thread synchronisation operations, which purge the data cache, before the data cache becomes full.

**Falling:** The performance of the mol_dyn, fft, lu and compress benchmarks are more heavily affected by the size of the data cache. These benchmarks would seem to benefit from an even larger data cache size than can be provided by the SPE's local memory.

**Flat:** Finally, the SPEC:monte_carlo, sor and sparse benchmarks exhibit little variation in performance as the cache sizes change, except at extremely small cache sizes. These benchmarks therefore have very small working sets.

Given the different behaviours of these benchmarks, it is clear that no fixed segregation of the code and data caches will provide the best performance for all applications. One possible solution to this would be to mix code and data in a single larger cache. A problem with this approach is that the data cache must be purged on thread synchronisation operations, whereas the code cache need not. With a single shared cache, either the code must be needlessly purged alongside data, or a more complex cache allocation and purging scheme must be employed.

Another approach would be to provide Hera-JVM with the capability to dynamically alter the code / data cache ratio, based upon runtime monitoring of a program's cache hit rates. Although a full investigation of is beyond the scope of this dissertation, Figure 5.16 gives an approximation of the performance improvement this approach could result in, by plotting the best performance results shown by these cache ratio experiments. Those benchmarks which perform worst on the SPE core see the greatest gains in performance when the code / data cache ratio is optimised. In fact, given the trends seen by the compress, antlr and hsqldb benchmarks, it would seem that these

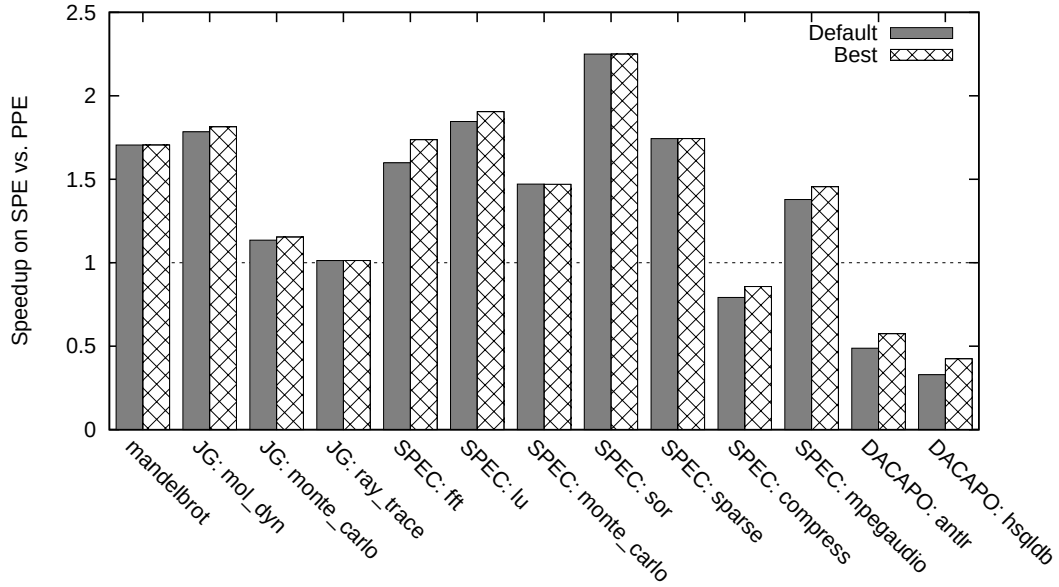



Figure 5.16: Performance with a per-benchmark optimal code / data cache ratio.

benchmarks would gain further performance improvements if the SPE's cache size was increased to the size of the PPE core (512KB). As it is, this optimisation could improve the performance of the hsqldb benchmark on the SPE core, such that it is within 2.25 times slower than when running on the PPE core. This is comparable to the 2.25x speedup observed by running the sor benchmark on the SPE core.

The provision of such a system for Hera-JVM is left for future research; the default code / data cache ratio of 88KB / 92KB is used in all subsequent experiments.

#### 5.5.3.3 Scalability

The preceding experiments compared the performance of a benchmark, run on a single SPE core, with that when run on a single PPE core. However, the Cell processor contains eight SPE cores and can provide significantly more computing power if an application can be parallelised. All of the benchmarks in the SpecJVM 2008 and the Java Grande Parallel suites can be parallelised, by passing the number of benchmark threads required as a parameter. The Dacapo antlr benchmark is single threaded and cannot be parallelised, however, the hsqldb benchmark already runs 20 client threads,

and thus should be parallelisable by increasing the number of SPE cores upon which these threads can be scheduled (this can be controlled with a parameter to Hera-JVM).

The Cell processor in the Playstation 3, used for these experiments, only provides six SPE cores for user applications (one core is disabled, due to manufacturing defects, and the other runs a secure hypervisor). Figures 5.17 to 5.19 show the speedup obtained by each benchmark as they are scaled from one to six SPE cores. This is shown both as a speedup against running on a single SPE core (a) and as the speedup against running on a single PPE core (b).

Most of the benchmarks scale well as the number of SPE cores increase. However, the lu, fft and both monte_carlo benchmarks stop scaling after four cores. The reason that these benchmarks stop scaling is believed to be due to garbage collection. These benchmarks allocate a large amount of data during their execution. When they are scaled to multiple threads, the amount of data increases with each benchmark thread created. This increased allocation pressure leads to much more frequent garbage collections, due to the small amount of memory available on the Playstation 3. Since the garbage collector currently runs on only the PPE core, this leads to a scaling bottleneck.

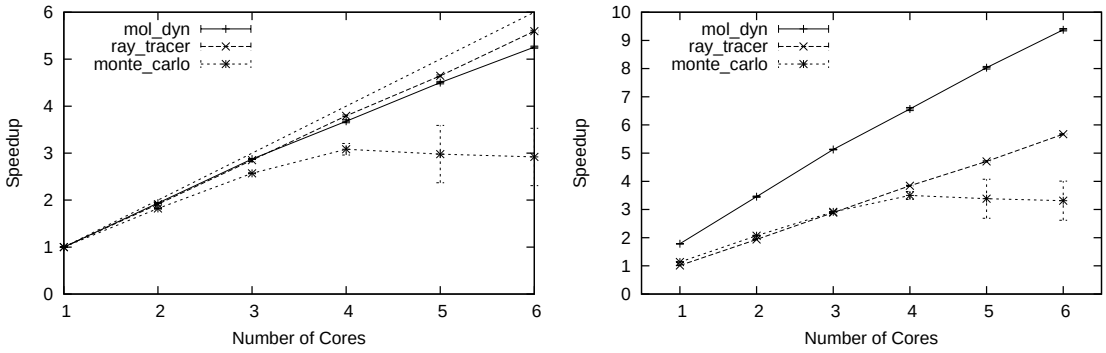
The hsqldb benchmark does not scale at all. This is believed to be due to the core engine of hsqldb not being multi-threaded[1], rather than any scalability issue within Hera-JVM.

Figure 5.20 provides an overview of the performance of running each benchmark on all six SPE cores, compared with running on the single PPE core. For those benchmarks which scale, running on all six SPE cores provides from a 3x to a 13x speedup, compared to running on the single PPE core.

## 5.6 Discussion

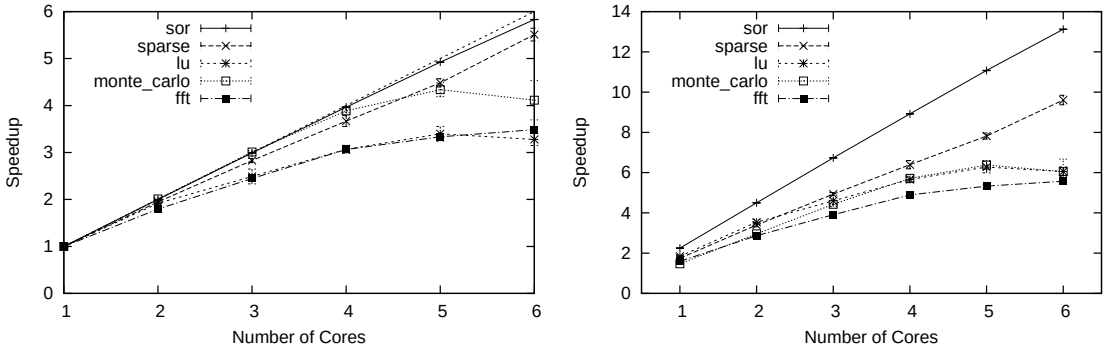
The SPE core is a challenging architecture on which to develop general purpose software. However, by abstracting this architecture behind a Java virtual machine, Hera-JVM hides the unusual features of the SPE core. This enables developers to write application code for this challenging architecture in the same manner as they would for more conventional architectures. This, in turn, enables the same code to be run on

[1]The hsqldb FAQ at `http://hsqldb.sourceforge.net/web/hsqlFAQ.html` states that the core engine is not yet multi-threaded.



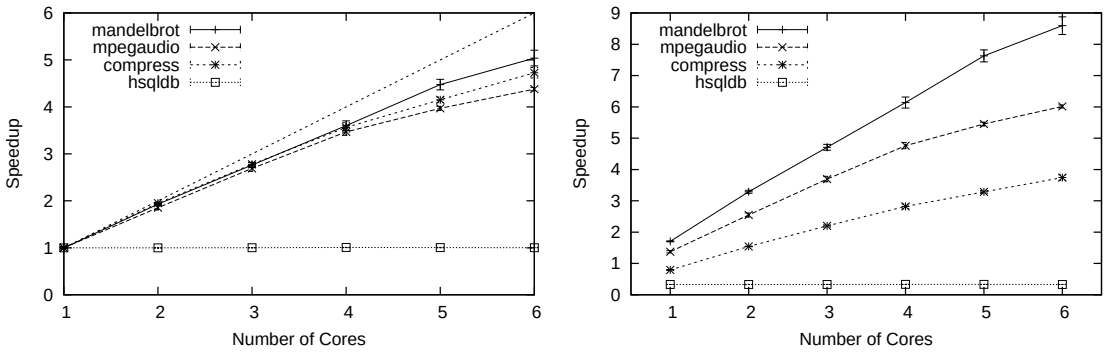

(a) Speedup vs. 1 SPE core (b) Speedup vs. 1 PPE core

Figure 5.17: Scalability of the Java Grande Parallel benchmarks.




(a) Speedup vs. 1 SPE core (b) Speedup vs. 1 PPE core

Figure 5.18: Scalability of the SpecJVM scimark benchmarks.




(a) Speedup vs. 1 SPE core (b) Speedup vs. 1 PPE core

Figure 5.19: Scalability of the remaining SpecJVM, Dacapo and mandelbrot benchmarks.

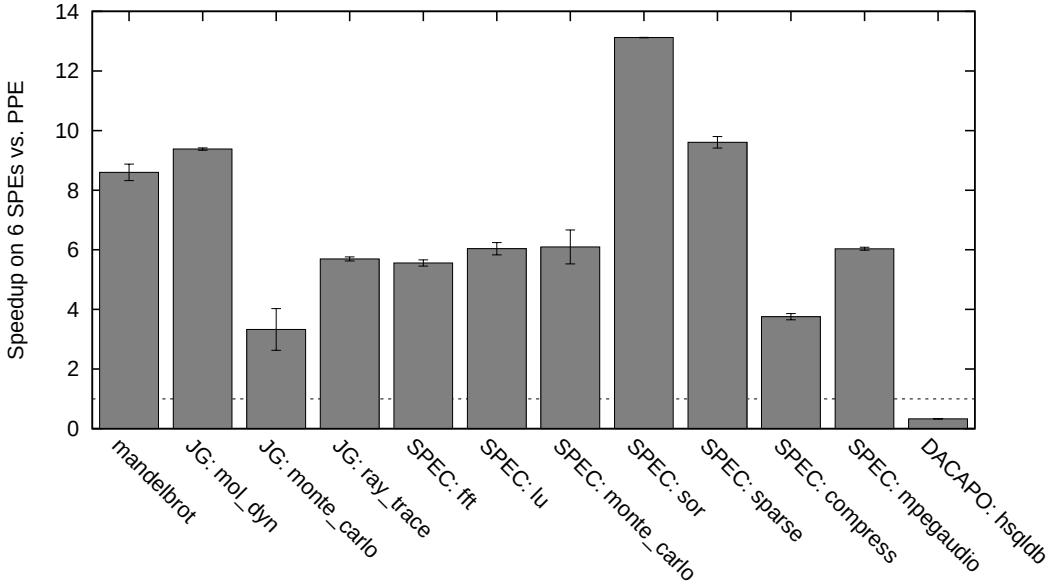



Figure 5.20: Performance comparison between benchmarks running on all 6 SPE cores and running on the single PPE core.

both the SPE core and the more conventional PPE core of the Cell processor, without requiring any modifications.

Of course, this abstraction does not come for free. Java is generally less efficient than lower level languages such as C or C++, due to features such as dynamic compilation, garbage collection and its object oriented nature (though recent advances (Blackburn *et al.*, 2004; Kotzmann *et al.*, 2008) are reducing this gap). Additionally, the wide impedance mismatch between the Java virtual machine abstraction and the SPE core's architecture is likely to increase the overhead of providing this abstraction.

However, the techniques described in Section 5.3, such as efficient stack management and software caching using high level type information, enable the SPE core to provide better performance than the PPE core for the majority of the benchmarks with which Hera-JVM was evaluated. Even those benchmarks which perform badly (antlr and hsqldb) have a similar slowdown on the SPE core as the speedup this core provides to more suitable benchmarks (the maximum performance disparity is about 2.25x on both sides).

This would suggest that, in most cases, the overhead of providing this abstraction is outweighed by the increased processing speed of the SPE core. This is illustrated

by the SPE core's performance in reading cached data from the heap (Figure 5.8(a)). Since the SPE must perform caching in software, even a cached object access requires eight machine instructions to look up the location of the cached data. The PPE core, on the other hand, can access the same data with a single memory access instruction, relying on hardware to automatically look up its cache. Even so, the SPE core performs better than the PPE core under this workload. The cost of providing this cached heap abstraction in software is amortised by the fact that accessing an SPE's local memory is significantly less expensive than accessing the PPE's data cache, due to its simple non-associative nature.

As the number of cores on a processor increases, hardware caches will become even more expensive and are likely to become scalability bottlenecks, due to the need for cache coherency between cores (Kumar *et al.*, 2005*b*). It may be that providing looser coherency guarantees in hardware, and relying on software abstractions to provide a similar programming experience, as with Hera-JVM, may be a better approach for future many-core architectures. While the Cell's approach is likely a step too far for most general purpose applications, these results with Hera-JVM show that, even with this architecture, it is possible to provide a relatively efficient caching abstraction in software.

Another advantage of the SPE core's simple design is that it requires much less area on a silicon die to implement. On the Cell processor die, the PPE core requires roughly the same area of silicon as 4 SPE cores. Thus, if an application can be parallelised, it can be executed on many more SPE cores than PPE cores, for the same sized processor. The scalability results show that, for those benchmarks which scale, even the worst performing benchmark provides a 3x speedup when running on four SPE cores, compared to running on a single PPE core, thus providing significantly better performance for the same silicon area.

Finally, the ability of Hera-JVM to transparently migrate a thread between the PPE and SPE core types provides the runtime system with the flexibility to fully utilise the heterogeneous core types of the Cell processor under varying conditions and workloads. Chapters 6 and 7 describe techniques for automatically selecting the most appropriate core type on which to schedule the different threads and phases of execution of a given application. This would not be possible without the homogeneous virtual machine interface and transparent migration provided by Hera-JVM.

# Chapter 6

# Migration Based Upon Behaviour Annotations

The Hera-JVM runtime system, described in the preceding chapter, hides much of the complexity involved in developing applications for heterogeneous multi-core architectures (HMAs). By providing a homogeneous virtual machine interface, an application developer no longer needs to develop, compile or package code specifically for a particular core type. However, given the heterogeneous nature of the cores, the same code will have different performance characteristics, depending upon the core type on which it is executed. For an application to effectively exploit an HMA system, the most appropriate core type for each thread or phase of execution must be chosen. If the application developer has to perform this choice manually (e.g., using the `@RunOnSubArch` and `@RunOnMainArch` annotations, as in the previous chapter), then he must still possess relatively in-depth knowledge of the capabilities of the different core types in order to best exploit the HMA processor. This would limit much of the benefit of hiding the processor's heterogeneity behind a homogeneous virtual machine interface.

This chapter discusses how the code behaviour characteristics, proposed in Chapter 4, can be employed by a runtime system to hide this partitioning process from the application developer, while still being able to exploit the cores' differing capabilities. These characteristics can be provided as code annotations, either explicitly added by a developer or automatically applied by a compiler, code analysis tool or a program profiling tool. This chapter focuses on annotations which are explicitly added by a developer; however, the same principles would apply for annotations added in another manner.

The first requirement for such a runtime system is a mechanism to track the set of behaviour characteristics which apply to a particular thread. Section 6.1 describes how Hera-JVM maintains this per-thread behaviour knowledge. This information is used by the runtime system to automatically infer the most appropriate core type for the different threads and execution phases which make up an application. Section 6.2 describes a thread migration policy, based upon thread behaviour cost functions, which decides whether a thread would benefit from being migrated to another core type. Finally, a mechanism is required to cause a thread, which has been selected for migration, to actually migrate at an appropriate point in its execution. A variety of migration strategies are investigated in Section 6.3.

## 6.1 Maintaining Per-Thread Behaviour Knowledge

To enable the runtime system to make scheduling decisions based upon the behavioural knowledge provided by code annotations, it must maintain the set of behaviour annotations which apply to each thread and track changes to this set as the thread executes. This section describes the mechanism used by the Hera-JVM runtime system to maintain this per-thread behaviour knowledge.

### 6.1.1 Set of Tracked Behaviour Annotations

The first design decision is to determine the set of annotations that the runtime system should track. The annotations chosen should be those which describe the execution behaviour patterns most likely to have a large impact on core placement. For example, if both core types have similar integer operation performance, then tracking the `@IntegerCode` annotation will be of little value, since this behaviour will have a negligible influence on core placement. The set of tracked annotations will therefore be dependent upon the characteristics of the HMA targeted by the runtime system.

Since Hera-JVM targets the Cell processor, the choice of tracked annotations reflect the most significant performance characteristic differences between the PPE and SPE core types. Both the micro-benchmarks in Section 5.5.2 and the macro-benchmarks in Section 5.5.3 suggest the most significant performance differences to be:

- Purely arithmetic operations (whether integer or floating point) have significantly faster performance on the SPE cores.

- Object and array accesses are slightly faster on the PPE core if the thread's working set fits in the SPE's on-core caches.
- Object and array accesses are significantly faster on the PPE core if the thread's working set does not fit in the SPE's on-core caches.

Therefore, the `@ArithmeticCode`, `@ObjectAccessCode` and `@LargeWorkingSet` behaviour annotations were selected to track each of these behaviours, respectively. The first two annotations describe processing requirement characteristics, as introduced in Section 4.2.1. The `@ArithmeticCode` annotation merges the `@IntegerCode` and `@FloatingPointCode` characteristics into a single annotation. The `@ObjectAccessCode` annotation is simply the `@DataAccessCode` annotation, renamed to suit Java parlance. Finally, the `@LargeWorkingSet` annotation describes an execution behaviour characteristic, as outlined in Section 4.2.2. This annotation is not parenthesised with the expected size of the working set. Instead, its presence is assumed to imply a working set size larger than the 96KB data cache provided by the SPE core type.

An application developer can tag a method with one (or more) of these annotations to signal the behaviour which is expected to dominate that method's execution. As a convenience, a class can also be annotated, which is interpreted by Hera-JVM as all the methods in that class being tagged with that behaviour annotation. A thread inherits a behaviour characteristic when it invokes a method annotated with the corresponding behaviour annotation, until that method returns (i.e., the method, and the full call tree of methods called by this method, inherit the behaviour characteristic). Therefore, to state that a thread has a given behaviour throughout its lifetime, the developer can simply annotate the thread's `run()` method, since this will be the first method invoked by the thread.

There may be situations where it is convenient to remove a behaviour characteristic from a thread for the duration of a particular method. For example, a thread which performs mainly arithmetic computations, and is thus tagged with the `@ArithmeticCode` annotation, may call a method which does very little arithmetic (e.g. a method to log the results to a file). The ability to remove the `@ArithmeticCode` behaviour characteristic from the thread when it invokes this method (and restore it once the method returns) would enable the runtime system to migrate this method to a more appropriate core type. Hera-JVM provides three further annotations for this purpose

(`@NonArithmeticCode`, `@NonObjectAccessCode` and `@SmallWorkingSet`). These annotations behave similarly to those above, but cause the thread to lose, rather than inherit, the corresponding behaviour characteristic for the duration of a tagged method.

### 6.1.2 Tracking Thread Behaviour at Runtime

The behaviour characteristics of a thread depend not only on the current method it is executing, but also on any behaviour annotations encountered in the sequence of methods called to reach this point. The runtime system must therefore maintain a representation of each thread's behaviour characteristics, and modify this representation whenever a behaviour annotation is encountered.

Hera-JVM maintains a behaviour bitmap for each thread. Each of the tracked behaviour characteristics is represented by a single bit — set for on, cleared for off. There are only two operations which can cause a thread's behaviour characteristics to change: the thread invokes a method tagged with a behaviour annotation; or the thread returns from a method tagged with a behaviour annotation. Therefore, the runtime system can maintain a thread's current behaviour characteristics by modifying method invocations and returns to update the behaviour bitmap appropriately.

When JIT compiling a method, the runtime system checks whether it has been tagged with any behaviour annotations. If so, a short sequence of machine code is added to the method's prologue (which is executed whenever the method is invoked). This code loads the executing thread's behaviour bitmap, sets and / or clears the bits which correspond to the behaviour annotations tagging this method, then stores the updated bitmap.

When a method returns, it must undo any changes it made to the thread's behaviour bitmap. However, simply clearing the behaviour characteristics which tag the method is not appropriate, since the thread may have already had some of these behaviour characteristics prior to the method's invocation. Instead, the method's prologue pushes the thread's behaviour bitmap onto its stack before it modifies the bitmap. During the method's epilogue (which is executed whenever the method returns), this value is popped from the stack, and saved in the thread's behaviour bitmap; thus restoring the thread's behaviour characteristics.

As an optimisation, rather than directly updating the thread's behaviour bitmap, Hera-JVM performs these updates on a per-core bitmap, which resides at a fixed loca-

tion (in local memory for the SPE cores). This optimisation saves one memory access per update on the PPE core, and an expensive software cache lookup on the SPE cores. During a thread switch operation, this bitmap is saved in the outgoing thread's control block, and replaced with the incoming thread's behaviour bitmap. Using this approach, behaviour tracking adds only seven PPE machine instructions or ten SPE machine instructions to methods tagged with behaviour annotations. Methods which are not tagged are unaffected.

## 6.2 A Cost Function-Based Migration Policy

Section 6.1 described the mechanism employed by Hera-JVM to track the behaviour characteristics of threads as they execute. This section describes a cost function based migration policy, which can use this thread behaviour knowledge to decide whether a thread should be migrated to a different core type.

The overall approach involves the runtime system periodically evaluating the *cost* that a thread incurs, based upon its current behaviour characteristics and the effect incurred by these characteristics on each of the different core types available to the runtime system. If this cost is higher for the thread's current core type than another core type, the runtime system may choose to migrate the thread to a more appropriate core type. To calculate these costs, each core type has an associated cost function, which takes a thread's current set of behaviour characteristics as its input.

The purpose of the cost function is to evaluate the most appropriate core type for a thread with a given set of behaviour characteristics. It is not necessary for the cost function to produce an absolute, or even particularly realistic measure of performance on a given core type. Instead, it need only produce a relative measure of the expected difference in performance between core types. As such, the costs associated with each behaviour characteristic are based upon the relative difference in performance observed for that behaviour between the different core types.

The micro-benchmark tests, performed in Section 5.5.2, provide an estimation of the relative performance disparity for these behaviour characteristics between the SPE and PPE core types under Hera-JVM. The *arithmetic* micro-benchmark suggests that when running purely arithmetic operations, the SPE is approximately four times faster than

| Behaviour | PPE Cost | SPE Cost |
|---|---|---|
| Arithmetic ($C_A$) | 4 | 1 |
| Object Access ($C_O$) | 1 | 2 |
| Large Working Set ($C_L$) | 2 | 8 |

Table 6.1: Costs associated with each core type.

the PPE. Similarly, the *assign* micro-benchmark suggests a 2x slowdown for purely object access code on the SPE versus the PPE. Finally, the *working set* micro-benchmarks show that when a thread's working set does not fit in the SPE's local memory, the overhead of software caching causes up to an 8x slowdown. The slowdown on the PPE, on the other hand, is only 2x. Using these results, the costs shown in Table 6.1 were chosen for these behaviour characteristics on each core type.

The total cost of a thread is calculated as the sum of the costs of those behaviour characteristics currently associated with that thread (Equation 6.1). In this equation, the $B_A$, $B_O$ and $B_L$ terms represent the Arithmetic, Object Access and Large Working Set behaviour characteristics, respectively. These terms are binary values, set to one if the thread has the corresponding behaviour characteristic, otherwise set to zero. The $C_A$, $C_O$ and $C_L$ terms are the behaviour costs as defined in Table 6.1. The thread's cost is calculated for each core type (the PPE and SPE cores in Hera-JVM), substituting the appropriate behaviour costs for that core type. The behaviour costs are summed, rather than being combined as their product, since these behaviours are independent from each other, and so a linear combination is the most appropriate.

$$C = B_A \cdot C_A + B_O \cdot C_O + B_L \cdot C_L \tag{6.1}$$

This *raw* cost function calculates the cost of the thread at a given moment in time. However, it may be appropriate to take the history of a thread's behaviour into account before making migration decisions. This will limit those migrations that occur due to very short behaviour phase changes, and are thus likely to incur more overhead than any benefit they might bring.

A thread's behaviour history is incorporated into the cost function using an exponential moving average function (Equation 6.2). For a given time $t$, a smoothed cost ($C'_t$) is calculated by combining the current raw cost from Equation 6.1 ($C_t$) with the previous time period's smoothed cost ($C'_{t-1}$). The value of $\alpha$ can be varied to affect the

degree to which the thread's behaviour history ($C'_{t-1}$) influences the thread's current behaviour cost ($C'_t$).

$$C'_t = (1 - \alpha) \cdot C_t + \alpha \cdot C'_{t-1} \tag{6.2}$$

Once a thread's cost has been calculated for both the SPE and PPE core types ($C'_{ppe}$ and $C'_{spe}$), these values can be compared to decide if migration would be beneficial. A target score ($S$) is calculated using Equation 6.3.

$$S = C'_{spe} - C'_{ppe} \tag{6.3}$$

A negative target score suggests the thread will run better on the SPE core, while a positive score recommends the PPE core. However, a policy which simply migrates the thread as soon as the target score switches from positive to negative (or vice-versa), could lead to unwanted oscillation of a thread between core types if its target score fluctuates around zero. *Hysteresis* can be used to counteract this unwanted oscillation.

A system which employs hysteresis takes into account its present state, as well as the value being measured, before deciding whether to transition to another state. Thus, the point at which a system changes state depends upon the system's current state. For this cost function, hysteresis is used to vary the target score at which a migration should occur, depending upon the current core type on which the thread is running (see Figure 6.1). A target score within the window of $-\delta$ to $+\delta$ will not trigger a migration. Thus $\delta$ can be thought of as the cost of migration; as $\delta$ is increased, a greater difference in cost between the core types is required to trigger a migration.

Given that the target score $S$ does not have a symmetrical absolute range, it is inappropriate to specify $\delta$ as an absolute value. Therefore, we define $\delta$ as a fraction ($\beta$) of the cost of the core on which the thread is currently executing (i.e. $\delta = \beta \cdot C'_{curr}$, where $C'_{curr}$ is $C'_{ppe}$ or $C'_{spe}$). The migration decision is thus captured by Equation 6.4 if the thread is currently executing on the PPE core, or Equation 6.5 if it is executing on an SPE core.

$$\text{migrate to SPE if} : S < -\beta \cdot C'_{ppe} \tag{6.4}$$

$$\text{migrate to PPE if} : S > +\beta \cdot C'_{spe} \tag{6.5}$$

Another issue which may affect a migration decision is introduced by including the history of a thread's behaviour into the cost function; a thread's target score may

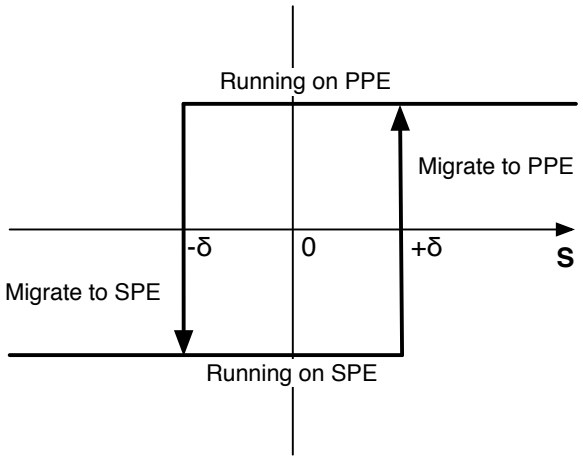



Figure 6.1: Migration with Hysteresis.

be slow to react to changes in the thread's current behaviour. While this is one of the reasons for including this history information — to provide greater stability by limiting short-lived migrations — it may result in migrations at inappropriate times. For example, a thread's behaviour history may cause its calculated target score to remain above the threshold required to trigger a migration, even though its current behaviour costs from Equation 6.1 suggest a migration is unnecessary.

To limit these inappropriate migrations, the runtime system can also take the trend in the direction of the target score into account. For example, if the target score is trending downwards, it may be better not to migrate a thread from an SPE core to the PPE core, even though its target score is above the PPE migration threshold. The downward trend means that the thread's current behaviour costs favour continued execution on the SPE core; however, a history of PPE favourable behaviour means the target score still lies above the PPE migration threshold[1].

