University of Glasgow

Dynamic Affinity Scheduling in Heterogeneous Multi-Core Processors

David Warnock

A Dissertation Submitted in Part-Fulfilment of the Requirements of the Degree of MSci Software Engineering at The University of Glasgow

Department of Computing Science,
University of Glasgow,
Lilybank Gardens,
Glasgow, G12 8QQ.

April 2009

Abstract

Heterogeneous processors designs are becoming increasingly common, and are likely to continue gaining popularity due to high-profile processors such as the Cell Broadband Engine. Traditionally programming for such a heterogeneous multi-core architecture requires separate design and implementation for each processing core, and programs are strictly bound to the core for which they were created. It is often the case that the tools available for each core are significantly different, and heterogeneous multi-core architectures have earned a reputation for being very difficult to develop for.

Recent work by Ross McIlroy has produced Hera JVM, a Java Virtual Machine which runs on the Cell processor and abstracts the complex hardware. Hera JVM uses annotations to schedule threads to cores, which prevents full hardware abstraction. This project aims to replace the annotation-based scheduler in Hera JVM with a dynamic affinity scheduler, allowing for a more complete abstraction of the heterogeneous processor.

A prototype scheduler was constructed, then its performance was compared to a static scheduler and the previous annotation-based scheduler. The dynamic affinity scheduler was found to be capable of matching or surpassing the performance of the alterantive schedulers while offering a greater degree of hardware abstraction.

Acknowledgements

There are a number of people that I would like to acknowledge for their help.

First, thanks to Professor Joe Sventek and Ross McIlroy for their guidance and assistance during the course of this Research Project. Without their expertise this project would not have been possible.

A special thanks to my friends and family, who have done their best to keep me sane for the past 5 years. In particular I'd like to thank Chris McAdam for his constant support during this project.

This Thesis is dedicated to the memory of my grandmother, Margaret Hutchison, who died April 27th 2009.

Contents

Chapter 1

Introduction

As processor designers approach the physical limitations of the materials available to them, they must look for new and inventive ways to make faster processors. Processor clock speeds have risen dramatically; the size of processors has been decreased in order to reduce latencies; large and efficient multi-layer hardware caches have been added to reduce memory bottlenecks. However, with each new processor more efficient than the last, it became increasingly obvious that monolithic processor designs would no longer be able to satisfy demands. Parallelism would be required to increase the performance of processors any further.

Multiple processor computers usually take the form of multiple processors on the same motherboard or a group of networked computers; as such these systems are expensive and are typically limited to servers and labs. As manufacturing processes evolved and processors became cheaper, major manufacturers such as AMD and Intel started to produce multi-core processors to meet the performance demands of their customers. These multi-core processors were homogeneous, containing two or more cores of the same specification.

Heterogeneous multi-core processors are designed such that the processing cores have different properties. In such heterogeneous processors, the different cores can have different instruction sets or performance ratings. Heterogeneous computer systems are not new to computer science; previous examples include grid computing, which employed a network of heterogeneous computers to achieve the highest possible throughput. Some grid systems are still in use today, such as the 'Folding@home'[1] project.

Heterogeneous processors have a number of advantages over their homogeneous counterparts. The main advantage is that workloads tend to be heterogeneous themselves; Kumar *et.al* demonstrated that a heterogeneous processor with affinity scheduling techniques would complete

[1] http://folding.stanford.edu/

its workload faster than an equivalent homogeneous processor, even though the heterogeneous processor has higher scheduling overheads [14].

Processors designed for specific tasks have a greater advantage, as the processor can be designed to have an affinity to the expected workload; this is the case with the Cell BE and the Intel IPX. If a processing core has been designed to have an affinity with a certain workload, components designed to improve general-purpose performance can be removed from certain cores. This will reduced the size and complexity of these processing cores, reducing costs and allowing more cores to be placed on a single chip. With more cores, each streamlined for a specific workload, the overall throughput for the target workloads would significantly increase when compared to a similar homogeneous design. For example, the Cell BE is designed specifically to increase the speed of processing media-based workloads.

However, there are a number of problems faced by the designers of heterogeneous processors. The main issues lie in programming; developers are often required to separately design and implement 'threads' that are explicitly bound to a particular core type. Often the programming language used for each thread is different due to variations between the tools available for each core type. Writing applications that can efficiently take advantage of heterogeneous processors can be difficult and error-prone.

Another part of the problem lies in scheduling threads. In order to correctly identify which core is best equipped to handle a given task, the programmer is required to have a deep understanding of a number of systems. First they must explore the architectural differences between the heterogeneous cores in order to understand their performance characteristics. Then they are required to identify the behavioural characteristics of their own code in order to identify which core should execute the job. Simply identifying program behaviour alone can be a complex task; it is often the case that behaviour cannot be predicted prior to runtime.

The techniques and algorithms employed when scheduling large distributed systems or multi-processor systems are of little use in the context of a heterogeneous multi-core architecture. As demonstrated by Kumar *et.al*, homogeneous scheduling techniques fail to take full advantage of heterogeneous architectures[14]. Therefore, there is a need to develop scheduling techniques and algorithms for heterogeneous processors that will ease the burden of programming for such architectures.

Assuming that a given piece of high-level code can be compiled to run on any of the given heterogeneous cores, automating the process of determining affinities would reduce the mental workload of the programmer substantially. If thread-to-processor affinities can be identified accurately and efficiently, threads could be automatically scheduled

based on these 'affinity scores'.

This could be extended to monitor thread behaviour at runtime, allowing the scheduler to handle changing affinities. If the overheads of such dynamic affinity scheduling are sufficiently small, it is likely that such a system would be more efficient than any manual alternative; a dynamic affinity scheduler would be able to migrate threads as affinities change, fix incorrectly specified affinities and even determine affinities where the programmer has not specified them.

This would abstract the heterogeneous architecture behind the high-level language, allowing the programmer to write multi-threaded code without requiring an understanding of the architecture it will run on. This would also allow code intended for a homogeneous architecture to be compiled for a heterogeneous architecture; dynamic affinity scheduling would ensure that each thread would be scheduled to the processor best equipped to run it, migrating them where necessary. This can be split into two distinct problems.

First, a set of program behavioural properties would need to be identified and their role in determining processor affinities explored. Drawing heavily from the affinity scheduling and runtime profiling research fields, this part of the problem would involve creating a taxonomy of code categories and identifying their processing requirements. For a given processor, the heterogeneous properties of each core would then need to be explored in order to identify the affinities between processing cores and the code categories in the taxonomy. This information would be used to design and build a scoring system capable of identifying affinity-altering behaviour and monitoring how affinities change as the thread runs.

The second part of the problem involves developing a scheduler that will make use of the information provided by the scoring system in order to make affinity scheduling decisions. It would also be a dynamic scheduler capable of reacting to thread behavioural changes, migrating threads if their affinities shift with their behaviour. This part of the problem is tied closely to the processor scheduling research field.

The main aim of the research is to develop a scheduler able to completely abstract the scheduling problem, allowing the programmer to completely disregard issues related to heterogeneous scheduling. The dynamic affinity scheduler would be expected to outperform or match the average performance of a static or manual heterogeneous scheduler.

If these objectives can be successfully achieved, then it would significantly reduce the difficulty of programming for such a processor. With programmers able to take advantage of a heterogeneous architecture without having to understand the underlying details, they would be able to develop for the platform as they would any other system. This may lead to increased adoption rates for heterogeneous architec-

tures, especially if code written for homogeneous architectures becomes portable to heterogeneous architectures without performance loss.

This work presented here is based on research carried out in the computer architecture, compiler design, scheduling and runtime profiling research fields. The results of this research will contribute chiefly to the scheduling and compiler design research fields, but it is likely to also have implications in the computer architecture field.

The remainder of this dissertation is as follows; in chapter 2 related work in these fields will be explored in detail in order to establish a starting point for the project and identify possible approaches. Chapter 3 will define a specific approach and link it to the available resources. In this chapter the design of the prototype scheduler will also be presented. Chapter 4 will discuss the design of any experiments, and their results will be presented in Chapter 5. Chapter 6 will discuss and interpret these results and compare them to the original objectives. Finally, Chapter 7 concludes the dissertation with a discussion of the project and an analysis of any future work.

Chapter 2

Literature Survey

This chapter explores other work related to the research problem. Presented here are papers by researchers working towards similar aims in order to establish the current state of the art. Section 2.1 will investigate models for the identification of program behaviour at runtime, while section 2.2 will explore research related to scheduling.

2.1 Runtime Profiling

Identifying runtime properties through runtime measurement is an area of active research, popular mainly because of the wide range of applications it has. Some techniques such as profiling are aimed towards offline optimization, while others such as phase and behaviour prediction are often employed in online dynamic optimisation systems such as Dynamo[1].

In [8], Duesterwald & Bala argue that effective software prediction can be performed using a very small amount of information. Their work demonstrates a technique for using small metrics in hot path prediction, which is calculated at runtime in order to assist runtime based optimisations. At the time, their Next Executing Tail system was used in the Dynamo[1] runtime optimization system; it has since been replaced with a Most Recently Executed Tail system. The path prediction system was only an example, as the point of the paper was to demonstrate that simple metrics can be more effective than complex ones by delivering information fast enough that it can be acted upon. The authors argue that simple metrics use less space and allow faster decision making, which reduces missed opportunity costs. If appropriate metrics are chosen prediction quality is not lowered, offering improved performance over complex prediction schemes. This is important research as it effectively demonstrates that simple metrics can be gathered and exploited to great effect at runtime. Duesterwald is a prolific researcher in this field and also works on the Dynamo project which this research

contributed to.

Another prolific researcher in the field is Timothy Sherwood. In [18], Sherwood *et.al* outlined their *basic block distribution analysis* method. Basic block analysis works by building a list of basic blocks at compile time. This list of basic blocks is transformed into a Basic Block Vector, or BBV, which contains a list of each basic block and how often it is run. When a basic block is entered at runtime its corresponding entry in the BBV is updated. This information identifies the parts of the code that are executed most frequently, which can be used to identify basic program phases. Despite requiring some work during the compiler stage, the authors argue that most optimizing compilers construct a list of these basic blocks when optimizing. As such, their technique could be used to identify program phases without incurring heavy costs.

In [17], Dhodapkar & Smith introduce a profiling technique they call *dynamic working set analysis*. In this, the working set of an executing program is monitored. The authors suggest that working set alterations can be used to represent phase changes. The authors note that working sets would require a quick lossy compression into working set signatures, as keeping records of them would be too resource intensive. A set of working set signatures can be used to represent the phases of a system. This method is unlikely to be effective in a heterogeneous system, and would require heavy modification.

Dhodapkar & Smith compare 3 phase detection techniques in [7]. The techniques are dynamic working set analysis, as outlined by themselves in [17]; basic block distribution analysis, as described by Sherwood *at al.* in [18] and conditional branch counts, a third system that uses cache hits and branch predictor hits to identify program phase changes. This technique may not be effective in heterogeneous processors, as the processors may not have the same properties; for example most of the cores in the Cell processor do not have a hardware cache or a branch predictor. Therefore, such hardware support cannot be guaranteed in heterogeneous systems.

The authors identified a number of properties that could be used to compare these very different profiling methods; comparing them on properties such as sensitivity, or how effectively phase changes are detected; stability, how effective the algorithm is at identifying stable phases; performance variance, which is a representation of how effectively the algorithm detects similar phases with varying performance values; and correlation, which is used to compare the phases detected by the 3 algorithms. Their experiment demonstrated roughly equal performance over all of the phase detection systems, with basic block analysis being slightly more effective at identifying phases than the other techniques.

In [19], Sherwood *at al.* discuss techniques that can be used to identify the behaviour of a program. Using the basic block vectors as described

in [18], the authors apply a clustering method which involves plotting the vectors in a vector space model and identifying concentrations of vectors by calculating the Manhattan distance between them. In this case the concentration of vectors is analogous to program phases. This technique could have a wide range of applications, depending on its use; unfortunately the algorithm used for clustering in this case has a background in machine learning and is resource heavy. As such it is unlikely that this system would be suitable for runtime profiling in a dynamic affinity scheduler.

Sherwood *at al.*[20] extends on the technique presented in [19]. They used the same basic block vector system in order to establish phases via clustering techniques, but in this paper they also extend this system to allow phase prediction to take place. Sherwood *at al.* implemented a Markhov chain predictor which allowed them to make estimates about the phase immediately following the current phase, based on the previous phases and their durations. They called this the *run length encoding Markhov predictor*. Upon evaluation, their system was demonstrated to be accurate when predicting phase changes. Unfortunately, it suffers from the same drawbacks as their clustering research, in that it is too complex to be suitable for runtime profiling of a dynamic affinity scheduler.

In [9], Duesterwald *at al.* discuss methods for identifying and predicting program behaviour at runtime. By taking samples from hardware counters, the authors were able to observe patterns in behaviour. They note that behaviour varies enough such that prediction is difficult; however, most programs displayed periodic behaviour. As such their future behaviour could be inferred from their histories. They also noted that different programs had similar periodic behaviour, and as such the behaviour of one program could be predicted from the history of another. They discussed prediction with simple statistical models such as a last value predictor, where the previous behaviour is taken to represent the future behaviour; they also explored more complex behavioural systems, such as table-based history predictors that identify behavioural patterns which repeat over time.

The implications of this paper are interesting; by demonstrating that program phases are usually cyclic, Duesterwald *at al.* have effectively suggested that very primitive techniques can be used in phase prediction. One of the strengths of the approach used by Duesterwald *at al.* in [9] is that it can function on any hardware counters that are available. While hardware counters cannot be guaranteed to be present between heterogeneous cores, this demonstrates that phase prediction and identification can be achieved at a low cost using very small amounts of data.

2.2 Scheduling

One of the first research groups to consider the advantages of heterogeneous single-chip multi-processors began by investigating how such a processor could be used to reduce power consumption. Led by Rakesh Kumar the team, published the results of their research in 2003 [13]. Their research demonstrated that heterogeneous cores are likely to have different power requirements, even in a single-chip multiprocessor. The authors demonstrated a dynamic scheduler could be used to reduce overall power use. Their scheduler made decisions based on power and performance requirements, re-scheduling where possible to reduce power usage without any major detriment to performance. When compared with a static scheduler, their power-aware dynamic scheduler demonstrated large energy savings with a small performance degradation.

The authors conclude that a heterogeneous chip multiprocessor could offer such improvements with as little as two heterogeneous cores, noting that more heterogeneous cores would offer greater bonuses. This paper is important because it compares a dynamic and static scheduler, highlighting a situation where a dynamic scheduler is better able to take advantage of a heterogeneous design. The paper is cited by a number of important research papers in this field, and the authors themselves used it as a basis for further research.

In 2004, Kumar *at al.* published another paper that built heavily upon their earlier research[14]. The authors compare a static scheduler in a homogeneous environment to both dynamic and static schedulers in a heterogeneous environment. When tested with a multi-threaded workload, the authors demonstrated considerable performance increases in the heterogeneous system for both static and dynamic scheduling, with the dynamic scheduling system offering the greatest performance increase. When comparing the architectures using static schedulers, the authors found that a statically scheduled homogeneous system would saturate its job queue long before a similar heterogeneous system. When compared to a dynamic scheduler, the throughput of the system increased again. In order to measure the performance of the systems, the authors measured the instructions completed per cycle, and compared the ratio between the systems; this seems a strong way to compare the performance of the system. The experimental workload was taken from the SPEC benchmark suite, and the jobs themselves selected and issued randomly; this helped to ensure that the results were free from bias. The authors acknowledged that they were using workload construction techniques similar to a number of other researchers. Their experiment appears thorough and the results are well presented.

This paper is important to my research problem given that it outlines a dynamic scheduler aimed at increasing performance in a heterogeneous environment. Their dynamic scheduler is very primitive, a fact

acknowledged by the authors. Considering future work the authors suggest that more complex heuristics could be employed to improve scheduling, such as "cache or branch behaviour, or basic block profiles...". Another suggestion is that a more complex model for phase prediction could be used to make pre-emptive scheduling decisions.

The work by M. Becchi & P. Crowley [2] describes an experiment comparing the differences between static allocation and dynamic allocation of processes in a heterogeneous environment. Their work bears a number of similarities to previous work by Kumar *at al.* and they cite their work amongst their main sources. There are a number of similarities between their paper and Kumar's 2004 paper, an issue which is addressed by the authors in this paper. The model presented in Becchi & Crowley's paper considers a number of additional factors, with the most prominent difference between the two being that Becchi & Crowley demonstrate a thread-driven dynamic scheduler that reschedules when conditions change in the environment. Comparatively, the technique employed by Kumar *at al.* in [14] creates new schedules in bounded time cycles, basing the new schedule on the execution behaviour during the previous cycle. Becchi & Crowley claim that this is no more than standard static scheduling at regular intervals, and that their system bears a greater similarity to a truly reactive dynamic system.

Becchi & Crowley demonstrate a more complete model in their paper, showing how a heterogeneous system could be modelled in a manner that allows it to react to changes in the system. While they are using the same metrics to trigger the change, they have improved the quality of the scheduler presented by Kumar *at al.* in [14]. At the thread level, this involves considering how to initially allocate threads to processors and how to store and work with the heuristics that will decide processor affinities. An important point is that their dynamic scheduler uses metrics collected entirely at runtime; this was necessary given the high level metrics the authors chose to work with, yet they still demonstrated important techniques for runtime profiling. The paper also discusses migration patterns and their overheads in considerable detail.

There are a number of published papers that address the issues of dynamic scheduling in a heterogeneous system. In a 2007 paper, Sondag *at al.* in [21] published the results of research into dynamic affinity scheduling in a heterogeneous single-chip multiprocessor. Their research bears a number of similarities to the research proposed here. Thread and core affinities are calculated based on behavioural properties; their research considers arithmetic, data and control based code categories. An initial offline analysis of randomly selected basic blocks identifies code behavioural clusters. The dynamic scheduler will hold a record of how well each core completed a cluster. If a core performed well on a cluster, then new code that is assigned to this cluster is also assigned to the core that particular cluster is attached to, with new code being evaluated in the same manner as the initial configuration

code. As a result of their approach, the authors claim their system will help to reduce profiling overheads while reducing the stress on the programmer by hiding the complexity of the system. However, the system proposed by the author has not been evaluated.

In [4], Blagojevic *at al.* demonstrates a multi-grain scheduling system for the Cell processor. Initially, they identified the tasks which represent a majority of executed code, and outsourced them to the Cell's SPEs, which are small streamlined processors designed for media-based workloads. This resulted in poorer performance compared to no outsourcing at all: the authors claimed this was due to the code not being optimized for the SPE's architecture. Once the code was optimized, they implemented Task Level Parallelism, or TLP, which outsourced tasks to SPEs. The PPE, which is intended to handle general purpose workloads and oversee the SPEs, was scheduled with more than one process so that execution on one could continue while the other was offloading a task. This also increased the number of tasks to improve the overall SPE utilization. They discovered that this frequently left some SPEs idle, so Loop Level Parallelism, or LLP, was also implemented. This allowed loops to be scheduled to idle SPEs when the SPE utilization was low. Testing their system against the standard Linux process scheduler, they concluded that theirs had greater performance.

The scheduler Blagojevic *at al.* outlined did not take full advantage of the heterogeneous nature of the Cell, as no workload to core affinities were considered when making scheduling decisions. Their system is also application dependant, requiring heavy modification of the original code to achieve compatibility with the Cell architecture.

Blagojevic *at al.* also published [5] in 2007, which built heavily upon their work in [4]. In this paper they modified their scheduler by adding a sampling phase, allowing them to determine the optimum number of SPEs to allocate to each process. The scheduler will re-evaluate periodically by searching over both task and loop level parallelism to determine the schedule with the highest utilization. There are a number of flaws with the method described. For example, the scheduler assumes that all tasks have the same execution time; another issue is that searching for optimal schedules uses an exhaustive search. While the authors acknowledge these issues and offer solutions, these issues damage the credibility of their work.

In [15], Li *at al.* demonstrate a scheduling system called AMPS which allows an operating system to exploit a heterogeneous architecture with minimal modification. AMPS provides load balancing by ensuring that the assigned workload matches the processing power of the core, which is calculated online when the operating system boots. AMPS also uses this information to provide a fastest-core-first algorithm that migrates threads to the fastest cores whenever it detects that they are underutilised. The third feature of AMPS is a system that aims to determine the cost of a migration in a Non-Uniform Memory Access based system.