The target score's trend in direction is incorporated into the migration decision by comparing a thread's current target score ($S_t$) to its previous target score ($S_{t-1}$). If the score is trending in the wrong direction, migration is prevented (Equations 6.6 and 6.7). The parameter $\gamma$ is introduced to control how strong the trend must be to

[1]In an ideal system the thread would have been migrated to the PPE core previously, when it first passed the migration threshold, and thus this situation would not occur. However, in a real system, factors such as non-migratable code sections or forced migration can cause this situation.

| Parameter | Effected Property | Prevents | Default Value | Value to turn off property |
|---|---|---|---|---|
| $\alpha$ | Behaviour History | Short-lived migrations | 0.8 | 0 |
| $\beta$ | Hysteresis | Oscillating migrations | 0.2 | 0 |
| $\gamma$ | Trend tracking | Badly timed migrations | 1 | 0 |

Table 6.2: Migration policy parameters.

prevent migration. Note, this trending information is never used to cause a migration, only to prevent a migration from occurring.

$$\text{do not migrate to SPE if} : S_t > \gamma \cdot S_{t-1} \tag{6.6}$$

$$\text{do not migrate to PPE if} : S_t < \gamma \cdot S_{t-1} \tag{6.7}$$

Thus, incorporating hysteresis from Equations 6.4 and 6.5, and trend tracking from Equations 6.6 and 6.7, the migration decision becomes:

$$\text{migrate to SPE if} : (S_t < -\beta \cdot C'_{ppe}) \text{ and not } (S_t > \gamma \cdot S_{t-1}) \tag{6.8}$$

$$\text{migrate to PPE if} : (S_t > +\beta \cdot C'_{spe}) \text{ and not } (S_t < \gamma \cdot S_{t-1}) \tag{6.9}$$

or, more simply:

$$\text{migrate to SPE if} : (S_t < -\beta \cdot C'_{ppe}) \text{ and } (S_t < \gamma \cdot S_{t-1}) \tag{6.10}$$

$$\text{migrate to PPE if} : (S_t > +\beta \cdot C'_{spe}) \text{ and } (S_t > \gamma \cdot S_{t-1}) \tag{6.11}$$

The degree to which history, hysteresis and trend information is taken into account by the migration policy can be tuned using the parameters $\alpha$, $\beta$ and $\gamma$, respectively. Indeed, these parameters can be used to turn off the associated property entirely. Table 6.2 outlines some of the characteristics of these parameters. In Section 6.4.3 these parameters are varied to investigate their effects and find their optimal default values.

## 6.3 Implementing Behaviour Based Thread Migration

This section describes the development of a concrete implementation of this migration policy for Hera-JVM. A number of design choices must be made to develop a real world implementation, such as how often a thread's cost should be evaluated and where in a

thread's execution it should be migrated. The decisions taken for these design choices trade off immediacy of reaction to behaviour changes with migration and behaviour measuring overheads. To investigate these trade-offs, this section explores a number of different mechanisms for evaluating a thread's cost and signalling it to migrate.

### 6.3.1 Evaluating a Thread's Cost

There are two main options which can be employed by Hera-JVM to evaluate the per-core-type costs associated with a thread's behaviour characteristics: the migration policy can be re-evaluated immediately, whenever a thread's set of behaviour characteristics change, or the runtime system can sample a thread's behaviour characteristics with a fixed time frequency, re-evaluating the thread's per-core-type cost at every sampling period.

Immediate cost evaluation has the advantage of allowing the runtime system to immediately react to a thread's changing behaviour. However, since any method invocation can change a thread's behaviour characteristics, the overhead involved in re-evaluating a thread's per-core-type cost after every behaviour change could become prohibitively expensive.

Evaluating a thread's cost at fixed time periods (e.g. at timer tick operations) limits the overhead of cost evaluation. However, the runtime system will be slower to react to changes in a thread's behaviour, and may not produce an accurate cost if it samples a thread's behaviour characteristics too infrequently, or at a time when the thread is behaving uncharacteristically.

Both approaches were implemented in Hera-JVM to evaluate these trade-offs. Sections 6.3.1.1 and 6.3.1.2 describe the implementations of each approach.

#### 6.3.1.1 Immediate Thread Cost Evaluation

To implement immediate thread cost evaluation, assembly code was added to the thread behaviour tracking code, described in Section 6.1.2. This code initiates a trap to a thread cost evaluation routine if the thread's behaviour characteristics have changed.

Given that every method invocation could change a thread's behaviour characteristics, thus triggering a thread cost evaluation, this cost evaluation routine was made as lightweight as possible. Therefore, this routine was also written in assembly code. As such, it was not feasible to implement the full cost function described in Section 6.2.

Instead, this routine just evaluates the raw per-core-type costs using Equation 6.1, without incorporating history, hysteresis or trend tracking into the migration decision (i.e., this implementation essentially fixes the parameters $\alpha$, $\beta$ and $\gamma$ to zero).

Using an exponential moving average to incorporate the history of a thread's behaviour into its cost function would, in any case, not provide a realistic smoothing effect for this immediate thread cost evaluation approach. The exponential moving average function assumes that each sample is equally significant when sampled, but becomes exponentially less significant each time a newer sample arrives. However, by evaluating (and thus sampling) a thread's cost whenever its behaviour changes, samples should not be treated equally. For example, a thread may inherit a behaviour characteristic for an insignificantly short time, e.g., when calling a short method. This sample should be treated with less significance than a longer lived characteristic; however, the immediate cost evaluation approach treats each behaviour change equally.

There are other possible approaches for incorporating history into this immediate cost evaluation approach (such as measuring the time between behaviour characteristic changes); however, these would be much more expensive to evaluate. Therefore, the immediate cost evaluation approach bases its migration decisions on only the thread's current behaviour characteristics. This is appropriate, given that this approach focuses on providing immediate reactions to thread behaviour changes.

#### 6.3.1.2 Thread Cost Evaluation at Timer Ticks

An alternative to evaluating a thread's cost every time its behaviour characteristics change is to sample the thread's behaviour characteristics at fixed time intervals and build up expected per-core-type costs, based upon these samples. There are two advantages to this approach. Firstly, the thread cost evaluation has a fixed overhead, rather than being dependent upon the placement of behaviour characteristic annotations. Secondly, the sampling process places less emphasis on short-lived behaviour characteristics over time, since they will have a lower probability of being sampled, given their short lifetime. The disadvantage is that the runtime system can only make migration decisions at these fixed sampling times, which may cause a delay in moving a thread to a more appropriate core type, or cause it to miss migration opportunities.

Hera-JVM already interrupts the running thread (by default, every 10ms) to make scheduling decisions and decide whether execution should switch to a different Java

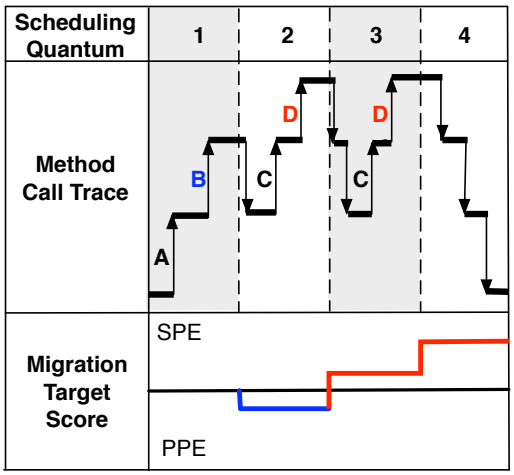



Figure 6.2: An example of thread behaviour sampling at timer ticks. Method B has behaviour characteristics which suit execution on the PPE core type, method D has behaviour characteristics that suit execution on the SPE, and all other methods do not have any behaviour characteristics.

thread[1]. To implement the sampling-based cost evaluation approach, this scheduling code was augmented to calculate the thread's per-core-type costs whenever it is called. The current thread's behaviour characteristics are sampled and then fed through the cost function migration policy. An example of this process is shown in Figure 6.2. This scheduling code is written in Java and runs relatively infrequently; therefore, the full cost function, described in Section 6.2, was implemented for this approach.

### 6.3.2 Triggering Thread Migration

Once the runtime system has decided that a thread will perform better on a different core type, it must trigger the thread to migrate at an appropriate point in its execution. Hera-JVM's migration mechanism (described in Section 5.4.1) places a number of constraints on this process. Firstly, migration can only occur at method invocation. Secondly, since stack frames created by the different core types are incompatible with each other, a thread must return to its original core type when it returns from the method which was migrated.

[1]The mechanism used to interrupt the current thread and invoke the scheduler code is described in Section 5.3.5.

This places conflicting demands on the migration triggering mechanism. Ideally, a thread should be migrated at the next available migration point (i.e. the next method call). However, if this method returns very quickly, the thread will be forced back to the original core type with little performance benefit and perhaps a performance loss, due to the migration overheads. It may be more appropriate to wait for a longer lived method before migrating.

Two migration triggering mechanisms are therefore investigated: triggering a migration on the next available method call; and targeting a longer lived method, further up the stack trace, for migration the next time it is executed by the thread.

#### 6.3.2.1 Migrate On Next Method Call

A per-core flag is added to signal that this core should migrate its current thread at the first opportunity. Code in the prologue of migratable methods checks this flag and, if it is set, initiates a migration by trapping to an out-of-line migration support routine. As with the thread's behaviour bitmap, this flag is placed in a reserved location, in local memory on the SPE cores, to avoid unnecessary memory accesses and software caching overheads. The overhead involved in checking this flag is minimal — two machine instructions on both architectures.

Some methods should never be migrated. For example, the runtime system runs scheduling code on each core to control thread switching. This code must run on the core it is controlling to function correctly, and therefore should not be migrated. Hera-JVM, therefore, prevents a thread from migrating when it is executing runtime system code by not including the migration flag check in any of the runtime system's methods. This means that the runtime system can trigger a thread to migrate during the scheduling code (e.g. when evaluating thread costs using the sampling-based approach in Section 6.3.1.2) and the thread will only be migrated when control returns to application or Java library code.

#### 6.3.2.2 Targeted Migration

Rather than migrate immediately, the targeted migration approach attempts to trigger migration on a method that is sufficiently long-lived to gain some benefit from the migration. The runtime system does not know how long future method calls are likely

to run, nor does it know if these future methods will exhibit the same behaviour characteristics as have triggered the thread's migration. However, it is possible to estimate the time at which each of the methods on the thread's stack was called. The runtime system can then select a method further up the thread's stack-trace which is likely to be both long-lived and to exhibit similar behaviour characteristics on future calls. This method can then be targeted for migration the next time it is called by the application. While this will not provide any immediate benefit to the thread, it should enable the runtime system to accumulate knowledge of which application methods are appropriate to migrate, thus improving the performance of subsequent calls to these methods.

To enable the runtime system to estimate the length of time for which a method has been executing, a slot is reserved in each method's stack-frame to store a representation of the time at which the method was invoked. However, calculating the current time at each method invocation would be an unnecessary overhead. An estimate of time's progression with an appropriate granularity is sufficient. Given that a migration is more expensive than a thread switch, it is appropriate to target only those methods which take at least one timer tick to complete for migration. Therefore, it is sufficient to estimate time with the granularity of a timer tick. As such, the runtime scheduler maintains a scheduling quantum counter, which is incremented at each timer tick. When a method is invoked, its prologue pushes the current value of this counter into the reserved slot of the method's stack-frame.

When the runtime system decides that a thread should be migrated, it selects a long-lived method by scanning through the thread's stack trace until it finds a method that was invoked one or more scheduling quanta before the current scheduling decision. Since this method has been executing for at least one scheduling quantum, it is likely to do so again the next time it is called, and thus is a good candidate for migration. Figure 6.3 shows an example of a method being selected for targeted migration.

Once a method has been targeted for migration, the runtime system must ensure that the thread migrates to the appropriate core type whenever this method is called. One possible approach for signalling this migration would be to set a flag which is checked when the method is invoked, as with the "migrate on next method call" approach. However, since a particular method is being targeted for migration, an individual flag would be required for each potentially migratable method. Storing this number of flags in the SPEs' local memory would impose an unnecessary burden on

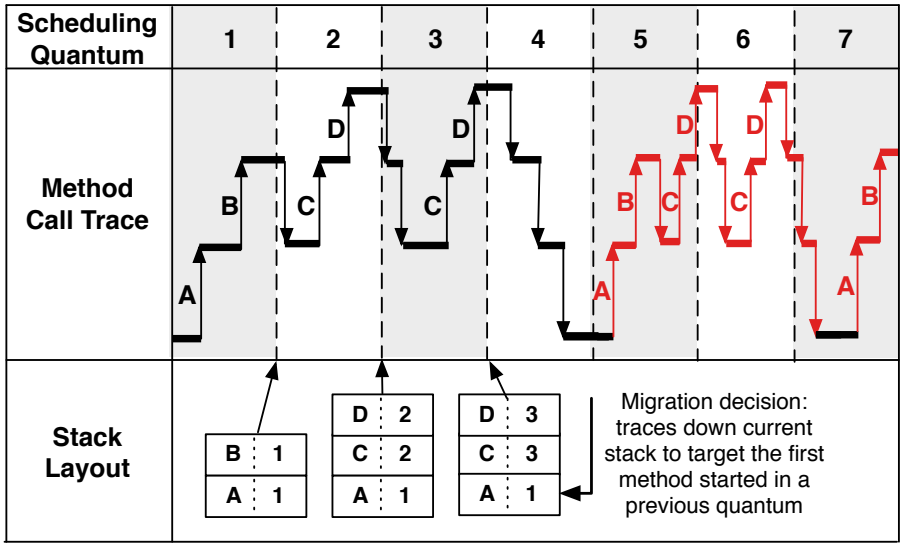



Figure 6.3: A targeted migration example. At the end of the third scheduling quantum, a migration decision is made. Method A is found to be the first method on the thread's stack, at that point, which was invoked in a previous scheduling quantum. Therefore, method A is targeted for migration and executes on the other core type (shown in red) on future invocations.

this constrained resource, while caching a flag when it is required would be unwieldy and cause additional software caching overheads at every method call. Even on the PPE core, this would be a cumbersome approach.

Instead, Hera-JVM employs runtime binary code re-writing to patch targeted methods, such that they trap to the migration support routine whenever they are invoked. Potentially migratable methods have a short[1] machine code sequence added to their prologue, which sets up a trap to the migration support routine. However, the branch instruction, which actually causes the trap, is substituted with a `nop` instruction. By default, therefore, the method will not cause a migration. To target this method for migration, the runtime system simply replaces this nop instruction in the method's compiled code with the branch instruction necessary to complete the trap operation. Since only a single instruction is being replaced, this can be done atomically, thus ensuring any other threads executing this code do not run inconsistent or invalid code during the update.

[1]Four machine instructions on the PPE core, three instructions on the SPE.

| | Thread Costing | Migration Mechanism |
|---|---|---|
| **AtAnnotation** | Immediate | Next Method |
| **AfterSched** | Sample Based | Next Method |
| **Targeted** | Sample Based | Targeted |

Table 6.3: Behaviour-based migration strategies implemented in Hera-JVM.

This approach changes the semantics of targeted migration somewhat, in that any thread which invokes this method will now be migrated, not just the thread which triggered the migration. This seems reasonable, however, given that a method is likely to perform similarly, even if called by a different thread.

### 6.3.3 Combining Thread Costing and Migration Triggering

The above thread costing and migration triggering mechanisms were combined to create three different implementations of Hera-JVM with support for behaviour-based thread migration. These behaviour-based migration strategies are outlined in Table 6.3.

These strategies progressively trade off immediacy in reacting to a thread's behaviour changes, with more informed (and expensive) decisions on when and where to migrate. The **AtAnnotation** strategy will immediately migrate on any method which has behaviour characteristic annotations that suggest the thread may perform better on a different core type. The **AfterSched** strategy samples a thread's behaviour characteristics at every scheduler timer tick, migrating, if appropriate, at the first opportunity after the scheduling code returns. Finally, the **Targeted** strategy also samples a thread's behaviour at timer ticks, but targets a long-lived method on the thread's stack for migration the next time it is called, rather than migrating immediately. The combination of immediate thread costing with targeted migration was not implemented because the overheads involved in selecting a method for targeted migration are too high to be executed each time a thread's behaviour characteristics change.

## 6.4 Experimental Analysis

In this section, the efficacy of employing behaviour characteristic annotations for automated exploitation of heterogeneous processing cores is investigated. The three strategies implemented by Hera-JVM (**AtAnnotation**, **AfterSched** and **Targeted**) are

compared and contrasted under a variety of synthetic benchmarks to discover their performance under a variety of different situations. The effectiveness of history, hysteresis and trend tracking are also investigated by varying the $\alpha$, $\beta$ and $\gamma$ parameters of the cost function. Finally, some real world benchmarks are annotated with behaviour characteristics to investigate the appropriateness of this approach in a realistic setting.

### 6.4.1 Experimental Setup

As with the experiments in the previous chapter, all the experiments are performed on a Playstation 3, with 256MB of RAM, running Fedora Linux 9. The baseline (non-optimising) compiler was used to compile both PPE and SPE machine code. Each experiment was repeated ten times, with the average being reported and the standard deviation between these runs shown using error bars. Unless otherwise noted, the thread behaviour cost function uses the default values of $\alpha$, $\beta$ and $\gamma$, as shown in Table 6.2.

### 6.4.2 Two Phase Synthetic Benchmark

The first synthetic benchmark is a simple program that exhibits two distinct phases of behaviour during its execution. One phase is heavily arithmetic, and therefore suited to the SPE cores; the other phase performs many object accesses, and therefore runs faster on the PPE core. This *two phase* benchmark is used to investigate the basic trade-offs between the different behaviour-based migration strategies, as well as exploring the effect of annotation placement and behaviour phase length.

Listing 6.1 outlines the code used for this benchmark program. The type of application that this benchmark attempts to model is one which performs some calculation over a given data-set (the `arithPhase()` method), then performs data update operations on this data-set (the `objAccessPhase()` method). This process is repeated over multiple data-sets. The actual calculation, or update operation, for each element of the data-set is performed by a separate method (`doArith` and `doObjAccess`, respectively).

The Arithmetic and Object Access workloads are tailored so that the SPE core executes the `doArith()` method 2.4 times as quickly as the PPE core, and conversely, the PPE core executes the `doObjAccess()` method 2.4 times as quickly as the SPE core. They are also tailored such that their execution times are roughly equal when run on their most appropriate core type. Table 6.4 shows the execution times of these

```
1  class TwoPhase {
2    void main() {
3      loop x times ... {
4        arithPhase();
5        objAccessPhase();
6    }}
7
8    void arithPhase() {
9      loop y times ... {
10       doArith();
11   }}
12
13   void objAccessPhase() {
14     loop y times ... {
15       doObjAccess();
16   }}
17
18   void doArith() {   Arithmetic workload ...   }
19   void doObjAccess() {   Object access workload ...   }
20 }
```

Listing 6.1: *Two phase* synthetic benchmark psuedo-code.

| | `doArith` ($\mu$s) | `doObjAccess` ($\mu$s) |
|---|---|---|
| PPE core | 61.75 ($\sigma = 0.684$) | 24.71($\sigma = 0.661$) |
| SPE core | 25.81 ($\sigma = 0.006$) | 59.33 ($\sigma = 0.007$) |

Table 6.4: Average execution time of the workload methods on each core type.

methods on the PPE and SPE core types. With perfect thread phase placement, ignoring overheads, the program should therefore run 1.7 times faster[1] than it would if run on a single core exclusively.

#### 6.4.2.1 Behaviour Annotation Placement

This benchmark clearly has an arithmetic phase and an object access phase, therefore these phases should be labelled using the `@ArithmeticCode` and `@ObjectAccessCode` appropriately. However, the choice of which methods are annotated with these behaviour characteristics will affect each of the migration decision strategies differently.

[1] If running on a single core, one of the phases will run at the faster speed, whereas placing each phase on the most appropriate core will result in both phases running at the faster speed. Therefore, only one of the phases will exhibit a speed-up relative to the single core case, leading to a total speed-up of $(1 + 2.4)/2 = 1.7$.

| | Workload method annotated | | Phase change method annotated | |
|---|---|---|---|---|
| | Time (s) | Migrations | Time (s) | Migrations |
| PPE | 17.69 (0.31) | 0(0) | 17.69 (0.3) | 0(0) |
| SPE | 17.21 (0.003) | 1 (0) | 17.21 (0.003) | 1 (0) |
| Manual | 10.42 (0.16) | 20 (0) | 10.42 (0.16) | 20 (0) |
| AtAnnotation | 278.7 (12.4) | 200,000 (0) | 10.66 (0.15) | 20 (0) |
| AfterSched | 52.05 (3.3) | 36,651 (5743) | 50.62 (2.21) | 42,373 (3301) |
| Targeted | 11.23 (0.09) | 19 (0) | 11.43 (0.16) | 37 (0) |

Table 6.5: Execution time and migration count for the *two phase* benchmark under different annotation placements (standard deviation in brackets).

The most intuitive approach is to annotate those methods which perform a workload that has a particular behavioural characteristic, with the appropriate annotation. In this case, the `doArith` and `doObjAccess` methods (lines 18 and 19 of Listing 6.1) are annotated with `@ArithmeticCode` and `@ObjectAccessCode` respectively. However, these methods are relatively short-lived, therefore, if the runtime system selects these annotated methods for migration, the overhead involved in migrating every time they are called would eradicate any potential performance improvement.

A developer who is aware of this fact could, instead, choose to annotate the longer lived methods, where the thread changes phase to a different behaviour. For this *two phase* benchmark, this would mean annotating the `arithPhase` and `objAccessPhase` methods (lines 8 and 13 of Listing 6.1). This is a more appropriate point for thread migration. However, this annotation placement is not as intuitive for the developer; the methods being annotated may not exhibit any of the behaviour characteristics with which they are being annotated, they just happen to call other methods which do.

The benchmark was annotated using both of these annotation placements, to investigate their effect. For these experiments, $x$ in line 3 of Listing 6.1 was set to 20, and $y$ on lines 9 and 14 was set to 10,000. This simulates a program working on 20 data-sets, each of which contains 10,000 elements.

Table 6.5 shows the execution times of this benchmark with both annotation placements, when running under the different migration decision strategies. The speedup, relative to the benchmark being run completely on the PPE core, is reported in Figure 6.4. As expected, running the program solely on the SPE core does not significantly improve performance. The **Manual** results show the maximum speedup of 1.7x, when

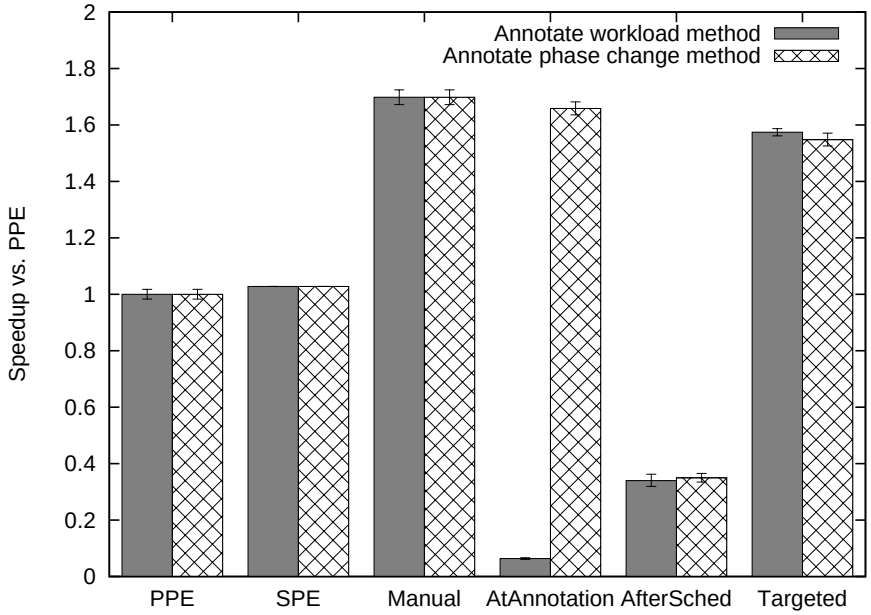



Figure 6.4: Speedup of the *two phase* benchmark as annotation placement is varied.

each phase is manually run on the most appropriate core type, without any migration or behaviour tracking overheads.

The **AtAnnotation** strategy almost reaches this maximum speedup (the migration and thread behaviour tracking overhead is just 2.25%), when the phase change methods are annotated. However, when the more intuitive approach of annotating the workload method is used, this strategy performs very badly, imposing a significant slowdown on the benchmark rather than improving performance. The overhead involved in migrating the short-lived workload methods, every time they are called, leads to this marked slowdown.

By sampling the thread's behaviour characteristics on timer ticks, rather than reacting immediately to annotations, the **AfterSched** strategy is much less affected by annotation placement. However, since this strategy simply migrates at the first opportunity, it is more likely to migrate the commonly called, short-lived methods than the less frequently called long-lived ones. The **AfterSched** strategy, therefore, does not improve performance in this benchmark, due to the migration overheads it causes.

The **Targeted** strategy attempts to overcome this deficiency by selecting relatively

long-lived methods for migration. This strategy works well, providing up to a 1.6x speedup for this benchmark, relative to being run entirely on the PPE core. It is also less affected by annotation placement than the **AtAnnotation** strategy - in this case performing slightly better with the more intuitive workload method annotation placement. The reason the **Targeted** strategy does not reach the theoretical maximum speedup is not purely down to overheads, but also due to the fact that a phase must be sampled before being migrated. Therefore, the first time a thread executes a phase which is better suited to a different core type, the first migration opportunity is missed while the phase is being sampled.

#### 6.4.2.2 Length of Behaviour Phases

The execution length of phases of behaviour in a thread influence whether or not these behaviour phases should be migrated. If the phase is short, the overheads of migration might outweigh the benefit of running that phase on a different core type. In this experiment, the effect of phase length of the *two phase* benchmark (annotated at the phase change methods) is investigated for the different migration strategies. The length of the two phases in this benchmark was varied by adjusting the ratio between $x$ and $y$ in Listing 6.1. As the length of a phase ($y$) was reduced, the number of phases ($x$) was increased proportionately, such that the same amount of work was being done ($x * y = 200000$ in all runs).

Figure 6.5 shows the speedup obtained by each migration strategy as the phase length is varied. The **AtAnnotation** strategy performs well when the length of the behaviour phase is greater than 1ms. However, by blindly migrating phases without taking their execution time into account, it incurs a significant slowdown when the phase length is less than 1ms, due to the thread migration overheads.

The **Targeted** strategy does not incur a significant slowdown when the phase length is short, because it does not target these short phases for migration. Once the phase length is long enough to overcome the migration overheads, the **Targeted** strategy has performance close to the **AtAnnotation** strategy. However, this speedup tails off as the phase length increases. This tail-off is not due to the length of the phase, but instead, due to the phases being repeated less often (the top x-axis of Figure 6.5 shows the number of phase repetitions for each run). Since the **Targeted** approach misses

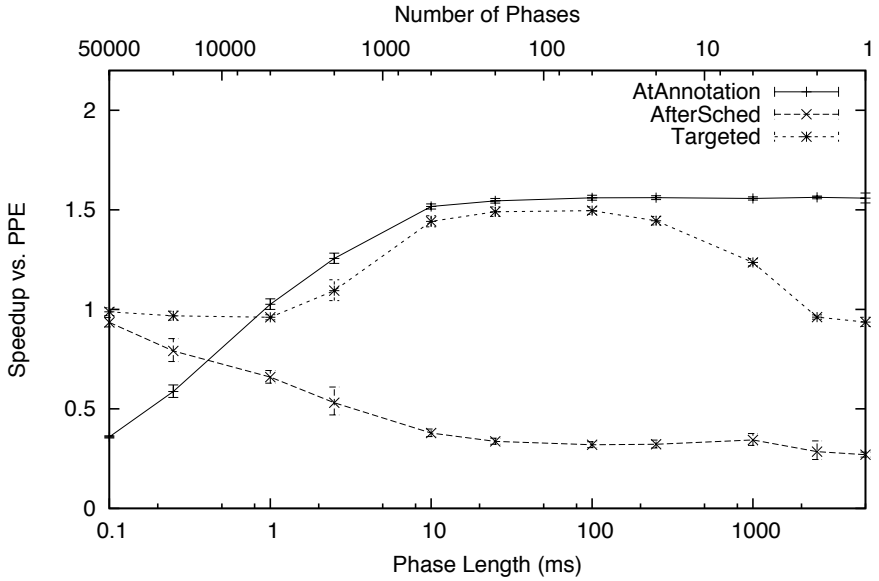



Figure 6.5: Speedup of the *two phase* benchmark as the phase length is varied. The *Phase Length* axis measures the average execution time of either phase when run on its most appropriate core type.

migration of the first occurrence of a particular behavioural phase, it relies on the phase being repeated a reasonable number of times to be effective.

The **AfterSched** strategy continues to perform badly here. As the phase length decreases, its performance does climb towards the reference performance, of running completely on the PPE core. However, this is only because the history, hysteresis and trend tracking element of the cost function — as intended — decrease the number of migrations taken as the phase length decreases.

### 6.4.3 XML Parsing Synthetic Benchmark

To investigate the influence of incorporating history, hysteresis and trend tracking into the thread behaviour cost function, a more complex and realistic benchmark was devised. This benchmark is based on a real world scenario — parsing of encrypted and compressed XML files. The workload for this benchmark is the complete set of the 36 Shakespeare plays that comprise the First Folio, marked up in XML[1] which are compressed and encrypted, with one file per play. The benchmark reads each of these

[1]The XML files can be found at: `http://www.cafeconleche.org/examples/shakespeare/`

files and decrypts and uncompresses the XML data. The XML is parsed to calculate the total number of lines spoken by each character in that play. Once the XML has been parsed, the mean number of lines spoken per character, and the standard deviation around this mean, is calculated for the play. This information, along with the count of lines spoken by each individual character, is inserted into the end of the XML document, which is then re-compressed, encrypted and written to a file on disk.

Whilst the benchmark itself is synthetic, parsing of encrypted and compressed XML files is a relatively common operation in many Web Service applications. Real world libraries were employed in the creation of the *XML parsing* benchmark to increase confidence that these results are applicable to real world applications. Encryption and decryption is performed using the triple DES symmetric key encryption algorithm (Coppersmith *et al.*, 1996), as implemented by the Java Cryptography Extension (JCE) library[1], which is part of the standard Java Library. The Lempel-Ziv-Welsh (LZW) algorithm (Welch, 1984) is employed for compression and decompression. Finally, the benchmark parses the XML using the SAX (Simple API for XML) parser in the Apache Xerces 2.9.1 library[2].

The benchmark has three main behaviour phases: the encryption and decryption routines, which perform best when run on the SPE cores; the compression and decompression routines, which perform approximately equally well on either core type; and the XML parsing, which runs faster on the PPE core. However, these phases, and the most appropriate core type for each, would not necessarily be clear to a non-specialist programmer. Therefore, this benchmark was annotated in the manner expected from a relatively competent, but non-specialist programmer: methods likely to significantly influence execution time were annotated with the behaviour characteristics of that method, without taking into account the phase to which the method may belong. In all, 14 methods were annotated with behaviour annotations.

#### 6.4.3.1 Exploration of the Cost Function Parameter Space

There are three independent variables in the migration decision cost function — the $\alpha$, $\beta$ and $\gamma$ parameters, controlling history, hysteresis and trend tracking, respectively. In this section, these parameters are varied to investigate the effectiveness of incorporating

[1] `http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html`
[2] `http://xerces.apache.org/xerces-j/`

| | Parameter set to zero | Parameter set to one |
|---|---|---|
| $\alpha$ | Only the current behaviour cost of a thread influences the migration decision. | Only past behaviour influences the cost function, therefore a thread will never migrate. |
| $\beta$ | The migration threshold is a *target score* $T = 0$ (i.e. no hysteresis). | The migration threshold is $T = C'_{currCore}$ (Given the definition of $T$, it can never reach this value and so migration never occurs.). |
| $\gamma$ | Trending direction of the target score is not taken into account. | Target score must be trending beyond the migration threshold to trigger a migration |

Table 6.6: Migration policy parameters.

each of these features into the cost function, and the degree to which they should influence the migration decision. The interaction between these parameters is also explored.

Of the three migration strategies employed by Hera-JVM, the **AtAnnotation** strategy does not incorporate history, hysteresis or trend tracking into its cost function, and the **AfterSched** performs badly under all scenarios. Therefore, the effect of cost function parameters are explored under only the **Targeted** migration strategy.

The range of all three parameters is from zero to one. Setting a parameter to zero *turns off* the associated part of the cost function. For example, when $\alpha = 0$, only the thread's current behaviour cost is incorporated into the cost function; the thread's previous behaviour is not taken into account. When $\alpha = 1$, the cost function only takes a thread's past behaviour into account, thus no new behaviour cost samples are incorporated into the cost function's output and the thread's cost value will never move from its initial value of zero. Table 6.6 describes the effect of setting each parameter to its minimum and maximum values.

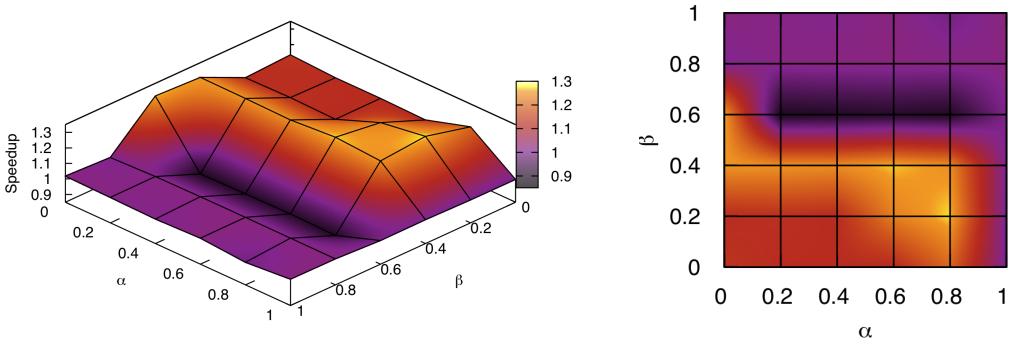
In the following experiments, the value of each parameter was varied in 0.2 increments between zero and one, and the XML Parsing benchmark was run under the **Targeted** migration strategy for each combination of parameter values. Figures 6.6 to 6.11 show the speedup obtained by this migration strategy, compared with running the benchmark entirely on the PPE core, for each combination of parameter values. These results are presented as both a 3D projection and a heat map, showing the effect

of varying $\alpha$ and $\beta$ on the speedup gained by the migration strategy, with $\gamma$ varied between figures. All figures have the same scale, with white representing the best possible speedup, down through orange, representing a modest speedup, then purple representing no speedup and finally black representing a slowdown.

It can be observed, for all values of $\gamma$, that when $\alpha = 1$ or $\beta \geq 0.8$, no speedup is observed. This is because, when these parameters are set so high, the target score never passes the migration threshold, and thus, the thread never migrates from the PPE core.

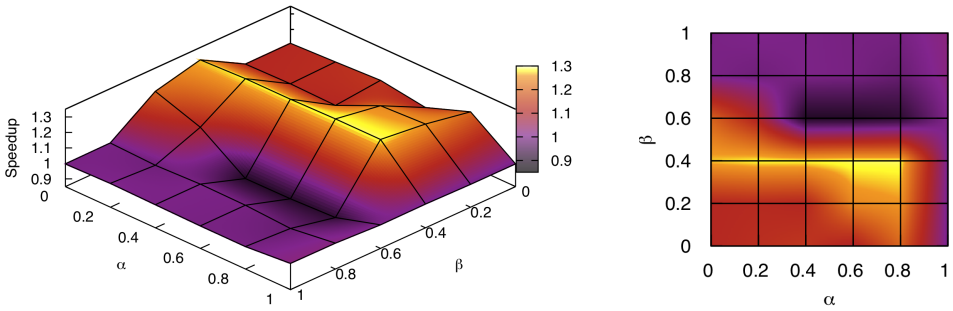
Across all values of $\gamma$, there is a relatively distinct *L-shaped ridge* of good speedup results moving along the $\beta = 0.4$ line, then down the $\alpha = 0.8$ line. The best speedup results generally occur in the corner of this *L-shaped ridge* ($0.6 \leq \alpha \leq 0.8$ and $0.2 \leq \beta \leq 0.4$). Behind this ridge (i.e. when $\alpha < 0.8$ and $\beta < 0.4$), the migration strategy does not perform as well, although it does still provide a modest speedup.

This supports the hypothesis that incorporating a certain amount of history (controlled by $\alpha$) and hysteresis ($\beta$) into the cost function does, indeed, improve the performance of the migration strategy. The reason for this becomes clear when considering an example routine in the XML Parsing benchmark — the calculation of the average and standard deviation of the number of lines spoken per character. This routine is an arithmetic operation, and is annotated as such. However, its execution time is too short for it to be a useful migration target. Whilst the **Targeted** migration strategy will prevent this short method from being migrated on its own, it can still influence a migration decision if the scheduler samples the behaviour characteristics of the thread whilst it is running. If the cost function does not incorporate the thread's behaviour history or hysteresis into its migration decision, sampling this method's arithmetic behaviour characteristic will immediately push the thread's score past the migration threshold. The **Targeted** migration strategy will try to find a longer-lived method, further up the thread's stack trace, to target for migration to the SPE core. In this case, the main XML parsing routine in the Xerces library is targeted — a method which performs much better on the PPE core. The mistaken targeting of this method for migration is likely to be corrected when it is next run (on the SPE core), but by then the performance has already suffered, due to this routine having been run on an inappropriate core type.



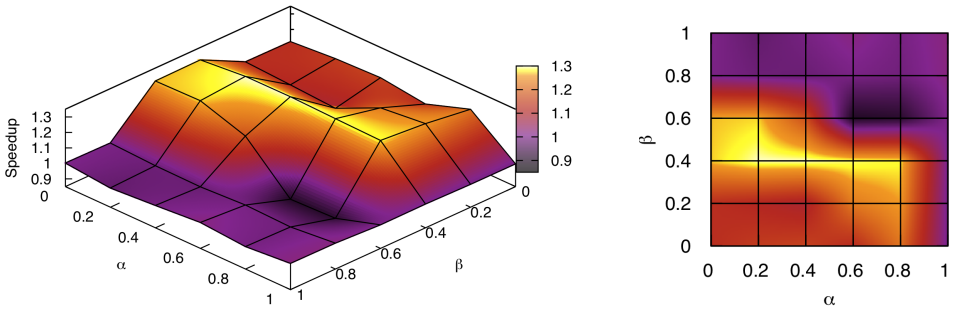

(a) 3D Projection (b) Heat Map

Figure 6.6: Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 0$.



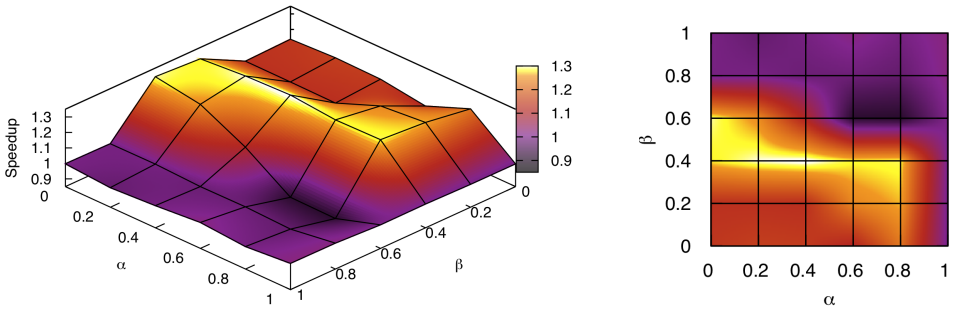

(a) 3D Projection (b) Heat Map

Figure 6.7: Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 0.2$.



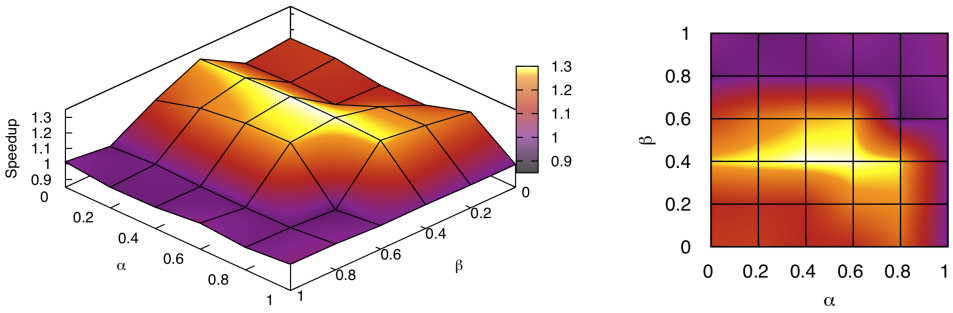

(a) 3D Projection (b) Heat Map

Figure 6.8: Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 0.4$.



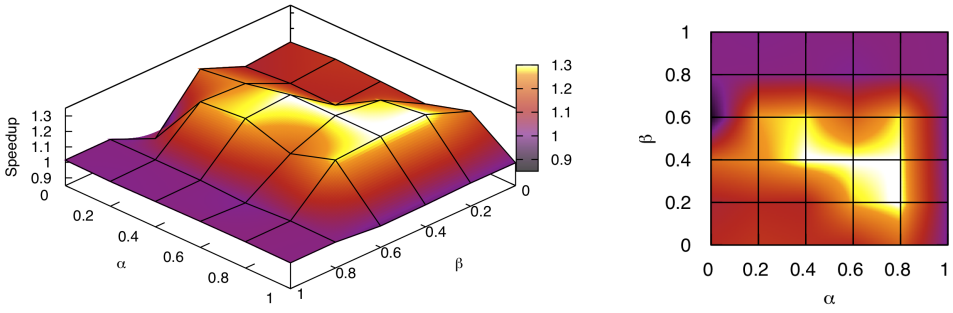

(a) 3D Projection (b) Heat Map

Figure 6.9: Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 0.6$.



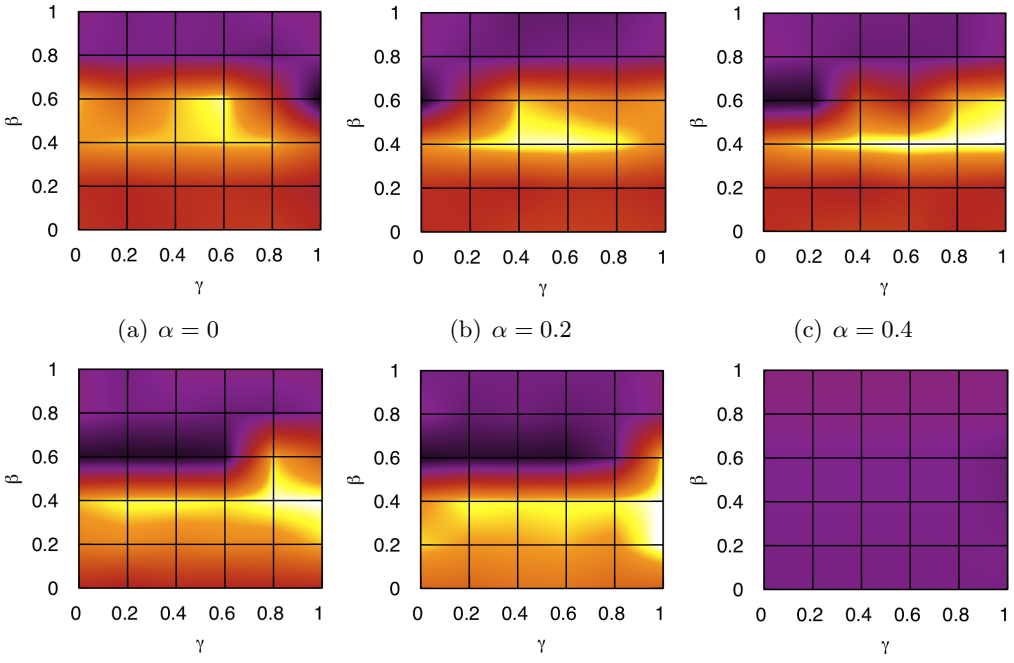

(a) 3D Projection (b) Heat Map

Figure 6.10: Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 0.8$.




(a) 3D Projection (b) Heat Map

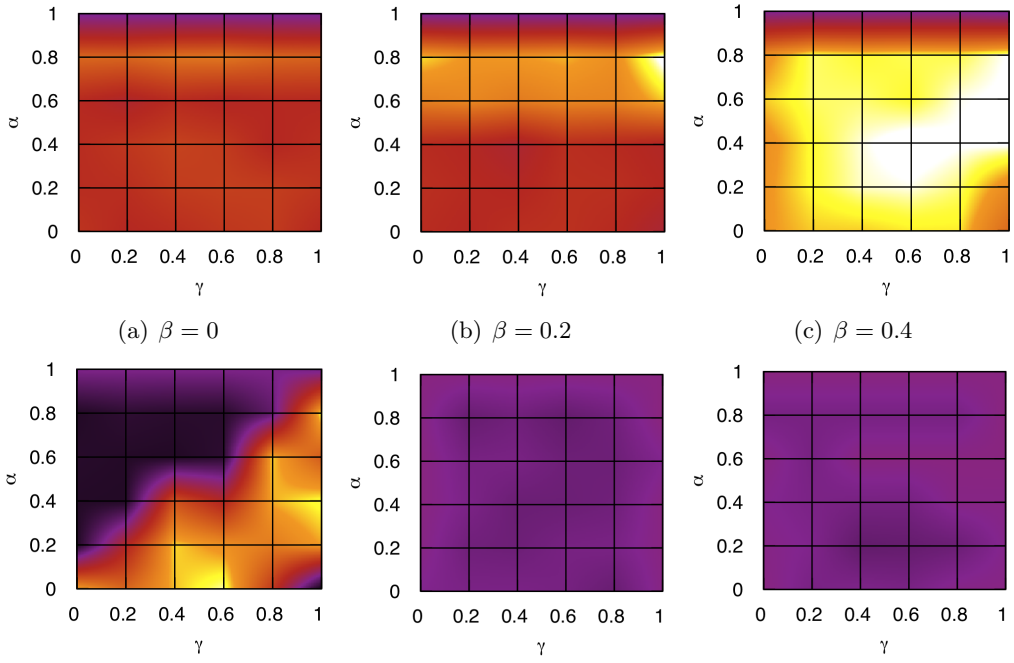
Figure 6.11: Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 1$.




(a) $\alpha = 0$ (b) $\alpha = 0.2$ (c) $\alpha = 0.4$

(d) $\alpha = 0.6$ (e) $\alpha = 0.8$ (f) $\alpha = 1$

Figure 6.12: Comparing the interaction between $\beta$ and $\gamma$.




(a) $\beta = 0$ (b) $\beta = 0.2$ (c) $\beta = 0.4$

(d) $\beta = 0.6$ (e) $\beta = 0.8$ (f) $\beta = 1$

Figure 6.13: Comparing the interaction between $\alpha$ and $\gamma$.

Incorporating the history of a thread's past behaviour into the cost function helps to mitigate this scenario. The influence of a single *rogue* sample is limited, because the thread's cost is calculated based upon multiple sample readings. Similarly, hysteresis can limit the above scenario by increasing the threshold required to trigger migration. The L-shape in these results suggests that both history and hysteresis can independently provide similar improvements in performance. The fact that the best speedups occur at the corner of the L suggests that these parameters complement each other, and both history and hysteresis are useful features to incorporate into the cost function.

The influence of trend tracking ($\gamma$) can be inferred by observing the evolution in the shape of the graphs, from Figure 6.6 to Figure 6.11, as $\gamma$ is increased. Whilst trend tracking slightly improves the best possible speedup result, its main effect is to broaden out the ridge of good speedup results across a greater range of $\alpha$ and $\beta$ parameter values. For example, a distinct *trough* of slowdowns occurs along the $\beta = 0.6$ line in Figure 6.6. This trough gradually disappears as $\gamma$ is increased, until in Figure 6.11 (where $\gamma = 1$), it has disappeared entirely. This suggests that trend tracking reduces the migration strategy's sensitivity to less than optimal values of the $\alpha$ or $\beta$ parameters.

Figures 6.12 and 6.13 show the same data, but with the $\alpha$ and $\beta$ parameters held constant in each graph. This enables the interaction between $\alpha$ and $\gamma$, and $\beta$ and $\gamma$ parameters to be seen more clearly. Figures 6.13(c) and 6.13(d) show that the $\alpha$ and $\gamma$ parameters are correlated to some extent. As more historical information is taken into account by the cost function (as $\alpha$ increases), a corresponding increase in trend tracking ($\gamma$) is required to provide stability against inappropriate migrations caused by out of date information.

Figure 6.14 summarises these results by showing the best results obtained when each of these features are enabled or disabled in the cost function. When all three features are enabled, the **Targeted** migration strategy performs almost as well as manually selecting the most appropriate core type for each phase. When only history is incorporated into the cost function, the best possible speedup drops from almost 1.3x to 1.2x. The best result when only hysteresis was incorporated into the cost function is slightly better, with a 1.27x speedup. However, the performance variance of this result is higher than when all three features are enabled, suggesting that the other features help improve stability. Finally, incorporating only trend tracking into the cost function does not significantly improve upon using only the raw behaviour characteristic costs

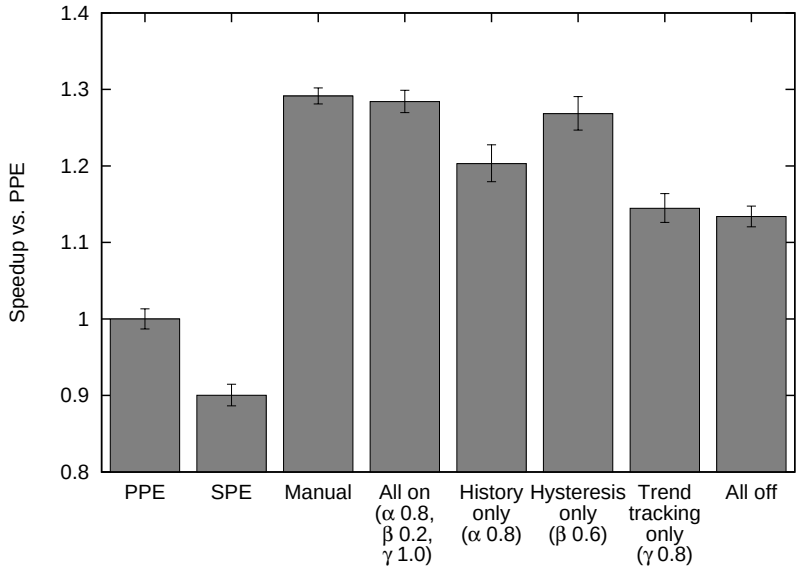



Figure 6.14: Results for the targeted migration strategy when history, hysteresis and trend tracking are enabled and disabled in the cost function.

(All off), with both approaches having a speedup of around 1.12x. In all, these results show that incorporating history, hysteresis and trend tracking into the cost function improves the performance of the **Targeted** migration strategy.

The most effective combination of these parameters ($\alpha = 0.8$, $\beta = 0.2$ and $\gamma = 1$) is used for all the remaining experiments in this chapter.

Finally, the **Targeted** migration strategy is compared with the **AtAnnotation** migration strategy in Figure 6.15. The **AtAnnotation** strategy again performs very poorly, due to the high number of short method migrations. When the behaviour annotations were removed from three frequently called methods of the XML Parsing benchmark (**AtAnnotation Improved**), this strategy comes close to reaching the **Targeted** migration strategy's performance. However, the **Targeted** migration strategy has better performance and, with its reduced susceptibility to annotation placement issues, is the migration strategy of choice.

### 6.4.4 Real World Benchmarks

In order to validate the use of behaviour characteristic annotations in a realistic setting, a subset of the real world benchmarks, first presented in Section 5.5.3, were annotated

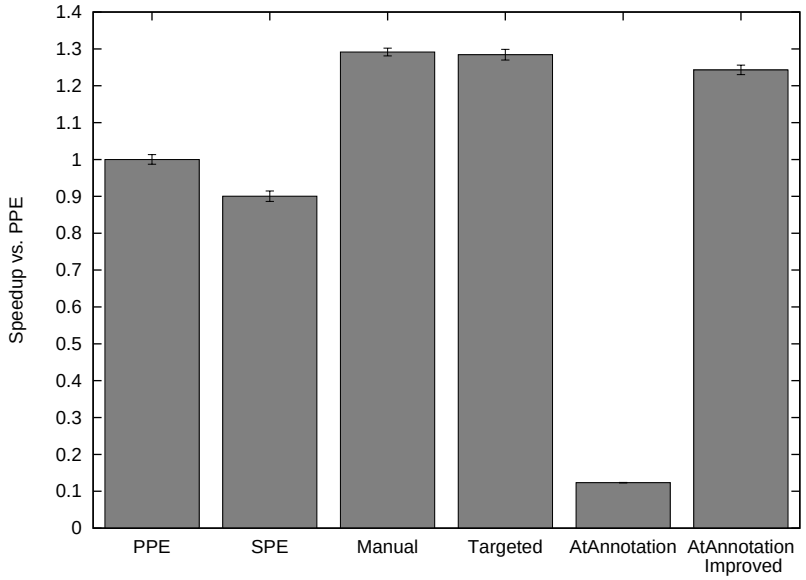



Figure 6.15: Speedup for the XML Parsing benchmark under different migration strategies.

with their behaviour characteristics. Annotating these benchmarks requires gaining an understanding of their code to be able to recognise the portions of the benchmark's code are likely to have a significant influence on execution time and characterise the behaviour of these execution phases. Therefore, to keep this experiment tractable, a representative subset of these benchmarks was selected for annotation: the Java Grande mol_dyn and ray_trace benchmarks; the SpecJVM 2008 sor, mpegaudio and compress benchmarks; and the Dacapo antlr benchmarks. These benchmarks were annotated, then executed, under Hera-JVM, with the behaviour characteristic annotations being used to influence migration decisions between the PPE and SPE cores.

To compare the effect of annotation placement on performance, each benchmark was annotated in two different ways. In the first approach, called **TagThread**, the benchmark's main thread is annotated with the behaviour characteristics that are expected to apply to the benchmark throughout its execution.

For the second approach, called **TagMethods**, the source code of each benchmark was examined to discover the set of methods that are likely to contribute significantly to the benchmark's execution time (i.e., computational and data-processing methods,

| Benchmark | TagThread | | TagMethods | |
|---|---|---|---|---|
| | Files Modified | Annotations | Files Modified | Annotations |
| JG: mol_dyn | 1 | 1 | 3 | 6 |
| JG: ray_trace | 1 | 2 | 4 | 28 |
| SPEC: sor | 1 | 1 | 1 | 2 |
| SPEC: mpegaudio | 1 | 2 | 4 | 23 |
| SPEC: compress | 1 | 2 | 1 | 11 |
| DACAPO: antlr | 1 | 1 | 17 | 37 |

Table 6.7: Behaviour characteristic annotations added to each benchmark.

as opposed to set-up, co-ordination and initialisation methods). These methods were then annotated with the characteristics applicable to them, as discovered through code inspection.

The **TagMethods** approach is a more accurate representation of the expected use-case of behaviour characteristic annotations, with methods being individually annotated to describe their behaviour as the program is developed. The **TagThread** approach approximates a profiling tool which profiles the behaviour of a program's threads and then automatically inserts appropriate behaviour characteristic annotations into the program's bytecode. Table 6.7 shows the number of files modified and the number of annotations added to each benchmark for each of these approaches.

Figure 6.16 shows the performance of each of these benchmarks under both the **AtAnnotation** and **Targeted** migration strategies, when annotated using either the TagThread or TagMethod approaches. The **AfterSched** migration strategy was not investigated, due to its poor performance in previous experiments. The performance of each benchmark is shown as the speedup gained relative to the benchmark's performance when executed entirely on the PPE core.

Each of these benchmarks are designed to examine a particular facet of system performance. Therefore, unlike the XML parse benchmark used above, they generally exhibit just one main behaviour phase, which performs better on either the SPE or PPE core. If the runtime system chooses the most appropriate core type on which to execute this phase, based upon the behaviour annotations, the benchmark's performance should approach the higher of either the SPE's or PPE's performance. Since Figure 6.16 measures performance relative to execution on the PPE core, the performance of the system should approach the higher of 1.0 or the SPE's performance.

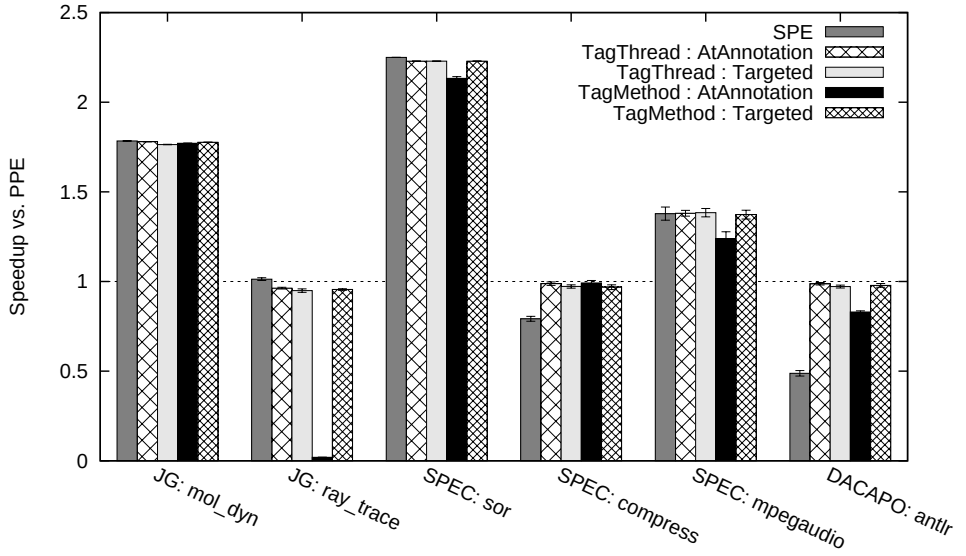



Figure 6.16: Performance of real world benchmarks, which have been annotated with their behaviour characteristics, when executed under Hera-JVM.

These results show that under both annotation approaches and migration strategies, all of the benchmarks (other than ray_trace under the **TagMethod:AtAnnotation** strategy) come close to the performance that they would achieve if they were manually placed on their most appropriate core type. This shows that the runtime system is choosing to migrate the benchmark threads correctly, based upon the information it has available through the behaviour code annotations. Therefore, the different migration strategies and annotation approaches can be compared with regards to their overhead and their intuitiveness for application developers.

Figure 6.17 shows the same results, but plotted as the overhead of each approach compared with manually placing the benchmark on the most appropriate core type. It is clear from this figure that the **TagMethod:AtAnnotation** approach frequently incurs a significantly larger overhead than other approaches. In the case of the ray_trace benchmark, this approach incurs in excess of a 50x slowdown relative to manual placement, rendering the program entirely unusable. This overhead is due to the fact that the **AtAnnotation** migration strategy immediately migrates any method that has been annotated with behaviour characteristics suggesting it would perform better on the other core type. If these methods are short-lived, the overhead of migration will be

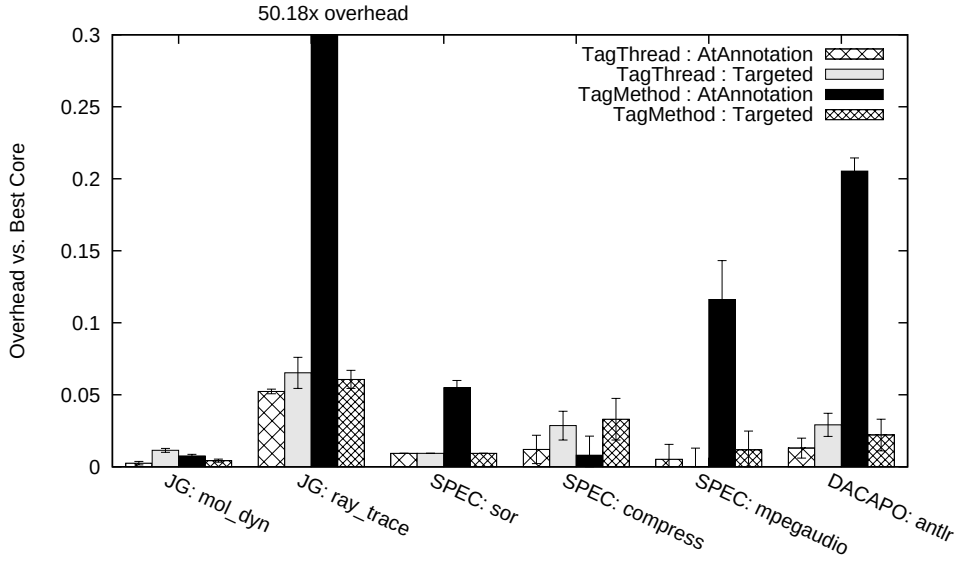



Figure 6.17: Overhead of tracking behaviour annotations.

greater than the benefit they gain from being executed on the other core type. If these short-lived methods are called very frequently, as with the ray_trace benchmark, this overhead can lead to more time being spent supporting thread migration than doing useful work.

The **Targeted** migration strategy does not encounter this problem, since it only targets long-lived methods for migration. The impact of each method's behaviour characteristics on overall program execution is built up over time and, if appropriate, a method is selected for migration. In some cases, such as the antlr benchmark, the shorter method's behaviour have a negligible impact on their parent methods and, therefore, no migration occurs. In others, such as ray_trace and mpegaudio, the behaviour characteristics of these short-lived methods have a large impact, thereby influencing the runtime system to migrate one of their longer lived parent methods.

In the **TagThread** approach, the program's behaviour characteristic annotations are attached to the main method of the benchmark's thread of execution. They are, therefore, encountered early in the benchmark's execution by the runtime system. The **Targeted** approach performs almost equally as well under either the **TagThread** or **TagMethod** approaches. The **AtAnnotation** approach, however, is better suited to the **TagThread** approach to annotation placement than it is to the **TagMethod** ap-

proach. Since the annotations tag the benchmark thread's run method, the whole of the benchmark's execution is migrated if the behaviour annotations suggest this would be appropriate. Since the annotations are only encountered once under the **TagThread** approach, the **AtAnnotation** strategy only makes a single migration decision during the benchmark's execution. It, therefore, has a slightly lower overhead than the **Targeted** strategy under this annotation placement approach, since the **Targeted** strategy continues to evaluate a thread's cost at each scheduling operation.