The system the authors describe is very simple, and only attempts to identify if a migration from one core to another will have a high or low cost. High costs were identified by determining with a few counters if the migration would cause a high number of cache misses and remote memory accesses; only low-cost migrations would be permitted. In evaluating their system, they demonstrated improved performance compared to stock Linux schedulers.

The system described by Li *at al.* in [15] for determining migration costs is lightweight and innovative, and a similar system could prove useful in another heterogeneous architecture. However, it would also require considerable modification; for example, in the Cell processor all the cores run at the same speed. The model used by Li *at al.* would require expansion in order to accommodate for different heterogeneous properties.

A project of interest is *CellSs*, an IBM-funded research project carried out by Bellens & Perez *at al.*. In their first paper [3] the authors apply superscalar techniques to the Cell processor. Their aim in this is to abstract the complexity of the Cell processor in order to reduce the mental workload of the programmer. In order to achieve this, CellSs uses simple tags inserted by the programmer to parallelize sequential code. These tags serve two roles. At compile time they are used to create two sets of files; the main program files which execute on the general purpose PPE core, and task programs that execute on the SPE cores designed for media processing. At runtime the tags are used to form a graph of the tasks and their data dependencies. However, all of this is transparent to the programmer, who simply uses annotations to indicate basic properties or signal that a task should be run on the SPE.

The following year, the same research group published another paper [17] which explored in greater detail the effects and applications of the CellSs model. They demonstrate that hardware trends are producing different kinds of multiple core systems, and argue that programmers require systems that will abstract the multi-core hardware. By allowing programmers to continue to write sequential code, leaving parallelisation to the compiler, they claim it will ease the burden of writing for a complex hardware system. They compare CellSs to other programming models for the Cell, identifying the differences between them.

It could be argued that the CellSs system is logically flawed. Although the system abstracts the complexities of the hardware layer, the programmer is still required to annotate code in order to allow the system to function. Without any understanding of the underlying hardware, programmers will be unable to tag their code in a manner that leads to efficient execution. Another issue is the encouragement of sequential programming: it is much better to promote a synchronous programming model when considering a multi-core processor. A third important issue is that their system does not take advantage of processor

affinities.

2.3 Conclusion

In part 2.2, a number of papers are examined that address the problem of heterogeneous scheduling. In [13, 14, 2, 21] runtime profiling is applied the basic heterogeneous schedulers in order to make more effective use of a heterogeneous system. However, these techniques are very basic and do not extend to architectures like the Cell, where the different cores have distinct differences. The CellSs [3, 17] project falls short of the mark in easing the programming burden, and also fails to take full advantage of the Cell's design.

Similarly, Section 2.1 revealed that runtime measuring techniques are well established for runtime optimisation systems such as Dynamo [1]. Behaviour tracking and prediction systems have been thoroughly investigated; however, it also appears that these techniques have never been used to determine runtime affinity properties, with one notable exception [21].

This demonstrates that a dynamic affinity scheduler for a heterogeneous system is an open research problem. While there is a lot of interest in heterogeneous processors, particularly the Cell, current research efforts do not offer any immediate solutions to problems outlined here.

Chapter 3

Investigation

This chapter will define the specific approach taken based on the results of the survey in chapter 2; this will be outlined in section 3.1. Section 3.2.2 will discuss the resources available and outline how the research will be undertaken, introducing the idea of a prototype scheduler and describing its functionality and design at a high level. Section 3.3.5 presents an investigation into the Cell processor, the heterogeneous processor that will be used during the project. Section 3.4.2 details the development of a prototype dynamic affinity scheduler, before Section 3.5 concludes this chapter.

3.1 Approach

In order to investigate the properties of dynamic affinity scheduling, a prototype dynamic affinity scheduler will be constructed for a heterogeneous architecture.

The literature survey of chapter 2 raised a number of important points that will impact upon the chosen approach. One of the first points raised was that of complexity; in particular, the complexity of the chosen metrics. While Duesterwald & Bala conclude in [8] that very basic metrics could be used just as effectively as complex ones, Kumar *et.al* in [14] believe that the efficiency of their heterogeneous scheduler could be improved with more complex metrics.

In any system that makes scheduling decisions at runtime, it is important to keep overheads and complexity to a minimum. Therefore, small and simple metrics would be desirable. As such, most researchers use power usage or processor speed in order to determine a schedule; however these metrics may be too basic for a number of processors, ignoring real heterogeneous properties.

Li *et.al* approached this issue in [15] by attempting to identify behavioural affinities between processor cores. Recognising that a het-

erogeneous processor may be heterogeneous in a number of ways, their system performs an investigation when the system boots in order to determine the type of processors present and their affinities to a particular workload. Unfortunately, their system will incur considerable overheads at runtime due to the algorithms and methods used.

Clustering techniques were employed by both Sondag *et.al* in [21] and by Li *et.al* in [15]. Using clustering techniques to identify and group common behavioural patterns has a number of problems in the context of a runtime scheduler. It's mathematically heavy, there may not be enough clusters to represent the entire range of program behaviours, behavioural clusters could be wrongly identified and correcting the initial cluster patterns may be expensive.

Most of the techniques described involved some level of basic block distribution analysis, first described by Sherwood *et.al* in [18]. This technique was demonstrated to be effective at by Dhodapkar & Smith in [7] at identifying program phases, although Sherwood *et.al* then began using basic block analysis as a base for developing clustering techniques[18].

Duesterwald *et.al* in [9] suggest that a simpler method may be possible. Using hardware counters to identify very basic metrics, they were able to demonstrate that most programs have a heterogeneous workload which utilizes the hardware in different ways depending on the current phase. They were also able to demonstrate that program behaviour is generally cyclic, and by extension they were able to conclude that previous program behaviour can be used to predict future program behaviour.

This implies that very basic metrics could be used to identify program behavioural phases, and that such metrics could also be used to predict future behaviour at a low cost. In [9] Duesterwald *et.al* use hardware counters to identify the specific runtime processing requirements of certain workloads; similarly, in [15] Li *et.al* used hardware performance tests to identify processing affinities.

A number of conclusions can be drawn from the results of previous research. First, that a set of simple metrics based on hardware requirements could be used to identify affinities. Second, that a technique similar to basic block distribution analysis could be used to determine the actual behaviour of running code. Thirdly, that the heterogeneous nature of the system must be well understood if affinities are to be identified.

Based on these conclusions, a system that determines affinities should have a prior knowledge of the processor's heterogeneous nature. It should also be able to 'score' a given basic block by identifying the core best equipped to run the code it contains. All of this could be accomplished at compile time by leaving 'tags' in the compiled code

that represents the affinities of each basic block.

At runtime, the scheduler could read these tags as they are found and use them to determine an overall score for the thread. An aging function would ensure that the score represents the thread's current behaviour. This would allow threads to be migrated as their affinities change over time.

This will be demonstrated through the implementation of a prototype dynamic affinity scheduler for the heterogeneous Cell processor. First, the heterogeneous properties of the Cell processor will be explored in order to identify the relationship between different workloads and the processing cores of the Cell.

Once these relationships have been identified, a scoring system will be developed that will tag code at compile time. Finally, a scheduler will be implemented that will use these tags to score threads, making scheduling decisions based on which core has the highest affinity for the threads perceived workload.

3.2 Foundation

The prototype scheduler will be developed for the Cell Broadband engine; the architecture of this processor will be explored in part 3.2.1. The Cell is a high-profile heterogeneous processor that powers the Playstation 3 games console. Designed primarily for media and games processing, it's also being deployed in servers and other systems as developers realise that it has considerable potential; the Cell processor contributes a considerable amount of power to the IBM Roadrunner, currently the fastest supercomputer in the world and the first supercomputer to exceed 1 Petaflop of processing power.[1] A network of Playstation 3 consoles also allowed the folding@home project to break the world record for distributed network processing power, by using a network of 670,000 Playstation 3 consoles to increase their throughput from 250 Teraflops to over 1 Petaflop.[2]

The work of McIlroy at the University of Glasgow involves modifying the Jikes Research Virtual Machine (RVM), a variation on the Java Virtual Machine (JVM), allowing it to run directly on the Cell processor. McIlroy has developed a system that translates Java Bytecode into the assembly language for each processing core of the Cell, meaning that normal Java code can be run on the Cell processor. The work presented here will involve further modification of Jikes RVM by replacing the annotation based scheduler designed by McIlroy with a dynamic affinity scheduler. Jikes RVM and McIlroy's modifications will be examined in part 3.2.2.

3.2.1 The Cell Broadband Engine

In the paper "Introduction to the Cell Multiprocessor"[11], Kahle *et.al* discuss the design and development of the Cell. Aiming for high performance in multimedia applications and games, the developers designed a processor with a single *Power Processor Element*, or PPE, and 8 '*Synergistic Processor Elements*', or SPEs. The PPE is a more general purpose processor based on the well established Power Architecture in order to make the system more accessible. Conversely, the SPEs are very small lightweight processors designed from the ground up to excel at a particular workload.

The PPE and the SPEs are arranged on a single die as shown in figure 3.1. The main components and their relative sizes are shown; the Synergistic Processing Elements (SPE); the Power Processing Element (PPE), with the its associated hardware cache; the Element Interconnect Bus (EIB); the input/output controllers and memory interface. The entire chip runs at the same speed, which is $3.2 GHz$ in the model

[1] Source: Top 500, June 2008. http://top500.org/lists/2008/06

[2] Source: BBC News, 2nd Nov 2007. http://news.bbc.co.uk/2/hi/technology/7074547.stm

used in the Playstation 3. The diagram in figure 3.2 shows how the processing cores are arranged and their relationship with the EIB.

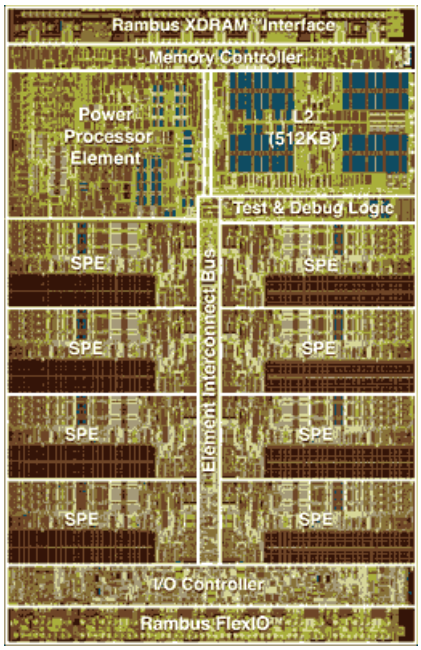

Figure 3.1: A close up of the Cell die, with the main components outlined.

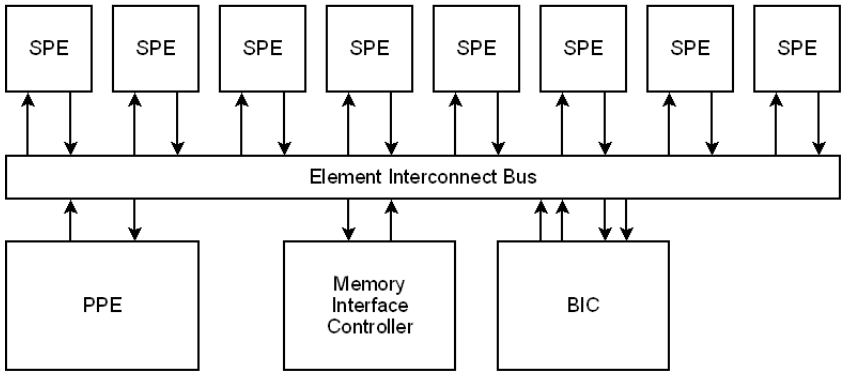

Figure 3.2: A diagram of the Cell's main components.

The PPE is part of the Power architecture family, and as such provides a familiar environment for Cell developers. It is fully compatible with the 64-bit Power architecture, which the developers hoped would help to ease the transition from conventional processor designs. Existing Power applications are able to run on the Cell without modification, but doing so would result in low utilization of the Cell's cores.

It was vital for the the Cell processor to have a familiar starting point to ease the transition from conventional processor designs. The Power architecture would be a good choice for this, as it dates back to 1990. The power architecture is also used by all of the Playstation 3's com-

petitors in the games console market, adding to the the appeal of the processor and encouraging cross-platform development.

The PPE is designed to be capable of managing the entire processor, and is equipped with additional instructions that allow it to control the SPEs; because of this, the SPEs are sometimes referred to as subordinate processors. It is equipped with a Direct Memory Access (DMA) controller allowing it to offload some of the work involved in fetching and storing from main memory; however it is also capable of accessing main memory through normal load and store instructions, along with the local memory of each SPE without requiring the DMA controller.

The PPE itself has 64KB of L1 cache split evenly between data and instructions, and 512KB of L2 cache. The Power core is equipped with a 23 stage pipeline, which uses a branch predictor to help protect the PPE from pipeline stalls. The core itself uses a dual-issue in-order design that interleaves instructions from two computational threads; this results in the processor appearing to have two distinct threads of execution, a technique intended to further increase the pipeline efficiency.

The Power Core of the PPE contains three execution units; an instruction unit for fetch, decode, branch, issue and completion; a fixed-point execution unit, which handles loads, stores and fixed-point arithmetic; and an AltiVec vector scalar unit, which is fully pipelined for all single-precision floating point arithmetic.

The SPE is not a familiar architecture for most developers, as it has been designed from the ground-up for a specific purpose. It is examined in detail in "The Microarchitecture of the Synergistic Processor for a Cell Processor"[10], Flachs *et.al* describe the SPE as a RISC-style processor that uses 32-bit fixed length Single-Instruction Multiple-Data, or SIMD, instructions. Programmers writing for the Cell would be required to understand both the Power instruction set and the SPE's SIMD instruction set. The SPE is designed to be managed externally; while it is a capable processor, it must be primed by an external processor before it is able to begin working. In the Cell, the external processor is the PPE.

Each SPE contains is comprised of two components; a *Synergistic Processing Unit* (SPU) and a Memory Flow Controller (MFC). The SPU uses the MFC to communicate with the rest of the system and main memory via the EIB; in fact, the SPU is completely reliant on the MFC and has no direct access to any other part of the system. The relationship between these components is shown in figure 3.3; the arrows show that communication is unidirectional through the MFC.

In "Cell Multiprocessor Communication Network: Built for Speed",[12] Kistler *et.al* describe the nature and capabilities of the MFC. The SPU communicates with the MFC through a unidirectional channel interface, which other SPE's and the PPE can write to. The MFC contains

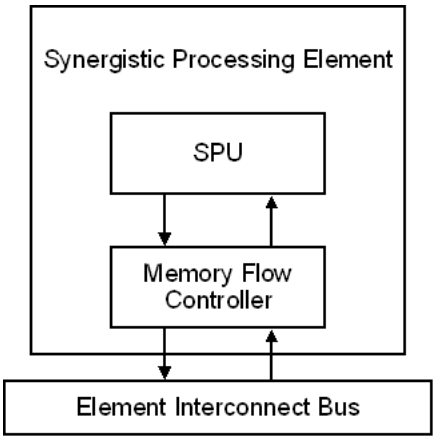

Figure 3.3: The components of the SPE.

a DMA controller, which SPU must use to access main memory. DMA requests queue up in the MFC, and can be issued by any of the other cores to a depth of 16 commands. The DMA controller can transfer up to 16KB of data in a single operation, which helps to compensate for the high costs of relying on DMA for all main memory access. The MFC also handles all signal notifications and mailboxes, providing a number of ways to manage communication between the Cell's cores.

The SPU represents the actual processing core of the SPE. Each SPU contains 256KB of memory for instructions and data known as the local store, a processing core, and a channel unit that communicates with the MFC. The local store is mapped onto the memory map of the processor, which allows the SPUs to access each other's local store through DMA requests. As the PPE can access memory through standard load and store operations, it is also capable of directly accessing the local store of each SPU. The local store contains no hardware cache, and is designed so that the programmer is responsible for managing this memory. Small, fast memory of this kind is known as *scratch-pad memory*. The local store itself is the largest part of the SPU, and as such was implemented using a single SRAM cell to minimise area and costs while maintaining a high level of efficiency.

The processing core of the SPU contains two pipelines, an 'even' pipeline and an 'odd' pipeline. The even pipeline handles arithmetic and word shift operations, while the odd pipeline handles quad-word-shift, branch, load and store operations. Instructions are dual-issued, allowing two instructions to be issued each cycle: one to each pipeline. Programs are executed in-order, with a logic unit that decides which instructions can be dual-issued. The SPE is fully pipelined for single-precision floating-point arithmetic, and is able to direct the output of an operation straight into another functional unit to reduce latencies.

The SPU is not equipped with a branch predictor, always assuming that each branch is not taken. To compensate for the lack of branch

predictor, the programmer is able to insert 'branch hints' that suggest which path is most likely to be taken. "The Microarchitecture of the Synergistic Processor for a Cell Processor"[10], specifies that each branch miss costs exactly 18 cycles in the SPE.

While each SPE is identical in the Cell, the differences between the SPEs and the PPE are very pronounced. While all the cores run at the same clock speed of 3.2GHz, the architectural differences between the PPE and the SPE mean that each core may be better suited to a particular workloads, an idea that underpins this project. The SPE and the PPE differ in their instruction sets and instruction types, pipeline construction, memory access capabilities and capacities, functional units and in many other ways. Due to these pronounced differences the Cell processor is considered to be a highly heterogeneous processor that is very difficult to program for; as such it is an ideal subject for this research.

3.2.2 Hera JVM

Hera JVM is a Java Virtual Machine, or JVM, written in Java. It is based on Jikes RVM, which is in turn based on the *Jalapeno Dynamic Optimizing Compiler*[6] which optimizes Java code at runtime. Jalapeno was built on two compilers; a baseline compiler which ran prior to runtime, and a dynamic re-compiler which handles any runtime optimizations.

Jikes RVM grew from the Jalapeno project as an internal IBM project, but was eventually released as open-source software in 2001 with support for the Power architecture. The Jikes RVM system has a number of notable features. 'VM Magic' allows the compiler to intervene during compilation and inject code at certain points. This allows the compiler to provide low-level mechanisms, such as providing direct access to memory. Jikes RVM also has a Memory Management Toolkit which heavily exploits 'VM Magic'. Jikes RVM also uses open-source alternatives to the standard Java class libraries, such as GNU Classpath.

Hera JVM is a research project built on Juikes RVM under construction at the University of Glasgow [16]. Hera JVM aims to demonstrate techniques for abstracting complex heterogeneous processors, in particular the Cell. Hera JVM uses the existing Jikes Power architecture compiler for the PPE, and contains a new compiler for the SPE. Threads are mapped to cores, and Java threads are able to migrate between the two cores types through just-in-time compilation.

In order to abstract the heterogeneous nature of the Cell further, Hera JVM also adds a considerable amount of functionality to the virtual machine. With Hera JVM, the SPEs are able to run multiple software threads, and a software cache in the local store of each SPE eliminates the need for the programmer to manage SPE memory themselves. This

allows Java code to run on either the SPE or the PPE without requiring heavy modification. Hera JVM abstracts most of the complexity including memory management, instruction set differences and thread migration.

Hera JVM contains a static scheduler that uses annotations to make scheduling decisions; annotations either directly specify the processor that code should run upon, or describe the expected program behaviour. In the latter case, scheduling decisions are made based on which processing core is best equipped to handle the workload described.

This means that the heterogeneous processor is abstracted between a combination of the annotations and automatic systems such as the software cache. The annotations themselves could be simplified, made optional or removed completely if the static scheduler was replaced with a dynamic affinity scheduler.

If the dynamic affinity scheduler was suitably efficient, it would be superior to an annotation-based scheduler in a number of ways. The most immediate benefit is that programmers are no longer required to add annotations to their code, allowing any Java program to run on the cores it has an affinity for. Affinity scheduling would correct mistakes made by the programmer when adding annotations, such as adding the wrong annotation or forgetting an annotation. A dynamic affinity scheduler would also allow threads to migrate as their affinities change; Hera JVM allows threads to migrate if they have the correct annotations.

The Cell processor is a high-profile heterogeneous architecture with a number of real-world applications, and Hera JVM provides an effective demonstration of how such an architecture could be abstracted. Implementing a dynamic affinity scheduler in the Hera JVM would complete the abstraction of the heterogeneous architecture, and demonstrate the effectiveness of affinity scheduling techniques.

3.3 Cell Investigation

In order to develop a suitable scheduler, it is important to first identify which heterogeneous properties of the Cell processor would offer the greatest benefits when exploited by a dynamic affinity scheduler. There are a wide range of potential affinities in the Cell processor, each likely to offer different levels of performance gain. An experiment will be run in order to determine exactly which affinities should be focused upon.