However, the **TagMethod** approach to annotation placement would seem to be more intuitive to developers than the **TagThread** approach, since they are only required to understand the behaviour characteristics of a single method, rather than trying to discover the behaviour characteristics of a thread throughout its execution. Also, if a thread exhibits different behaviour phases, these can be captured by the **TagMethod** approach, while they might be lost in the **TagThread** approach. For these reasons, the **Targeted** migration strategy provides the best trade-off in terms of ease-of-use, flexibility and performance.

## 6.5 Summary

This chapter investigated the use of behaviour characteristic code annotations to inform a runtime system's selection of an appropriate core type on which to execute a program's code for an HMA system. The heterogeneous core types of the Cell processor have different performance characteristics, depending upon the behaviour of the code executed upon them, even under the homogeneous virtual machine interface of Hera-JVM.

To explore the efficacy of scheduling based upon code behaviour annotations, Hera-JVM was augmented to track code annotations describing behaviour characteristics that are applicable to code performance on the Cell processor's two core types. A cost function was developed in Section 6.2 which enables the runtime system to decide whether a thread should be migrated to a different core type based upon its current behaviour characteristics. Historical data, hysteresis and trend tracking were applied to this cost function in order to increase the system's stability and limit migrations that are likely to be detrimental to overall performance.

Three different migration strategies (**AtAnnotation**, **AfterSched** and **Targeted**) were implemented for the Hera-JVM runtime system. These strategies progressively trade off immediacy in reacting to behaviour changes with making more informed decisions about the point in a thread's execution at which it is most appropriate to migrate. Experiments, both on synthetic and real world benchmarks, showed that the **AtAnnotation** strategy performs very well when the behaviour annotations are at appropriate migration points. However, it is very sensitive to annotation placement and behaviour phase length. This makes it a less appropriate migration strategy for a program which has been annotated by a non-specialist developer. The **AfterSched** strategy was found to perform poorly in all situations, since it is likely to migrate short-lived methods. The **Targeted** strategy is much less sensitive to annotation placement; it also does not incur significant slowdowns when the behaviour phase length is too short to benefit from migration. This makes it a good *do no harm* migration decision strategy, suitable for executing applications annotated by non-specialist developers. However, it misses the first opportunity to migrate a thread when its behaviour changes, and thus relies on this behaviour phase being repeated multiple times if it is to be effective.

A strategy that combines the immediacy of the **AtAnnotation** strategy with the *do no harm* characteristics of the **Targeted** strategy would provide the best of both worlds.

# Chapter 7

# Monitoring Program Behaviour at Runtime

The previous chapter shows that a runtime system can use knowledge of a program's expected behaviour, provided by code annotations, to inform its thread scheduling and migration decisions on a heterogeneous multi-core architecture. However, code annotations are not the only way of providing this behaviour information. Instead, the runtime system could directly monitor certain aspects of a program's behaviour at runtime. By using this monitored information to infer a program's behaviour, the runtime system can enhance the behaviour information provided by annotations and even support efficient exploitation of heterogeneous cores by completely unmodified applications that have not been augmented with behaviour annotations.

This chapter investigates the extension of Hera-JVM to support automatic monitoring of a thread's behaviour, based upon the frequency with which it executes different classes of bytecode. With this information, it can make informed scheduling and migration decisions, even when the program is not annotated with the behaviour annotations described in the previous chapter.

Section 7.1 describes the mechanism used to monitor a thread's behaviour at runtime. The means by which this monitoring information informs migration decisions is described in Section 7.2. An experimental analysis of these techniques under both synthetic and real world benchmarks is presented in Section 7.3.

## 7.1 Monitoring Execution of Different Bytecode Types

Hera-JVM was augmented to enable investigation of the use of runtime monitoring of a program's behaviour to inform thread placement on the Cell's heterogeneous core types. Only two behaviour characteristics are monitored by Hera-JVM — the proportion of arithmetic and object access operations performed by a thread. By tracking these two values, the runtime system can infer highly arithmetic phases of a thread, which are likely to benefit from migration to an SPE core. Similarly, phases which perform a significant number of object accesses can be identified, to ensure they are run on the PPE core.

It is important that the runtime monitoring of a thread's behaviour is lightweight and efficient, otherwise any gains in performance, realised by improved scheduling decisions, will be offset by the overheads involved in performing this monitoring. Thus, while an approach which directly counts the number of arithmetic and object access bytecodes executed by a thread would provide precise measurement of the proportion of each operation type being performed by a thread, the overheads of doing so would likely outweigh any benefits.

To reduce this monitoring overhead, Hera-JVM employs a two stage process. The first stage involves analysing a method's bytecodes to *score* the behaviour of blocks of code within the method. This stage occurs only once per method, when it is being compiled from bytecode to machine code by the runtime system. The second stage involves using these scores to update a pair of per-thread counters when the method is executed. An approximate count of the number of arithmetic and object access bytecodes executed by a thread can be calculated, based upon the values of these two counters. Section 7.1.1 describes the system used by Hera-JVM to score methods at compile time. The mechanism used to monitor a thread's behaviour, based upon these scores, is described in Section 7.1.2.

### 7.1.1 Scoring Methods

Each method is scored immediately before being compiled to machine code by the runtime system's JIT compiler. The scores which are generated for a method represent the number of arithmetic and object access bytecodes that will be executed whenever

the method is invoked. The purpose of generating these scores is to provide a summary of the types of operations which will be performed when this method is executed. Updates to the runtime monitoring counters can then be performed in aggregate, using this summary score information, rather than having to monitor every operation performed.

A single arithmetic / object access score could be created per-method, with the monitoring code updating a thread's behaviour, based upon this score, whenever the method is invoked. However, loops, if / else blocks and case statements mean that the actual number of operations which a method executes is not necessarily the same as the number of bytecode operations in that method's code.

A more accurate approach would be to score each of the *basic blocks* of the method[1], and update the thread's behaviour, using this score, whenever the basic block is executed (this approach is similar to that employed by Sherwood *et al.* (2001) to find application phases). While this approach would provide an accurate representation of a thread's behaviour, updating the runtime monitoring counters every time a basic block executes would lead to significant monitoring overheads.

Instead, the approach taken by Hera-JVM is to approximate the behaviour of a thread, by scoring methods and loops, but amalgamating the scores of other basic blocks, such as if / else and case statements, into their parent loop's or method's score. The reasoning behind this approach is that loops and method invocations can have a significant effect on a thread's behaviour, whereas other basic blocks are unlikely to significantly influence the thread's overall behaviour.

For example, a highly arithmetic loop could execute many more arithmetic operations than are accounted for by the method's score, since the loop's body will be repeated many times. However, the execution of a particular conditional block of code (e.g. regardless of whether the if block or the else block of an if statement is executed) should not, by itself, significantly affect a thread's behaviour[2].

The scoring system, therefore, calculates multiple scores for each method; one for the method's main score and one for each of the loops found within the method. When a method is invoked, the monitoring system updates the thread's behaviour score using

[1] A basic block is a code section which contains only a single entry point and a single exit point (i.e., there are no jumps within the block). It is therefore either executed in its entirety, or not at all.

[2] If the conditional block invokes a method or executes a loop, this could have a significant effect on behaviour. However, this effect will be captured by the method or loop scoring.

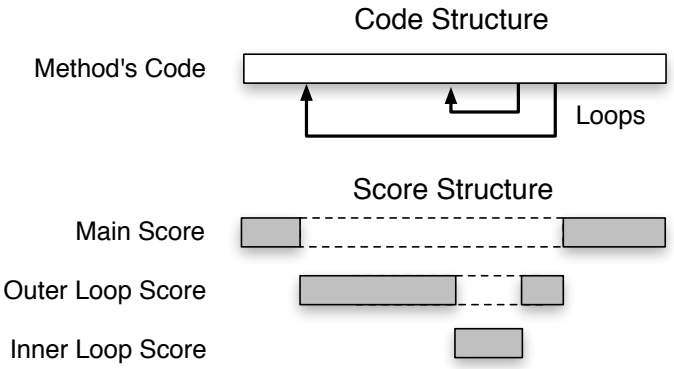



Figure 7.1: The structure of a method's score, related to its code structure.

the method's main score. The monitoring system also updates a thread's behaviour score after each iteration of a loop's body, using the score associated with the loop.

Each of these scores represent a distinct, non-overlapping block of code within a method. The method's main score is calculated based upon code which is not part of any loop body. Likewise, a loop's score is based upon the code in its body, minus any code which forms an inner loop. Invocations of other methods do not contribute to this method's score; these method's will be scored separately and only taken into account if the method is actually invoked. Figure 7.1 demonstrates which parts of a method's code is used to calculate each score.

To monitor a thread's behaviour, each score consists of both a count of arithmetic bytecodes and a count of object access bytecodes within the block of code that this score represents. To calculate the set of scores associated with a particular method's code, its bytecode stream is scanned to classify each bytecode. During this classification process, branch operations are also inspected. Those branches which target an earlier point in the stream (i.e. backward branches) are indicative of loops; therefore, the source and target address of these branches are noted as a loop iteration boundary. A score is created for each of these loop iterations, as well as for the method itself.

Two counters are created for each of the loops found and for the method's body itself. The counters represent the number of arithmetic and object access bytecodes in the associated body of code. The array of bytecode classifications is scanned and, for each bytecode, the score associated with the innermost loop at that bytecode's position

(or the method's main score if the bytecode is not part of a loop's body) is updated by incrementing the appropriate counter.

This scoring process calculates these scores based upon a method's bytecode; therefore, it cannot be used for native methods or system call methods, which have no associated bytecode. Most of these methods can be safely ignored, since they are generally not arithmetic or object accessing in nature and are infrequently executed. Therefore, they have minimal influence on the aspects of a thread's behaviour that are being monitored by this system.

However, the Java Math package (`java.lang.Math` is highly arithmetic, implemented as native code and likely to be invoked frequently by some applications. To enable the influence of this package to be monitored, the scoring system treats an invocation of a method in the Math package specially. Since a Math method cannot be scored directly, its influence is subsumed into the score of any invoking method. Method invocation bytecodes are checked to discover if they invoke a method in the Math package. If so, the bytecode is scored as if it is multiple bytecodes (one hundred arithmetic bytecodes by default, chosen as an approximation of the work done by the Math methods). This approach could also be used for other intrinsic functions (e.g. Magic methods) or system calls if these were found to significantly influence a thread's behaviour.

### 7.1.2 Monitoring a Thread's Behaviour

Once these scores have been calculated, they are used by the runtime system to approximate the number of arithmetic and object access bytecodes that have been executed by a thread at runtime. An execution count for each type of bytecode is maintained in per-thread runtime counters. Whenever a block of code, represented by a particular score, is executed (e.g. a method body returns, or a loop iteration is executed), these counters are updated with the values held in the appropriate score. Since the score holds the count of each class of bytecode for the block of code that was just executed, this update approximates the amount, and type, of work performed during this code block's execution.

To ensure low-overhead monitoring, these updates are performed inline, alongside the rest of the method's code. Hera-JVM's bytecode to machine code compiler inserts machine code that updates the runtime monitoring counters at appropriate points in

a thread's execution, namely, at a method's epilogue and backward branches (i.e., the end of every loop body iteration). Each update involves simply incrementing both the arithmetic and object access bytecode counters by a compile time constant value, based upon the current code block's score.

As with the behaviour bitmap, used in the previous chapter to track the set of behaviour annotations which apply to a thread, these bytecode counters reside at a fixed per-core location (in local memory for the SPE cores and directly on the stack for the PPE core). During a thread switch operation, the values in these per-core counters are saved in the outgoing thread's control block, and replaced with the incoming thread's previously saved counters. The code necessary to update these counters is eight machine instructions on the PPE core and fourteen instructions on the SPE core. This runtime monitoring system has a relatively low overhead, due to these updates being lightweight, and only being necessary at a method's epilogue and loop iterations.

## 7.2 Migration Decisions

The monitoring system, described above, enables the runtime system to infer a thread's behaviour with regard to both the number of arithmetic and object access bytecode operations executed. Hera-JVM uses this knowledge to inform thread migration decisions between the PPE and SPE cores. The migration triggering infrastructure, described in the previous chapter, is reused by Hera-JVM for this runtime behavioural monitoring approach.

### 7.2.1 The Cost Function

The cost function, described in Section 6.2, is adapted for use with the information provided by monitoring of a thread's arithmetic and object access behaviour at runtime. A cost is associated with arithmetic operations ($C_A$) and object access operations ($C_O$) for both core types. The runtime monitoring system provides a count of the number of arithmetic ($M_A$) and object access ($M_O$) bytecodes executed since the last cost evaluation. The predicted cost of executing a thread on each core, during this time period, is calculated as the proportion of each type of bytecode operation, multiplied by the operation type's respective cost:

$$C = \frac{M_A}{M_A + M_O} \cdot C_A + \frac{M_O}{M_A + M_O} \cdot C_O \tag{7.1}$$

| Behaviour | PPE Cost | SPE Cost |
|---|---|---|
| Arithmetic ($C_A$) | 4 | 1 |
| Object Access ($C_O$) | 1 | 2 |

Table 7.1: Costs associated with each core type.

The costs for arithmetic and object access operations on each core type are shown in Table 7.1. These costs are the same as those used for the `@ArithmeticCode` and `@ObjectAccessCode` behaviour annotations in Chapter 6. As with the annotation-based cost function, the reasoning behind these costs is that arithmetic operations are up to four times faster on the SPE core, while object access operations (when cached) are two times faster on the PPE core.

Dividing the monitored count of executed arithmetic and object access bytecodes by the total monitored count of executed bytecodes leads to these values being normalised between zero and one. This is useful when runtime monitoring is combined with explicitly annotated behaviour characteristics, where the coefficients used to compute the cost function have values of either zero or one (see Section 7.2.2).

The annotation-based cost function measures the influence of a particular behaviour characteristic in a binary manner. That is, a thread is either affected by a particular behaviour, if it has invoked a method tagged with the associated behaviour annotation, or it is not. The cost function for this runtime monitoring approach, on the other hand, represents the arithmetic and object access behaviour characteristics as a fraction between zero and one, based upon the proportion of each type of work performed in a given time period. Thus, rather than simply categorising a thread as either having, or not having, a particular behaviour characteristic, this approach can provide some sense of the degree to which the thread is affected by the characteristic.

Once a thread's raw cost has been calculated for each core type, it is smoothed using the exponential moving average function, previously described in Equation 6.2. These costs are converted into a target score, which the runtime system uses to make a migration decision. Hysteresis and trend tracking are used in this migration decision to limit unnecessary migrations, as is the case with the annotation-based cost function (Equations 6.10 and 6.11 are reused by this monitoring-based cost function). The history, hysteresis and trend tracking processes of this cost function are controlled using the $\alpha$, $\beta$ and $\gamma$ parameters, as described in the previous chapter.

### 7.2.2 Combining Annotations with Runtime Monitoring

Whilst a runtime system can use runtime monitoring to learn certain behavioural characteristics of a program, there may be other characteristics which are overly expensive or impractical to monitor at runtime or for which runtime monitoring has simply not yet been implemented. In these cases, it may be useful to combine the behavioural information provided by code annotations with the characteristics monitored at runtime, in order to inform thread migration and placement decisions. This runtime monitoring approach makes migration decisions based upon a very similar cost function to that used by the annotation-based approach. Therefore, the cost function can be easily augmented to incorporate behaviour characteristic information provided by code annotations into its migration decisions.

The behaviour characteristics $B_A$, $B_O$ and $B_L$, used by the annotation-based cost function, are either zero or one, depending upon whether the characteristic is set or un-set for the thread in question. Similarly, the monitored proportion of arithmetic and object access operations are normalised to a fraction between zero and one (i.e., the values of $\frac{M_A}{M_A+M_O}$ and $\frac{M_O}{M_A+M_O}$ will always be between zero and one), before being used in the runtime monitoring cost function. Thus, the value of $\frac{M_A}{M_A+M_O}$ can be seen as equivalent to $B_A$, and $\frac{M_O}{M_A+M_O}$ as equivalent to $B_O$, except that they provide a continuous range between zero and one, rather than a binary choice. This enables the `@LargeWorkingSet` behaviour annotation to be combined into the runtime monitoring approach's cost function, in the same form as in the annotation-based cost function. This leads to the following raw cost function:

$$C = \frac{M_A}{M_A + M_O} \cdot C_A + \frac{M_O}{M_A + M_O} \cdot C_O + B_L \cdot C_L \tag{7.2}$$

where, $B_L$ is set to zero or one, depending upon whether the thread has inherited the large working set characteristic through the program's code annotations. The cost value used for $C_L$ is the same as used in the annotation-based approach (2 for the PPE core and 8 for the SPE core).

### 7.2.3 Triggering Thread Migration

The same thread migration triggering infrastructure, that was developed for Hera-JVM as part of the annotation-based migration approach (described in Section 6.3),

can also be used for this runtime monitoring-based approach. Three different migration strategies were developed for the annotation-based approach — **AtAnnotation**, **AfterSched** and **Targetted**. However, since this runtime monitoring approach does not employ behaviour annotations for migration decisions, there is no equivalent to the **AtAnnotation** migration strategy for runtime monitoring. The **AfterSched** migration strategy was also not employed in this chapter, due to its poor performance in the previous chapter. Thus, only the **Targetted** approach is used by Hera-JVM for runtime monitoring-based migration.

The cost of running a thread is calculated each time it completes execution of its assigned scheduling quantum. If the cost function decides that the thread would benefit from migration, it selects a long-lived method from the thread's current stack trace and targets that method to be a migration point when it is next invoked (the process is the same as described in Section 6.3). This strategy is particularly suitable, since the migration decision is directly based upon the work which was performed during the thread's previous scheduling quantum. Thus, targeting one of the methods on the thread's call stack that led to this behaviour is an appropriate course of action.

## 7.3 Experimental Analysis

In this section, the effectiveness of exploiting runtime monitoring information to trigger thread migrations is investigated. Suitable values are established for the cost function's $\alpha$, $\beta$ and $\gamma$ parameters, which control history, hysteresis and trend tracking, respectively. Unmodified and unannotated real world benchmarks are then executed using this system to validate this approach with realistic applications. The experiments with these benchmarks examine whether monitoring an application's behaviour at runtime can enable the runtime system to select appropriate migration points. These experiments also examine the overhead incurred by monitoring a thread's behaviour at runtime. All the experiments up until Section 7.3.3 employ the runtime monitoring information exclusively, using the cost function defined in Equation 7.1. In Section 7.3.3, the information provided by behaviour code annotations is combined into this cost function, as described in Section 7.2.2.

The same experimental set-up is used as in the previous two chapters, with experiments being performed on a Sony Playstation 3, with 256MB of RAM, running Fedora Linux 9. Error bars represent the standard deviation of ten runs of each experiment.

### 7.3.1 XML Parsing Synthetic Benchmark

The first set of experiments uses the XML parsing benchmark, described in Section 6.4.3. This benchmark has three distinct behaviour phases: the encryption and decryption routines, which perform best when run on the SPE cores; the compression and decompression routines, which perform approximately equally well on either core type; and the XML parsing, which runs faster on the PPE core. These phases all exhibit a different proportion of arithmetic and object access operations, which should enable the runtime monitoring system to detect them and migrate the benchmark execution appropriately.

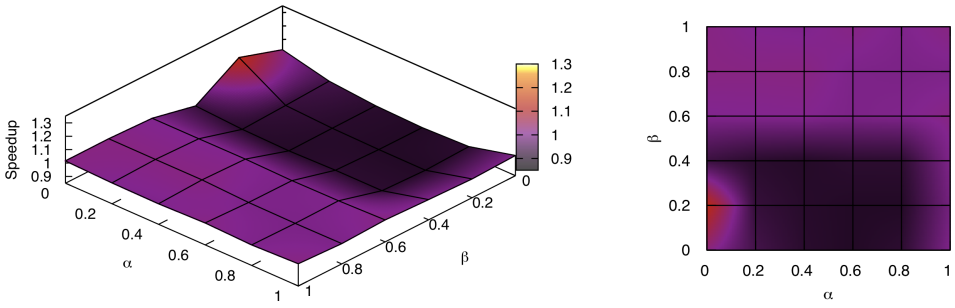
#### 7.3.1.1 Exploration of the Cost Function Parameter Space

As with the annotation-based cost function, the three independent variables ($\alpha$, $\beta$ and $\gamma$) control history, hysteresis, and trend tracking in the runtime monitoring-based migration decision cost function. While these parameters control the same aspects of the cost function as with the annotation-based approach, the most effective values for each parameter may differ from those found in Section 6.4.3.1, due to the different source of information being provided to the cost function. Therefore, this section re-explores this parameter space to find the most appropriate values for this runtime monitoring approach.

As in Section 6.4.3.1, the value of each parameter was varied in 0.2 increments between 0 and 1, and the XML Parsing benchmark was run under the **Targeted** migration strategy for each combination of these parameter values. The results of each parameter setting are shown in Figures 7.2 to 7.9. The same scale is used as in Figures 6.6 to 6.11, with white representing the best possible speedup achieved by the runtime monitoring approach, down through orange, representing a modest speedup, then purple, representing no speedup and finally black, representing a slowdown.

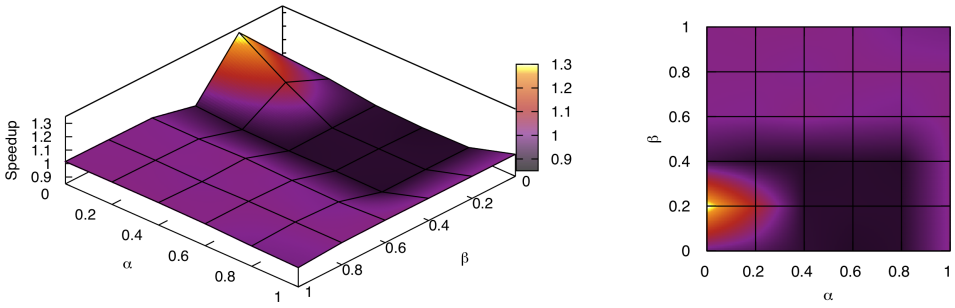
Comparing these results to those of the annotation approach in Section 6.4.3.1, it is clear that the performance of the runtime monitoring cost function is more sensitive to the value of these parameters.

This is particularly noticeable as the value of $\gamma$, which controls trend tracking, is increased between Figures 7.2 and 7.7. When no trend tracking is employed ($\gamma = 0$), the benchmark performs almost uniformly poorly. As the influence of trend tracking is



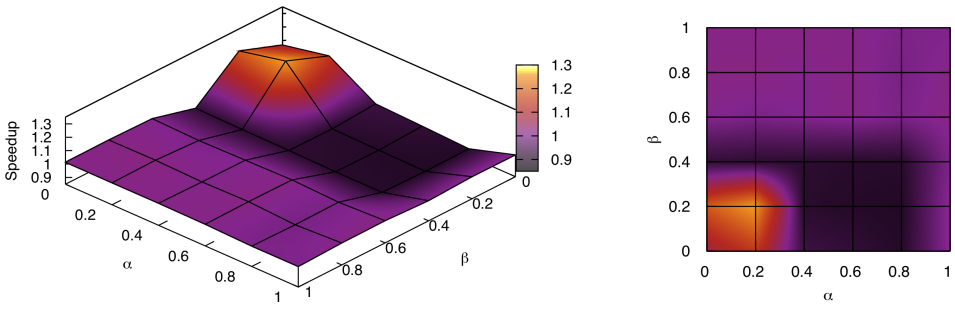

(a) 3D Projection (b) Heat Map

Figure 7.2: Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 0$



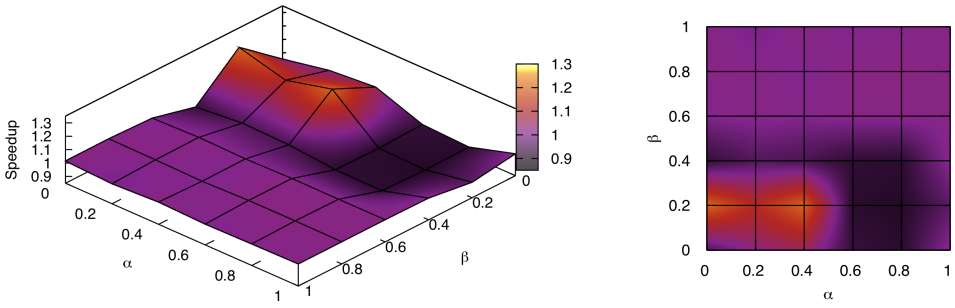

(a) 3D Projection (b) Heat Map

Figure 7.3: Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 0.2$



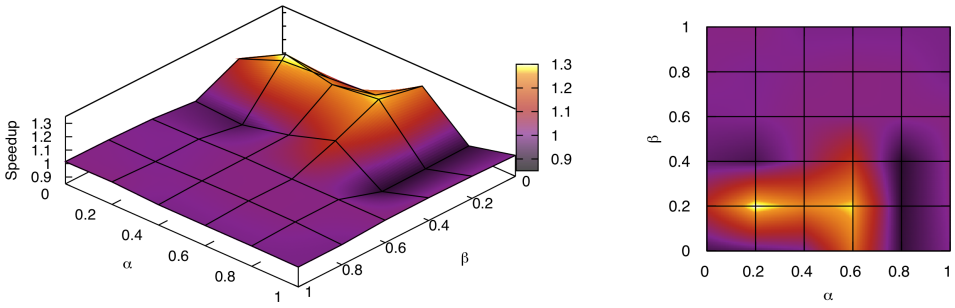

(a) 3D Projection (b) Heat Map

Figure 7.4: Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 0.4$



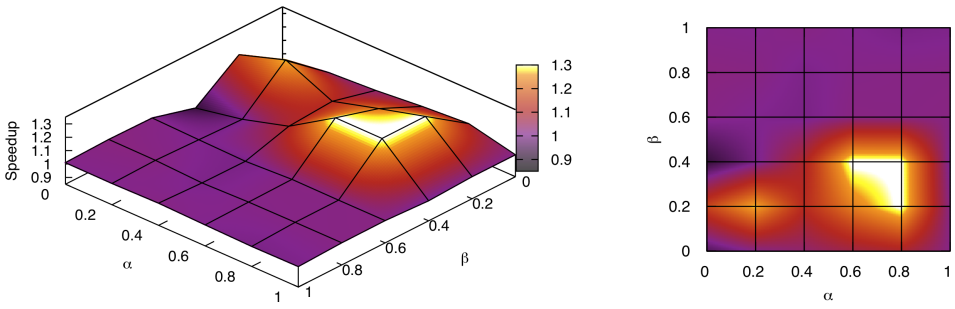

(a) 3D Projection (b) Heat Map

Figure 7.5: Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 0.6$




(a) 3D Projection (b) Heat Map

Figure 7.6: Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 0.8$




(a) 3D Projection (b) Heat Map

Figure 7.7: Speedup as $\alpha$ and $\beta$ parameters are varied, with $\gamma = 1$

increased, the range of the $\alpha$ and $\beta$ parameter values that provide good performance increases. The best speedup results are seen when $\gamma = 1$.

The increased positive influence of trend tracking on this cost function, compared to the annotation-based approach, highlights the noisier nature of the data provided by runtime monitoring. The monitored arithmetic and object access execution counts fluctuate significantly between sampling periods, even within a single *behaviour phase* of an application. Trend tracking limits migrations to only those cost updates which increase the system's confidence that a thread will perform better on the other core type. This increases the system's robustness to inaccurate monitored behaviour, caused by an erroneous or uncharacteristic sampling period.

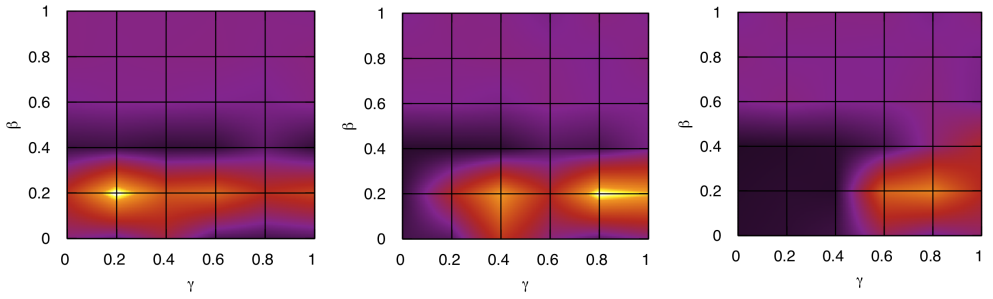
Figures 7.8 and 7.9 show the same data, but with the $\alpha$ and $\beta$ parameters held constant in each graph. Figure 7.9 shows two interesting interactions between these parameters. Firstly, whenever $\beta \geq 0.6$, the system shows no speedup or slowdown. Since $\beta$ controls the hysteresis threshold required to trigger migration, setting this parameter too high means that the monitored behaviour will never trigger a migration. Secondly, as with the annotation-based approach, the $\alpha$ and $\gamma$ parameters are correlated, suggesting that trend tracking is required to provide stability against inappropriate migrations caused by out of date information.

The most effective combination of these parameters is $\alpha = 0.8$, $\beta = 0.2$ and $\gamma = 1$. Interestingly, this is also the most appropriate combination of parameters for the annotation-based approach. The remaining experiments all use these values, unless otherwise stated.

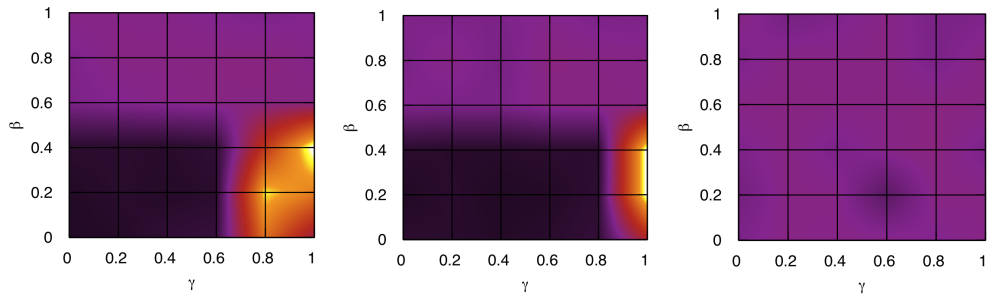
#### 7.3.1.2 Comparing Runtime Monitoring to Annotations

Figure 7.10 compares the performance of the XML parsing benchmark, under the runtime monitoring migration strategy, to that of the annotation tracking strategy.

The "Monitoring (Overhead)" result shows the overhead incurred by the runtime monitoring system, under this benchmark. To uncover this overhead, Hera-JVM was run with the $\alpha$, $\beta$ and $\gamma$ parameters set deliberately high (all set to one), such that the thread will never migrate and instead always runs on the PPE core. Thus, the difference in performance between this run and the non-monitored PPE core run reflects the overhead in monitoring a thread's behaviour, without involving thread migration. This overhead is about 2% for this XML parsing benchmark.

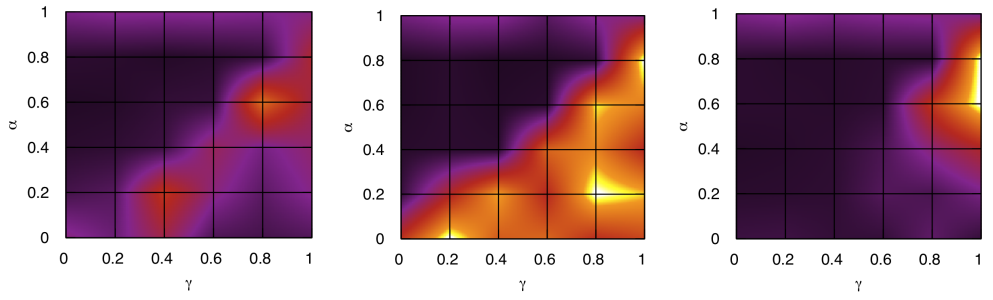



(a) $\alpha = 0$ (b) $\alpha = 0.2$ (c) $\alpha = 0.4$

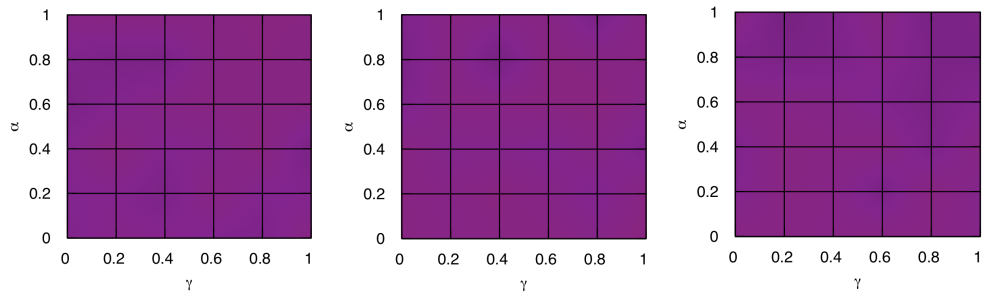



(d) $\alpha = 0.6$ (e) $\alpha = 0.8$ (f) $\alpha = 1$

Figure 7.8: Comparing the interaction between the $\beta$ and $\gamma$




(a) $\beta = 0$ (b) $\beta = 0.2$ (c) $\beta = 0.4$




(d) $\beta = 0.6$ (e) $\beta = 0.8$ (f) $\beta = 1$

Figure 7.9: Comparing the interaction between the $\alpha$ and $\gamma$

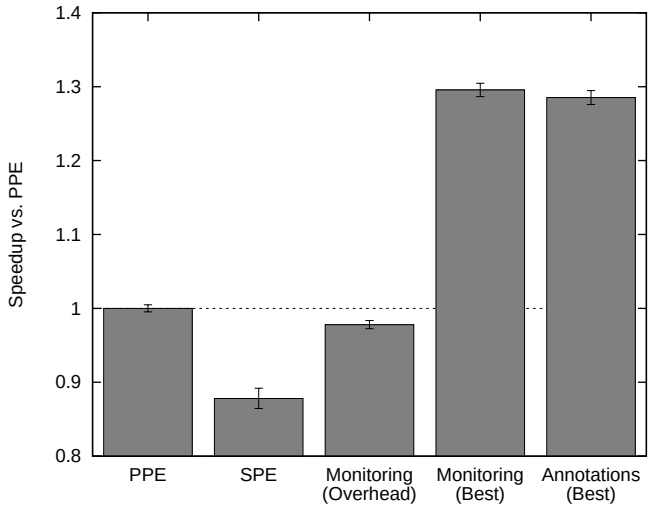



Figure 7.10: Comparing runtime monitoring and behaviour annotations for the XML Parsing benchmark.

The "Monitoring (Best)" result presents the speedup provided by the runtime monitoring system when its parameters are set correctly ($\alpha = 0.8$, $\beta = 0.2$ and $\gamma = 1$). This speedup is very close to that provided by the annotation-based approach, in fact it slightly exceeds its performance. The increase in performance compared to "Annotation (Best)" is relatively insignificant; however, considering that this approach is also overcoming the added overhead of its runtime monitoring, this increase in performance is surprising.

### 7.3.2 Real World Benchmarks

To validate this use of runtime behaviour monitoring, the set of real world benchmarks, introduced in Section 5.5.3, are run under this modified version of Hera-JVM. The benchmarks are unmodified and have not been annotated with their behaviour characteristics; only the runtime monitoring data is used by Hera-JVM to influence its migration decisions.

Figure 7.11 presents the performance of these benchmarks when migration is controlled by runtime monitoring, compared to manually selecting the SPE core for their execution. These results are shown as a speedup, relative to execution on only the PPE core.

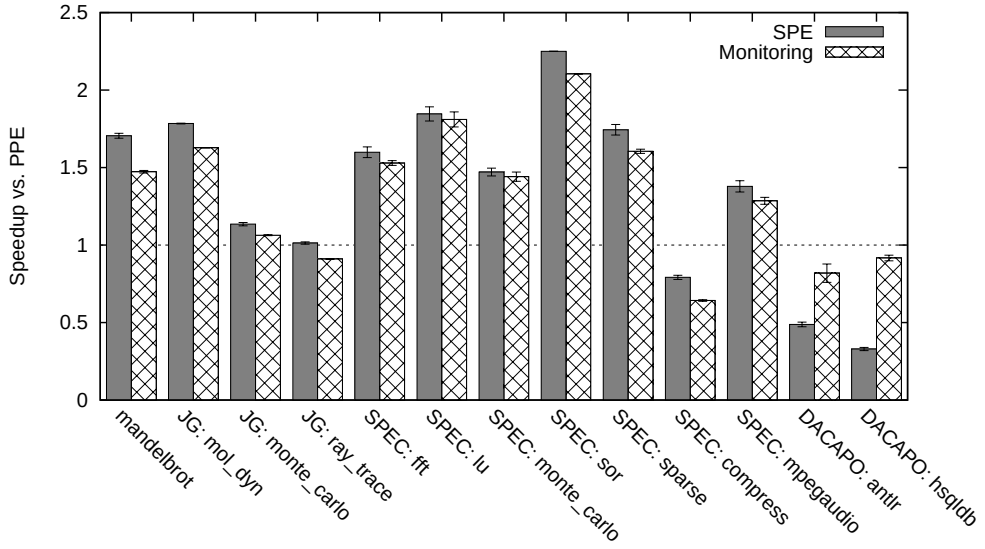



Figure 7.11: Performance of real world benchmarks running under Hera-JVM, when using behaviour runtime monitoring.

As discussed in Section 6.4.4, these benchmarks exhibit just one main behaviour phase, which performs better on either the SPE or PPE core. Therefore, if the runtime system chooses the most appropriate core type on which to execute this phase, the benchmark's performance will approach whichever is the higher of either the SPE's or the PPE's performance.

The results show that almost all of these benchmarks approach the performance possible when they are manually run on their most appropriate core type. Therefore, the runtime system has chosen the most appropriate core type for each benchmark's main phase of execution, based upon the monitored arithmetic and object access operation count.

The one exception is the compress benchmark, which the runtime system has migrated to the SPE core, even though it has poorer performance on this core type than the PPE core. This highlights one of the deficiencies of an approach which exclusively monitors program behaviour at runtime. The compress benchmark does have a high proportion of arithmetic operations. Therefore, based upon the information at its disposal, the runtime system has made the correct choice. However, this benchmark also operates over a large amount of data, leading to its poor performance on the SPE core.

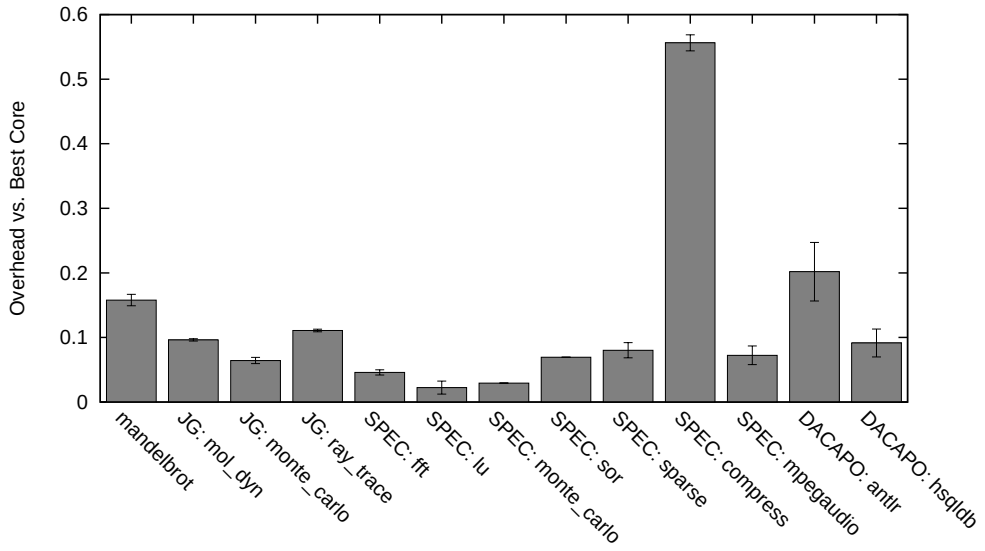



Figure 7.12: Overhead of runtime monitoring for real world benchmarks.

Since this behaviour characteristic is not tracked by the monitoring system, it is not taken into account for migration decisions. The annotation-based approach, however, can exploit the `@LargeWorkingSet` annotation to provide information about this behaviour characteristic. Section 7.3.3 investigates the performance when this annotation characteristic is taken into account alongside the runtime monitoring.

Figure 7.12 shows these results plotted as the relative overhead incurred by the runtime monitoring system, against manually selecting the *best* core on which to run each benchmark. The runtime monitoring system incurs between a 3% and 10% overhead on the majority of these benchmarks.

For those with a higher overhead, this cost is not purely down to the runtime monitoring itself. As explained above, the compress benchmark is executed on the sub-optimal core type, leading to its high apparent overhead. In the case of the antlr benchmark, a subset of this benchmark's methods are arithmetic in nature, and are therefore migrated to the SPE core. While these methods have similar performance on either core type, the overhead of migrating them leads to a decrease in this benchmark's performance. These methods are not always selected for migration (which explains the high variance in this benchmark's performance), suggesting their proportion of arithmetic operations is likely to be only slightly above the migration threshold. Finally,

some of the additional overhead of the mandelbrot benchmark is due to a relatively frequently called method being selected for migration, thus increasing the number of migrations performed.

### 7.3.3 Combining Annotations and Runtime Monitoring

As exemplified by the compress benchmark, runtime monitoring cannot always provide sufficient information about a program's behaviour to enable effective migration decisions by the runtime system. The runtime system requires knowledge of the compress benchmark's large working set behaviour, in order to choose a more appropriate core type for its execution. In this case, the runtime system could be augmented to provide runtime monitoring of this large working set behaviour characteristic, by tracking cache miss information at runtime. However, it may not always be feasible or prudent to directly monitor, at runtime, a particular behaviour characteristic that is required to inform thread core-type placement decisions. Therefore, this section investigates the incorporation of information about this characteristic, provided by the `@LargeWorkingSet` code annotation, with the arithmetic and object access behaviour characteristics, provided by runtime monitoring, in the manner described in Section 7.2.2.

The performance of each of the real world benchmarks annotated with behaviour characteristic annotations in Section 6.4.4 is measured using this combination of runtime monitoring and annotation-based behaviour information. The source code annotated using the **TagMethod** approach (rather than the **TagThread** approach) was used in these experiments because it most closely resembles the expected use case of the behaviour annotations and means that the set of behaviour annotations that apply to a thread change throughout its execution, rather than being fixed once the thread has been created.

Figure 7.13 compares the performance of these benchmarks when Hera-JVM uses either annotation, runtime monitoring or a combination of both to inform its thread migration decisions. Figure 7.14 shows the same results, but presented as the overhead incurred by each approach, compared to manually selecting the benchmark's best core type.

The only annotation that is being tracked by the *monitoring with annotations* approach is the `@LargeWorkingSet` annotation. The majority of the benchmarks only

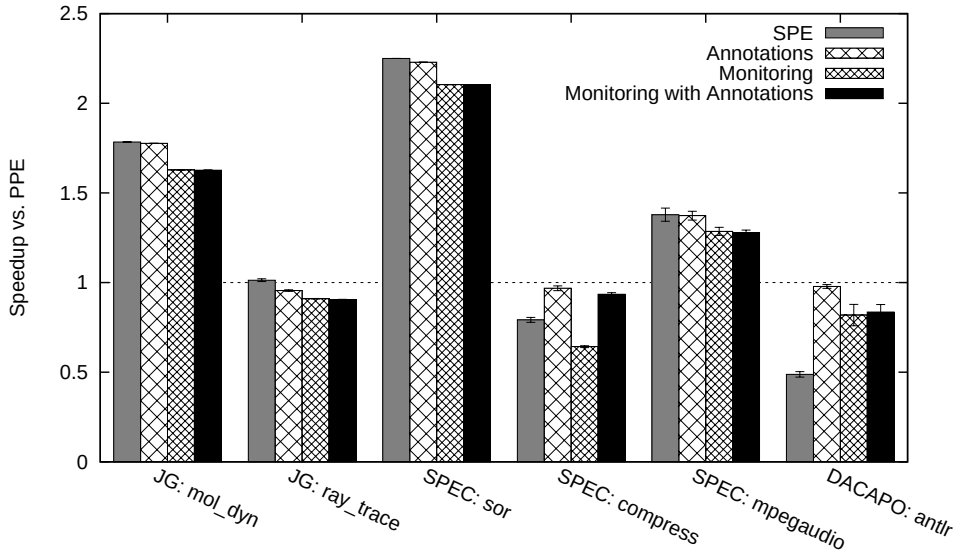



Figure 7.13: Performance of the Hera-JVM when runtime monitoring and annotation behaviour information are combined.

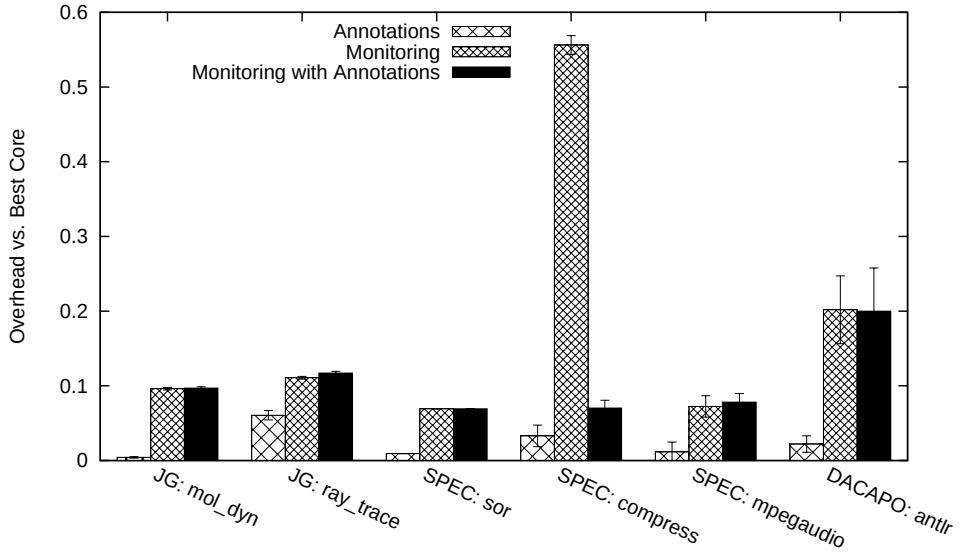



Figure 7.14: Overhead of runtime monitoring with annotations approach.

employ this behaviour annotation for short periods of their execution. Thus, its influence on migration decisions is negligible and the runtime system makes the same migration decisions as in the purely runtime monitoring approach.

The one benchmark which employs the `@LargeWorkingSet` annotation for a significant fraction of its execution is the compress benchmark, which performed badly under the purely runtime monitoring approach. When the runtime system incorporates the `@LargeWorkingSet` characteristic into migration decisions, alongside runtime monitoring of arithmetic and object access execution behaviour, this benchmark sees a considerable increase in its performance. By taking this annotation into account, the runtime system correctly decides that the compress benchmark is better suited to execution on the PPE core than the SPE core.

The overheads incurred by tracking behaviour annotations throughout a thread's execution are small, compared with those incurred by the monitoring of arithmetic and object access code behaviour at runtime. Therefore, incorporating behaviour annotations into migration decisions, alongside runtime monitoring of behaviour characteristics, imposes a negligible additional overhead over the monitoring-only approach.

## 7.4 Summary

This chapter investigated whether information gained from monitoring a thread's behaviour at runtime can inform a runtime system selection of the most appropriate core type for different phases of a program's execution. Doing so enables the runtime system to make appropriate thread migration decisions between the different core types of an HMA without requiring any input from the application developer. The architecture's heterogeneity can be hidden from the developer entirely; it is not even necessary for the developer to annotate an application with its behaviour characteristics.

To verify whether runtime monitoring can provide the information required by a runtime system to inform its thread migration decisions, Hera-JVM was augmented to support runtime monitoring of the proportion of arithmetic and object access operations executed by each thread. The experiments presented in Section 7.3 demonstrate, on real world benchmarking applications, that a runtime system can use the information provided by this form of runtime monitoring to select a suitable core type for a thread's execution in an HMA system.

Monitoring a thread's behaviour at runtime does impose some overhead on this system. However, the monitoring techniques described in Section 7.1.2 limit this overhead to between 3% and 10% for real world benchmarks. This overhead could be reduced even further by exploiting the fact that, in most JVM systems (including JikesRVM), *hot* methods are recompiled using an optimising compiler. Since a method will only be recompiled if it has been executed many times, the runtime system should already have built up knowledge of this method's behaviour. There is, therefore, no need to continue monitoring its behaviour; the method's influence on program behaviour can be summarised and updates performed in aggregate. Thus, when the method is recompiled to its optimised form, the runtime monitoring code in its prologue, epilogue and backward branches can be removed such that no monitoring overhead is incurred during the method's execution. These hot methods are responsible for the majority of the program's execution time — this is why the runtime system has selected them for recompilation — so removing the runtime monitoring overhead from these methods should have a significant beneficial influence on performance. This technique is regularly employed for other forms of runtime monitoring in runtime systems — e.g., in edge count profiling used to guide basic block reordering, to reduce mis-predicted branches and instruction cache misses (Burger & Dybvig, 1998).

It may not always be possible to fully capture the behaviour characteristics required to characterise a program's behaviour through runtime monitoring alone. Section 7.3.3 shows that additional information, provided by explicit annotations, can be combined with the information obtained through runtime monitoring, to improve thread placement and migration decisions. As well as using annotations for characteristics that are not tracked by runtime monitoring, annotations could be used to augment the information provided through runtime monitoring and vice versa. Annotations could be used to provide ahead-of-time hints of the expected behaviour of a program. Runtime monitoring could then refine, or, in the case of inaccurate annotations, correct this information, thus providing the best of both approaches.

# Chapter 8

# Inter-Thread Communication on NUMA Architectures

Chapters 5 to 7 have focused on abstracting the heterogeneity of the Cell processor, which is an extreme example of a heterogeneous multi-core architecture (HMA). However, other more conventional processor architectures are being introduced which also exhibit heterogeneity. An aspect of heterogeneity, now commonly found in multi-processor systems, is a non-uniform arrangement of system memory. These *non-uniform memory access* (NUMA) architectures have homogeneous processing core capabilities; however, their memory is laid out such that the speed of data access depends upon the location of the core accessing the data and the data's location in memory.

While not as demanding as the Cell processor, with regards to application development, a NUMA system still requires the developer to have knowledge of the system's memory layout in order to extract its maximum performance. This chapter investigates the use of *thread team* annotations, outlined in Section 4.2.3, as a means of expressing thread communication information to a runtime system. The runtime system can use this information to automatically optimise thread and data placement, such that the overheads incurred by the system's non-uniform memory layout are minimised.

Investigating the use of thread communication characteristics in the context of a NUMA system, rather than on the Cell processor, enables the applicability of processor heterogeneity abstraction through behaviour characteristics to be examined on a different class of heterogeneous architecture. Given that not all HMA systems are as heterogeneous as the Cell processor, investigating the effectiveness of behaviour char-

acteristics on a less heterogeneous system increases confidence that behaviour characteristic knowledge can be usefully applied in a broad range of different system types.

Section 8.1 describes the typical characteristics of non-uniform memory access architectures and outlines the effect such a structure can have on inter-thread communication. Section 8.2 presents thread team behaviour characteristics as a means of abstracting NUMA node placement decisions, by enabling programmers to express a program's inter-thread communication behaviour to the runtime system. Section 8.3 describes how these thread team characteristics can be used by a runtime system to inform its thread placement decisions, in order to reduce inter-thread communication overheads. An experimental analysis of scheduling using thread teams on a NUMA machine is presented in Section 8.4.

## 8.1 Non-Uniform Memory Access Architectures

As systems scale to an increasing number of processing cores, a single shared memory bus can quickly become a key scalability bottleneck (Archibald & Baer, 1986; Rettberg & Thomas, 1986). To overcome this bottleneck, a number of commodity multi-processor systems are now employing non-uniform memory access architectures (Cox & Fowler, 1989; Laudon *et al.*, 1997). These NUMA architectures contain multiple nodes, each having their own memory bus and associated memory. Cores on a given NUMA node can directly access data located on their node's memory (local memory); however, to access data on a different node (remote memory), they must communicate with that node over an inter-node interconnect. This leads to non-uniform memory access times, depending upon whether local or remote memory is being accessed.

These architectures commonly have a shared global address space and employ a cache coherency system to ensure that the processing cores share a consistant view of memory[1]. As such, a shared memory programming model can be used for application development on these architectures. However, to gain the maximum performance from such a system, a program must be carefully designed, such that it minimises remote memory accesses.

A typical example of a ccNUMA architecture, used by commodity servers, is the AMD Opteron architecture (Keltcher *et al.*, 2003). Figure 8.1 outlines the system ar-

[1]This type of architecture is known as cache coherent NUMA or ccNUMA.

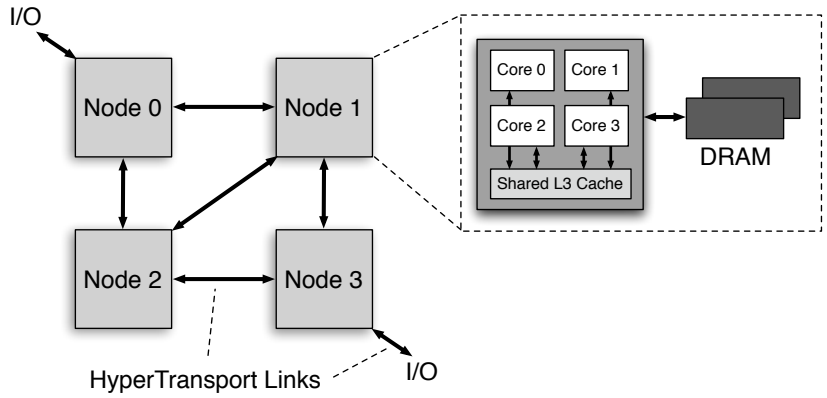



Figure 8.1: The NUMA node layout of a 4x4 core AMD Opteron system.

chitecture of a Dell PowerEdge 6950 server. It contains four quad-core AMD opteron processors. Each processor socket has a direct connection to its own memory node, but to access data located on another processor's memory node, the data must be transferred across the inter-processor HyperTransport links (hidden from software behind a hardware cache coherency protocol). Most nodes are a single HyperTransport hop away from each other; however, communication between nodes 0 and 3 requires two hops.

Communication between threads of a program often occurs through shared data objects. Data access latency depends upon the location of a thread and the data it is accessing on a NUMA architecture. Therefore, the overhead of inter-thread communication will depend upon the layout of threads and their shared data on the system's NUMA nodes.

A micro-benchmark was created to investigate the effect such a NUMA architecture has on inter-thread communication overheads. This benchmark spawns two threads, which *ping-pong* messages between each other, by alternately incrementing a shared counter, which is protected by token passing. The algorithm executed by each thread is outlined in Listing 8.1, as pseudo-code.

The micro-benchmark was written in C and compiled using gcc version 4.1.2. It was run on a Dell PowerEdge 6950 server containing four quad-core AMD Opteron 8350 processors. The libnuma library[1] was used to control the NUMA node on which each of

[1] `http://oss.sgi.com/projects/libnuma/`

```
1 threadRunFunc(volatile int * ctr, volatile int * token) {
2   for (i=0; i<5000000; i++) {
3     while (token != myID) { /* Spin */ }
4     (*ctr)++;
5     (*token) = otherID;
6   }
7 }
```

Listing 8.1: NUMA inter-thread communication micro-benchmark pseudo-code.

the two threads execute, as well as the node on which the shared data is allocated. The micro-benchmark was run for each combination of thread_1, thread_2 and shared data node placements[1], with each combination being run 10 times to provide an average and standard deviation.

Figure 8.2 shows the number of cycles required to perform a two way message (averaged over 5 million messages), as the location of the communicating threads and their shared data is moved between different NUMA nodes. The results are categorised into the set of placements which performed similarly — All on Same Node, Two on Same Node and All on Different Nodes. Placing the shared data on nodes 0 or 3 led to a higher overhead than if it was placed on nodes 1 or 2; therefore, these results are separated out accordingly in Figure 8.2. This overhead is likely due to the fact that nodes 0 and 3 are connected to I/O devices, unlike nodes 1 and 2, leading to higher contention on these nodes.

Placing both threads and the shared data on the same NUMA node (All on Same Node) results in the best performance[2]. The four cores on each NUMA node share a 2MB L3 cache. This means that if both the threads and their shared data reside on the same NUMA node, the threads can communicate through this shared cache, leading to reduced communication overheads.

The communication overhead increases if either one of the threads or the shared data resides on a different NUMA node (Two on Same Node). Placing the threads on different nodes leads to each message requiring communication over a HyperTransport

[1] With four nodes on which to place the two threads and the shared data, this leads to 64 possible combinations.

[2] Each NUMA node contains four processing cores; therefore each thread executes on its own core, even if the threads are on the same NUMA node.

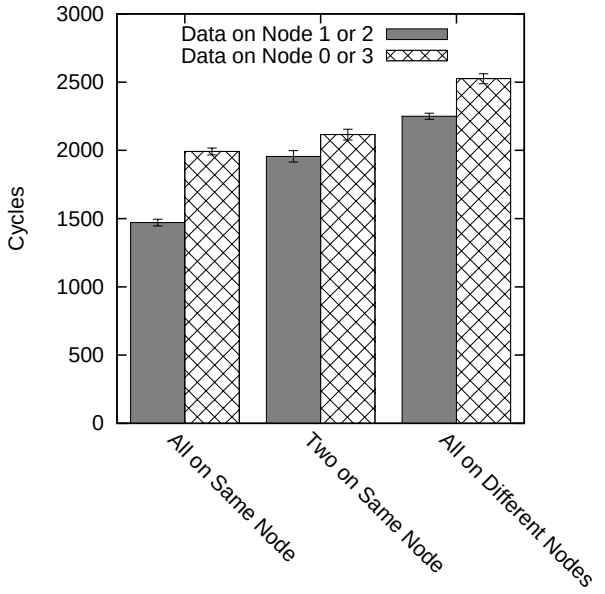



Figure 8.2: The effect of NUMA on inter-thread communication.

link for cache fetch and invalidation operations. Placing the shared data on a different NUMA node from the threads leads to a HyperTransport message being required for each write to this shared data.

Placing each of the threads and the shared data on different NUMA nodes results in the worst performance, since both cache operations and memory updates require HyperTransport communication. Of note, communication between node 0 and node 3 requires two hops on the system's HyperTransport network, leading to a distinctly higher overhead when the data must be transferred between these nodes (the final bar in Figure 8.2).

These results show that the selection of the NUMA node on which to allocate threads and data can have a significant impact on inter-thread communication overheads. This micro-benchmark shows up to a 60% increase in inter-thread communication overhead, depending upon thread and data placement. This result motivated the development of a runtime system which can use thread communication behaviour characteristics to inform its thread and data placement decisions.

## 8.2 Abstracting NUMA Node Placement Decisions

The approach taken in this work is to abstract NUMA node placement decisions from the application developer, by providing a mechanism for defining thread teams using code annotations. The runtime system will preferentially choose to execute a thread on a core that is close to, or on the same NUMA node as other threads of the same team, thus reducing their inter-thread communication overheads.

A thread team is a set of threads which are expected to communicate with each other or share access to data during their execution. A thread can be a member of more than one thread team. This provides a mechanism for describing asymmetric communication patterns. For example, if thread 1 is a member of team A, thread 2 is a member of team B, and thread 3 is a member of both team A and team B, then threads 1 and 2 both communicate with thread 3, but thread 1 and 2 do not communicate with each other.

In this work, a thread is assigned to a team using code annotations. Java threads are defined by a class that either extends the `Thread` class or implements the `Runnable` interface. To declare a thread as being a member of a team, the declaration of the class used to create this thread is annotated using the `@ThreadTeam(name="<name of team>")` annotation. The string literal, assigned to the `name` parameter, defines the team to which this thread is being tagged as a member. Any other threads annotated with a team which has the same name are members of the same thread team. If a thread is a member of multiple teams, then its class declaration will be annotated with a multiple of these annotations, one for each team of which it is a member.

The use of compile time string literals to define the name of the team of which a given thread is a member is limiting, since different threads created from the same thread class must always be members of the same set of teams. Unfortunately, this is an inherent limitation of annotations in Java. This limitation can be overcome by creating differently annotated sub-classes of a given thread class, although this is a relatively clunky solution. An approach with more flexibility would be to augment the thread object's constructor method with an argument, holding a dynamically generated list of thread teams of which the newly created thread should be a member. A method could also be added to enable a thread to change the set of teams of which it is a member during its execution. However, for the purposes of this work, the annotations are sufficient to demonstrate thread team-based scheduling on NUMA systems.

## 8.3 Scheduling based upon Thread Teams

This section describes how a runtime system can be augmented so that it makes informed thread placement and scheduling decisions on a multi-core NUMA architecture, based upon thread team behaviour characteristics. Hera-JVM was modified to enable investigation of this approach. Since Hera-JVM is based upon JikesRVM, it supports the x86 processor architecture, as well as the PowerPC architecture and Cell SPE support, described in Chapter 5. It can therefore execute Java applications on an x86 AMD Opteron NUMA server. This section details the modifications that were made to Hera-JVM to enable it to use thread team information to inform its thread scheduling decisions.