3.3.1 Predicted Affinities

Hera JVM has already tackled a number of heterogeneous issues, such as adding a software cache for the SPE's local memory. In order to abstract heterogeneous behaviour, McIlroy intended to add an annotation based static scheduler to Hera JVM. In his second year PhD report, McIlroy specifies a number of annotations along with the behaviour they are intended to represent. These annotations would be an ideal starting point for predicting the affinity properties should be used to make scheduling decisions.

The proposed annotations are capable of identifying arithmetic, memory access, blocking, data ownership, and thread communication properties. For example, some of these annotations allow the programmer to identify floating point or integer based code, while another group allows the author to identify producer/consumer relationships between threads. As the dynamic scheduler aims to use only simple metrics, only the metrics which will offer the highest returns should be considered.

Arithmetic annotations consider floating point and integer calculations. The SPEs of the Cell are designed to be substantially faster than the PPE at arithmetic, in particular single-precision floating-point workloads. In "The potential of the cell processor for scientific computing"[22], Williams *et.al* discuss the floating point abilities of the Cell and demonstrate that the SPEs of the Cell are extremely efficient at floating point arithmetic when compared to other processors such as the Intel Itanium. As discussed in section 3.2.1, the SPE's of the Cell were designed specifically for this purpose. This suggests that arithmetic affinities should be the first area of investigation for determining affinities.

Execution behaviour annotations concern memory access, the use of heap and stack memory, input/output access and blocking behaviour. The SPEs within the Cell are limited to accessing only their own local memory directly; for everything else they are forced to use DMA accesses. Through DMA, they are capable of reading and writing large pieces of data in a single DMA request. However, for smaller pieces of data the overheads of making a DMA request quickly become very

high. The MFC unit of each SPE is only capable of holding 16 outstanding DMA requests, and coupled with the relatively large amount of memory available per SPE, it is clear that the SPE is not intended for applications that make frequent memory accesses. The PPE has no issue with memory access, having both a DMA controller and direct access to all the memory in the system. Thread memory access patterns are a difficult thing to measure, especially if heap or stack usage are considered. However, it may be possible to use a low-level metric to represent memory access frequency, which may reflect these affinity properties to some degree.

Data ownership annotations allow the static scheduler to assign threads based on information sharing. Annotations are suggested to signify when data is shared or local, and whether shared data can be accessed simultaneously by different sources. Theoretically, threads which share data will have lower overheads if they are assigned to the same core. Unfortunately, Hera JVM clears the software cache on a context switch, causing a number of problems. If two threads sharing data were both on the SPE, then the local cache would be cleared at every context switch. If on different SPEs, then DMA would be required for each access. If on the PPE, the problems would be reduced. If a mixture of PPE and SPE, it would be a one-way arrangement with only the PPE able to quickly access the data. Therefore, any threads which have a data-sharing arrangement would need to run on the PPE. This would be a complex system to model, and as such representing it in a low-level metric would be difficult.

Thread communication annotations are also specified, able to signify that threads belong to a particular group and the communications between these groups. Annotations are also outlined that can signify producer consumer relationships, or threads which are a part of more than one group. As with data ownership, this information can have an impact on scheduling when exploited. If threads are assigned to cores so that one thread runs while another is blocked, then it reduces resource contention. Unfortunately, this suffers from the same problems of data ownership; cache-clearing on the SPE's and the complexity of modelling this behaviour make it unsuitable for low-level metrics.

The annotations outlined by McIlroy suggest that arithmetic and memory access operations are likely to demonstrate the most clear affinities. However, it is also worth considering the architecture of the Cell processor as discussed in section 3.2.1 for other potential affinity properties. One of most notable differences between the PPE and the SPE is the structure of their pipelines. In particular their sizes; the PPE has a long pipeline that, when stalled, would result in significant overheads. To combat this, the PPE is equipped with a branch predictor. The SPE has a very short and simple pipeline that dual-issues instructions. While pipeline stalls cost less on the SPE, they will happen much more often because it is not equipped with a branch predictor. This implies that branch-heavy code would show an affinity for the PPE, where it

could be fully pipelined.

There are three heterogeneous properties that should be investigated. The first is the difference in handling arithmetic; the SPE is designed to be considerably faster than the PPE at this task. The second property is memory usage, which is likely to demonstrate an affinity for the PPE. The third property considers branches in the code, which are also likely to demonstrate an affinity for the PPE.

Memory access could be represented in Java through the usage of Objects. This would include instantiating objects, calling methods and accessing fields. Each of these implies a memory operation, and as such monitoring the usage of object could be used to represent the overall memory usage of a particular thread. This would effectively encompass any cache affinities and the SPE memory access limitations. Therefore, code that handles a large number of object code would be likely to demonstrate an affinity for the PPE due to the combined effects of a hardware cache and direct access to memory.

Most Java code can be represented as either arithmetic, branches, or object code. A given program could be analysed to work out how often each operation appears; these 'code densities' could then be used to calculate an affinity for each program to a particular processor. Hera JVM takes Java Bytecode as an input, so these three properties could be mapped directly to low-level instructions.

This would mean that thread to core affinities could be calculated by monitoring the bytecodes of a program; if there are more bytecodes that show an affinity for the SPE, then the thread should be scheduled to the SPE. Such thread monitoring could happen online or offline; some runtime monitoring would obviously be required, but a system such as basic block distribution analysis, as outlined by Sherwood *et.al* in "Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications"[18] could be used to move the majority of the work to the compilation stage.

Scheduling decisions based on these three properties would depend entirely on the performance gap between the two processors. In order to determine the exact performance ratios between these instructions, an experiment was run to determine performance differences between the SPE and the PPE when faced with arithmetic, object and branch-based workloads.

3.3.2 Arithmetic Affinities

An experiment was written to determine the affinities over different categories of Java bytecode. The first of these categories concerns arithmetic bytecodes, of which there are three sub-categories; integer, long, single-precision floating-point, and double-precision floating-point. For

the remainder of this report, single-precision floating-point will be referred to as simply 'float' and double-precision floating-point will be referred to as simply 'double'.

The aim of the arithmetic test was to write a piece of code and run it on both the PPE and the SPE, recording the time it takes to complete on each core. The magnitude difference between the resulting times is a good representative of code to core affinities.

All the tests were written in Java, with each line of the test designed to invoke a particular java bytecode. For the arithmetic tests, all the primitive types have similar bytecode commands; therefore there was little difficulty in creating similar tests. At the Java level, the code appears to be 100% arithmetic code. At the bytecode level, there are additional load and store operations; for the arithmetic tests, these additional operations reduced the arithmetic operation density to around 25%. Table 3.1 demonstrates how a simple operation, a number multiplied by itself, is compiled into bytecode.

Java	Bytecode
`varA *= varB`	`ILOAD x:varA`
	`ILOAD x:varB`
	`IMUL`
	`ISTORE x:varA`

Table 3.1: *Java Code* Compared to Equivalent *Java Bytecode*.

The aim of each test was to investigate the relationship between the percentage of arithmetic code and the processing core the code was executed on. Four individual tests were run, one for each basic type; integer, long, float and double.

For each basic type, 101 loops over the test were made, with each consecutive test increasing the arithmetic workload by 1%; the remainder of the test code was built from simple get and put field operations. The first iteration comprised 0% of arithmetic, while the final iteration comprised 100% arithmetic code. The entire test was run 5 times for each basic type in order to provide a high level of accuracy.

Each test ran the same code on the PPE and the SPE and recorded the times. When the test code consists of 100% arithmetic code, the speed difference between the processing cores becomes apparent. Table 3.2 shows the results of the final iteration, when the test code contains 100% arithmetic; that is, Table 3.2 demonstrates the actual affinities for arithmetic code.

These results demonstrate that the SPE will always be superior to the PPE with any kind of arithmetic, although with varying degrees of success. It is not surprising that the SPE is significantly faster with floating-point arithmetic, as it was designed specifically for a floating-

point workload. However, when it is considered that both cores run at the same clock speed, a 468% increase in speed is impressive. Double-precision arithmetic demonstrates the lowest increase, yet still runs 269% faster on the SPE than it does on the PPE. Therefore, all arithmetic operations demonstrate a significant affinity towards the SPE cores.

Instruction Type	PPE Time (ms)	SPE Time (ms)	SPE Difference
Integer:	279.2	79.6	351%
Long:	590.2	156.8	375%
Float:	492	105.2	468%
Double:	466.2	173.2	269%

Table 3.2: A Comparison of the Arithmetic Abilities of each Core Type.

Figure 3.4 shows the relationship between the two cores as the percentage of arithmetic code is increased. The long and double-precision arithmetic relationships increase in a linear fashion, which is to be expected given a linear increase in the percentage of arithmetic code. However, basic integer and floating point operations demonstrate a slight curve, showing that a mixture of the two operations produces a smaller increase in the SPE.

There are numerous possible reasons for this slight curve; it could be related to DMA requests in the SPE, or the difference in pipeline structures. However, it lies outside the scope of this experiment to address this question; the aim is to identify affinities, and the results of this test clearly demonstrates that the SPE has a strong affinity for arithmetic workloads.

As these results demonstrate a similar trend, they can be aggregated into a single result. Figure 3.5 shows the average relationship between arithmetic code and the core that it runs on, demonstrating the superiority of the SPE for arithmetic-heavy workloads. The individual results of these tests, along with graphs showing the timings of the results, can be found in Appendix A.

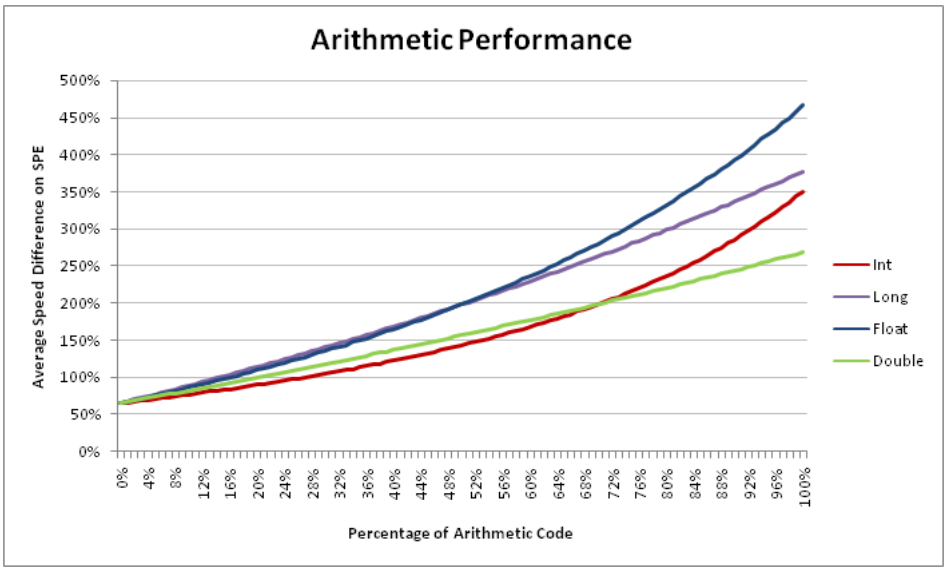

Figure 3.4: Individual Arithmetic Performance Differences between the Cores of the Cell Processor

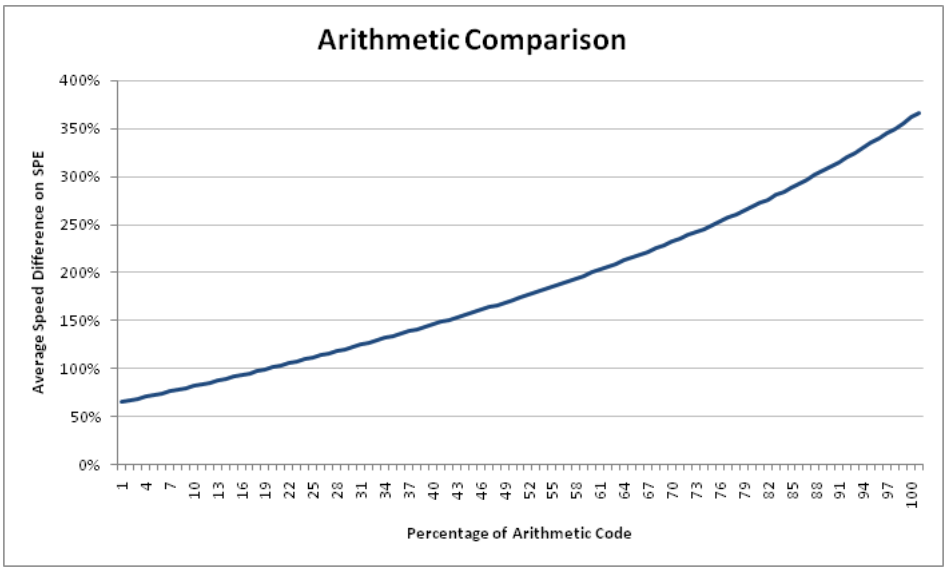

Figure 3.5: Average Arithmetic Performance Differences between the Cores of the Cell Processor

3.3.3 Object Affinities

In Java, object code is can be split into three distinct categories; instantiation, method calls and field references. In Java objects and instantiated through the `new` command, which invokes the constructor method and allocates memory for the new object; this can be quite an expensive operation, and as such may highlight significant affinity differences.

Field references simply mean referring to a non-local variable, which usually require a DMA request on the SPE. Method references are basic function calls and will also require a DMA request on the SPE. Test code was written such that each test contained roughly the same amount of special bytecodes, and covered the whole range of bytecodes for each category. This meant that some of the test methods had to repeat the same operations to ensure that all the tests were of similar size, ensuring that the results could be accurately compared.

When first running this test, the 'new' part of the object test took an extremely long time to run and produced highly inconsistent results. This was due to the Java Garbage Collector, which automatically clears memory when an object is no longer in use. The Java Garbage Collector automatically suspends the running thread in order to clear unused items from the memory. The high cost of repeatedly creating new objects coupled with the efforts of the garbage collector to clean up the unused objects created long testing times and unreliable results.

This problem was solved by reducing the depth of the test; while each phase during execution consists of 5000 iterations, the phase length for any 'new' operations was reduced by a factor of 10 to 500. This allowed for the reading of more accurate results, although the test still took a significant amount of time to complete. As with the arithmetic test, the object test was run 5 times to ensure accuracy.

As with the arithmetic test, this test consists of increasing the percentage of 'special bytecodes' in the test from between 0% and 100%. Similarly, the values at 100% object code are used to describe the actual affinities between the code and the processing cores. Table 3.3 lists the results of this test, and demonstrate that object code will run considerably faster on the PPE.

The methods used for this test were small, but as the SPE is capable of making very wide DMA transfers this is acceptable; also, the methods could not have been made any larger without adding other bytecodes and operations to the test. It is surprising that the result for method invocations is so high, as each invocation is likely to require a DMA access.

It is not altogether surprising that the PPE is significantly faster than the SPE at making field accesses; with direct access to all the memory

in the system, it is a simple matter for the PPE to access any field regardless of where it is kept.

That 'new' operations demonstrate a significant affinity for the PPE is also an expected result. This is likely due to the hardware support and caching abilities of the PPE, while the SPE is slowed by DMA requests and a software cache.

Instruction Type	PPE Time (ms)	SPE Time (ms)	SPE Difference
Method	394.4	467	84%
Field	137.2	212.2	65%
New	2687	4784	56%

Table 3.3: A Comparison of the Ability of each Core to handle Object Code.

Figure 3.6 demonstrates the performance differences and how they change as the percentage of object bytecodes is increased. As before, the remainder of the test code is made up of simple get and set and set operations. However, this means that the field test would be made entirely of the same test code for every iteration; that is, every iteration will contain 100% field code. As this has been used as a basis of comparison for all the other tests, changing the test to prevent this would make comparison difficult. As a direct result, the field test appears as a straight line on the graphs.

The 'new' test also has an unusual curve; it has a somewhat inconstant downwards curve until the test code contains roughly 40% 'new' operations. At this point, the graph levels out and becomes a straight line, demonstrating that a peak has been reached; at over 40% 'new' operations the SPE will be around 44% slower than the PPE. Identifying why this peak exists lies outside the scope of this investigation.

The combined scores of these three tests are shown in Figure 3.7. These results are adequate to show that the PPE has an affinity for object code. In the best case, object code will still run 16% slower on the SPE; therefore, where possible object code should be run on the PPE. The full results of each individual test can be found in Appendix A.

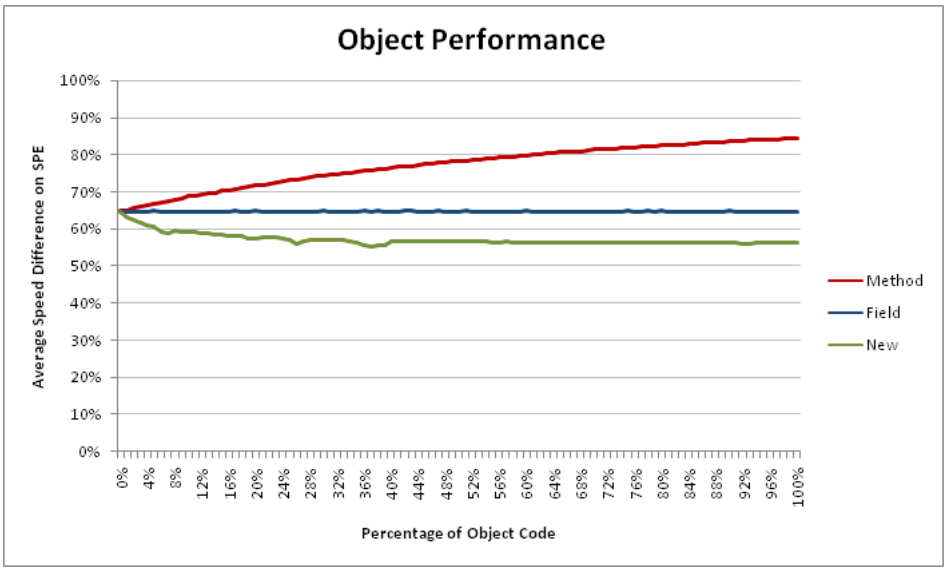

Figure 3.6: Individual Object Performance Differences between the Cores of the Cell Processor

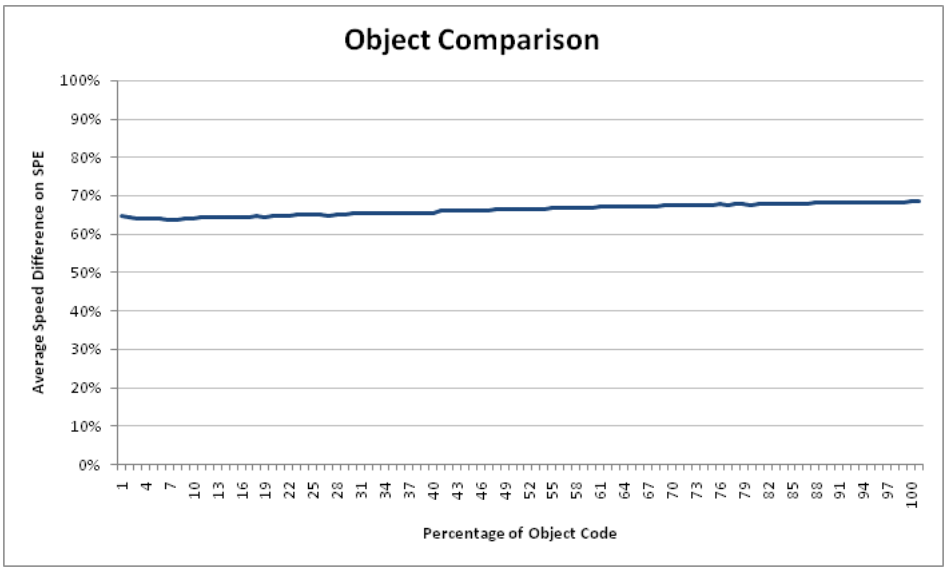

Figure 3.7: Average Object Performance Differences between the Cores of the Cell Processor

3.3.4 Branch Affinities

As the SPE is not equipped with a branch predictor, branch misses would create pipeline stalls. However, the SPE has a significantly different pipeline structure from that of the PPE, which is equipped with a branch predictor. The SPE instruction set provides a 'branch hint' ability, which allows the programmer to suggest the likelihood of a given branch being taken. The presence of this system suggests that branch misses would have a significant impact on the performance of code running on the SPE.