The goal of this scheduling system is to place threads which communicate with each other on the same or neighbouring NUMA nodes, to reduce their communication overheads. A secondary goal is to explicitly separate non-communicating threads, to reduce cache pollution on the L3 cache shared by all the cores of a NUMA node on the Opteron system used in this work. Finally, data should be allocated on the NUMA node upon which the threads most likely to access this data are executing.

To achieve these goals, Hera-JVM assigns each team to a *preferred* node. When a thread is created, the preferred nodes of each of the teams to which it belongs are used to calculate the preferred ordering of NUMA nodes on which to execute this thread. When making scheduling decisions, Hera-JVM uses a thread's preferred NUMA node order to preferentially execute this thread on a core that is simultaneously close to other members of its team and is not oversubscribed.

The changes made to Hera-JVM, to provide it with knowledge of the NUMA environment on which it is executing and enable it to control the NUMA node on which data is allocated, are described in Section 8.3.1. The algorithm used to generate a preferred node order for newly created threads is detailed in Section 8.3.2. Finally, Section 8.3.3 outlines the changes made to Hera-JVM's scheduling algorithm to incorporate a thread's preferred node order into the decision as to the core on which it should be scheduled.

### 8.3.1 Making Hera-JVM NUMA Aware

To exploit a non-uniform memory access architecture, a runtime system must have mechanisms for controlling the NUMA node on which threads are executed and memory is allocated. In this section, the provisioning of these mechanisms for Hera-JVM is described.

Before describing the changes made to Hera-JVM to enable NUMA aware scheduling and memory allocation, it is helpful to review the scheduling and memory allocation approach taken by JikesRVM, on which Hera-JVM is built. As described in Section 5.3.5, JikesRVM uses an m-to-n green threading model, where multiple Java threads are mapped onto each per-core OS thread, and scheduled by the Java runtime system, rather than the underlying operating system. Each of these OS threads is known as a *virtual processor* and is pinned to a single core on the system.

Thread memory allocation requests are managed at the virtual processor level. When a thread requests memory, it is allocated from a block of the heap space that is reserved for exclusive use by the virtual processor on which it is executing. Since only one thread can allocate memory from this virtual processor's heap space at a time (thread switching is disabled on this virtual processor for the duration of the allocation), this removes the need for inter-thread synchronisation on memory allocations. A virtual processor's heap space can be expanded if an allocation request cannot be accommodated in its remaining capacity. This is achieved by performing an anonymous (i.e., non-file backed) memory map operation, through the `mmap` system call. This provides the virtual processor with an additional range of physically mapped virtual memory from which to allocate data[1].

This approach suggests a virtual processor centric approach to NUMA control for the runtime system. Each virtual processor is already bound to a particular NUMA node by virtue of its execution being pinned to a particular core. To provide the runtime system with knowledge of the virtual processors to NUMA node bindings, the native *system call* method, called during a virtual processor's initialisation to pin its execution to a core, was augmented to return the ID of the NUMA node on which this core is located. With this knowledge, the runtime system can control the NUMA node

[1]Requests by virtual processors to grow their heap space are protected by inter-thread synchronisation, since multiple virtual processors could try to grow their own heap space simultaneously and contend for the same range of virtual memory.

on which a thread executes or allocates memory, by restricting it to the set of virtual processors which are bound to that NUMA node.

To control the NUMA node on which data is allocated, Hera-JVM was modified to enable it to control the NUMA node of memory mapped virtual addresses when they are mapped to physical memory. The Linux libnuma library[1] can restrict the mapping of a given virtual address range to ensure that it is mapped to physical memory that is located on a particular NUMA node. Since a virtual processor expands its heap space by mapping additional virtual memory, this ability to restrict the physical pages on which virtual addresses are mapped enables the runtime system to restrict the virtual processor's heap space to memory on its own NUMA node. Thus, the NUMA node on which a thread's data is allocated can be limited to its local node, assuming a thread is only executed on that node's virtual processors.

However, not all data should be allocated locally. Some of the runtime system's data-structures are shared by all threads (e.g., class type information blocks and the shared *table of contents* data-structure are accessed by all threads). Machine code is also accessed by many different threads. If the shared data and machine code is stored on a single NUMA node, then contention at that system inter-connect is likely to occur. To spread this load across all NUMA nodes, the Hera-JVM was modified to *interleave* this data evenly across all NUMA nodes. This was achieved by using libnuma to mark the range of virtual memory addresses used for shared data and machine code, such that they are mapped to physical pages that are interleaved across the NUMA nodes of the system.

Another runtime system data-structure that should not necessarily be allocated on the local NUMA node is the stack for a newly created thread. The memory required for a new thread's stack will be allocated by its parent thread, before this new thread is started. There is no guarantee that this new thread's preferred NUMA node will be the same as the node on which its parent is executing when its stack is allocated. Since a stack is only ever accessed by its own thread, and is accessed very frequently by that thread, it should be allocated on the node on which this thread is most likely to be run. Thus, a newly created thread's preferred NUMA node is calculated (using the algorithm described in the following section) before its stack is allocated, such that its

[1] `http://oss.sgi.com/projects/libnuma/`

stack can be allocated on this preferred node, rather than the node on which its parent thread is currently executing.

A similar approach could be used to select the NUMA node on which data, which is itself tagged with the `@ThreadTeam` annotation, is allocated. Currently Hera-JVM does not treat annotated object declarations specially; however, this annotation could be used to enable a thread to allocate data on behalf of another thread or thread team. The runtime system could calculate the preferred node for a given object, based upon the thread team annotations attached to the object's declaration. This object can then be allocated on its preferred NUMA node, using a similar mechanism as is used to allocate threads' stacks. The implementation of off-node allocation is left as future work[1].

### 8.3.2 Applying a Cost to a Thread's Placement

Hera-JVM attempts to inform its thread placement decisions based upon both the structure of the non-uniform memory architecture on which it is executing and its knowledge of thread teams, as provided by the `@ThreadTeam` annotation. Its aim is to place threads which are members of the same team on cores in the same NUMA node, while aiming to place non-communicating threads on different NUMA nodes. To do this, it calculates a preferred ordering of NUMA nodes for any thread with one or more thread team annotations. This calculation is performed when a thread is created, based upon the current placement of the teams with which it will communicate. It is only performed for threads which have been annotated with thread team information; unannotated threads will not be assigned a preferred node order. The scheduler can use this preferred order to influence the core on which it will schedule the thread's execution. This section describes the algorithm used to calculate a thread's preferred node order placement, while Section 8.3.3 outlines how the Hera-JVM's scheduler uses this information to inform thread placement decisions.

Hera-JVM calculates a thread's preferred node order using a cost function, which evaluates the cost of placing a thread on each of the nodes on the system, based upon the thread teams of which this thread is a member. The cost of placing a particular thread on a node is based on two factors: the distance from this node to each of

[1]This off-node heap allocation mechanism can be simulated by employing a separate proxy thread, itself annotated in the same manner as the data, to allocate this data.

```
1  int calcPrefNodeOrder(Thread thread) {
2    int costs[] = new int [numberOfNodes];
3    for (node in nodes) {
4      for (threadTeam in threadTeams) {
5        if (thread.memberOf(threadTeam) {
6          costs[node] += node.distanceFrom(threadTeam.prefNode);
7        } else if (threadTeam.prefNode == node) {
8          costs[node] += threadTeam.numberOfThreads;
9        }
10     }
11   }
12   return nodeOrder(costs);
13 }
```

Listing 8.2: Pseudo-code of algorithm used to calculate the preferred NUMA node order of a thread.

the thread teams of which the thread is a member; and the presence of any non-communicating thread teams on this node. These two factors try to, respectively, cluster communicating threads onto the same or nearby NUMA nodes, while pushing apart non-communicating threads onto different NUMA nodes.

In order to calculate a node's distance from a thread team, the location of each thread team must be defined. To this end, each team is assigned a *preferred node*. A thread team is assigned a preferred node when it is first encountered by the runtime system. A thread's preferred node order is calculated when it is created, based upon the teams with which it is annotated. If this thread is a member of any thread teams that the runtime system does not yet know of, these new teams are not used in this calculation. However, once the thread's preferred node has been calculated, these new teams are assigned the same preferred node as this newly created thread. Once assigned, a thread team's preferred node does not change, but is used to influence the placement of newly created threads annotated as being a member of this team.

Listing 8.2 shows the algorithm used by Hera-JVM to calculate a thread's preferred NUMA node order. A cost is calculated for each node, by iterating over each thread team in the system. For each team of which the thread is a member (line 6), the node's cost is increased, based upon its distance from that team's preferred node. For the 4x4 core AMD Opteron NUMA system (Figure 8.1), this distance is defined as the number of HyperTransport hops over which inter-core messages between these two nodes must

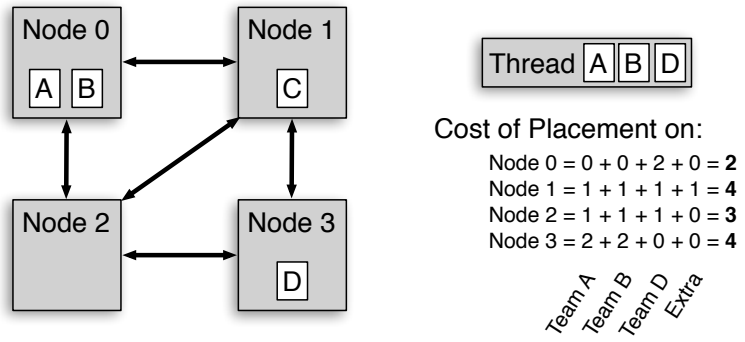



Figure 8.3: Using thread teams to place threads on a NUMA system.

traverse (this distance can be zero if the thread team is on the node currently being costed). This cost penalises placements that would place this thread on a NUMA node that is far away from the threads with which it communicates.

For thread teams of which this thread is not a member, but which are located on the node that is being costed (line 8), a cost is added based upon the number of threads in that team. The aim of this cost is to penalise a placement in which this thread is located on a preferred node of a team of which it is not a member. By making this cost equal to the number of threads in the team, this means that nodes which are lightly loaded with non-communicating threads are preferable to heavily loaded nodes.

Figure 8.3 shows an example of this algorithm in use. Once the cost of each node has been calculated, the nodes are ordered, based upon these costs. If two nodes have equal costs, a decision as to their order is taken randomly. In the case of Figure 8.3, this thread's preferred node order could be either 0, 2, 1, 3 or 0, 2, 3, 1, depending upon the random choice for the two nodes which have an equal cost.

In Hera-JVM, a thread's preferred node order is compressed into a single word sized integer. Since the NUMA system supported by Hera-JVM has only four nodes, two bits can be used to represent a node's ID. Thus, a preferred order of nodes can be represented as a sequence of these two-bit IDs, in the order required. Bit shifting and masking is used to select the ID of a node at a particular point in this order. If a system has too many nodes to make this approach practical, the state required to store a thread's preferred node ordering can be reduced by storing only the least costly subset of this order, and treating other nodes as equally expensive.

This algorithm has an algorithmic complexity of $O(n \times g)$, where $n$ is the number of nodes in the system and $g$ is the number of thread teams. This may appear to limit its scalability. However, this calculation is only ever performed once per thread: the preferred node order is used to inform scheduling decisions but is only calculated when the thread is created. Additionally, the number of nodes in a system is likely to be a fraction of the number of cores on the system. Similarly, the number of thread teams will be far fewer than the number of threads executed by a program. Thus, this complexity should not limit a system's scalability in practice.

### 8.3.3 Scheduling Threads with Per-Node Costs

The algorithm described in the previous section calculates a preferred ordering of nodes for each thread. However, the runtime system's scheduler must still decide upon the processing core on which to execute each of the program's threads. Blindly choosing to schedule a thread on a core in its preferred node will not always result in the best utilisation of all the processing cores available in the system. Such an approach could lead to the processing cores on a *popular* node becoming over-subscribed, while cores on other nodes lie idle. Instead, a thread's preferred node order is used to advise Hera-JVM's scheduling decisions, alongside other information, such as core utilisation.

The scheduling algorithm implemented by JikesRVM forms the basis of the NUMA node aware scheduler in Hera-JVM. Thread scheduling in JikesRVM is distributed across the processing cores of the system. Each processing core (represented by a virtual processor object) has its own local run-queue, from which it choses threads to execute. A thread is only in one virtual processor's run-queue at any time; however, it may be moved to a different virtual processor's run-queues by the runtime system in order to balance load across all the available processing cores. The thread placement decision is made by a method called `scheduleThread`. It is called whenever a thread is initially registered, becomes unblocked, or yields (either voluntarily or due to a timer tick), to choose the virtual processor on which the thread should be run when it is next scheduled. In JikesRVM, the primary aim of this method is to balance a program's processing load evenly across the available processing cores of the system. Hera-JVM, augments this `scheduleThread` method to try to preferentially group communicating nodes onto the same NUMA node, by taking each thread's preferred node order into account.

```
1  VirtualProcessor scheduleThread(Thread thread) {
2
3    // if the thread has a processor affinity, select it
4    if (thread.hasProcessorAffinity())
5      return thread.processorAffinity();
6
7    // if the thread is not on its preferred node, check if it can
8    // be transferred to an idle processor on its preferred node
9    if (thread.hasPreferredNodeOrder() &&
10       thread.getPreferredNode() != thisProc.numaNode) {
11     int prefNode = thread.getPreferredNode();
12     VirtualProcessor idleProc = getIdleProcessor(prefNode);
13     if (idleProc != null) {
14       return idleProc;
15     }
16   }
17
18   // if this thread is the only runnable thread on the current
19   // processor, leave the thread here
20   if (runQueue.isEmpty()) {
21     return this;
22   }
23
24   // try to find an idle processor
25   // check in preferred node order
26   for (prefOrder = 0; prefOrder < numNodes; prefOrder ++) {
27     int prefNode = thread.getPreferredNode(prefOrder);
28     VirtualProcessor idleProc = getIdleProcessor(prefNode);
29     if (idleProc != null) {
30       return idleProc;
31     }
32   }
33
34   // schedule round-robin on next processor to load balance
35   if (thread.hasPreferredNodeOrder())
36     // choose only from processors on the thread's preferred node
37     return nextProcessorOnNode(thread.getPreferredNode());
38   else
39     // choose from any processor
40     return nextProcessor();
41 }
```

Listing 8.3: Pseudo-code of Hera-JVM scheduling algorithm. The additions made to support NUMA aware scheduling are shown in red.

Listing 8.3 presents the `scheduleThread` method in pseudo code. The original algorithm used by JikesRVM is shown in black, with the changes added to enable NUMA node aware scheduling in Hera-JVM shown in red.

The approach taken by the original algorithm is relatively simple. If the thread has an affinity for a particular processing core (line 4), it is always scheduled on this processor. Otherwise, if this thread is the only thread in its virtual processor's run-queue (line 20), it is kept on its current virtual processor. If not, the thread is offloaded to another processor to attempt to load balance. If another processor is currently idle, the thread is scheduled on this idle processor (line 28). Otherwise, this load balancing is achieved by scheduling the thread on the next processor in a round-robin fashion (line 40).

To enable Hera-JVM to preferentially schedule a thread on its preferred node, this algorithm was augmented in three places. Firstly, a check is made as to whether a thread is currently executing on a core located on its preferred node (lines 9 to 16). If not, and there is an idle core available on this node, the scheduler will move the thread onto this idle core. This ensures that a thread will execute on its preferred node if the cores on that node are not over-subscribed.

Secondly, when attempting to balance the system's load by offloading a thread to another idle processor (line 28), the scheduler does not simply select the first idle processing core available. Instead, it checks each node for idle processors in the order preferred by the thread being scheduled. Thus, while a thread's preferred node order is taken into account, an idle core will always be given work if there are enough threads in the system, even if no threads prefer that core's node. This approach is the most appropriate, since a program's performance will suffer more by not fully utilising the available processing cores than it will from the additional inter-core communication overheads incurred by not placing its threads on their preferred nodes.

Finally, if the current core is over-subscribed and no other cores are idle, load balancing is performed by moving the current thread onto the next processing core located on the thread's preferred node (line 37). Thus, if no other cores are idle, the thread is kept on its preferred node, even if it moves to a different core for the purposes of load balancing. This approach provides intra-node load balancing for threads which have a preferred node, and retains full system load balancing for threads without a preferred node. However, some workloads may require an additional form of inter-node

load balancing for threads with preferred nodes. Such workloads were not encountered during the experiments performed using this scheduling algorithm, and so inter-node load balancing is left as future work.

These three additions are all conditional upon the thread that is being scheduled having a preferred node order. Any thread which has not been annotated with thread team information will be scheduled as it would have been under JikesRVM — based upon processing core load, without any preference for a core on a particular node.

## 8.4 Experimental Analysis

This section investigates the effectiveness of using thread team annotations to inform scheduling decisions and improve system-wide performance on NUMA architectures. The experimental setup for these experiments is described in Section 8.4.1. Section 8.4.2 examines the scalability improvements provided by scheduling threads on a NUMA architecture based upon thread team information. Finally, Section 8.4.3 examines Hera-JVM's effectivness at clustering threads that are members of multiple teams, as well as its sensitivity to thread team configurations that preclude optimal clustering of threads on NUMA nodes.

### 8.4.1 Experimental Setup

The experiments in this section are performed on a Dell PowerEdge 6950 server running Centos 5.4, with version 2.6.26 of the Linux kernel. The Dell PowerEdge 6950 has the NUMA architecture shown in Figure 8.1, with four NUMA nodes. The machine used in these experiments is equipped with four quad-core, x86-based, Opteron processors, each of which is its own NUMA node. This gives a total of sixteen processing cores, each clocked at 2GHz. Each NUMA node contains 4GB of RAM, providing the system with 16GB of memory overall.

A variation of the mandelbrot benchmark, first presented in Section 5.5.3, is used to investigate the performance of various thread and data placement strategies on this NUMA architecture. To increase the influence of non-uniform memory access latency on this benchmark's performance, it was modified, such that, instead of producing an $800 \times 600$ pixel image with a range of 200 colour levels, it draws a $12,288 \times 12,288$ pixel image with only 20 colour levels. The benchmark can spawn multiple threads,

which co-operate to draw a single image. Each thread computes the colour of a subset of the image's pixels. Since each thread has its own unique set of pixels to draw, no synchronisation is required for pixel updates. Therefore, the benchmark's scalability is primarily limited by the NUMA architecture's inherent scalability and the runtime system's choice of thread and data placement.

Three runtime system configurations, each with different thread and data placement strategies, are compared by these experiments. The first, Hera-JVM, employs the approach described in Section 8.3 — scheduling threads based upon the teams to which they belong. JikesRVM_Local is the default policy employed by JikesRVM, with thread scheduling **not** taking the system's NUMA architecture into account and data being allocated on the NUMA node on which the thread requesting this allocation is executing. Finally, the JikesRVM_Interleave configuration uses the numactl tool[1] to force memory allocations to be interleaved across all NUMA nodes, no matter which thread requests the memory.

Other than the additions described in Section 8.3, Hera-JVM is identical to version 3.0 of JikesRVM, with which it is compared in these experiments. Both runtime systems are built using the production configuration of JikesRVM, and so employ the optimising compiler to recompile *hot* methods[2].

All experimental runs were performed ten times, with the average being reported and the standard deviation between these runs shown using error bars. The selection of ten runs was chosen as it was found to be sufficient to lead to a stable average and standard deviation.

### 8.4.2 Scalability

To investigate the effectiveness of thread team awareness on a runtime system's ability to scale an application across a NUMA architecture, the mandelbrot benchmark was executed under the Hera-JVM, JikesRVM_Local and JikesRVM_Interleave runtime system configurations, while being scaled from one to sixteen co-operating threads.

Figure 8.4 shows the performance of this benchmark as it is scaled across the cores of the system, measured as the number of $12,288 \times 12,288$ pixel mandelbrot images

[1] `http://oss.sgi.com/projects/libnuma/`

[2] This differs from the previous chapters, where only the baseline compiler was used, due to the lack of an optimising compiler for the Cell processor's SPE core types.

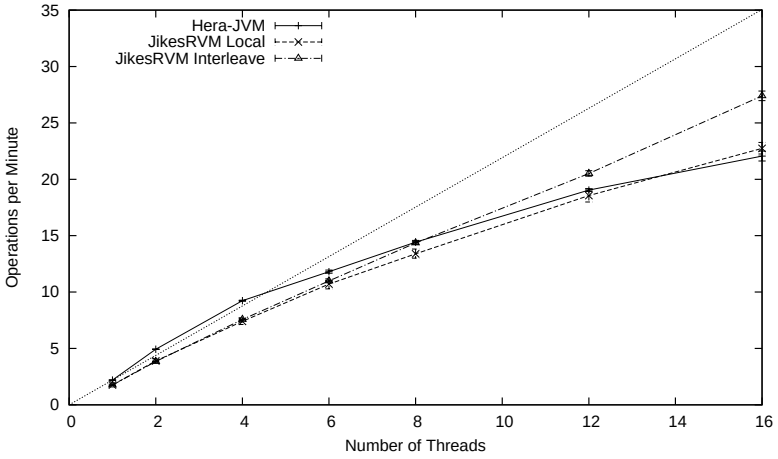
that can be generated by the benchmark per minute. Scalability is investigated under different configurations of thread co-operation. Figure 8.4(a) shows the scaling when all the threads are annotated as being members of the same thread team. For this configuration, the image's data is allocated as a single block, with all threads co-operating towards the drawing of the whole image. Figure 8.4(b) shows scalability when the threads are split into two equal teams. Threads only co-operate towards the drawing of the half of the image allotted to their team. Finally, Figure 8.4(c) shows the result of splitting the threads into four equal teams and allotting a quarter of the image to each team.

With only a single team of threads (Figure 8.4(a)), the image's data is allocated by a single thread. Thus, both Hera-JVM and JikesRVM_Local allocate the entirety of the image's data on a single NUMA node (the node on which the thread performing the allocation is executing at the time). Since the threads are all members of the same thread team, Hera-JVM attempts to cluster the threads on the same NUMA node as the image's data. This leads to better scaling of the application up until four threads[1], since all the threads can be executed by cores on the same NUMA node. When scaled to more than four threads, some of the application's threads must be placed on a different NUMA node than the node hosting the image's data, and must therefore access this data remotely. Since the data is on a single NUMA node, these remote accesses contend at that NUMA node, leading to a regression of Hera-JVM's scalability compared to that of JikesRVM_Local by sixteen threads.

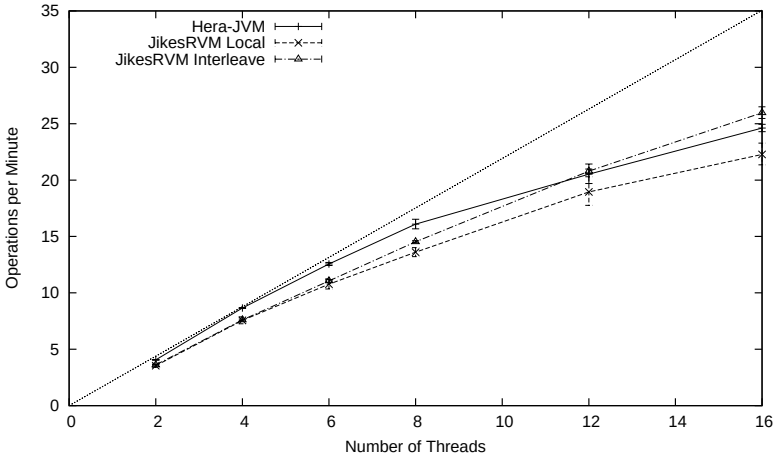
The JikesRVM_Interleave strategy automatically interleaves the image's data across all four NUMA nodes. This leads to inferior scaling when the threads could be executed on a single node; however, by sixteen threads, JikesRVM_Interleave performs better than the other approaches, since the data is spread across all the NUMA nodes and remote access to a single node is avoided, preventing this bottleneck.

When the image is split into fractions and each fraction is drawn by an independent team of threads (Figures 8.4(b) and 8.4(c)), the benchmark's scalability under Hera-JVM improves. Hera-JVM's scheduling algorithm attempts to place threads from different teams on different NUMA nodes. Therefore, each team is assigned a different

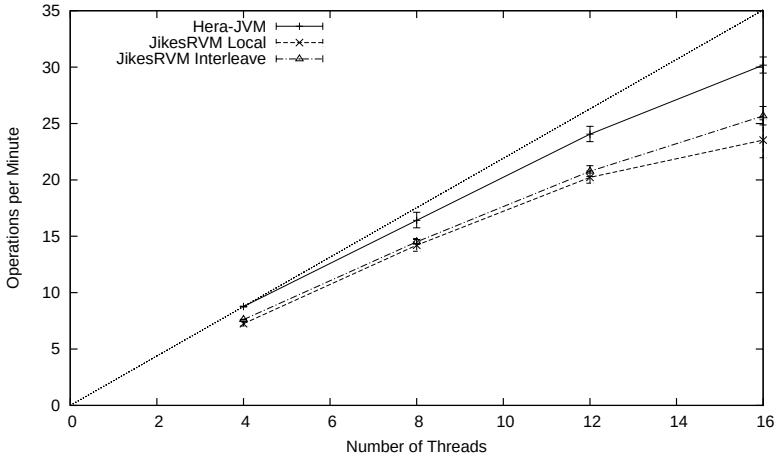
[1]The mandelbrot benchmark actually scales super-linearly up until four threads under Hera-JVM, likely due to beneficial cache effects relating to all four cores sharing an L3 cache.




(a) One thread team.




(b) Two thread teams.




(c) Four thread teams.

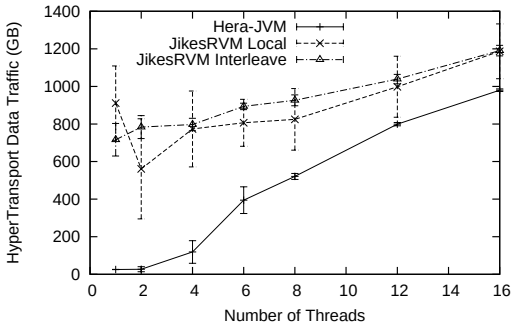
Figure 8.4: Scaling performance of mandelbrot benchmark on a NUMA system (higher is better). The dotted diagonal line shows perfect linear scaling.

preferred NUMA node on which its fraction of the image's data is allocated. Hera-JVM attempts to cluster a team's threads on its preferred NUMA node, close to the fraction of the image's data that they access. With two thread teams (Figure 8.4(b)), the image's data is spread across two NUMA nodes. Hera-JVM's thread team aware scheduling means that this configuration scales better under Hera-JVM than JikesRVM_Local and JikesRVM_Interleave up to eight threads, after which it is necessary to schedule threads on a non-preferred NUMA node. However, Hera-JVM continues to scale better than the default JikesRVM_Local strategy after this point. When the benchmark's work is split between four thread teams (Figure 8.4(c)), Hera-JVM can allot a thread team to each of the four NUMA nodes of the system. Thus, the benchmark scalability is significantly improved under Hera-JVM, having 28% better performance than JikesRVM_Local and 17.5% better performance than JikesRVM_Interleave at 16 threads.

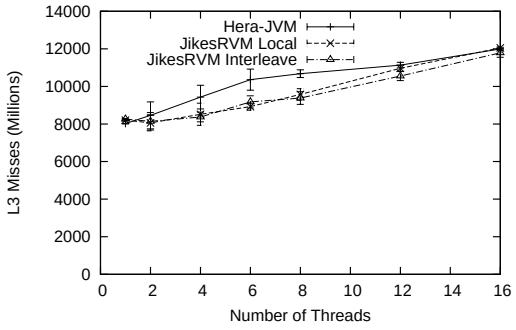
To verify that Hera-JVM's improved scalability is due to a reduction in inter-core data traffic, version 0.96 of OProfile was used to measure system-wide inter-core HyperTransport data traffic and L3 cache misses during the benchmark's execution[1]. Figure 8.5 shows the system-wide HyperTransport data traffic ((a), (c) and (e)) and L3 cache misses ((b), (d) and (f)) that result from running the previously described benchmark configurations under Hera-JVM, JikesRVM_Local and JikesRVM_Interleave, respectively.

The volume of inter-core HyperTransport traffic, generated by the benchmark, is consistently less when it is executed under Hera-JVM than under either of the JikesRVM configurations. This supports the premise behind Hera-JVM's scheduling algorithm — that laying out data and threads based upon thread team information reduces inter-core data traffic. However, as demonstrated by these figures, a lower system-wide level of inter-core traffic does not always lead to better performance. With one team of sixteen threads, the benchmark performs better under JikesRVM_Interleave than it does under Hera-JVM, even though the system-wide HyperTransport traffic is lower under Hera-JVM. This occurs because, under JikesRVM_Interleave, the traffic is evenly spread across all the links in the system, whereas under Hera-JVM, all the traffic is

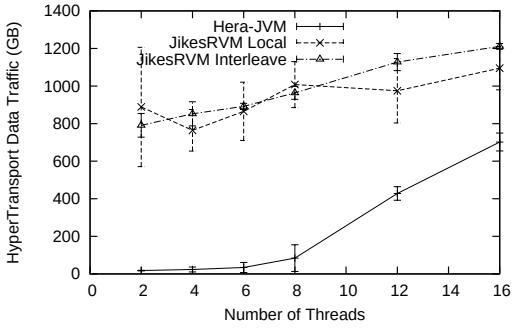
[1] OProfile uses hardware performance counters provided by the AMD Opteron processors to measure these metrics.



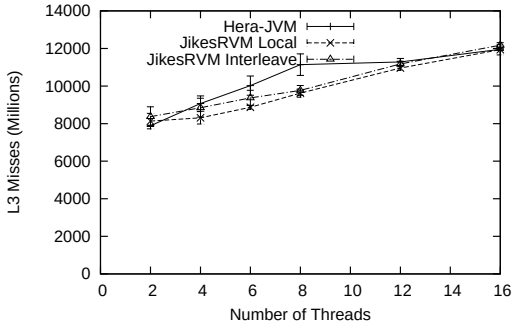

(a) One team - HyperTransport traffic.



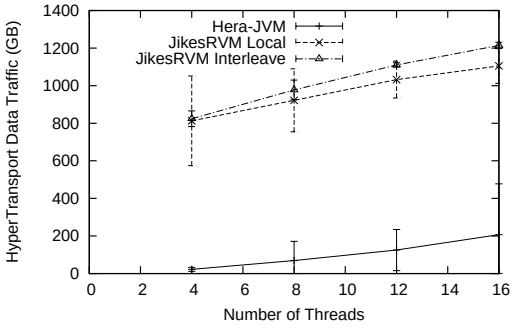

(b) One team - L3 cache misses.



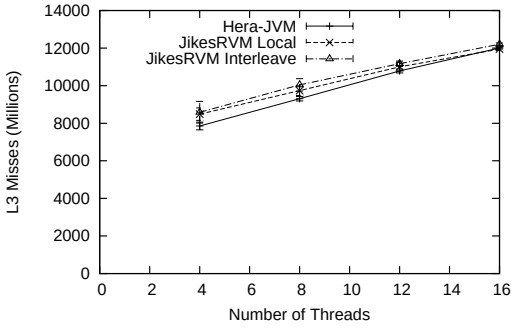

(c) Two teams - HyperTransport traffic.




(d) Two team - L3 cache misses.




(e) Four teams - HyperTransport traffic.




(f) Four team - L3 cache misses.

Figure 8.5: HyperTransport inter-core data traffic and L3 cache misses for mandelbrot benchmark on NUMA system (lower is better).

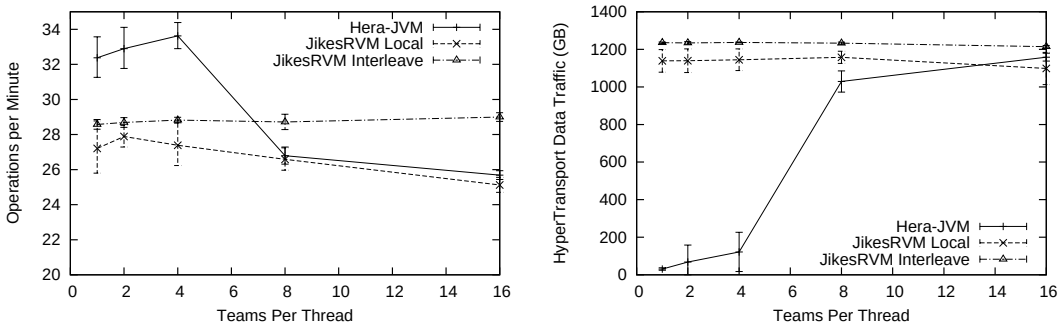
directed towards the NUMA node holding the benchmark's image data, leading to a single bottleneck.

The L3 cache miss count results reveal another somewhat counter-intuitive behaviour. When the benchmark is configured with one or two thread teams, and less than 8 threads, Hera-JVM provides better performance than JikesRVM, even though it incurs a higher cache miss rate. The reason for this becomes clear when its thread placement decisions are taken into account. Hera-JVM attempts to cluster threads of the same team onto the same NUMA node. Since the four cores of a NUMA node share a single L3 cache in the architecture used by these experiments, this clustering of threads leads to higher L3 cache contention, and therefore higher miss rates. However, since these cache misses can be serviced by the node's local memory, they have less impact than the high latency remote memory cache misses incurred under JikesRVM. This illustrates that lower cache miss rates do not always lead to higher performance. If, as in a NUMA architecture, the location of data affects the time required to service a cache miss, then thread and data placement can have a more significant performance impact than cache miss rates on an application's performance.

### 8.4.3 Multiple Teams per Thread

The experiments in the previous section involved threads that were a member of at most one thread team; however, the approach described in this chapter enables a thread to be annotated as being a member of multiple thread teams. To investigate Hera-JVM's effectiveness at clustering threads that are members of multiple thread teams, the mandelbrot benchmark was modified as follows. The generated image is split into sixteen fractions, with each fraction being associated with a particular thread team. Sixteen threads are started, with each thread being a member of $x$ thread teams, where $x$ can be varied between one and sixteen. A thread then contributes towards drawing $\frac{1}{x}$ of each of the image fractions with which it is associated, through the thread teams of which it is a member. Therefore, the same amount of work is performed by the same number of threads throughout these experiments. However, the degree of co-operation between these threads can be varied by changing the number of teams to which each thread is assigned.

The assignment of teams to threads was varied randomly between runs, with the same set of random assignments being run under all three runtime configurations. The



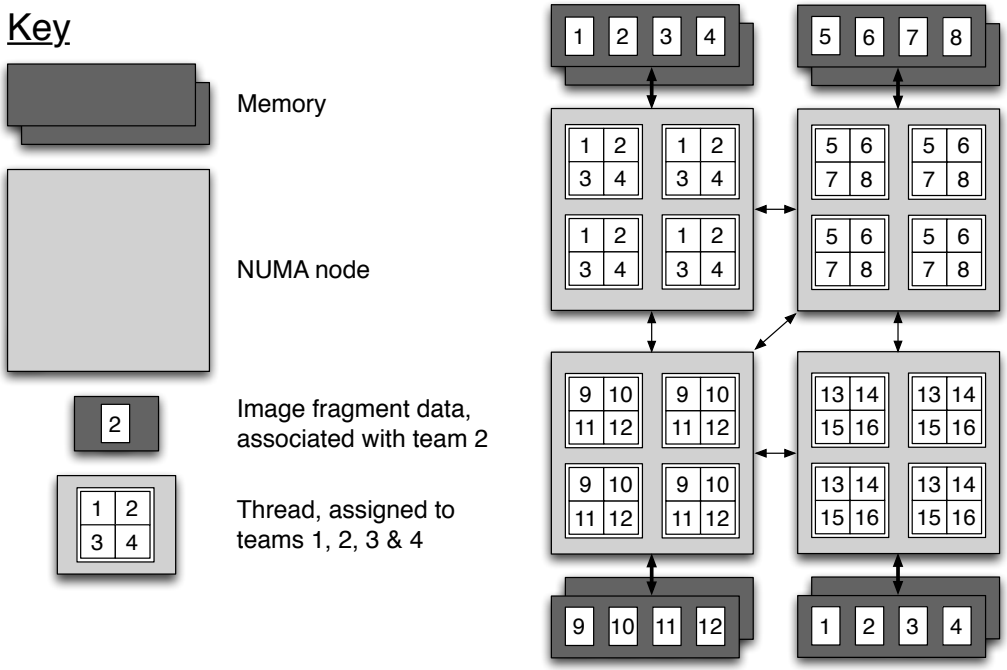

(a) Performance.

(b) HyperTransport inter-core data traffic.

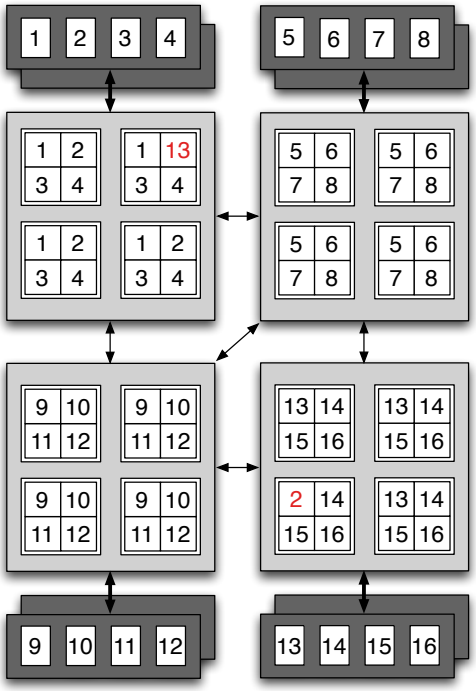
Figure 8.6: The performance of the mandelbrot benchmark on a NUMA architecture as the number of thread teams assigned to each thread is varied.

team assignments were constrained such that threads can be clusterable into $\frac{1}{x}$ groups, with all $x$ threads in each group being members of the same $x$ thread teams (Hera-JVM's sensitivity to less optimal thread team assignments is investigated subsequently). To create each pseudo-random team assignment, $x$ team names are randomly selected and assigned to $x$ randomly chosen threads. These threads and team names are removed from the sets from which they are chosen, and the process is repeated until no threads or team names remain to be assigned.

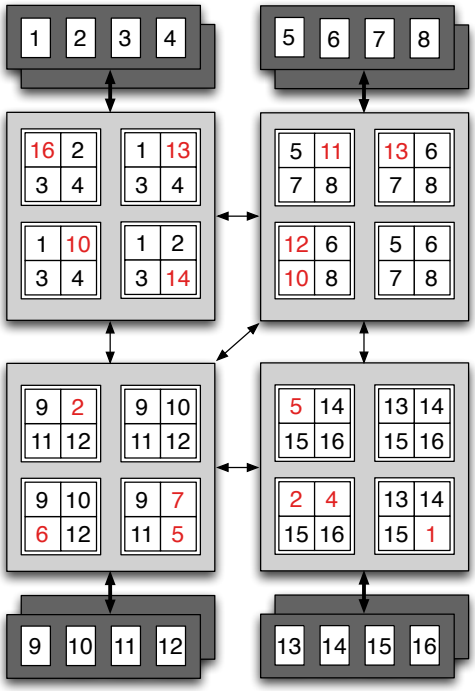
Figure 8.6 shows the level of performance and volume of HyperTransport traffic that results from running this benchmark under Hera-JVM, JikesRVM_Local and JikesRVM_Interleave, as the number of teams with which each thread is associated is varied. These results show that Hera-JVM can support clustering of multi-team threads on NUMA nodes in order to realise greater performance than either of the JikesRVM configurations. However, once more than four thread teams are assigned to each thread, co-operating threads can no longer be clustered on a single NUMA node (more than four threads co-operate; therefore, since there are only four processing cores on each NUMA node, some of the co-operating threads must be placed on a different NUMA node). After this point, Hera-JVM's performance reverts to that of JikesRVM_Local. This demonstrates that the use of thread teams enables Hera-JVM to improve an application's performance by clustering co-operating threads, but only if the structure of co-ordination between an application's threads permits clustering.




(a) Zero swaps - 100% clusterable

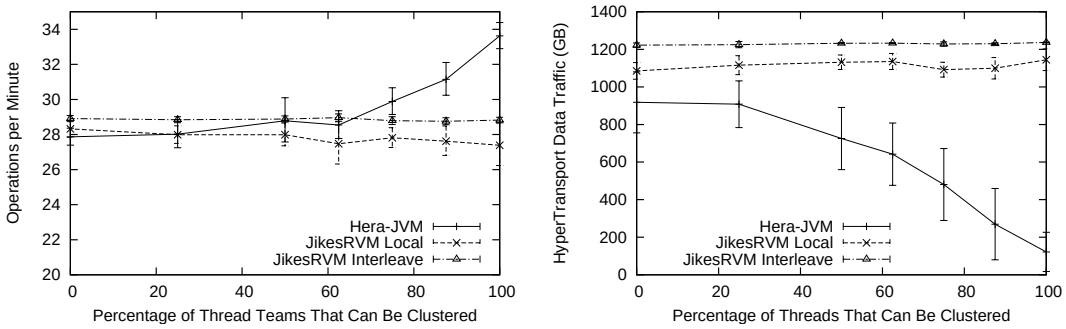


(b) One swap - 97% clusterable



(c) Eight swaps - 75% clusterable

Figure 8.7: Example of reducing the clusterablity of thread team assignments by shuffling thread / team assignments. Each number represents a thread team name (and associated image fragment). Thread / team assignment swaps are highlighted in red.




(a) Performance.

(b) HyperTransport inter-core data traffic.

Figure 8.8: The performance of the mandelbrot benchmark on a NUMA architecture, with each thread assigned to four thread teams, as the proportion of threads that can be clustered on a single NUMA node is varied.

To investigate Hera-JVM's sensitivity to non-optimal thread team assignments, the algorithm used to randomly assign teams to threads in the mandelbrot benchmark was modified such that teams are **not** always assigned with $\frac{1}{x}$ groups of $x$ threads that share the same $x$ thread team assignments. A second phase was added to the random team assignment algorithm, in which a set number of randomly chosen thread / team assignments are shuffled. By increasing the number of assignments that are shuffled during this stage, the benchmark's work can be made less clusterable. Figure 8.7 shows the effect of this assignment shuffling on the potential clustering of threads and image fragment data onto NUMA nodes.

Figure 8.8 shows the effect on the performance of the mandelbrot benchmark when the proportion of threads that can be clustered is varied. The benchmark is configured with sixteen threads, each of which is assigned to four thread teams. As expected, the less potential there is to cluster co-ordinating threads onto the same NUMA node, the lower the performance gains achieved by Hera-JVM compared with JikesRVM. However, the performance advantage of Hera-JVM degrades gracefully, with some benefit provided by Hera-JVM's approach as long as at least 50% of the thread teams can be clustered. This shows that the algorithm employed by Hera-JVM can improve performance on NUMA architectures even when an application's inter-thead communication patterns are relatively irregular.

## 8.5 Summary

This chapter examined the use of behaviour characteristics as a means of improving performance on a heterogeneous multi-core architecture that has very different characteristics from the Cell processor, used in Chapters 5 to 7. Specifically, this chapter investigated the use of thread team annotations as a means of expressing inter-thread communication patterns. This thread team information is used by the runtime system to inform its thread and data placement decisions in order to improve an application's performance under a NUMA architecture.

The experiments presented in Section 8.4 demonstrate that, by taking thread teams into account when choosing the core on which a thread should be scheduled, Hera-JVM can reduce inter-core data traffic and, consequently, improve application performance. Performance of a multi-threaded mandlebrot benchmark can be improved by up to 28% under Hera-JVM, compared to a runtime system which is unaware of threads' communication patterns.

However, the use of thread teams does not completely eliminate the need for an application to be structured in a manner amenable to scaling on a NUMA architecture. For Hera-JVM's approach to be effective, an application's shared data must be partitionable across multiple NUMA nodes, and its threads clusterable, such that inter-thread communication outwith a NUMA node can be limited. When an application is not structured in a NUMA-friendly fashion, the performance provided by Hera-JVM reverts to that of the JikesRVM_Local configuration, on which it is based. Under these circumstances, simply interleaving an application's data evenly across all of the NUMA nodes on the system (JikesRVM_Interleave) can provide better performance. This suggests that the most appropriate approach may involve a hybrid scheme, in which thread and data placement is based upon thread team annotations when threads can be clustered appropriately, but reverts to automatic fine grained interleaving of memory pages for less NUMA-frendly application structures.

Further experimental analysis on the use of thread teams, using more realistic benchmarks, is required before its effectiveness on real-world applications can be established. However, the results presented in this chapter provide a first step in understanding where this approach can be useful in improving an application's performance on a

NUMA architecture. This work also enabled investigation into the effectiveness of behaviour characteristics as a means of abstracting a very different heterogeneous multi-core architecture from the Cell processor, that was investigated in the previous chapters. Thus, it provides confidence that using behaviour characteristics to inform a runtime system's thread and data placement decisions can improve application performance under a variety of different heterogeneous multi-core architectures.

# Chapter 9

# Conclusion and Future Work

A number of modern computer systems have been designed with multiple different types of processing cores, each of which has different capabilities and drawbacks. This trend towards increasing processing core heterogeneity is likely to continue and find its way into commodity computer systems, thanks to the proliferation of graphics processing units and the increasing number of processing cores employed by commodity processors. While these heterogeneous multi-core architectures have the potential to provide significantly better performance and efficiency, compared to homogeneous architectures, it is notoriously difficult to develop applications that can exploit this potential. Non-specialist application developers must be provided with better abstractions and runtime support if they are to take advantage of these architectures.

The goal of this work is to provide a runtime system that ameliorates many of the difficulties encountered in application development on heterogeneous multi-core architectures through abstraction, thereby enabling non-specialist application developers to exploit their potential, without requiring in-depth knowledge of their design.

## 9.1 Thesis Statement Revisited

In this section, the thesis statement that was presented at the start of this dissertation is revisited in light of the work presented by this dissertation. This thesis statement is restated below:

> "Given the difficulties involved in programming and managing heterogeneous multi-core architectures (HMAs), their use is currently limited to specialist applications. I assert that a homogeneous multi-core virtual machine abstraction can be employed to reduce the burden of developing applications for HMAs, while still enabling the disparate processing resources of an HMA to be exploited. By tracking a program's behaviour, a runtime system can make informed thread and data placement decisions, enabling the program to make effective use of heterogeneous processing resources. By employing program behaviour characteristics to guide the partitioning of a program's execution across heterogeneous processing cores, application developers do not require in-depth knowledge of an HMA's design in order to exploit it effectively."

To prove this assertion, a behaviour-aware runtime system was created and its effectiveness at enabling applications to exploit the performance provided by two different HMA systems, with minimal developer effort, was investigated.

Chapter 4 presented the overall approach for abstracting heterogeneous processors proposed by this work. This involves hiding the heterogeneous nature of an HMA processor behind a homogeneous virtual machine abstraction and having a runtime system map a program's execution onto the most appropriate core types, taking the program's behaviour into account. A set of behaviour characteristics were proposed that capture the aspects of a program's behaviour most likely to influence its execution performance on different core types. These characteristics include a program's processing requirements, execution behaviour and inter-thread communication patterns. The chapter proposed a variety of means for inferring a program's behaviour characteristics, including explicit code annotations, static code analysis and runtime monitoring. Finally, the use of cost functions, to enable a runtime system to make informed thread and data placement decisions, based upon a program's behaviour characteristics, was presented.

To investigate this premise, a Java runtime system called Hera-JVM was developed. In Chapter 5, the techniques used to enable Hera-JVM to provide a homogeneous virtual machine abstraction on the highly heterogeneous IBM Cell processor were presented. These include the provision of the same Java virtual machine interface on both of the Cell processor's core types, transparent migration between these cores and the

provision of efficient support of the Java memory model under the unusual memory hierarchy of the Cell processor's SPE cores, using a type-aware software caching system. This support enables Hera-JVM to execute unmodified Java programs on either of the Cell processor's core types and have both core types co-operate in the execution of a single application. To back up this claim, a number of real-world benchmarks were executed on both core types under Hera-JVM, thus ensuring that the heterogeneous cores produce consistent results for real-world applications. This verifies the first part of the thesis statement — that a heterogeneous multi-core architecture can be successfully hidden behind a homogeneous virtual machine interface.

Chapter 5 also investigated the effectiveness of the software caching scheme that Hera-JVM employs to hide the SPE core type's unusual memory hierarchy, using both synthetic micro-benchmarks and real-world benchmarks. These experiments show that, when a program's working set could fit in the SPE's local memory, this scheme was very effective. In fact, for read-only operations, this software caching scheme can outperform the hardware cache of the Cell processor's PPE core. However, its performance drops for write-heavy workloads[1], or workloads in which the program's working set does not fit in the SPE's local memory.

Finally, the performance of the two core types of the Cell processor, under a variety of different program behaviours, were characterised using both synthetic micro-benchmarks and real-world benchmarks. This information was used to uncover the set of behaviour characteristics that have the most influence on the relative performance that a program can expect to achieve on either core type. This analysis showed that the Cell's SPE cores are much better suited to arithmetic computations (no matter whether these involved integer, floating point or bitwise-based operations), while the PPE core has a slight advantage when accessing data objects, and a significant advantage if a program's working set cannot fit in the SPE core's local memory cache.

In Chapter 6, Hera-JVM was extended so that it can be made aware of a program's behaviour, through explicit annotations in the program's code. Hera-JVM uses this behavioural information, alongside its knowledge of the performance characteristics of each of core type on the system, to select the most appropriate core type on which to execute each of the program's threads and execution phases.

[1] However, a preliminary investigation into the use of a write-back caching scheme suggests it may be possible to significantly improve the software cache's write performance.

A cost function was developed to enable the runtime system to decide whether a thread should be migrated to a different core type, based upon its current behaviour characteristics. It was found that by incorporating past history, hysteresis and trend tracking into this cost function, its stability can be increased, leading to a reduction in the number of detrimental migrations. A number of different migration strategies (**AtAnnotation**, **AfterSched** and **Targeted**) were proposed and implemented in the Hera-JVM runtime system. These enabled investigation into the trade-off between immediacy in reacting to behaviour changes, against making more informed migration decisions. The **Targeted** migration strategy was found to be the most effective, due to its ability to target a suitable *long-lived* method for migration, and thereby avoid migrating methods that are too short to benefit from execution on a different core type.

To evaluate the efficacy of thread placement based upon behaviour characteristics, a number of real-world benchmarks were annotated with their behaviour characteristics and executed under Hera-JVM. The runtime system was able to use this behavioural knowledge to automatically migrate each benchmark's threads across the Cell processor's heterogeneous cores. Doing so, it can achieve performance that is comparable to that of manual partitioning of the benchmark, based upon specialist knowledge of the capabilities of the Cell processor's different core types. This validates the second part of the thesis statement — that a behaviour-aware runtime system can make informed thread and data placement decisions that enable a program to transparently exploit heterogeneous processing resources.

To further reduce the burden of application development on HMA processors, Chapter 7 investigated the use of runtime monitoring to automatically infer a program's behaviour characteristics as it executes, without the need for explicit code annotations. A lightweight runtime behaviour monitoring system was developed, that enables Hera-JVM to automatically measure the proportion of arithmetic and object access operations being performed by a program's threads throughout their execution. The overhead of this runtime monitoring system was reduced by *scoring* code blocks as they are compiled and updating a thread's behaviour using these aggregate scores. The cost function and migration strategies employed by the annotation-based behaviour-aware runtime system were reused to enable Hera-JVM to base its thread migration decisions upon the behaviour characteristics that it monitors at runtime. Chapter 7 also examined combining annotation-based behaviour characteristics with those monitored

at runtime, to provide additional program behaviour information to the runtime system. This process was simplified by the fact that the same cost function approach is shared by both the annotation-based and runtime monitoring-based behaviour-aware approaches.

A number of unmodified real-world benchmarks were executed under Hera-JVM, to evaluate the effectiveness of runtime behaviour monitoring in enabling automatic thread placement and migration decisions on the Cell Processor. These experiments showed that runtime monitoring could be as effective as explicit behaviour annotations in this regard, subject to a small (3-10%) runtime monitoring overhead. This validates the third part of the thesis statement — that a behaviour-aware runtime system can eliminate the burden of application development on an HMA processor, while still enabling applications to effectively exploit its potential performance.

Finally, Chapter 8 performed an initial investigation of the use of thread team annotations as a means of improving an application's performance under a NUMA architecture. When a thread is created, a cost function is used to evaluate its preferred NUMA node ordering, based upon the teams of which it is a member. Hera-JVM uses this information to influence its scheduling decisions, such that communicating threads are clustered onto the same NUMA node, thereby reducing inter-node data traffic and improving overall performance.

By examining the use of behaviour characteristics to improve performance on a heterogeneous multi-core architecture that has very different characteristics from the Cell processor, the work in Chapter 8 increases confidence that the assertions made in the thesis statement apply under a variety of different heterogeneous multi-core architectures.

## 9.2 Contributions

This work contributes towards the abstraction of heterogeneous multi-core architectures in the following ways:

- **Hiding a heterogeneous multi-core architecture behind a homogeneous virtual machine abstraction**

  This work demonstrated the abstraction of heterogeneous multi-core architectures by hiding their heterogeneity behind a homogeneous virtual machine architecture,

using the Hera-JVM runtime system. By providing the same virtual machine interface on heterogeneous core types and transparent migration between core types, an application's execution can be spread across the heterogeneous cores of an HMA processor, without requiring any knowledge of the architecture's heterogeneity from the application. The feasibility of this approach was evaluated by executing a number of unmodified real world benchmarks across the heterogeneous core types of the Cell processor.

- **Behaviour-aware runtime system**

  A set of behaviour characteristics was presented, that enables a runtime system to make effective use of heterogeneous processing cores, without burdening application developers with details of an HMA's underlying design. Three sets of behaviour characteristics were defined, namely, processing requirements, execution behaviour and inter-thread communication. The use of both explicit code annotations and runtime monitoring, as a means of providing this behaviour information, was explored. Finally, the use of these behaviour characteristics, to inform a runtime system's thread and data placement decisions, was investigated under two distinct heterogeneous architectures — the Cell processor and an x86 NUMA machine.

- **Behaviour characteristic cost function**

  A cost function based approach for informing thread placement and migration decisions, based upon a program's behaviour characteristics, was presented. The effectiveness of history, hysteresis and trend tracking, as means of improving the cost functions' stability, was investigated and found to be beneficial in reducing unproductive migrations and improving overall application performance.

- **Migration strategies**

  A number of strategies for triggering a thread's migration from one core type of the Cell processor to the other were explored. A targeted migration strategy, which enables a method to be modified at runtime, such that it automatically triggers a thread's migration whenever it is invoked, was found to be the most effective. By scanning a thread's call stack and targeting a suitable *long-lived* method for migration, the overheads incurred by thread migration can be reduced.

- **Lightweight program behaviour runtime monitoring system**

  A lightweight runtime monitoring system was developed, that enables a runtime system to automatically infer a program's behaviour during its execution. By measuring the proportion of a thread's execution spent performing different types of computation, the runtime system can automatically select the most appropriate core type on which to execute the thread and, if necessary, when to migrate it. The use of explicit behaviour characteristic code annotations, alongside runtime monitoring, was also demonstrated.

- **Inter-thread communication aware scheduling on NUMA architectures**

  An initial investigation was carried out into the use of thread team behaviour characteristics, as a means of expressing inter-thread communication patterns to a runtime system. This knowledge was employed by the runtime system to enable it to automatically optimise thread and data layout on a NUMA architecture. This was found to improve the performance of programs that can be partitioned across the nodes of a NUMA machine, by reducing the volume of inter-NUMA node traffic.

- **A Java compiler for the Cell's SPE core type**

  The creation of a Java compiler for the SPE core type of the Cell processor was required for this work. This involved the development of a software caching scheme that exploits high level type information to reduce the overhead of DMAing data between main memory and the SPE core's local memory, while respecting the Java memory model.

## 9.3 Future Work

This dissertation has shown that a behaviour-aware runtime system can be used to reduce the burden of developing applications that exploit a heterogeneous multi-core architecture. Yet, there is considerable scope for future work that builds upon the work presented in this dissertation. In this section, some of these opportunities for future work are outlined.

### 9.3.1 Other Heterogeneous Architectures

This dissertation investigated the use of a behaviour-aware runtime system under two different heterogeneous multi-core architectures — the Cell processor and a NUMA architecture. There are many other examples of heterogeneous multi-core architectures that could benefit from this approach. An interesting area of future work would be to examine the effectiveness of behaviour characteristics at enabling a runtime system to exploit the performance potential of other HMA systems.

One of the most interesting systems on which to investigate the use of a behaviour-aware runtime system would be a processor that consists of both CPU and GPU (graphics processing unit) core types. GPUs are becoming prevalent in commodity PC systems and there are a number of proposed processor designs that will incorporate GPU cores into a multi-core processor (AMD, 2008; Nvidia, 2009*b*; Seiler *et al.*, 2008). Given their ubiquity and high floating point performance, GPU's have the potential to greatly improve the performance of many general purpose applications. However, this performance can only be exploited by programs that perform certain types of computation (e.g., highly data-parallel computations). In addition, making use of a GPU for general purpose computation currently requires adapting the computation to fit a streaming processing model (Buck *et al.*, 2004; Munshi, 2009; Ryoo *et al.*, 2008).

Extending this work to enable the exploitation of GPUs by general purpose applications will involve tackling a number of additional challenges. To take advantage of all of the processing cores on a GPU requires a highly parallel computation. In addition, the different processing cores of a GPU often share functional units, which makes these architectures highly sensitive to branching program control flow (Fung *et al.*, 2007). For a runtime system to successfully hide these challenges from an application developer, while still providing the full performance of the GPU architecture, will require further research into areas such as automatic parallelisation and ahead-of-time branch prediction techniques.

### 9.3.2 Other Behaviour Characteristics

This work concentrated on evaluating the effectiveness of four behaviour characteristics in informing a runtime system's thread placement decisions, namely `@ArithmeticCode`,

`@ObjectAccessCode`, `@LargeWorkingSet` and `@ThreadTeam`. A worthwhile area of future work would be to investigate whether the other behaviour characteristics, proposed in Chapter 4, are also effective in guiding a runtime system's resource allocation decisions.

The behaviour characteristics that were evaluated during this dissertation were chosen because they were expected to provide the most benefit for the Cell processor and NUMA architecture on which Hera-JVM was evaluated. However, a runtime system that executes under a different heterogeneous multi-core architecture is likely to benefit from knowledge of a different set of program behaviour characteristics. For example, as discussed above, GPU processing cores are typically very sensitive to branching program control flow. Therefore, a behaviour characteristic that identifies code which is likely to have highly branching control flow would be beneficial to a runtime system that executes on an HMA with both CPU and GPU type processing cores. It is likely that additional behaviour characteristics will be identified by extending this work to other architectures.

### 9.3.3 Tagging Data with Behaviour Characteristics

The focus of this work involved tagging code with its expected behaviour characteristics; however, there are a number of situations in which it may also be beneficial to have the ability to tag data with behaviour characteristics. As discussed in Chapter 8, the `@ThreadTeam` behaviour characteristic could be applied to a piece of data to signal that it is likely to be accessed by threads from that team more often than the thread which allocated the data. This information could be used by a runtime system to select the most appropriate memory location (or NUMA node) on which to allocate the data, such that the application's overall memory access time is minimised.

The `@SequentialAccessBehaviour` and `@RandomAccessBehaviour` characteristics could also be used to tag data, in addition to code. If a runtime system is provided with ahead-of-time knowledge of a data-structure's expected access patterns using these behaviour characteristics, it could optimise the strategy it uses to cache the data-structure. If a data-structure is accessed sequentially, a prefetching caching strategy is very effective in reducing memory access times (Smith, 1978). On the other hand, if a large data-structure is accessed randomly, it may be non-beneficial or even detrimental to cache its elements as they are accessed. This is because these elements are likely

to have been evicted from the cache before they are next accessed, meaning these cached copies only serve to pollute the cache. If a runtime system were provided with the expected access behaviour of a given data-structure, it could use a variety of mechanisms to influence the data-structure's caching strategy, such as inserting data prefetch instructions or marking memory regions as non-cacheable.

There are a number of challenges which must be tackled before a runtime system can be provided with knowledge of data-structure's behaviour characteristics. A mechanism must be provided to enable data-structures to be tagged with their behaviour characteristics. One option would be to enable behaviour characteristic annotations to target data-variable declarations. Data-structures then inherit the behaviour characteristics of the variable to which they are assigned. However, data aliasing issues could complicate this approach — if a data-structure is pointed to by more than one variable, it may inherit conflicting behaviour characteristics from each of these variable's declarations. Further work is required to overcome these challenges and investigate the effectiveness of using data behaviour characteristics to inform runtime system operations.

### 9.3.4 Inferring Behaviour Characteristics Through Static Analysis

In this work, both explicit code annotations and runtime monitoring were investigated as a means of providing a runtime system with information about a program's behaviour. Another option would be to provide source code analysis tools that can infer a program's behaviour ahead of its execution, and automatically insert appropriate annotations into its source code. The use of ahead-of-time program analysis tools would relieve application developers from having to explicitly annotate their programs, while avoiding the overheads incurred by monitoring a program's behaviour at runtime.

Section 4.3.2 discussed a number of static analysis techniques which could be employed to uncover a program's behaviour characteristics, such as data-flow analysis (Allen & Cocke, 1976), loop bounding analysis (Healy *et al.*, 1998) and escape analysis (Choi *et al.*, 1999). An interesting area of future research would be to investigate the effectiveness of these static analysis techniques at inferring program behaviour, compared with explicit code annotations or runtime monitoring.