To determine the affinity of branch-heavy code, four branch tests were written to demonstrate different types of branching behaviour. The first of these tests was designed to be perfective, in that the SPE pipeline would stall as little as possible. The SPE always assumes that a branch is not taken, so test code was organised such that this would be the case; the SPE pipeline would theoretically run at optimal efficiency.

The second of these tests was designed to be as destructive as possible to the SPE, by ensuring that branches disrupted the pipeline. It would be expected that this would demonstrate a considerable overhead for the SPE, and as a result of this branch-heavy code would demonstrate an affinity for the PPE.

The third test was an alternating test, which alternated between branch hit and branch misses in order to test the effectiveness of the PPE branch predictor. While the SPE would be expected to demonstrate uniform behaviour without a branch predictor, the PPE's branch predictor would be expected to produce similar results regardless of the path taken. However, if the branch predictor fails on the PPE it will be much more expensive to recover than if it happened on the SPE.

The fourth and final test aims to investigate the ability of both cores to process Java 'case' statements, which are also known as 'switch' statements. This consisted of a number of nested case statements to determine the ability of both processing cores to handle such code.

Table 3.4 shows the results of this experiment. Using the same format as before, the results demonstrate that at 100% branch code, there is very little difference between the two cores. This could be because the branch predictor on the PPE is primitive, or because the SPE's short pipeline allows it to recover quickly from errors. Section 3.2.1 noted that branch misses cost exactly 18 cycles to rectify on the SPE; it would seem that this is not a major issue, and that regardless of behaviour branch-heavy code does not show a particular affinity for either core.

As with the previous experiments, Figure 3.8 demonstrates how the performance changes as the percentage of branch code increases, while Figure 3.9 shows the average affinity of branch code. Based on the re-

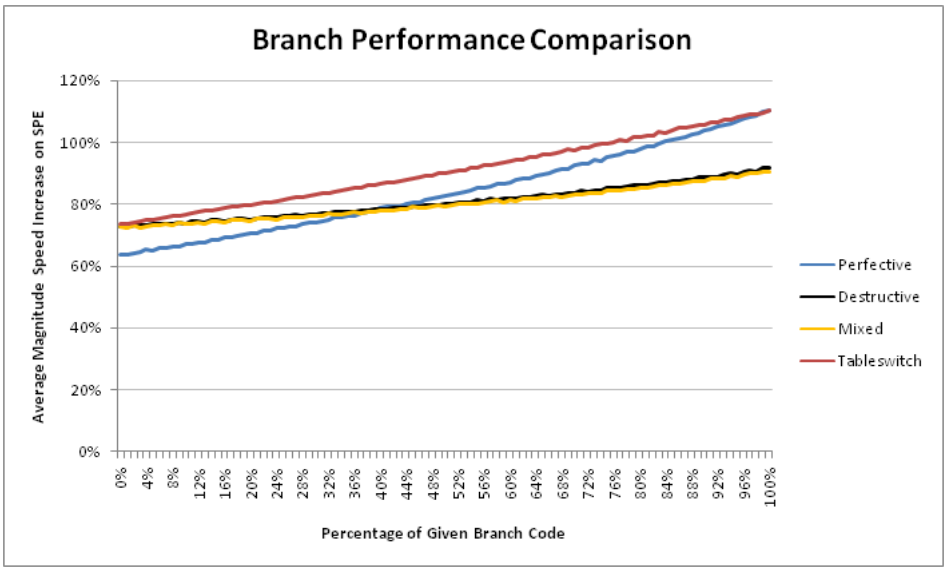

Figure 3.8: Individual Branch Performance Differences between the Cores of the Cell Processor

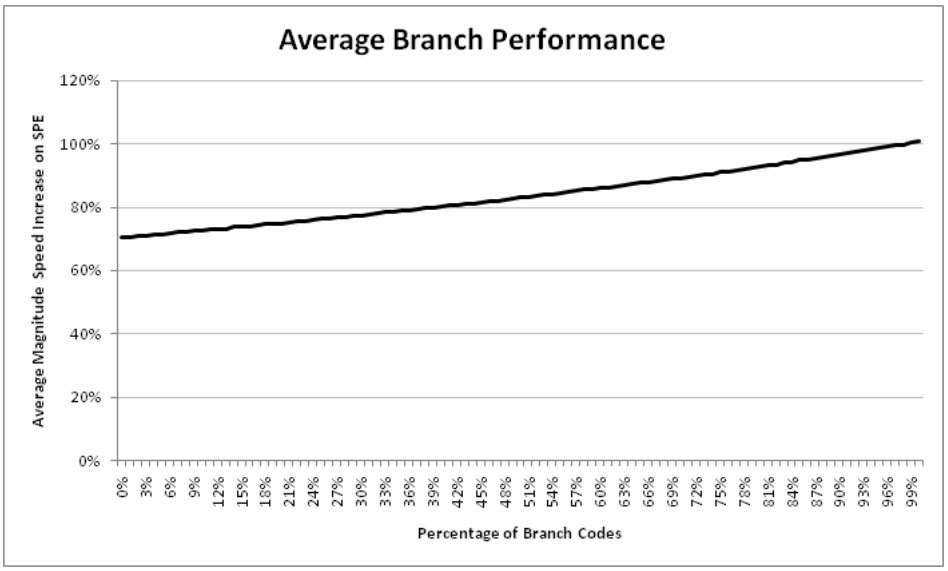

Figure 3.9: Average Branch Performance Differences between the Cores of the Cell Processor

Instruction Type	PPE Time (ms)	SPE Time (ms)	SPE Difference
Perfective:	161	146	110%
Destructive:	118.8	129.6	92%
Alternating:	117.25	129.5	91%
Case/Switch:	195.8	177.4	110%

Table 3.4: The Impact of Branching on the Two Cores.

sults of my test, it seems that branches cost little and have little affinity for either the PPE or the SPE. As with the previous experiments, more detailed results can be found in Appendix A.

3.3.5 Affinity Conclusions

These tests have made the affinity properties of each code group clear. Shown together in Figure 3.10, the relationship between the code type and processing core is quite clear. The 100% line demonstrates that the code has no particular affinity for either core. Results above this line demonstrate an affinity for the SPE, while results below show an affinity for the PPE.

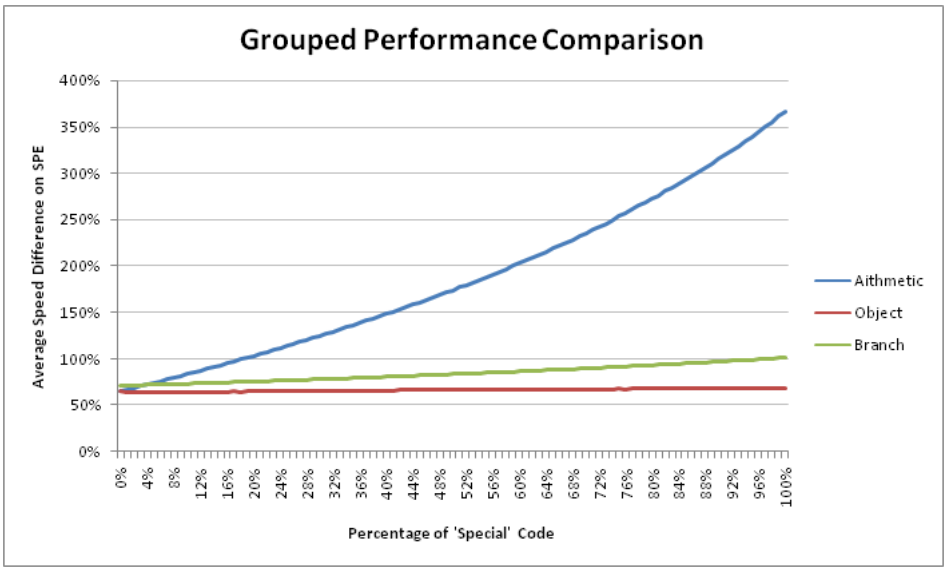

Figure 3.10: Comparison of Average Instruction Performances

At 100%, the actual affinities of each type of code can be compared. Table 3.5 demonstrates the final results of this test; the magnitude performance difference of the same code when run on either core. The most striking of these results is for arithmetic code, which will demonstrate an average speed increase of 366% when run on the SPE. Conversely, object code will run 150% faster on the PPE. Branches in code seems make little difference and can be scheduled to either processor.

Code Category	SPE	PPE
Arithmetic:	3.66	0.28
Object:	0.68	1.5
Branch:	1.01	0.99

Table 3.5: Code to Core Affinities for each Bytecode Category.

Figure 3.11 shows the speed difference for each type of code when scheduled on the SPE. This information would allow a scheduler that monitors java bytecode to make scheduling decisions based on the fraction of either arithmetic or object code. While a scheduler could run it's own tests to determine code to core affinities, as Li *et.al* demonstrated with their AMPS system[15]; however, running such tests is

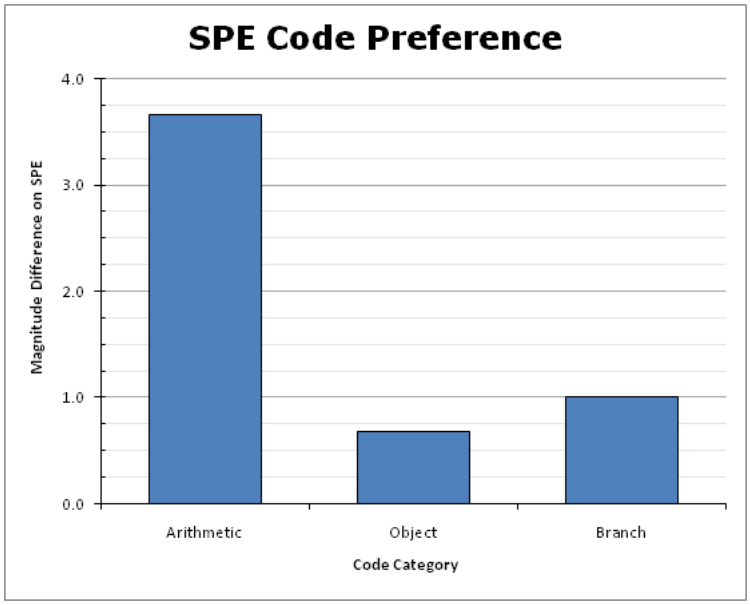

Figure 3.11: Instruction Type to Core Affinities

costly and the same effect could be achieved through offline analysis of the architecture.

This information will provide the basis for a prototype dynamic affinity scheduler; this experiment has demonstrated not only that the Cell's cores have heterogeneous processing abilities, but also that the performance difference between cores is significant.

3.4 Prototype Design

This section will detail the design and development of a prototype dynamic affinity scheduler. In section 3.1, it was concluded that the prototype scheduler will analyse the input code and create tags that signify the processing core affinities of that code. Section 3.2.2 concluded that this scheduler will be built into the Hera JVM[16] and will run on the Cell Processor. Section 3.3.5 detailed an investigation into the Cell processor and its heterogeneous properties, and demonstrated that bytecode monitoring could be used to identify code-to-core affinities.

Therefore, the prototype system will monitor the bytecodes of a program. When it is compiled into PPE or SPE assembly, the code can be 'scored', attaching a value to the code that can be used to determine which core this code should be scheduled on. When a thread runs a piece of code, the scores for that code are added to the thread. This information can be used to predict the future behaviour of the thread, as demonstrated by Duesterwald *et.al* in "Phase Tracking and Prediction"[9]. As a thread's affinities change over time, it will migrate between processors.

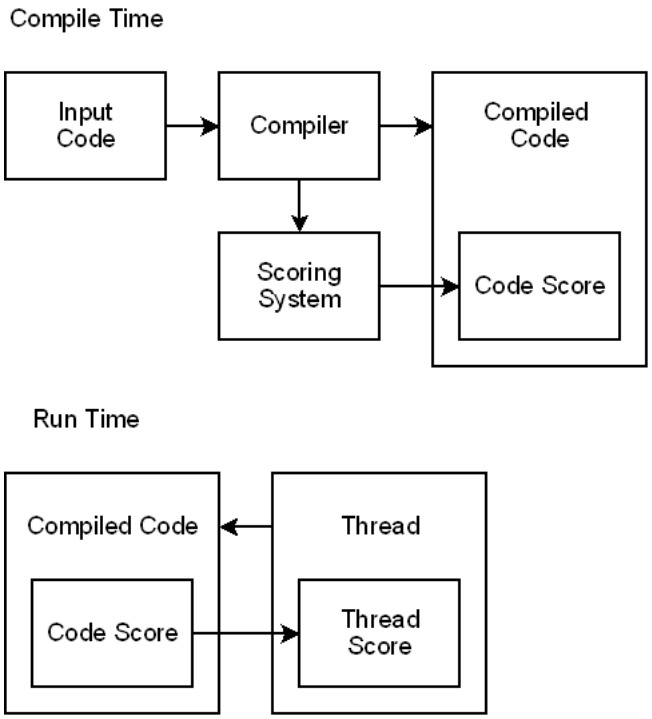

Figure 3.12: The Generation of Thread Affinities

This system would be comprised of two parts; the compile-time scoring system, which will score a given piece of code, and a runtime scheduler which will use these scores to make scheduling decisions. Figure 3.12 demonstrates how each thread will eventually build up a 'score' that

represents where each thread should be scheduled.

Each component of this system will be explored seperately. The compile-time scoring system will be discussed first, in section 3.4.1. The actual dynamic affinity scheduler itself will be discussed in section 3.4.2.

3.4.1 Scoring System

The scoring system will run at compile time and generate 'scores' for each piece of code that it is given. These scores will then be used at runtime to determine the thread-to-core affinities. Hera JVM is a Just-In-Time (JIT) compiler, meaning that code is compiled just before it is run. It takes Java class files as an input, which contains a Java program in *Java Bytecode*. Java bytecode, or simply bytecode, is an intermediate language that Java programs are compiled into to run on a JVM.

In Java, all code is contained in a method. As Hera JVM is a JIT compiler, it compiles each method from bytecode just before it is run. In Hera JVM, each method needs to be compiled twice; once for the PPE, and once for the SPE. The method scoring system runs during the compilation stage, and attaches scores to each compiled method.

Program behaviour is dynamic, so attaching scores to methods is not entirely straightforward; even something as simple as a loop would make a significant difference to the behaviour and affinities of a method. Scores could be attached to basic blocks, in a system similar to the one suggested by Sherwood *et.al* in *"Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications"*[18].

However, such a system would likely impose significant overheads. It was decided that the most important type of dynamic behaviour to consider would be signalled by a backwards branch; a backwards branch would suggests a loop, which is the programming construct most likely to have a significant impact on the runtime behaviour. This would ignore conditional statements, reducing the accuracy of the resulting behavioural description. It was decided that modelling simply the entire method along with backward branches would provide an adequate description of that method's actual runtime behaviour.

Section 3.3.5 concludes that arithmetic code will always run significantly faster on the SPE; therefore the scoring system will aim to calculate exactly how many arithmetic operations are in a given piece of code. It will do this by counting the number of arithmetic bytecodes along with the number of actual bytecodes, which can be used to work out the percentage of arithmetic in the given method.

To account for backwards branches, the code between each backwards branch would also be counted. The score of a backwards branch would

be subtracted from it's parent, and the score for a given branch would be added every time that backwards branch is taken. Figure 3.13 demonstrates how each method would then be scored.

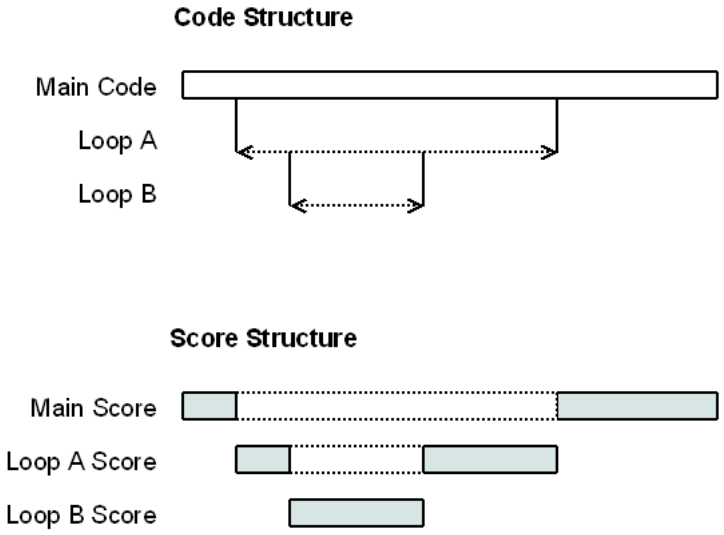

Figure 3.13: Map of a Method Score

The method scoring system itself is reasonably complex; it is required to analyse all of the bytecodes and recognise particular situations, such as backwards branches, and generate scores accordingly. However, a number of interesting issues arose when developing the method scoring system.

The first issue was immediately obvious when the scheduler was first run; it produced unrealistically high results. After using a bytecode viewer to verify that the test code was not signifcantly larger than it was believed to be, it became apparent that the additional bytecodes were in fact coming from the JVM itself. Hera JVM and the Sun Java JVM, along with most of the given APIs, are written in Java. As a result of this, a simple piece of test code that produces "Hello World" as an output produces an extremely high score. This is because prior to running the actual method, the JVM must start up and compile the code. When the eventual score is released, it includes the results of all of these operations.

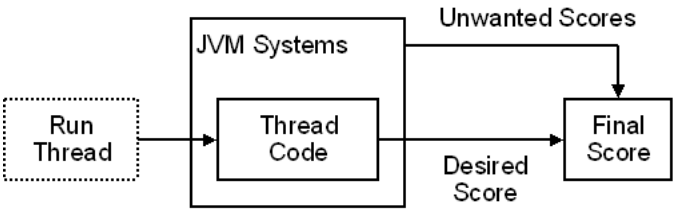

Figure 3.14: The JVM Scoring Problem

This problem was addressed by modifying the scoring system in a num-

ber of ways; first, it was modified to ignore any code which belongs to the JVM. However, it could not be programmed to ignore the standard Java libraries; this would go against the aims of the research. Unfortunately, some parts of the JVM will invoke methods from the standard Java libraries; this cannot be prevented. To address this issue, the score was cleared at certain points to reduce the effects. It was eventually possible to verify that a given piece of test code was completely free from mitigating scores from the JVM by counting bytecodes and verifying output manually; while it would be unwise to claim that this demonstrates the scoring system is completely accurate, it can be claimed that any remaining external influences would be quite unlikely. Figure 3.14 demonstrates this issue.

Another interesting issue was that of method invocations. When a method is invoked, it runs 'seperately' from the method which invoked it. Figure3.15 highlights this problem; should the scores for a method be counted towards the score if its invoker? It was decided that methods should be self-contained, and such their scores should only affect the thread scores, and not the scores of the methods that call them. This was extended so that the method invocations are not counted at all when counting the total number of bytecodes.

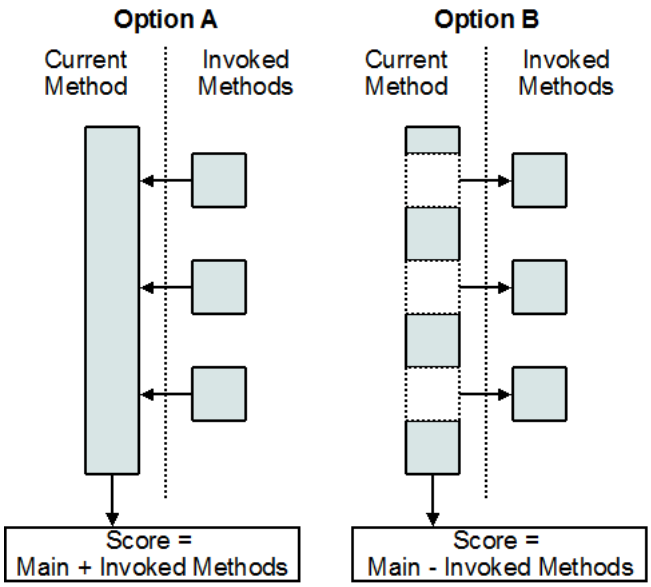

Figure 3.15: The Method Invocation Problem

One test gave very surprising results when it was initially run. The test in question contained advanced mathematical formulae concerning the mass and luminosity of black holes. However, when this test was run it produced a very low score. This low score is attributed to the Java math package, which is written in C; as this package is written in C, the scoring system is unable to score its methods. To counter this, the method scoring system automatically adds a set score to any method which invokes the math package. The added scores are high enough to

ensure that a small number of math package invocations would quickly produce a very high arithmetic score for the method.

This information, once it has been generated, must be accessible by the thread. This is done by adding assembly code to the generated output that contains the method's scores. When the code is run, the values are read out of the assembly code and used to determine the thread affinities. As the SPE and PPE have different instruction sets, two different implementations are required.

Both implementations would require a similar set of modifications. First, the stack would need to be modified for each method invocation to create a space that could hold the scores. Three parts of each compiler would need to be changed; the prologue, which sets up the stack; the epilogue, which finishes a method invocation and returns the results; and the backwards branch generator, where backwards branches are generated.

Altering the PPE stack was not a complex task; it was achieved by adding two extra slots to the stack header, as shown in Figure 3.16. These two extra slots created enough space to store the total bytecodes for a method, and the total number of arithmetic bytecodes for a method. During runtime, this information can be written to and read from the stack by assembly code.

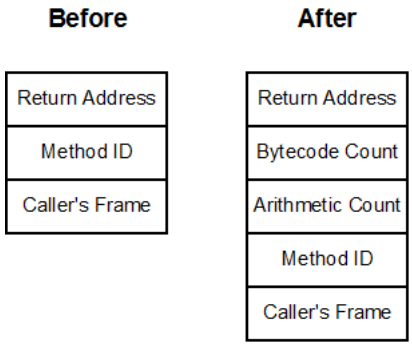

Figure 3.16: Modifications to the PPE Stack

The prologue is the first part of the compiler to be called, and it generates the code that sets up the stack header. This was expanded so that when the prologue is called, it additionally sets up the extra two slots as shown in Figure 3.16. The prologue then requests the method scores from the compiled method object; these starting scores are then written directly into the stack header. If the code contains no backwards branches, then these are the scores that will eventually be passed back to the thread itself.

When the compiler generates the assembly code for a backwards branch, it also requests the scores for this particular branch from the compiled method. Each branch is identified by the bytecode index, that is the position of the branch in the source bytecode. Adding this informa-

tion to the stack header means the current values must be taken from the stack and the new values added to it. Then, the new scores are returned to the stack header. This process is shown in Figure 3.17.

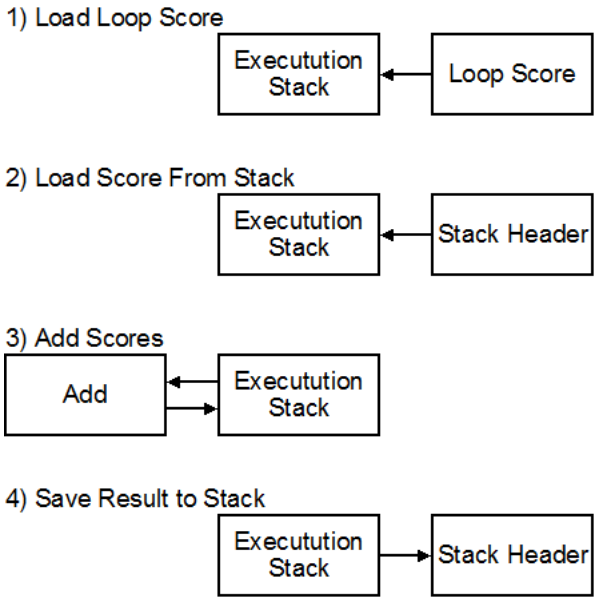

Figure 3.17: Adding a Branch Score to the Current Method Score

At the epilogue for the PPE, the information is taken from the stack and written to the PPE processor object. To do this, two entrypoints were created; entrypoints are accessible from both the assembly code and the high level system. Two additional entrypoints were created on the processor object, which could be addressed during the epilogue section of code generation. This meant that when the epilogue is run, the values are taken from the stack header and written to a field in the processor object; this will allow the scheduler, and subsequently the threads, to access them. This process is demonstrated in Figure 3.18.

The process is slightly different when the SPE is concerned, as it has a different stack structure and a different instruction set. As shown in Figure 3.19, the stack appears very similar, and creating space for the new values did not pose a major challenge. As before, two extra spaces were created on the stack; one for the total bytecode count, and one to hold the number of arithmetic operations that were encountered.

The process for altering the prologue on the SPE was only slightly different from the method used in the PPE, and these changes are related to the instruction set. Similarly, loops were handled as shown in Figure 3.17; the differences between the SPE and the PPE were again related to the difference in instruction sets.

The epilogue for the SPE was significantly different however, as entrypoints are not compatible with the SPE. While this technique was effective for the PPE, the SPE would require a different approach. It is for such problems that the SPE was equipped with such a large

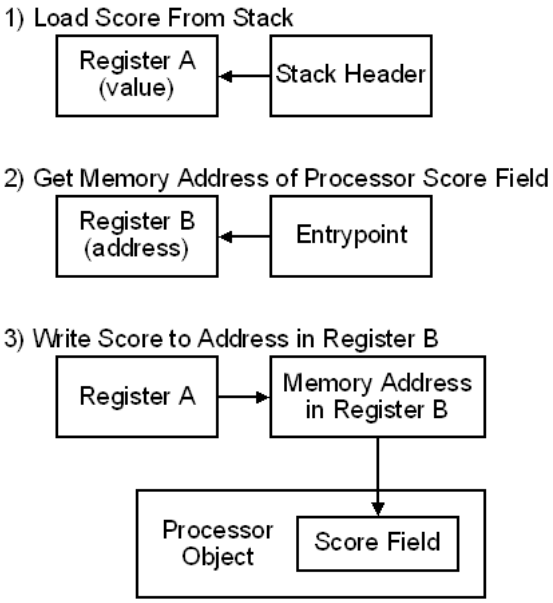

Figure 3.18: The PPE Epilogue

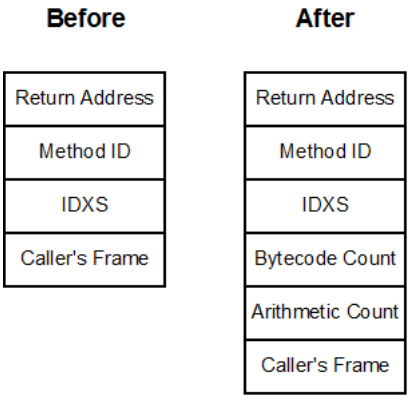

Figure 3.19: Modifications to the SPE Stack

amount of scratch-pad memory; this memory can be used for whatever the programmer desires.

At first, enough space was reserved in the memory to provide space for two integer values. However, when this was tested the numbers that were read from the memory had been corrupted. This was because the SPE can only read and write from the local memory in 128-bit wide segments or greater. Initially, 64 bits had been reserved, which is enough space for two integers. This problem was solved by rotating the fetched quadword. In order to access values smaller than 128 bits, the SPEs use an 'offset' system; however, Hera JVM distributes the scratch-pad memory when the SPE is initialized. There were considerable amounts of space reserved for later expansion, so 128 bits were reserved for each integer. This kept the system simple, and reserved space for more data should the system be expanded at a later point.

Figure 3.20 demonstrates how the memory access patterns of the SPE led to the scores becoming corrupt, while Figure 3.21 shows how the epilogue takes the scores from the stack header and writes them to these reserved slots in the scratch-pad memory.

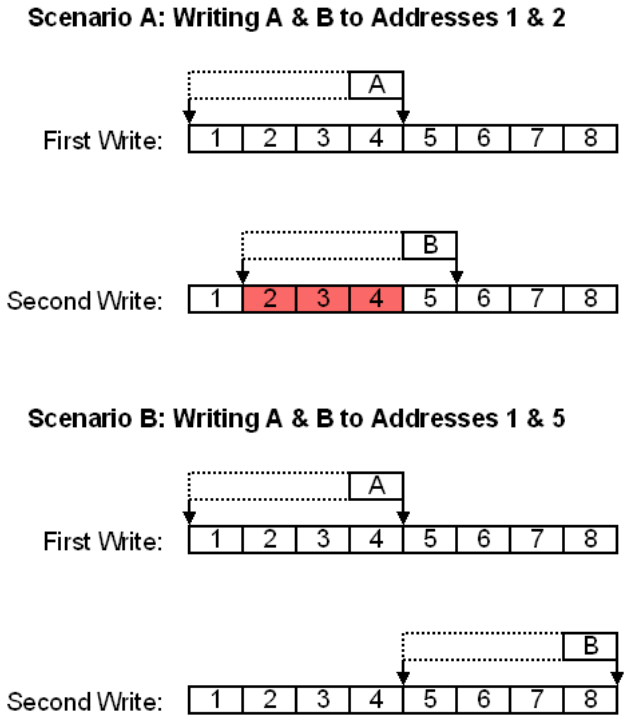

Figure 3.20: Reading and Writing from the SPE Scratch-Pad Memory

The entire process of thread scoring is reasonably straightforward. Figure 3.22 offers an overview of the process. At compile time, regardless of the target architecture, the method is scored and these scores are added to the method object. When the time comes to generate the assembly code for this method's bytecodes, additional instructions are generated at certain points that add the scores to the output.

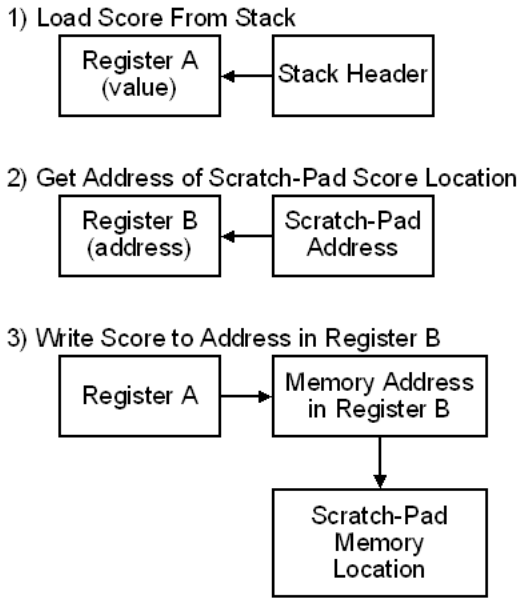

Figure 3.21: The SPE Epilogue

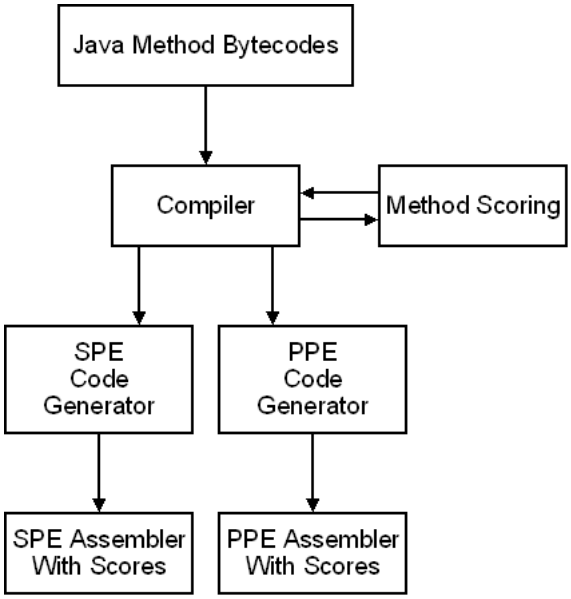

Figure 3.22: Thread Scoring Overview

At runtime, the inserted code manipulates the stack header, producing two values that represent the amount of arithmetic operations in the source code. The scheduler will then read these values and use them to make scheduling decisions.

3.4.2 Scheduling System

The scheduler will make scheduling decisions based on the scores it receives. If the given scores show a high amount of arithmetic code, these operations will be scheduled to the SPE where they will complete faster. This should increase the overall speed of the system.

Section 3.4.1 discusses how scores are generated and placed onto the processor object. For each core, this data is in a different location. The PPE has this data immediately accessible through reference fields, while the SPE can fetch the data via a memory operation.

Whenever a scheduling quantum expires, a command is issued for each processor object to get the score that it holds. The processor object then invokes a method in the thread object that saves these scores. The processor will then clear the scores before it continues execution; it may continue executing the same thread, or it may switch to another thread. This process is illustrated in Figure 3.23.

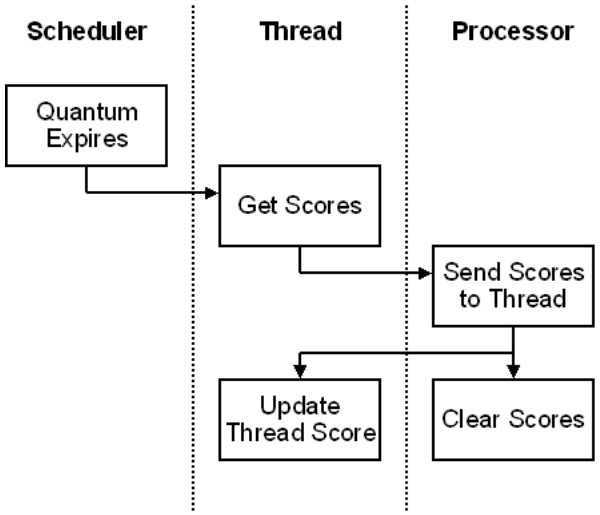

Figure 3.23: Data Movement from Processors to Threads

When the scheduler receives a score, it first adds the new score to the score currently held. An exponential averaging function is used so that the new values are regarded as the most important, as shown.

$$newScore = (\alpha * newScore) + ((1 - \alpha) * oldScore)$$

In this function, α is a number such that $0 < \alpha < 1$. A higher value for α places more emphasis on the current result, while a lower value places more emphasis on the previous results.

Once the score has been evaluated, a decision is made on whether the current thread should migrate or not. This depends upon two factors; the core the thread is currently running on, and the score relative to

a pre-set migration threshold. Migration thresholds can be projected from the results of the tests in Section 3.3.5.

The migration system in Hera JVM is simple; if the thread qualifies for migration, then it can simply invoke a method called '`takeNextMigration()`'. Threads are only allowed to migrate at method invocations, at which point the thread call tree migrates to the destination core. Subsequent method calls within the outsourced method will also run on the target processor; however, when the method returns the thread will return to the processor upon which it was initially scheduled. Figure 3.24 shows how method invocations are handled during a thread migration.

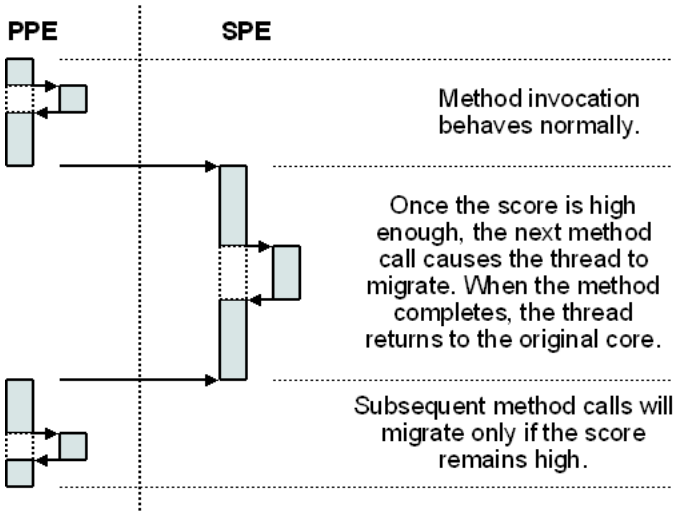

Figure 3.24: Thread Migration Behaviour

Unfortunately, this system faced a number of significant problems. This flag could be set at any point, but there is no guarantee that the next method to run in that thread would be capable of migrating. For example, the next method could invoke the JIT compiler. Migrations have been prevented during such vital JVM code, ensuring that JVM methods do not migrate to other cores accidentally. However, JVM code often calls standard Java operations, such as string operations, which are not safe from this issue. If the migration flag is set during such an invocation, then this method will migrate causing the JVM to crash. This problem is illustrated in Figure 3.25

In order to combat this, migrations were prevented from happening at certain points. These points include any JVM code that is called during runtime, and all JVM code that runs before and after the runtime. This was a very time-consuming task, as it was difficult to tell which methods required the additional flags that prevented then from being migrated.

Two additional flags were added to the system. The first was set when the actual input code began execution and was unset when the code completed, thus preventing JVM setup and completion code from being

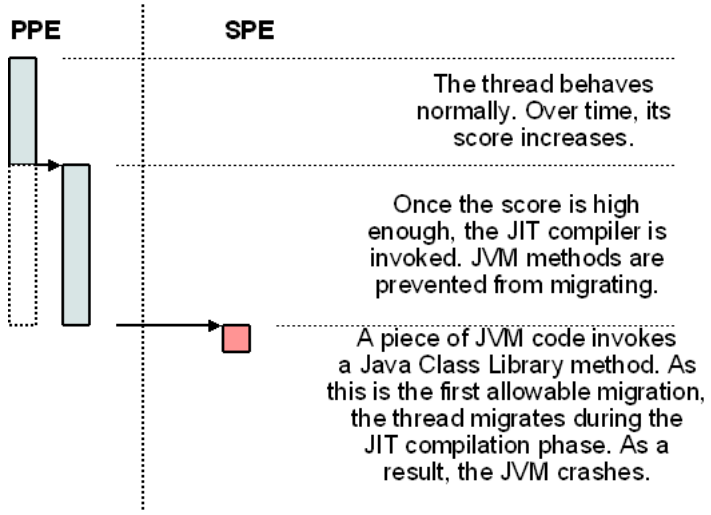

Figure 3.25: Harmful Thread Migration

accidentally migrated. Figure 3.26 illustrates this principle.

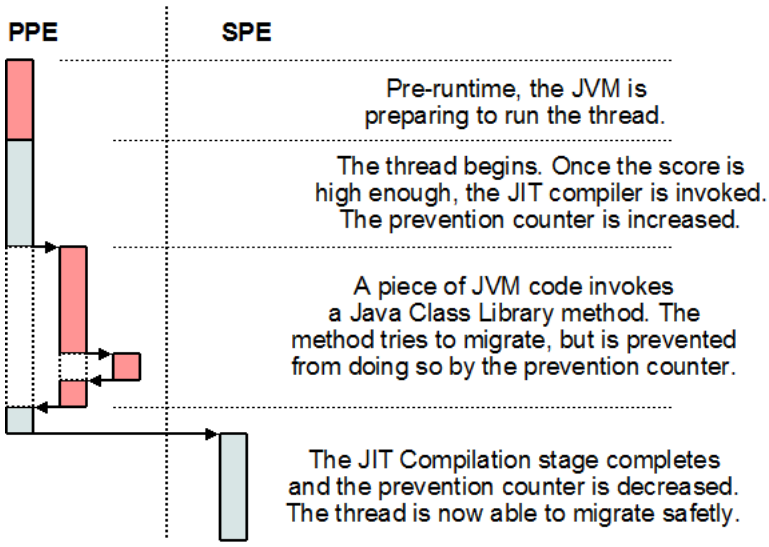

Figure 3.26: Runtime Migration Control

The other flag took the form of a counter, which was incremented whenever a piece of runtime JVM code was invoked. When the JVM code completed, the counter was decremented again. While the counter was at 0, migrations were allowed. Runtime code could be invoked at almost any time in an unpredictable manner, so this system was required to ensure migrations could not occur at certain points.

When a migration flag is set, it will automatically be cleared when the prevent migrations counter is increased. However, the thread will set another flag allowing it to remember that it planned to migrate. When the counter reaches 0 and migrations are allowed again, the thread will check if this flag is true. If it is true, then the thread will check if

its current score permits migration and that it is still in the program runtime. If these checks fail, then it will clear the flag, resetting the migration system to its default mode. This process is illustrated in Figure 3.27.

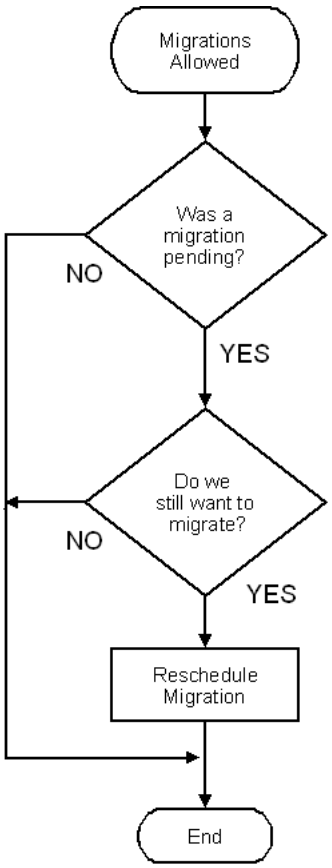

Figure 3.27: JVM Runtime Migration Control

The thread scoring system and the runtime scheduler work together to decide where each thread should be scheduled. Thread-to-core affinities take the form of a simple value representing the volume of arithmetic code in the system; if this is over a certain threshold, then the thread will migrate. This prototype scheduler provides an effective base for further experimentation.