### 9.3.5 Low-Level Abstraction of HMAs

Throughout this dissertation, the Java programming language was used for application development. Java was chosen because of its portable nature and standardised virtual machine interface, which enables the same application code to be executed on heterogeneous processing cores that have different instruction set architectures. However, the heterogeneous processing cores of future HMA processors may not employ different instruction set architectures and, therefore, may not require a homogeneous virtual machine to abstract their differences. An interesting area of future work would be to investigate whether behaviour characteristics can be employed to inform runtime allocation decisions under lower-level languages, such as C and C++.

By extending this work to a lower-level language, it could be employed in situations where the high-level nature of Java is not appropriate, either for performance reasons or because of the lack of control afforded by Java (e.g., lack of pointers). It would also enable investigation of the abstraction of HMA processors by an underlying operating system, as apposed to a runtime system, using the techniques examined in this work.

Without the high-level information provided by a language such as Java, it may be more challenging to infer a program's behaviour. For example, in C, data-access operations are untyped, making it more difficult to infer the data-structure that is being accessed and tailor the operation appropriately. As such, additional methods for expressing and tracking program behaviour would be required to apply this work to a language such as C or C++.

### 9.3.6 Summary

There are a number of interesting future directions in which to take the work presented by this dissertation. The use of behaviour characteristics for the abstraction of heterogeneous features can be applied to different HMA processors, and be extended to make use of additional behaviour characteristics. The use of behaviour characteristics targeted at data-structures, rather than code, would be another interesting avenue of future research. Static analysis techniques that would enable tools and compilers to automatically tag programs with their expected behaviour characteristics, ahead of their execution, would broaden the appeal of this work and further reduce the application development burden. Finally, perhaps the most interesting area of potential future

work would be to apply the techniques and overall approach described by this dissertation to a more low-level programming language and runtime environment. This would enable operating systems to abstract HMA processors, thereby providing any program running on such a system with the opportunity of exploiting an HMA's heterogeneous processing resources.

# References


Adiletta, M., Rosenbluth, M., Bernstein, D., Wolrich, G. & Wilkinson, H. (2002). The Next Generation of Intel IXP Network Processors. *Intel Tech. Journal*, **6**(3), 6–18. Cited in Sections 2.2 and 5.1.

Ainsworth, T. & Pinkston, T. (2007). Characterizing the Cell EIB On-Chip Network. *IEEE Micro*, **27**(5), 6–14. Cited in Sections 4.1.3 and 5.1.

Allen, F. E. & Cocke, J. (1976). A Program Data Flow Analysis Procedure. *Communications of the ACM*, **19**(3), 137–147. Cited in Sections 4.3.2 and 9.3.4.

Allen, J., Bass, B., Basso, C., Boivie, R., Calvignac, J., Davis, G., Frelechoux, L., Heddes, M., Herkersdorf, A., Kind, A. *et al.* (2003). IBM PowerNP Network Processor: Hardware, Software, and Applications. *IBM Journal of Research and Development*, **47**(2), 177–194. Cited in Section 2.2.

Alpern, B., Augart, S., Blackburn, S., Butrico, M., Cocchi, A., Cheng, P., Dolby, J., Fink, S., Grove, D., Hind, M. *et al.* (2005). The Jikes Research Virtual Machine Project: Building an Open-Source Research Community. *IBM Systems Journal*, **44**(2), 399–417. Cited in Section 5.2.

AMD (2008). 'AMD Fusion: The Industry-Changing Impact of Accelerated Computing'. White Paper. Cited in Sections 2.3.1 and 9.3.1.

Amdahl, G. (1967). Validity of the Single Processor Approach to Achieving Large Scale Computing capabilities. *In Proceedings of the Spring Joint Computer Conference*. pp. 483–485. Cited in Section 2.1.

Archibald, J. & Baer, J. (1986). Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, **4**(4), 273–298. Cited in Sections 2.3.2 and 8.1.

Armstrong, J. (2003). Making Reliable Distributed Systems in the Presence of Software Errors. PhD thesis. The Royal Institute of Technology, Stockholm, Sweden. Cited in Section 3.1.

Bader, D. & Agarwal, V. (2007). FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine. *Lecture Notes in Computer Science*, **4873**, 172. Cited in Section 3.2.1.3.

Balakrishnan, S., Rajwar, R., Upton, M. & Lai, K. (2005). The impact of performance asymmetry in emerging multicore architectures. *SIGARCH Computer Architecture News*, **33**(2), 506–517. Cited in Section 3.

Barker, K. J., Davis, K., Hoisie, A., Kerbyson, D. J., Lang, M., Pakin, S. & Sancho, J. C. (2008). Entering the petaflop era: the architecture and performance of Roadrunner. *In Proceedings of the Conference on Supercomputing (SC'08)*. pp. 1–11. Cited in Sections 2.2, 3.2.1.3, and 5.1.

Baumann, A., Barham, P., Dagand, P., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A. & Singhania, A. (2009). The Multikernel: A New OS Architecture for Scalable Multicore Systems. *In Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP'09)*. Cited in Sections 3.2.2 and 3.4.

Becchi, M. & Crowley, P. (2006). Dynamic thread assignment on heterogeneous multiprocessor architectures. *In Proceedings of the 3rd conference on Computing Frontiers*. ACM. p. 40. Cited in Section 3.3.

Bellens, P., Perez, J. M., Badia, R. M. & Labarta, J. (2006). CellSs: A Programming Model for the Cell BE Architecture. *In Proceedings of the Conference on Supercomputing (SC'06)*. p. 86. Cited in Section 3.2.1.3.

Benthin, C., Wald, I., Scherbaum, M. & Friedrich, H. (2006). Ray tracing on the Cell processor. *In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*. pp. 25–23. Cited in Section 3.2.1.3.

Blackburn, S., Garner, R., Hoffmann, C., Khang, A., McKinley, K., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. *et al.* (2006). The DaCapo benchmarks: Java benchmarking development and analysis. *In Proceedings of the 21st Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*. pp. 169–190. Cited in Section 5.5.3.

Blackburn, S. M., Cheng, P. & McKinley, K. S. (2004). Myths and realities: the performance impact of garbage collection. *SIGMETRICS Performance Evalaluation Review*, **32**(1), 25–36. Cited in Section 5.6.

Blagojevic, F., Nikolopoulos, D., Stamatakis, A. *et al.* (2007*a*). Dynamic Multigrain Parallelization on the Cell Broadband Engine. *In Proceedings of the 12th Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*. ACM. p. 100. Cited in Section 3.3.

Blagojevic, F., Nikolopoulos, D., Stamatakis, A. *et al.* (2007*b*). Runtime Scheduling of Dynamic Parallelism on Accelerator-based Multi-Core Systems. *Parallel Computing*, **33**(10-11), 700–719. Cited in Section 3.3.

Blythe, D. (2006). The Direct3D 10 System. *In Proceedings of the Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'06)*. pp. 724–734. Cited in Section 2.3.1.

Bolosky, W., Fitzgerald, R. & Scott, M. (1989). Simple but effective techniques for NUMA memory management. *In Proceedings of the 12th Symposium on Operating systems Principles (SOSP'89)*. pp. 19–31. Cited in Section 3.4.

Borkar, S. (2007). Thousand core chips: a technology perspective. *In DAC '07: Proceedings of the 44th annual Design Automation Conference*. pp. 746–749. Cited in Section 2.1.

Bower, F., Sorin, D. & Cox, L. (2008). The Impact of Dynamically Heterogeneous Multicore Processors on Thread Scheduling. *IEEE Micro-Institute of Electrical and Electronics Engineers*, **28**(3), 17–25. Cited in Section 3.

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M. & Hanrahan, P. (2004). Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics*. Cited in Sections 2.3.1, 3.2.1.1, and 9.3.1.

Bugnion, E., Devine, S., Govil, K. & Rosenblum, M. (1997). Disco: running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, **15**(4), 412–447. Cited in Section 3.4.

Burger, R. & Dybvig, R. (1998). An infrastructure for profile-driven dynamic recompilation. *In Proceedings of the 1998 International Conference on Computer Languages*. p. 240. Cited in Section 7.4.

Case, R. P. & Padegs, A. (1978). Architecture of the IBM system/370. *Communications of the ACM*, **21**(1), 73–96. Cited in Section 2.2.

Chandra, P. (1988). Programming the 80387 coprocessor. *Byte*, **13**(3), 207–215. Cited in Section 2.2.

Chapin, J., Rosenblum, M., Devine, S., Lahiri, T., Teodosiu, D. & Gupta, A. (1995). Hive: Fault Containment for Shared-Memory Multiprocessors. *In Proceedings of the 15th Symposium on Operating Systems Principles (SOSP'95)*. pp. 12–25. Cited in Section 3.4.

Charles, J., Jassi, P., Narayan, A., Sadat, A. & Fedorova, A. (2009). Evaluation of the Intel Core i7 Turbo Boost feature. *In IEEE International Symposium on Workload Characterization*. Cited in Section 2.3.2.

Chen, T., Raghavan, R., Dale, J. N. & Iwata, E. (2007). Cell Broadband Engine Architecture and its First Implementation: A Performance View. *IBM Journal of Research and Development*, **51**(5), 559–572. Cited in Sections 2.2 and 5.1.

Cho, Y. & Mangione-Smith, W. (2005). A pattern matching coprocessor for network security. *In DAC '07: Proceedings of the 42nd annual Design Automation Conference*. pp. 234–239. Cited in Section 2.2.

Choi, J., Gupta, M., Serrano, M., Sreedhar, V. & Midkiff, S. (1999). Escape analysis for Java. *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 1–19. Cited in Sections 4.3.2 and 9.3.4.

Coppersmith, D., Johnson, D. & Matyas, S. (1996). Triple DES cipher block chaining with output feedback masking. *In IBM Journal of Research and Development*. Vol. 40. Cited in Section 6.4.3.

Coulson, G., Blair, G., Gomes, A., Joolia, A., Lee, K., Ueyama, J. & Ye, Y. (2003). A Reflective Middleware-based Approach to Programmable Networking. *Proceedings of 2nd Intl. Workshop on Reflective and Adaptive Middleware*. Cited in Section 3.2.1.2.

Cox, A. & Fowler, R. (1989). The implementation of a coherent memory abstraction on a NUMA multiprocessor: experiences with Platinum. *ACM SIGOPS Operating Systems Review*, **23**(5), 32–44. Cited in Sections 2.3.2, 3.4, and 8.1.

Dagum, L. & Menon, R. (1998). OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, **5**(1), 46–55. Cited in Sections 3.1, 3.2.1.1, and 4.5.

Dai, J., Huang, B., Li, L. & Harrison, L. (2005). Automatically partitioning packet processing applications for pipelined architectures. *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'05)*, pp. 237–248. Cited in Section 3.2.1.2.

Dean, J. & Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. *In Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'04)*. Cited in Section 3.2.1.1.

Donaldson, A., Riley, C., Lokhmotov, A. & Cook, A. (2008). Auto-parallelisation of Sieve C++ programs. *Lecture Notes in Computer Science*, **4854**, 18. Cited in Section 3.2.1.3.

Eichenberger, A., O'Brien, J., O'Brien, K., Wu, P., Chen, T., Oden, P., Prener, D., Shepherd, J., So, B., Sura, Z. *et al.* (2006). Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture. *IBM Systems Journal*. Cited in Section 3.2.1.3.

Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G. C., Larus, J. R. & Levi, S. (2006). Language Support for Fast and Reliable Message Based Communication in Singularity OS. *Proceedings of the EuroSys Conference*. Cited in Section 3.2.2.

Feghali, W., Burres, B., Wolrich, G. & Carrigan, D. (2002). Security: Adding Protection to the Network via the Network Processor. *Intel Technology Journal*, **6**(3), 40–49. Cited in Section 3.2.1.1.

Flynn, M. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, **21**(9), 948–960. Cited in Section 5.3.2.

Foster, I. & Karonis, N. (1998). A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. *In Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. pp. 1–11. Cited in Section 3.1.

Fung, W., Sham, I., Yuan, G. & Aamodt, T. M. (2007). Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. *IEEE/ACM International Symposium on Microarchitecture*, pp. 407–420. Cited in Section 9.3.1.

Gamsa, B., Krieger, O., Appavoo, J. & Stumm, M. (1999). Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. *In Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*. pp. 87–100. Cited in Section 3.4.

George, L. & Blume, M. (2003). Taming the IXP network processor. *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'03)*, pp. 26–37. Cited in Section 2.2.

Govil, K., Teodosiu, D., Huang, Y. & Rosenblum, M. (2000). Cellular Disco: Resource Management using Virtual Clusters on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, **18**(3), 229–262. Cited in Section 3.4.

Greenstadt, J. (1957). The IBM 709 Computer. *In New Computers, Report from the Manufacturers ACM Conference*. pp. 92–98. Cited in Section 2.2.

Gropp, W., Lusk, E. & Skjellum, A. (1994). *Using MPI: portable parallel programming with the message-passing interface*. the MIT Press. Cited in Section 3.1.

Gustafson, J. L. (1988). Reevaluating Amdahl's law. *Communications of the ACM*, **31**(5), 532–533. Cited in Section 2.1.

Harris, T., Marlow, S. & Jones, S. P. (2005). Haskell on a shared-memory multiprocessor. *In Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*. pp. 49–61. Cited in Section 3.1.

Healy, C., Sjodin, M., Rustagi, V. & Whalley, D. (1998). Bounding loop iterations for timing analysis. *In Fourth IEEE Real-Time Technology and Applications Symposium*. pp. 12–21. Cited in Sections 4.3.2 and 9.3.4.

Hewitt, C., Bishop, P. & Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. *In Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI'73)*. pp. 235–245. Cited in Section 3.1.

Hill, M. & Marty, M. (2008). Amdahl's Law in the Multicore Era. *Computer*, **41**(7), 33–38. Cited in Section 2.1.

Hiranandani, S., Kennedy, K. & Tseng, C.-W. (1992). Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, **35**(8), 66–80. Cited in Section 3.2.1.2.

Hoare, C. (1978). Communicating sequential processes. *Communications of the ACM*, **21**(8), 677. Cited in Section 3.1.

Hofstee, H. (2005). Power efficient processor architecture and the Cell processor. *11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, pp. 258–262. Cited in Sections 2.2 and 5.1.

Hunt, G. C. & Larus, J. R. (2007). Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.* Cited in Section 3.2.2.

Ipek, E., Kirman, M., Kirman, N. & Martinez, J. F. (2007). Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. *In ISCA '07: Proceedings of the International Symposium on Computer Architecture*. pp. 186–197. Cited in Section 2.3.2.

Kapasi, U., Rixner, S., Dally, W., Khailany, B., Ahn, J., Mattson, P. & Owens, J. (2003). Programmable Stream Processors. *IEEE Computer*, **36**(8), 54–62. Cited in Sections 3.2.1.1 and 4.2.2.

Kelley, M., Winner, S. & Gould, K. (1992). A scalable hardware render accelerator using a modified scanline algorithm. *Proceedings of the Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'92)*, **26**(2), 241–248. Cited in Section 2.3.1.

Keltcher, C., McGrath, K., Ahmed, A. & Conway, P. (2003). The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, **23**(2), 66–76. Cited in Section 8.1.

Kotzmann, T., Wimmer, C., Mössenböck, H., Rodriguez, T., Russell, K. & Cox, D. (2008). Design of the Java HotSpot$^{\text{TM}}$ client compiler for Java 6. *ACM Transactions Architecture Code Optimization*, **5**(1), 1–32. Cited in Section 5.6.

Krintz, C. & Calder, B. (2001). Using annotations to reduce dynamic optimization time. *In Proceedings of the Conference on Programming Language Design and Implementation (PLDI '01)*. pp. 156–167. Cited in Section 1.

Krishnaswamy, D., Stevens, R., Hasbun, R., Revilla, J. & Hagan, C. (2003). The Intel PXA800F wireless Internet-on-a-chip architecture and design. *In IEEE Custom Integrated Circuits*. Cited in Section 2.2.

Kumar, R., Farkas, K. I., Jouppi, N. P., Ranganathan, P. & Tullsen, D. M. (2003). Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. *In MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. Cited in Sections 2.2 and 3.3.

Kumar, R., Tullsen, D. M., Jouppi, N. P. & Ranganathan, P. (2005*a*). Heterogeneous chip multiprocessors. *IEEE Computer*, **38**(11), 32–38. Cited in Section 2.2.

Kumar, R., Tullsen, D. M., Ranganathan, P., Jouppi, N. P. & Farkas, K. I. (2004). Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. *SIGARCH Computer Architecture News*, **32**(2), 64. Cited in Section 3.3.

Kumar, R., Zyuban, V. & Tullsen, D. (2005*b*). Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. *In Proceedings of the 32nd International Symposium on Computer Architecture*. pp. 408–419. Cited in Section 5.6.

Lattner, C. & Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. *In Proceedings of the International Symposium on Code Generation and Optimization*. Cited in Section 3.2.1.1.

Laudon, J., Lenoski, D., Syst, S. & View, M. (1997). System overview of the SGI Origin 200/2000 product line. *In Proceedings of IEEE Compcon'97*. pp. 150–156. Cited in Sections 2.3.2 and 8.1.

Li, T., Baumberger, D., Koufaty, D. A. & Hahn, S. (2007). Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures. *In Proceedings of the Conference on Supercomputing (SC'07)*. pp. 1–11. Cited in Section 3.3.

Liao, S.-W., Diwan, A., Bosch, Jr., R. P., Ghuloum, A. & Lam, M. S. (1999). SUIF Explorer: an interactive and interprocedural parallelizer. *In Proceedings of the 7th Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*. pp. 37–48. Cited in Section 3.2.1.2.

Liedtke, J. (1995). On $\mu$-kernel construction. *In Proceeding of 15th ACM Symposium on Operating System Principles (SOSP'95)*. pp. 237–250. Cited in Section 3.2.2.

Linderman, M. D., Collins, J. D., Wang, H. & Meng, T. H. (2008). Merge: a programming model for heterogeneous multi-core systems. *SIGOPS Operating Systems Review*, **42**(2), 287–296. Cited in Section 3.2.1.1.

Lindholm, E., Kligard, M. J. & Moreton, H. (2001). A user-programmable vertex engine. *In Proceedings of the Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'01)*. pp. 149–158. Cited in Section 2.3.1.

Lindholm, E., Nickolls, J., Oberman, S. & Montrym, J. (2008). NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, pp. 39–55. Cited in Section 2.3.1.

Liu, Y., Jones, H., Vaidya, S., Perrone, M., Tydlitát, B. & Nanda, A. (2007). Speech Recognition Systems on the Cell Broadband Engine Processor. *IBM Journal of Research and Development*, **51**(5), 583–591. Cited in Section 3.2.1.3.

Manson, J., Pugh, W. & Adve, S. V. (2005). The Java Memory Model. *In Proceedings of the 32nd Symposium on Principles of Programming Languages (POPL'05)*. pp. 378–391. Cited in Sections 5.3.3, 5.3.3.3, and 5.3.3.4.

Mark, W., Glanville, R., Akeley, K. & Kilgard, M. (2003). Cg: A system for programming graphics hardware in a C-like language. *In Proceedings of the Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'03)*. p. 907. Cited in Section 3.2.1.1.

Mathew, J. A., Coddington, P. D. & Hawick, K. A. (1999). Analysis and development of Java Grande benchmarks. *In JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*. ACM. pp. 72–80. Cited in Section 5.5.2.

Mattson, T. G., Van der Wijngaart, R. & Frumkin, M. (2008). Programming the Intel 80-core network-on-a-chip terascale processor. *In Proceedings of the Conference on Supercomputing (SC'08)*. pp. 1–11. Cited in Section 2.3.2.

McCool, M. (2006). Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform. *In GSPx Multicore Applications Conference*. Cited in Section 3.2.1.1.

McCool, M. & Du Toit, S. (2004). *Metaprogramming GPUs with Sh*. AK Peters, Ltd. Cited in Section 3.2.1.1.

McIlroy, R. & Hodson, O. (2007). Subordinate Kernels: Application Offloading in Asymmetric Multi-Processor Systems. *In Workshop on Operating System Support for Heterogeneous Multi-Core Architectures*. Cited in Section 1.3.

McIlroy, R. & Sventek, J. (2009*a*). Abstracting Heterogeneous Multi-Core Architectures using a Code Annotation Aware Runtime System (Poster). *In The EuroSys conference (EuroSys'09)*. Cited in Section 1.3.

McIlroy, R. & Sventek, J. (2009*b*). Hera-JVM: Abstracting Processor Heterogeneity Behind a Virtual Machine. *In Workshop on Hot Topics in Operating Systems (HotOS)*. Cited in Section 1.3.

McIlroy, R., Dickman, P. & Sventek, J. (2008). Efficient Dynamic Heap Allocation of Scratch-Pad Memory. *In Proceedings of the International Symposium on Memory Management*. Cited in Sections 1.3 and 5.3.3.2.

Microsoft (2002). 'DirectX 9.0 graphics'. Available online at `http://msdn.microsoft.com/directx`. Cited in Section 3.2.1.1.

Mogul, J. C., Mudigonda, J., Binkert, N., Ranganathan, P. & Talwar, V. (2008). Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems. *IEEE Micro*, **28**(3), 26–41. Cited in Section 3.3.

Montrym, J. & Moreton, H. (2005). The Geforce 6800. *IEEE Micro*, pp. 41–51. Cited in Section 2.3.1.

Morris, R., Kohler, E., Jannotti, J. & Kaashoek, M. F. (1999). The Click modular router. *In Proceeding of 17th ACM Symposium on Operating System Principles (SOSP'99)*. Cited in Section 3.2.1.2.

Motorola (2001). 'C-5 Network Processor Architecture Guide'. Cited in Section 2.2.

Munshi, A. (2009). The OpenCL Specification. *Khronos OpenCL Working Group*. Cited in Sections 2.3.1, 3.2.1.1, 4.5, and 9.3.1.

Nightingale, E., Hodson, O., McIlroy, R., Hawblitzel, C. & Hunt, G. (2009). Helios: Heterogeneous multiprocessing with satellite kernels. *In Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP'09)*. Cited in Sections 1.3, 3.2.2, and 3.4.

Noll, A., Gal, A. & Franz, M. (2008). CellVM: A Homogeneous Virtual Machine Runtime System for a Heterogeneous Single-Chip Multiprocessor. *In Workshop on Cell Systems and Applications*. Cited in Sections 3.2.2, 5.2, and 5.3.4.4.

Nvidia (2009*a*). 'Nvidia Fermi Compute Architecture'. White Paper. Cited in Section 2.3.1.

Nvidia (2009*b*). 'Nvidia Tegra: Visual computing for mobile devices'. available online at `http://www.nvidia.com/page/handheld.html`. Cited in Sections 2.3.1 and 9.3.1.

Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E. & Zenger, M. (2004). An overview of the Scala programming language. Technical Report LAMP-REPORT-2006-001. École Polytechnique Fédérale de Lausanne (EPFL). Cited in Section 3.1.

Olukotun, K., Nayfeh, B. A., Hammond, L., Wilson, K. & Chang, K. (1996). The case for a single-chip multiprocessor. *SIGPLAN Not.*, **31**(9), 2–11. Cited in Section 2.

Papakipos, M. (2006). The PeakStream Platform. *In LACSI Workshop on Heterogeneous Computing*. Cited in Section 3.2.1.1.

Peercy, M. S., Olano, M., Airey, J. & Ungar, P. J. (2000). Interactive multi-pass programmable shading. *In Proceedings of the Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'00)*. pp. 425–432. Cited in Section 3.2.1.1.

Penry, D. A. (2009). Multicore diversity: a software developer's nightmare. *SIGOPS Operating Systems Review*, **43**(2), 100–101. Cited in Section 3.

Perez, J., Bellens, P., Badia, R. & Labarta, J. (2007). CellSs: Making it easier to program the Cell Broadband Engine processor. *IBM Journal of Research and Development*, **51**(5), 593–604. Cited in Section 3.2.1.3.

Peyton Jones, S., Gordon, A. & Finne, S. (1996). Concurrent Haskell. *In Proceedings of the 23rd Symposium on Principles of Programming Languages (POPL'96)*. pp. 295–308. Cited in Section 3.1.

Pham, D., Asano, S., Bolliger, M., Day, M., Hofstee, H., Johns, C., Kahle, J., Kameyama, A., Keaty, J., Masubuchi, Y. *et al.* (2005). The design and implementation of a first-generation Cell processor. *IEEE Solid-State Circuits Conference*. Cited in Sections 2.2 and 5.1.

Rettberg, R. & Thomas, R. (1986). Contention is no obstacle to shared-memory multiprocessing. *Communications of the ACM*, **29**(12), 1202–1212. Cited in Sections 2.3.2 and 8.1.

Roscoe, A. W. & Hoare, C. A. R. (1988). The laws of occam programming. *Theoretical Computing Science*, **60**(2), 177–229. Cited in Section 3.1.

Ross, P. (2008). Why CPU Frequency Stalled. *IEEE Spectrum*, **45**(4), 72–72. Cited in Section 2.

Ryoo, S., Rodrigues, C., Baghsorkhi, S., Stone, S., Kirk, D. & Wen-Mei, H. (2008). Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. *In Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming (PPoPP'058)*. pp. 73–82. Cited in Sections 2.3.1, 3.2.1.1, and 9.3.1.

Saha, B., Adl-Tabatabai, A., Ghuloum, A., Rajagopalan, M., Hudson, R., Petersen, L., Menon, V., Murphy, B., Shpeisman, T., Sprangle, E. *et al.* (2007). Enabling Scalability and Performance in a Large Scale CMP Environment. *In Proceedings of the EuroSys Conference*. pp. 73–86. Cited in Section 3.2.2.

Schüpbach, A., Peter, S., Baumann, A., Roscoe, T., Barham, P., Harris, T. & Isaacs, R. (2008). Embracing diversity in the Barrelfish manycore operating system. *In Proceedings of the Workshop on Managed Many-Core Systems (MMCS), Boston, MA, USA, June*. Cited in Sections 3.2.2 and 4.5.

Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T. & Hanrahan, P. (2008). Larrabee: a many-core x86 architecture for visual computing. *In Proceedings of the Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'08)*. pp. 1–15. Cited in Sections 2.3.1 and 9.3.1.

Shah, N. (2001). Understanding Network Processors. *Master's thesis, University of California, Berkeley, Sep*. Cited in Section 2.2.

Shah, N., Plishker, W. & Keutzer, K. (2003). NP-Click: A Programming Model for the Intel IXP1200. *2nd Workshop on Network Processors (NP-2)*. Cited in Section 3.2.1.2.

Shelepov, D., Saez Alcaide, J. C., Jeffery, S., Fedorova, A., Perez, N., Huang, Z. F., Blagodurov, S. & Kumar, V. (2009). HASS: A Scheduler for Heterogeneous Multicore Systems. *SIGOPS Operating Systems Review*, **43**(2), 66–75. Cited in Section 3.3.

Sherwood, T., Perelman, E. & Calder, B. (2001). Basic block distribution analysis to find periodic behavior and simulation points in applications. *In International Conference on Parallel Architectures and Compilation Techniques*. pp. 3–14. Cited in Section 7.1.1.

Shiv, K., Chow, K., Wang, Y. & Petrochenko, D. (2009). SPECjvm2008 Performance Characterization. *In Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*. Springer. pp. 17–35. Cited in Section 5.5.3.

Smith, A. J. (1978). Sequential program prefetching in memory hierarchies. *Computer*, **11**(12), 7–21. Cited in Section 9.3.3.

Smith, A. J. (1982). Cache memories. *ACM Computing Surveys (CSUR)*, **14**(3), 473–530. Cited in Section 4.2.2.

Smith, L., Bull, J. & Obdrizalek, J. (2001). A parallel Java Grande benchmark suite. *In Proceedings of the Conference on Supercomputing (SC'01)*. pp. 6–6. Cited in Section 5.5.3.

Sondag, T. & Rajan, H. (2009). Phase-guided thread-to-core assignment for improved utilization of performance-asymmetric multi-core processors. *In IWMSE '09: Proceedings of the 2nd International Workshop on Multicore Software Engineering*. Cited in Section 3.3.

Sondag, T., Krishnamurthy, V. & Rajan, H. (2007). Predictive thread-to-core assignment on a heterogeneous multi-core processor. *In Proceedings of the 4th workshop on Programming Languages and Operating Systems*. Cited in Section 3.3.

Strong, R., Mudigonda, J., Mogul, J. C., Binkert, N. & Tullsen, D. (2009). Fast switching of threads between cores. *SIGOPS Operating Systems Review*, **43**(2), 35–45. Cited in Section 3.3.

Tarditi, D., Puri, S. & Oglesby, J. (2006). Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. Technical Report MSR-TR-2005-184. Microsoft Research. Cited in Section 3.2.1.1.

Thornton, J. E. (1970). *Design of a Computer—The Control Data 6600*. Scott Foresman & Co. Cited in Section 2.2.

Trinder, P., Barry Jr, E., Davis, M., Hammond, K., Junaidu, S., Klusik, U., Loidl, H. & Jones, S. (1999). GpH: An Architecture-independent Functional Language. *IEEE Transactions on Software Engineering*. Cited in Section 3.1.

Vangal, S., Howard, J., Ruhl, G., Dighe, S., Wilson, H., Tschanz, J., Finan, D., Singh, A., Jacob, T., Jain, S., Erraguntla, V., Roberts, C., Hoskote, Y., Borkar, N. & Borkar, S. (2008). An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, **43**(1), 29–41. Cited in Section 2.3.2.

Vaughn-Nichols, S. J. (2009). Vendors draw up a new graphics-hardware approach. *IEEE Computer*, **42**, 11–13. Cited in Section 2.3.1.

Wall, D. W. (1991). Limits of instruction-level parallelism. *SIGARCH Computer Architecture News*, **19**(2), 176–188. Cited in Section 2.

Wang, P., Collins, J., Chinya, G., Jiang, H., Tian, X., Girkar, M., Yang, N., Lueh, G. & Wang, H. (2007). EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-Core Multithreaded System. *In Proceedings of the Conference on Programming Language Design and Implementation (PLDI'07)*. p. 166. Cited in Section 3.2.1.1.

Welch, T. (1984). A technique for high-performance data compression. *Computer*, **17**(6), 8–19. Cited in Section 6.4.3.

Wentzlaff, D., Griffin, P., Hoffmann, H., Bao, L., Edwards, B., Ramey, C., Mattina, M., Miao, C.-C., Brown, J. & Agarwal, A. (2007). On-chip interconnection architecture of the tile processor. *IEEE Micro*, **27**(5), 15–31. Cited in Section 2.3.2.

Woo, M., Neider, J., Davis, T. & Shreiner, D. (1999). *OpenGL Programming Guide: The Official Guide to Learning OpenGL*. Addison-Wesley Longman Publishing Co., Inc.. Boston, MA, USA. Cited in Section 3.2.1.1.

Yee, B. (1994). Using Secure Coprocessors. PhD thesis. Carnegie-Mellon University. Carnegie-Mellon University. Cited in Section 2.2.