3.5 Conclusion

In this Chapter the nature of the research to be undertaken was outlined. In section 3.1, the literature survey was discussed, revealing a knowledge gap and outlining what could be done to fill this gap. In Section 3.2.2, it was concluded that this research will be carried out by designing a dynamic scheduler for the Cell processor. Section 3.3.5 contains an investigation into the Cell in order to fully understand its hetergeneous properties, and Section 3.4.2 discusses how a prototype

scheduler was designed, implemented, and configured.

Chapter 4 will build on the results of the work carried out in this Chapter, and details the design of an experiment that will allow the dynamic affinity scheduler developed in section 3.4.2 to be evaluated and compared to similar schedulers.

Chapter 4

Experimental Design

The aim of this project is to investigate the capabilities of dynamic affinity scheduling techniques when applied to heterogeneous multi-core processors. The work carried out in Section 3 has produced a working and configured prototype dynamic affinity scheduler. This scheduler is built into Hera JVM, originally designed with a static annotation-based scheduler.

In this Chapter an experiment will be developed to investigate the properties of this prototype dynamic affinity scheduler, and judge its merits when compared with two other schedulers. Section 4.1 presents the aims of the experiment and specifies the hypotheses, followed by Section 4.2 where the variables in the experiment are outlined and discussed. Section 4.3 presents the design and specifications of the experiment, before Section 4.4 summarises and concludes this Chapter.

4.1 Experimental Aims

The aim of the experiment is to demonstrate the potential of dynamic affinity scheduling techniques. This will be done by comparing the prototype *Dynamic Affinity Scheduler* presented in Section 3.4.2 with the two other schedulers available in Hera JVM; a *Static Scheduler* and an *Annotation-Based Scheduler*.

These three schedulers differ in a number of ways. The dynamic and the static scheduler require no modification of the source code, making them the simplest for the programmer to use. However, only the annotation-based and dynamic schedulers provide the ability to take advantage of the Cell's multiple cores. The static scheduler does not provide migration abilities and therefore has minimal overheads, while the other schedulers incur penalties for every thread migration that occurs.

The actual merit of each scheduler depends entirely on the code it is

trying to schedule. If the code provides a high volume of migration opportunities, migrating at all of them would incur heavy overheads that would likely negate any benefit from affinity scheduling. Similarly, if a program contains few migration points then the dynamic affinity scheduler would not be able to respond quickly to changing affinities.

The dynamic affinity scheduler aims to identify code phases and schedule based on processor affinities; if this is successful, then the phase code would run faster. However, the overheads involved may result in the cost of migration being greater than the benefit of migrating. The longer each code phase is, the higher the benefit of correct affinity scheduling.

While identifying the code phases is automatic in the dynamic affinity scheduler, the annotation-based scheduler relies entirely on the programmer to identify both the program phases and the correct location to place annotations; this requires a deep understanding of the program's runtime behaviour, the affinity properties of the processor and the costs and benefits of each migration.

It is the differences between the three schedulers that the experiment should explore. It is unlikely that any single scheduler would be superior in every case, therefore the experiment should be cover a number of cases in order to evaluate each scheduler in terms of their abilities and limitations. Based the information presented in Chapter 3, the following hypotheses concerning the performance of each scheduler can be formed.

- `H1`: The dynamic affinity scheduler will be superior to the annotation-based scheduler when creating affinity-based schedules.
- `H2`: The performance of the dynamic affinity and annotation-based schedulers will improve for well-defined workloads with clear phase boundaries.
- `H3`: The dynamic affinity scheduler and the annotation-based scheduler will have better multi-threaded performance than the static scheduler.
- `H4`: For both the dynamic affinity scheduler and the annotation-based scheduler, the benefit of affinity scheduling will be greater than the cost.

Exploring these hypotheses would allow for a broad investigation into the capabilities of dynamic affinity scheduling techniques. The results of an experiment into these properties would provide enough information to analyse dynamic affinity scheduling techniques at a higher level.

All of the hypotheses presented here will be investigated in a similar manner. First, a performance test will be developed that will stress all three schedulers. The performance of each scheduler will be evaluated

in terms of this performance test, allowing a comparison of each schedulers abilities. The details of this experiment will be explored in the remainder of this chapter.

4.2 Variables

Prior to designing the experiment, the variables involved must be considered with respect to the desired output. In this section the different variables present in the experiment will be introduced and discussed in order to evaluate their potential impact on the experiment.

4.2.1 Indepedent Variable

The independent variable in the experiment is the type of scheduler used. There are three different schedulers that will take part in the experiment; the prototype dynamic affinity scheduler developed in Section 3.4.2, the annotation-based scheduler and the static scheduler, both of which are built into Hera JVM .

The static scheduler in Hera JVM, when faced with unannotated code, will allow the code to continue running on the current core; assuming no annotations are present, a given thread will complete on the same core it was created on. Threads will not migrate between cores without annotations, so there are no migration overheads; the overheads related to calculating the affinities will also not be present.

The annotation-based scheduler requires hand-tuning, as location of the annotations will have a significant impact on the results of the test. Therefore, different configurations should be used when running the experiment on the annotation-based scheduler. A hand-tuned schedule should also be able to make the most effective use of the hardware. To allow for a meaningful comparison, only configurations that will be highly effective should be tested.

The dynamic affinity scheduler is more complex, as it has to score the code and calculate affinities. If scoring is too complex, then the overheads will result in lower performance compared to the static schedulers. It's also possible that the overheads of migration may create a significant performance drop. While different configurations can result in different levels of performance, the prototype affinity scheduler was configured during its implementation as detailed in Section 3.4.2. The configuration of the dynamic scheduler should remain constant throughout the experiment.

4.2.2 Dependent Variable

The dependent variable in this case is the performance of the test program. Taken as a high-level view, 'performance' in this respect refers to the time taken for the test program to complete. The lower the time taken to complete, the better the performance of the scheduler. However, there exists the possibility of confounding variables if the performance measurement is limited to completion time alone.

For example, the unannotated static scheduler would have no scheduling overheads as it does not allow migrations. The annotation and dynamic schedulers would be expected to reduce the time spent in certain areas of the code, however they would incur greater overheads. This creates a confounding variable; if performance is lowered, it could be because of a poor scheduling decision or high overheads. There is also the chance that the cost of migration is equal to the benefit, disguising the effects of affinity scheduling. This implies a need for two separate timing metrics; one that measures the time spent running the test code itself, and another metric that measures the total time taken to run the test.

4.2.3 Extraneous Variables

There are a number of extraneous variables present in the system that would be likely to have an impact on the performance of the schedulers. The most obvious extraneous variable is the code used when testing the system. Some test programs will be ideally suited for a particular scheduler, and will always produce better results when scheduled by it; for example, pure object code in a single thread would be expected to run best on the static scheduler, as there are no overheads and the PPE has an affinity for this workload.

There are also some JVM effects that should be considered. For example, the JIT compiler itself would have an impact on performance if not accounted for. The first time each method is invoked, the JIT compiler will be activated; is this happens during a timed part of the experiment, it would interfere with the results.

Another JVM factor that must be considered is the garbage collector. In Java, the garbage collector is a separate thread which intermittently blocks running threads to clear up memory space. If garbage collection takes place during a timed section, it could significantly modify the results of the experiment, as garbage collection is often a lengthy and time-intensive task. These JVM variables should be controlled as much as possible to ensure that they do not interfere with any experimental results.

Other programs running on the Cell could also impact on the results of the experiments. This should be controlled as much as possible, en-

suring that for each experiment is carried out in a similar environment.

As the dynamic affinity scheduler builds up a score over time, each thread has an associated score. If different tests are run using the same threads, then the scores of one test would be passed to the next, creating a 'learning effect'. This could impact on performance in a number of ways, improving it in some cases and decreasing it in others. This learning effect must be eliminated for any experiments to produce valid results.

The dynamic scheduler must be provided with sufficient opportunities to migrate if it is to construct efficient schedules. Therefore, the number of migration opportunities will have a significant impact on performance; if few opportunities, it is vital that the decisions made at those points are correct; with abundant migration points, the scheduler may be able to migrate often enough that migration overheads quickly outweigh any performance gains. This variable is important and should be controlled, allowing any relationship between the number of migration points and the performance of the dynamic scheduler to be explored.

The number of threads is also expected to have a significant impact on the performance of each scheduler. As the number of threads is increased, it is predicted that both the annotation scheduler and the dynamic affinity scheduler would quickly outperform the static scheduler, as they are able to distribute their workload over the other cores. The number of threads present in the system should be controlled.

As the Cell is a heterogeneous processor, running the experiment from different starting cores would result in performance differences. This should be controlled, ensuring that each thread starts execution in the same environment.

Another factor to control during testing is the actual cost of the measurements themselves. As much as possible, measurements should not interfere with the running of the test; interference such as writing results to a file during a timed area would have an impact on the results of that test.

These variables should all be considered and controlled where possible. Some variables, such as the number of threads, workload and migration opportunities should be explicitly controlled during the test to provide different testing environments for each scheduler. However, these are not independent variables; it should be the scheduler alone that creates any performance differences. Each part of the experiment should be performed in similar circumstances, with each of the variables discussed in this section well defined and controlled.

4.3 Experimental Design

The main observed variable is the time taken to complete the test; yet this value alone is not sufficient for a full analysis. As described in section 4.2, there are two factors to be measured for each scheduler; the time spent running the test code, and the overheads associated with each scheduler. It is important that both these values are considered separately, and as such the experiment must be able to produce both of these values.

Section 4.2 also outlined that the experiment should consider different workloads, variations in migration opportunities, and the number of test threads.

In order to test the schedulers, a benchmark program will be developed that has been specifically designed for this experiment. This benchmark program will directly control a large number of the variables discussed in section 4.2, and will produce all the information required for an evaluation.

The most important feature of this benchmark program is the ability to generate a number of different workloads that can be used to stress the three schedulers. The construction of the workloads will be varied in a number of ways, based on the two workload variables discussed in section 4.2; the processing requirements of the workload and the number of migration opportunities presented by the workload. Both of these variables will be controlled in order to model a variety of workloads, each one dynamically generated.

The first of these variables is the runtime behaviour of the workload. Two different workloads of equal size will be developed. The first, an arithmetic workload with an affinity for the SPE; the second, an object-based workload with an affinity for the PPE. These workloads have been developed based on the results of the investigation into the Cell's affinity properties carried out in section 3.3.5. The two workloads can be combined in a number of ways to create program-emulating workloads that vary in their runtime behaviour and processing core affinities.

Given that `A` represents an arithmetic workload and `B` represents an object-based workload, then the two workload components can be combined to create 16 different workload patterns as shown below. Workloads will be dynamically built by iterating over each one of these patterns. Such dynamically created workloads allows the experiment to be run over a wide range of input, each one representing a different set of affinity properties.

AAAA	ABAA	BAAA	BBAA
AAAB	ABAB	BAAB	BBAB
AABA	ABBA	BABA	BBBA
AABB	ABBB	BABB	BBBB

The second variation between the workloads concerns the number of migration opportunities in the test code. Hera JVM is currently limited to migrating only at the start of method invocations, so large monolithic programs will be able to migrate less often than small compartmentalized ones.

The experiment will account for this by extending the amount of time spent processing each pattern, while reducing the number of iterations; this has the effect of increasing the weight of the workload while reducing the migration opportunities it presents. The actual length of the experiment will remain constant each time, allowing the results to be compared. Table 4.1 shows the weights of the different workloads to be used during the experiment.

Workload Name	Weight	Iterations	Patterns per Test
Very Heavy	10000	1	10000
Heavy	2000	5	10000
Medium	400	25	10000
Light	200	50	10000
Very Light	100	100	10000

Table 4.1: The Weights of the Different Workloads

The benchmark program will also control a number of the other other variables discussed in section 4.2. The effects of garbage collection will reduced as much as possible by invoking the `System.gc()` method where possible, which will notify the JVM that it should run the garbage collector. This does not guarantee that the garbage collector will not be run during the tests, but it will help to ensure that garbage collection occurs outside of timed areas. Unfortunately it is no longer possible to control garbage collection once there is more than one thread, as any other thread could call this method during a given thread's timed section. Therefore, enough iterations of the experiment test should be performed to identify times that have been inflated by the garbage collector.

The JIT compiler effects will be explicitly managed by running a warm-up system which runs all the code on both processors prior to beginning the test. This will cause the JIT compiler to pre-compile the benchmark methods, ensuring that the JIT compiler will not be called during any of the timed sections.

The learning effect will be eliminated by creating a new thread for each part of the test. Each thread will run a test and report the results to a static scoring system. At the end of the thread's runtime it will create

the next test thread, which it will start immediately before it completes. Once the next thread begins running, it will call `System.gc()` to try and clean up the previous thread; it will run the experiment. The final thread will call a static finalising method, which will cause the scoring system in of the benchmark to write its results to a file. As the scores are linked to the thread, this will eliminate any learning effects from the experimental results. The process of submitting scores to another part of the program at the end of the test also controls any timing anomalies that would have occurred as a result of directly measuring the performance of the threads.

There are few variables not directly controlled by the benchmark, and they will be controlled during the experiment by other means. The tests themselves will be run on the same Cell processor when the Cell is not being used for other tasks, ensuring that external programs will not affect the outcome of the tests.

Controlling these variables ensures a similar test environment between experiments, and also ensures that a wide number of variables are controlled during each experiment, helping to ensuring that the results are free from confounding variables and external influences.

4.4 Conclusion

The experiment described in section 4.3 will be ideal for investigating the hypotheses specified in section 4.1.

In the case of `H1`, the experiment will provide both the time spent in the target code and the overheads of the scheduler. It is predicted that the annotation based scheduler will have higher overheads than the affinity scheduler; however, if it can reduce the time taken to complete each phase, then the dynamic affinity scheduler would be superior as it would both complete phase code faster and have smaller overheads. Over the various workloads, if this property can be shown to exist then it would provide evidence that supports the first hypothesis.

The second hypothesis, `H2`, will be investigated through directly controlling the type of workload. This is directly related to the number of migration opportunities present in the given workload. As the experiment involves testing over a range of generated workloads of varying weight, the information gathered from the experiment should provide sufficient evidence to support any evaluation of this hypothesis.

The third hypothesis, `H3`, will be investigated by comparing the performance of each scheduler over a range of threads. As the number of threads increase, it is expected that the phase completion times of the dynamic and annotated schedulers will be lower than the static scheduler; if the overall completion times are also lower, it supports

the theory that the dynamic and annotation-based schedulers offer superior multi-threaded performance.

The fourth hypothesis, `H4`, represents a core issue in this project. The experiment will provide information on the overheads, and through comparison with a static scheduler it will be possible to determine if the price of affinity scheduling can be justified. Section 3.3.5 has already shown that the performance difference between the two cores of the Cell processor are considerable; investigating this hypothesis will reflect upon the value of both annotation and dynamic affinity scheduling techniques.

The results of this experiment will be presented in chapter 5, and these results will be interpreted in Chapter 6.

Chapter 5

Results

In this Chapter the results of the experiment described in Chapter 4 are presented. Section 5.1 present the results of the static scheduler experiment. Section 5.2 presents the results of two different kinds of annotation tests, as described in Section 4.3. Section 5.3 will present the results of the Dynamic Scheduler experiment, before Section 5.4 concludes the Chapter.

5.1 Static Scheduler Results

The static scheduler is the most basic of the three schedulers. As such, it is expected to have the lowest overheads. However, the static scheduler is a poor choice for any heterogeneous multi-core processor as it will prevent programs from taking full advantage of the resources available to them.

Three tests were carried out on the static scheduler; a workload based test, to determine its ability to handle different workloads, presented in Section 5.1.1; a workload weight test, to determine the effects on the static scheduler with different phase lengths and migration points, presented in Section 5.1.2; and a thread test, to investigate the capabilities of the static scheduler when working with multi-threaded code, presented in Section 5.1.3.

5.1.1 Static Scheduler Workload Pattern Test Results

The two workloads were designed to run with roughly equal times on the static scheduler, regardless of their weight or pattern. However, there remained slight differences between each combination of patterns.

Table 5.1 presents the average timing results for each combination of patterns. The full test comprised of all permutations, but for simplicity these results are aggregated into simple combinations here. Therefore,

Pattern Type	Average Time (ms)
AAAA	120
AAAB	129
AABB	124
ABBB	126
BBBB	128

Table 5.1: Static Scheduler Performance by Workload Construction

entry AAAB represents all combinations that contain 75% type A code and 25% type B code. The full results are presented in Appendix B.

Table 5.1 shows that there is very little difference between the performance over a single thread, regardless of the workload. There is an 8 ms difference between code of type A and code of type B.

There is an almost visible trend as the weights are altered, with the time increasing by 2 ms towards the BBBB pattern; there is one outlying result here, which is the AAAB patterns. Upon further examination of the expanded graphs found in section B, a spike in performance can be seen specifically in the second pattern which is AAAB. As the JIT compiler effects are controlled, this effect is attributed to the Hera JVM garbage collector. As a result of this, AAAB type patterns have a high standard deviation of 11%.

These results will form the bases upon which the other results are compared in Chapter 6. This spike appears over all the scheduling tests with a similar weight, and as such will not have a major impact on any comparisons drawn from these results.

5.1.2 Static Scheduler Workload Weight Test Results

The workload based test is designed to highlight the effects of different phase lengths. However, the overall workload will remain the same, and as such the static scheduler is expected to produce similar times over all the results.

Workload	Total (ms)	Phase (ms)
Very Light	125	123
Light	124	122
Medium	127	124
Heavy	126	124
Very Heavy	124	124

Table 5.2: Static Scheduler Completion Times for Various Workload Weights

The results of this test are presented in Table 5.2. This presents both the total time taken to carry out the test and the time spent inside

each phase, averaged out over all 16 patterns described in Section 4.2.

As expected, the results demonstrate that the length of the phase has little effect on the actual time taken to complete the given piece of code. The standard deviation between the total time taken to complete the test is 0.93%, while the standard deviation between phase times is 0.65%.

The graph in Figure 5.1 presents this information visually, demonstrating that there is an insignificant difference between between each workload.

5.1.3 Static Scheduler Multi-Threaded Test Results

The static scheduler is not capable of making migrations without the use of annotations. In effect, the static scheduler is simply the annotation scheduler presented with unannotated code. It is expected to have significantly fewer overheads than the other schedulers, but the drawbacks of being unable to migrate make it entirely unsuitable for a multi-core heterogeneous processor.

As such, it is expected that each new thread will increase the average completion time in a direct relationship with the number of threads present. The multi-threaded experiment will consider up to 8 test threads, each one built from the "Heavy" workload type.

Threads	Total (ms)	Phase (ms)
1	126	124
2	197	137
3	289	212
4	473	472
5	510	415
6	625	524
7	814	813
8	858	755

Table 5.3: Static Scheduler Thread Performance

Each thread ran a heavy workload over all 16 patterns, and reported the average overall time and average phase time once it had completed. The average completion time of a thread was then calculated, and organised by the number of threads present in the test. This information is presented in Table 5.3; as with the previous results, this is an abridged version of the full results which can be found in Appendix C.

As was predicted, increasing the volume of threads appears to have a direct impact on the average completion time. As each thread is forced to share resources, each new thread imposes a significant performance penalty. The overheads of the system remain quite low, however the

time taken to complete each phase rises significantly as the threads are increased.

Figure 5.2 presents a visual comparison between the overheads and the performance of the system, demonstrating how it varies with the number of threads in the system. Figure 5.3 isolates the performance, showing that performance decreases as the number of threads is increased.

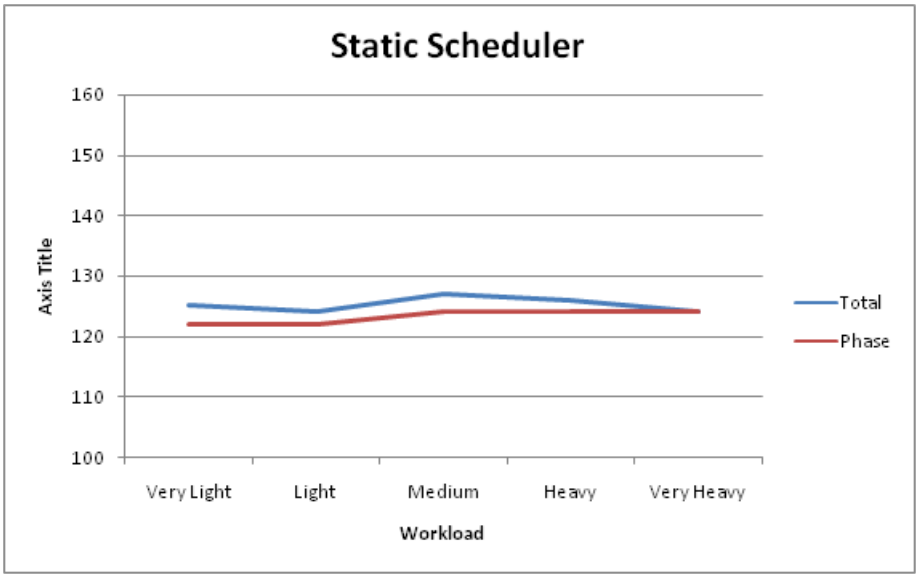

Figure 5.1: Static Scheduler Performance for Various Workload Weights

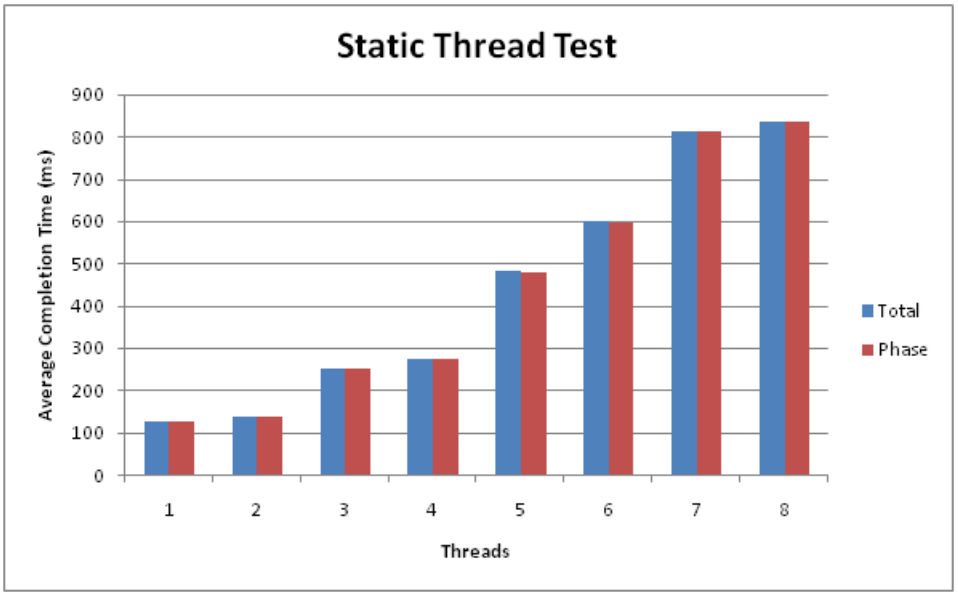

Figure 5.2: Static Scheduler Performance and Overheads over Multiple Threads

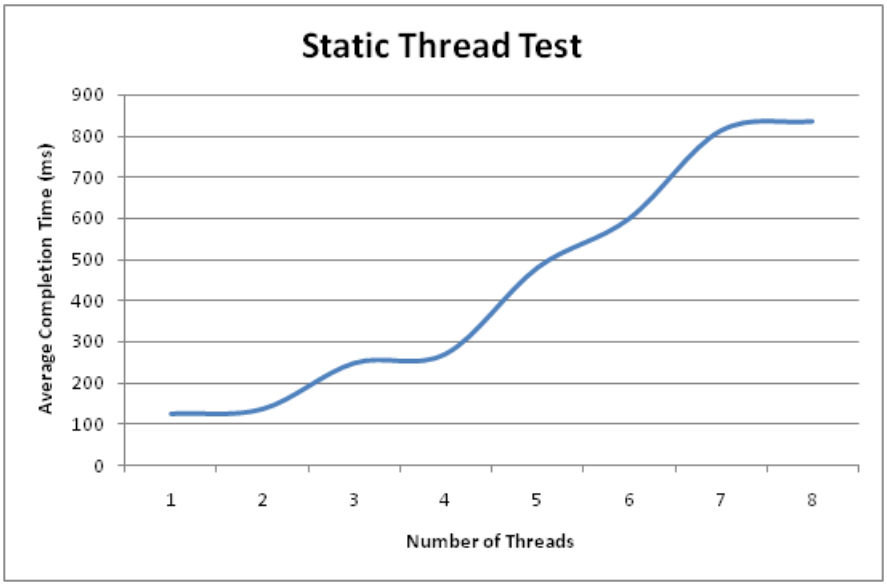

Figure 5.3: Static Scheduler Performance over Multiple Threads

5.2 Annotated Scheduler Results

The annotated scheduler is a variant of the static scheduler, using code annotations to make scheduling decisions. The annotations were placed in two places when testing the annotation scheduler; in an intuitive location likely to reduce phase time yet incurring heavy overhead costs, and in a counter-intuitive location that aims to minimise overheads through a knowledge of the workloads. These two annotations are tested separately so that a broader overview of the annotation based scheduler can be developed.

The results of both annotation tests will be presented together here; the results of the workload based test will be presented in Section 5.2.1, followed by the results of the workload weight test in Section 5.2.2, and finally the results of the multi-threading test are presented in Section 5.2.3.

5.2.1 Annotation Scheduler Workload Pattern Test Results

The annotated scheduler will incur significant overheads as it migrates at every annotation, even if the annotation is incorrect. For the purposes of this experiment only useful annotations are considered. As the intuitive model migrates very often, it is expected that it will suffer from extremely high overheads when faced with workloads that contain more type `A` code.

Conversely, the counter-intuitive annotation placement will reduce the migration overheads significantly by migrating the thread at the highest level. This is analogous to a programmer with a high understanding of the system ignoring type `B` affinities, which may be a reasonable trade-off.

Workload	Intuitive Time (ms)	Counter-Intuitive Time (ms)
`AAAA`	649	62
`AAAB`	505	92
`AABB`	412	124
`ABBB`	302	156
`BBBB`	139	188

Table 5.4: Annotated Scheduler Performance by Workload Pattern

Table 5.4 presents the results of the experiment. As expected, the intuitive annotations incur very heavy overheads, which outweigh the savings made through affinity scheduling. However, the counter-intuitive scheduler saves on migrations through a knowledge of the system. All code is run on the SPE which results in a net saving; the cost of migration and the penalty for incorrectly scheduling type `B` workloads is easily accounted for by the significant savings of running type `A` code

on the SPE.

Through ignoring the affinities of type `B` code, counter-intuitive scheduler completes type `AAAA` patterns almost ten times faster than the intuitive scheduler. However, the intuitive scheduler makes a small saving in type `BBBB` code, completing it 26% faster. More information about the results of this test can be found in Appendix A.

These results demonstrate that the type of workload has a significant impact on an annotation-based scheduler; however it can also be asserted that the annotation-based scheduler relies heavily on a deep understanding of the underlying system and the overheads of migration. It is likely that someone with this knowledge would annotate code in counter-intuitive places, as shown, in order to achieve the greatest performance. However, it is often the case that the actual runtime behavior of a program is hidden; it could be a dynamic program, or the program could use library code which is transparent to the programmer. These factors should be taken into account when evaluating the effectiveness of an annotation-based scheduler.

5.2.2 Annotation Scheduler Workload Weight Test Results

The results presented in Section 5.2.1 suggest that migration costs are quite high in Hera JVM. This test will highlight this issue by varying the number of migration points present in the test code. It is expected that with more migrations, the overheads will quickly start to outweigh the benefits of migration.

Intuitively Annotated		
Workload	Total	Phase
Very Light	1050	118
Light	505	119
Medium	252	94
Heavy	121	89
Very Heavy	97	91

Counter-Intuitively Annotated		
Workload	Total	Phase
Very Light	133	126
Light	125	121
Medium	123	118
Heavy	120	115
Very Heavy	119	115

Table 5.5: Annotated Scheduler Performance and Workload Size

Table 5.5 presents the results of this test for both annotations. The "Very Light" workload category causes a very large number of migrations to take place in the intuitively annotated test, resulting in

extremely poor performance. However, the time spent in each phase does not suffer as phase code is always allocated to the best processor regardless of overheads.

The overheads of intuitively annotated code are compared to the phase times in Figure 5.4, which shows that good performance is only achieved when the program is large and monolithic. When working with small compartmentalized workloads, scheduling overheads represent 88.8% of the total running time.

The counter-intuitive annotations have slightly longer phase times, as they ignore some affinity properties in order to reduce the overheads of scheduling. On average, the overheads of scheduling in this manner represent 4% of the total running time. Figure 5.5 highlights this, showing that the overheads of scheduling in this manner are reasonably constant, showing a standard deviation of 0.93 ms in overhead costs.

The full results of this test can be found in Appendix B.

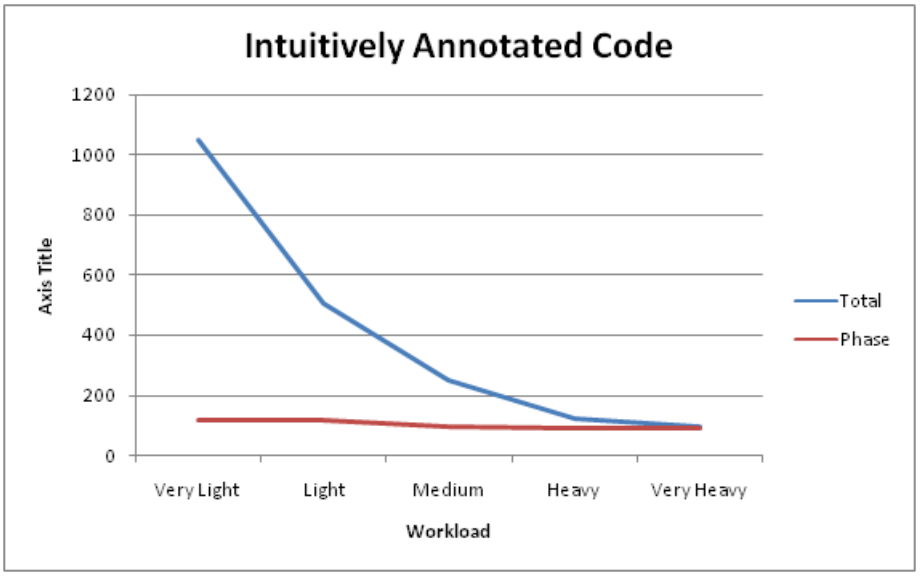

Figure 5.4: Intuitively Placed Annotations and Workload Size

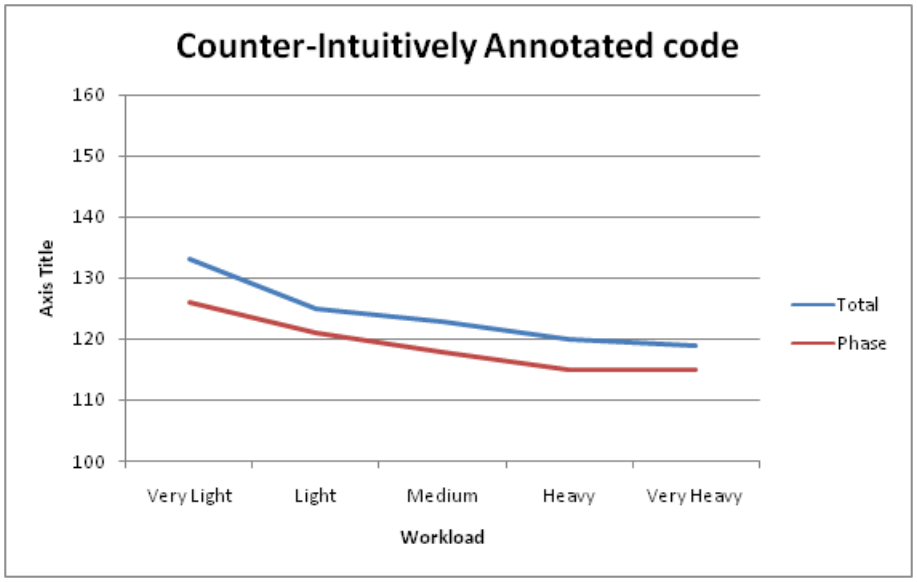

Figure 5.5: Counter-Intuitively Placed Annotations and Workload Size

5.2.3 Annotation Scheduler Multi-Threaded Test Results

As the annotation based scheduler is able to distribute its workload, it is to be expected that the annotation based scheduler will not incur heavy penalties when faced with a multi-threaded workload. As in the static scheduler test, 8 threads are tested over all of the patterns using the "Heavy" workload.

Threads	Total (ms)	Phase (ms)
1	121	89
2	158	93
3	238	99
4	299	96
5	354	94
6	393	95
7	609	99
8	775	98

Table 5.6: Annotated Scheduler Performance and Overheads over Multiple Threads with Intuitive Annotations

The results of the test on the intuitively annotated benchmark are shown in Table 5.6. The phase time is very small, as the workload is distributed over all of the processing cores based on the affinities of that code. However, the scheduler retains high overheads that rise with the number of threads in the system. Taking full advantage of the Cell's heterogeneous processors has resulted in very good performance for the phase code, which suffers very little as the number of threads increases. The overheads are compared to the phase times in Figure 5.6, while Figure 5.8 shows the overall performance trend as the number of threads is increased.

Threads	Total (ms)	Phase (ms)
1	120	115
2	122	116
3	145	139
4	192	164
5	207	202
6	235	232
7	273	235
8	311	235

Table 5.7: Annotated Scheduler Performance and Overheads over Multiple Threads with Counter-Intuitive Annotations

For the benchmark with counter-intuitive annotations, the trend is quite different Continuing the analogy that the counter-intuitive annotations have been written by a programmer with a deep knowledge of the Cell, another factor becomes apparent when the results in Table 5.7 are considered. Initially, the phase times remain reasonably low

along with the overheads. However, the Cell processor available only has 6 SPEs in working condition, implying that thread switching starts to take place as the number of threads reaches this point. Surprisingly, this does not manifest as a sudden performance drop, but instead as a non-linear performance drop relative to the number of threads.

It is likely that this effect occurs because multiple threads are created and migrated between the cores. The Element Interconnect Bus, described in Section 3.2.1, runs at half the speed of the processing cores. The EIB is responsible for all communication in the Cell, and it may be responsible for the steady curve shown. With the combined effects of increasing DMA requests, migrating new threads between cores, and eventually managing thread switches on the cores, a saturation point may be reached. An increase in phase times can be attributed to rising numbers of DMA requests, while the rise in overheads that occurs at the introduction of the fourth thread may represent the EIB beginning the struggle with the high workload.

When these factors are considered, it becomes apparent that using an annotation system correctly requires a great deal of knowledge concerning the underlying system, and presents a serious issue when attempting to abstract the heterogeneous hardware. Figure 5.7 compares the overheads and phase times of this test, while Figure 5.9 demonstrates the performance trend as the number of threads in the test rises. The full results of this experiment are included in Appendix C.

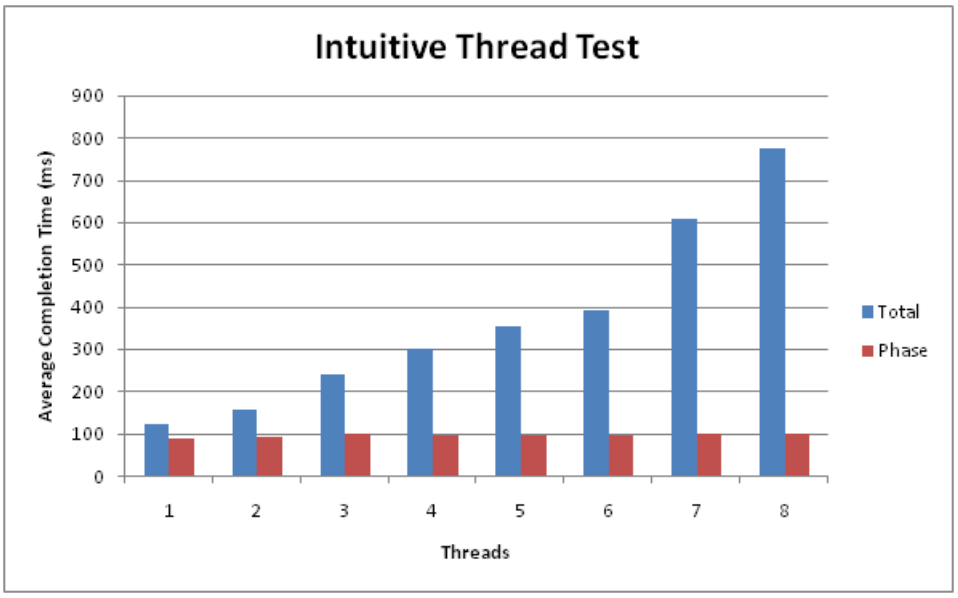

Figure 5.6: Annotated Scheduler Performance and Overheads over Multiple Threads with Intuitive Annotations

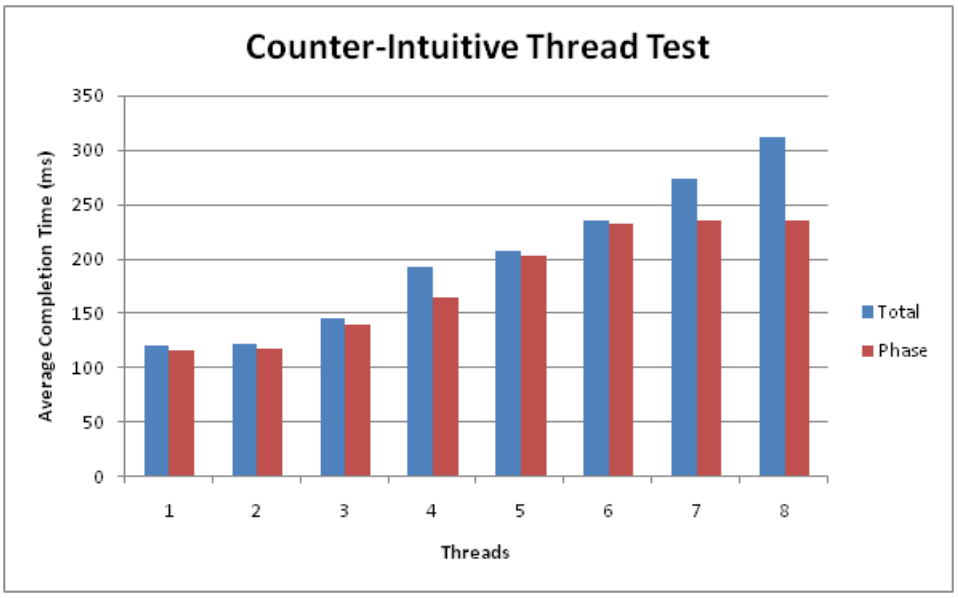

Figure 5.7: Annotated Scheduler Performance and Overheads over Multiple Threads with Counter-Intuitive Annotations

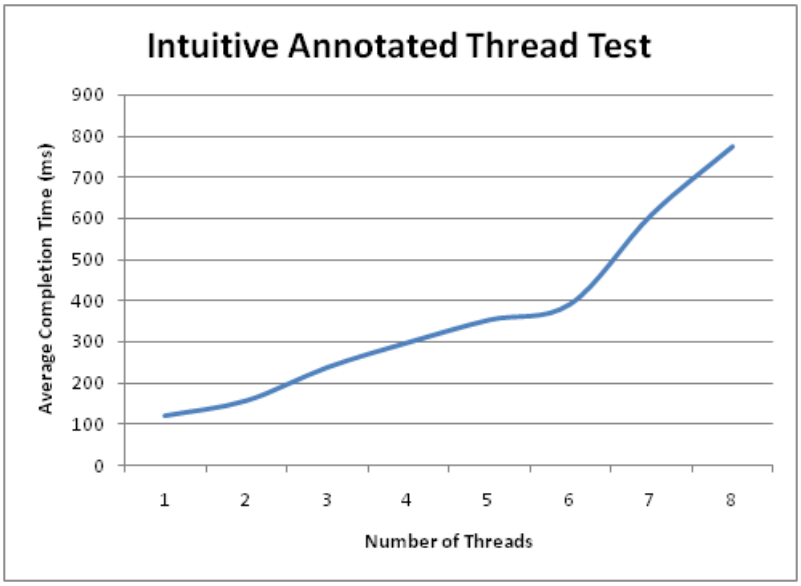

Figure 5.8: Annotated Scheduler Performance over Multiple Threads with Intuitive Annotations

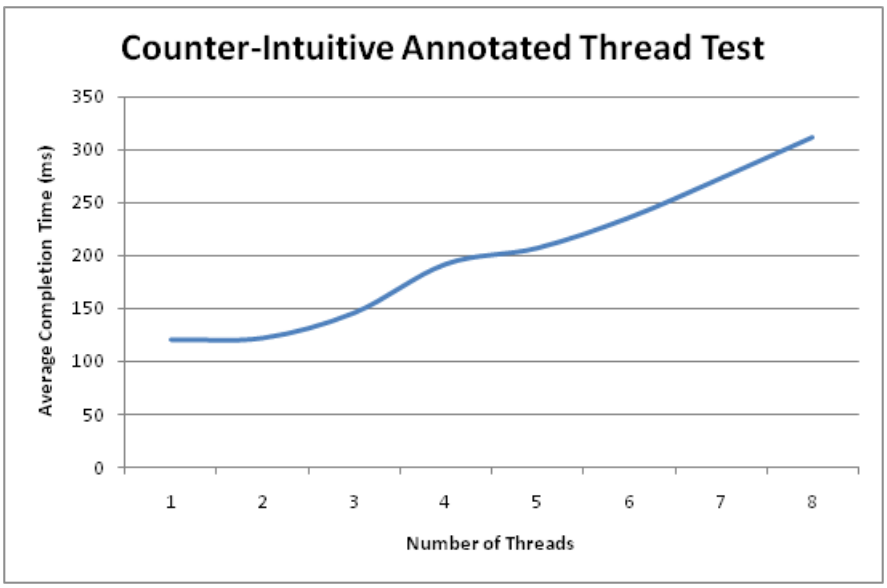

Figure 5.9: Annotated Scheduler Performance over Multiple Threads with Counter-Intuitive Annotations

5.3 Dynamic Affinity Scheduler Results

The dynamic scheduler is the most advanced scheduler, attempting to make scheduling decisions based on a pre-programmed knowledge of hardware properties. of the three schedulers. It is likely to present some overheads, but it is hoped that these overheads will be smaller than the savings of affinity scheduling.

Again, three tests were carried out. The results of the workload pattern test are presented in Section 5.3.1, followed by the results of the workload weight test in Section 5.3.2, and finally the results of the multi-threading test in section 5.3.3.

5.3.1 Dynamic Affinity Scheduler Workload Pattern Test Results

The dynamic affinity scheduler is expected to spend a small amount of time 'learning' the affinities and then scheduling based on prior behavior. As the test program exhibits cyclic behaviour, the system will learn where the current code should be scheduled after it has run for a short amount of time. This should allow the scheduler to migrate code only when the migration is beneficial.

Workload	Average Time (ms)
AAAA	161
AAAB	148
AABB	129
ABBB	130
BBBB	132

Table 5.8: Dynamic Scheduler Performance over Various Workload Constructions

Table 5.8 presents the average times of the dynamic scheduler over all workloads, organised by the construction of the workload. This highlights some interesting points; firstly, that the dynamic scheduler is able to recognise that type `B` workloads should not be migrated; it is able to complete these workloads very quickly. It was expected that type `AAAA` and `AAAB` workloads would be very fast in the dynamic scheduler; however, the actual cost of migration in a single thread actually outweighs the benefits of the migration.

The full results of this test are presented in Appendix B.

5.3.2 Dynamic Affinity Scheduler Workload Weight Test Results

While Section 5.3.1 demonstrates that the dynamic scheduler is capable of recognising the different workloads and adapting the scheduling policy to take advantage of affinities, the actual benefit is tied to the scheduling overheads. Given more opportunities to migrate, the dynamic scheduler is likely to incur higher initial overheads as it learns the behaviour of the system.

Workload	Total (ms)	Phase (ms)
Very Light	144	121
Light	136	125
Medium	137	123
Heavy	128	120
Very Heavy	126	123

Table 5.9: Dynamic Scheduler Performance over Various Workload Weights

Table 5.9 gives the time spent in each phase and the overall time as the workload's weight is varied. These results suggest that the dynamic affinity scheduler is affected very little by the weight of the workload and the definition of the phases. This behaviour is contrary to the expected behaviour of the dynamic affinity scheduler, as it was expected that different migration opportunities would result in significantly different scheduling abilities. It is also surprising that the type of workload has a greater impact on performance than the number of migration opportunities.

Figure 5.10 presents this information visually, showing that the dynamic scheduler is not significantly affected by the number of migration opportunities or by the length of the phase. The full results of this test can be found in Appendix B.

5.3.3 Dynamic Affinity Scheduler Multi-Threaded Test Results

The dynamic affinity scheduler is able to distribute its workload based on core affinities, and as such it is expected to perform well in the multi-threaded test.

Table 5.10 presents the results of this test; as with the previous tests, the table presents the average performance of a thread relative to the number of threads in the system. There are some interesting points to highlight in these results; firstly, there is a sudden jump in performance once seven threads are reached. This can be attributed to the six SPE's in the Cell; when trying to schedule seven threads, the system will begin to slow down. This can be clearly seen in both the phase times and

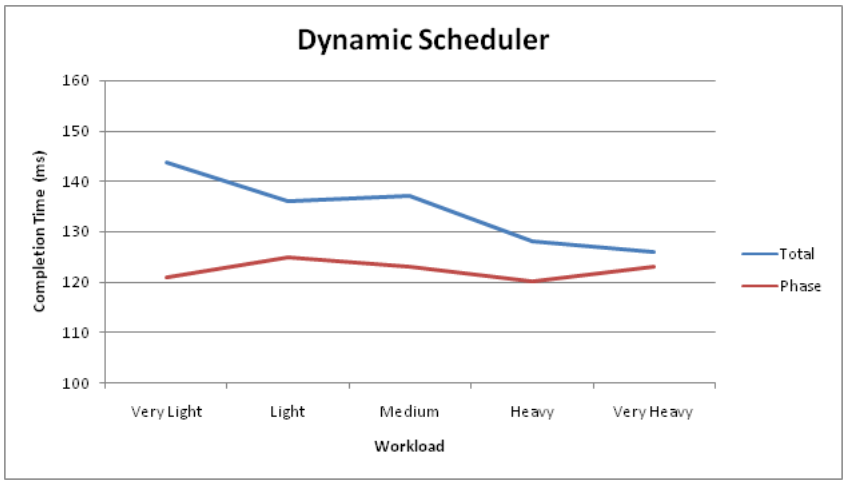

Figure 5.10: Dynamic Scheduler Performance over Various Workload Weights

Threads	Total (ms)	Phase (ms)
1	126	121
2	159	121
3	170	126
4	210	124
5	223	123
6	232	126
7	300	135
8	330	132

Table 5.10: Dynamic Scheduler Performance over Multiple Threads

the overall times.

Another interesting point is the gradual increase in the overall time, up until seven threads. This trend can be attributed to the increase in EIB activity; as discussed in Section 3.2.1, the EIB runs at a much slower speed than the cores; the programmer is responsible for making efficient use of the EIB, as described by Kistler *et.al*[12]. With the dynamic scheduler making a number of migrations, the overheads of migrating will increase with the use of the EIB; similar behaviour was observed in the annotated scheduler in Section 5.2.3.

These results are presented in Figure 5.11, showing how the performance and the phase times change with the number of threads. Figure 5.12 shows the actual performance trend as the number of threads present in the system is increased. The full results of this test can be found in Appendix C.

5.4 Conclusion

In this Chapter, the results of the experiment described in Chapter 4 are presented. The results of this experiment will be evaluated in the following chapter. The full results of these tests can be found in Appendices B and C.

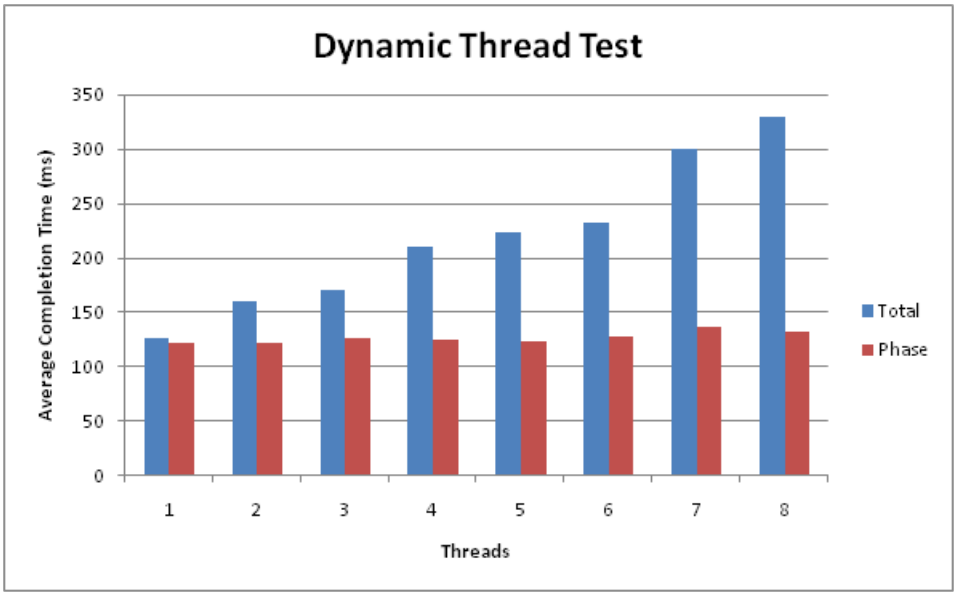

Figure 5.11: Dynamic Scheduler Performance and Overheads over Multiple Threads

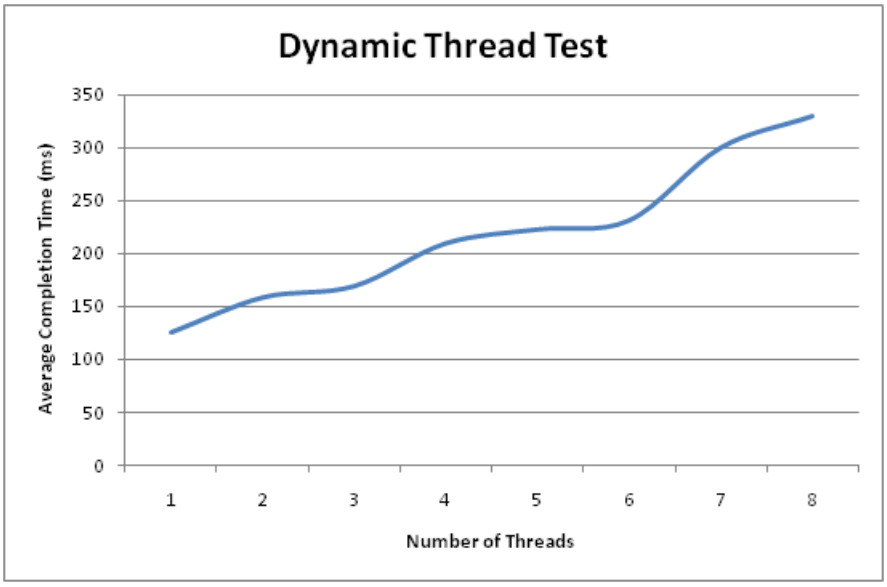

Figure 5.12: Dynamic Scheduler Performance over Multiple Threads

Chapter 6

Evaluation

In this chapter, an evaluation based on the results presented in Chapter 5 is presented. Sections 6.1, 6.2, 6.3 and 6.4 will evaluate the hypotheses laid out in Section 4.1, before Section 6.5 concludes this chapter with a summary of the main points presented here.

6.1 Evaluation of H1

`H1`: The dynamic affinity scheduler will be superior to the annotation-based scheduler when creating affinity-based schedules.

Scheduler	AAAA	AAAB	AABB	ABBB	BBBB
Dynamic	158	149	127	127	129
Static	120	129	124	126	128
I-Annotated	649	505	412	302	139
C-Annotated	62	92	124	156	188

Table 6.1: Finish Times by Workload

To evaluate this hypothesis, the ability of each scheduler to adapt to different workloads is considered. Table 6.1 compares the finishing times of each scheduler over different types of workload. This information is also presented in the graph in Figure 6.1.

It is immediately obvious that the annotated scheduler, when given intuitively annotated code, has the worst performance. While Figure 6.5 shows that this scheduler will provide the best performance in-phase, the large overheads involved in achieving this performance make it a wasted effort. Effectively, this means that any developer lacking an in-depth knowledge of the Cell's hardware would find it very difficult to create an efficient schedule using annotations.

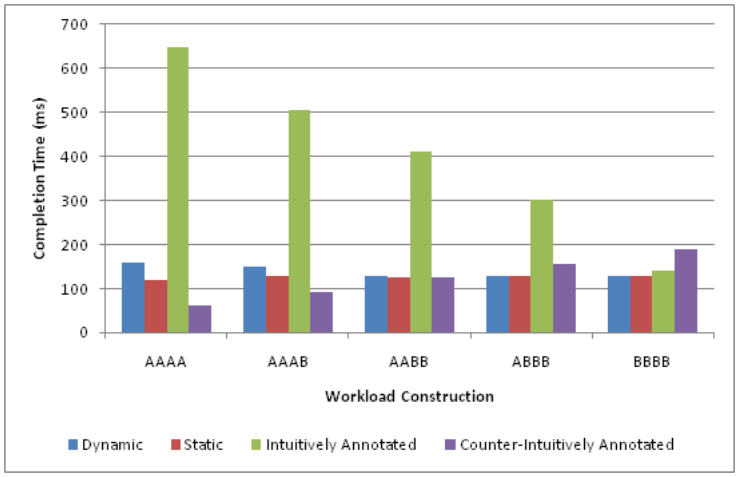

Figure 6.1: Performance of each Scheduler grouped by Workload

However, given this knowledge a counter-intuitive placement of the annotations can result in a very well performing schedule in a single thread. As is shown in Figure 6.5, this will provide performance that is comparable to the static and dynamic schedulers; however, a closer look at Table 6.1 and Figure 6.1 reveal that this technique factors affinities for the best net gain and does not allow different workloads to migrate for improved performance; it also relies entirely on a knowledge of the runtime behaviour of the program, which is why it suffers from poor performance in `BBBB` style workloads.

The dynamic affinity scheduler, however, is able to identify the affinity properties of the running code and migrate the thread based on its processing core affinities. This means that it is significantly faster than the intuitively annotated scheduler in `AAAA` workloads, while performing equally well with `BBBB` style workloads. This suggests that the dynamic affinity scheduler offers better performance on unmodified code than an annotation based scheduler offers with intuitively annotated code.

Comparing performance with the counter-intuitively annotated schedule is more difficult. The counter-intuitive schedule offers the best performance, as it reduces the number of migrations to the smallest possibly amount. As a result, it ignores `BBBB` affinities and schedules all the code to run on the SPE. As a result, it outperforms the dynamic affinity scheduler on 40% of the given workloads, matches its performance in `AABB` style workloads, and is unable to match it in the `BBBB` style workloads.

The average performance difference between the counter-intuitive and dynamic affinity schedules shows that the counter-intuitive scheduler is slightly superior in this case, as is shown in Figure 6.1. Yet this only considers a single thread of execution.

Figure 6.3 shows that the dynamic affinity and the counter-intuitive annotation schedules are almost evenly matched as the number of threads increases; as with the single thread, the counter-intuitive annotated scheduler provides slightly faster performance. However, Figure 6.2 reveals the penalties of ignoring type `B` affinities. Even up to eight threads, the actual time spent in-phase is consistently low for the dynamic affinity scheduler.

The counter-intuitively annotated schedule, in ignoring `B` affinities, creates a large amount of traffic on the Element Interconnect Bus, a phenomenon outlined in Section 5.2.3. The additional pressure on the EIB means that `B` type workloads take much longer than expected to finish on the SPE; in effect, their affinity to the PPE actually increases with the number of threads.

The dynamic affinity and intuitively annotated schedules both provide the best in-phase performance; however, the high overheads of the intuitive schedule make it entirely useless.

The evidence in this case supports the hypothesis; the dynamic affinity scheduler is superior to the annotation based scheduler in creating affinity schedules. If the code is annotated intuitively enormous scheduling penalties will render is effectively useless; however, it is the only way to achieve the lowest in-phase performance. If the code is annotated counter-intuitively, it relies on a deep understanding of the system. Yet it ignores some affinity properties, and as a result suffers poor in-phase performance.

The evidence suggests that only the dynamic affinity scheduler is able to distribute the workload in a manner that maintains low overheads and high in-phase performance; as such, it can be concluded that it is better than the annotated scheduler at creating affinity-based schedules, validating the hypothesis.

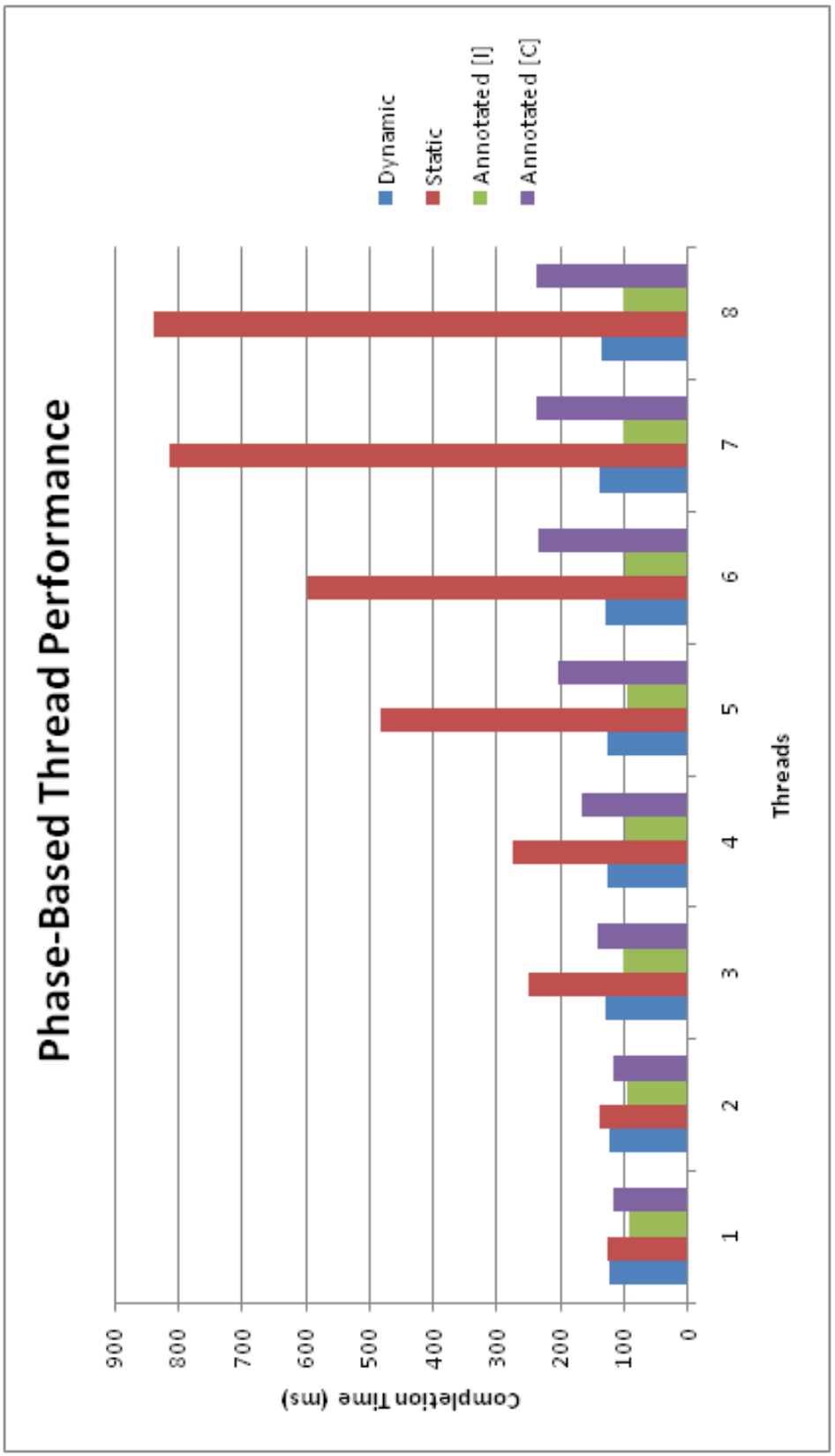

Figure 6.2: Phase Performance of each Scheduler over Multiple Threads

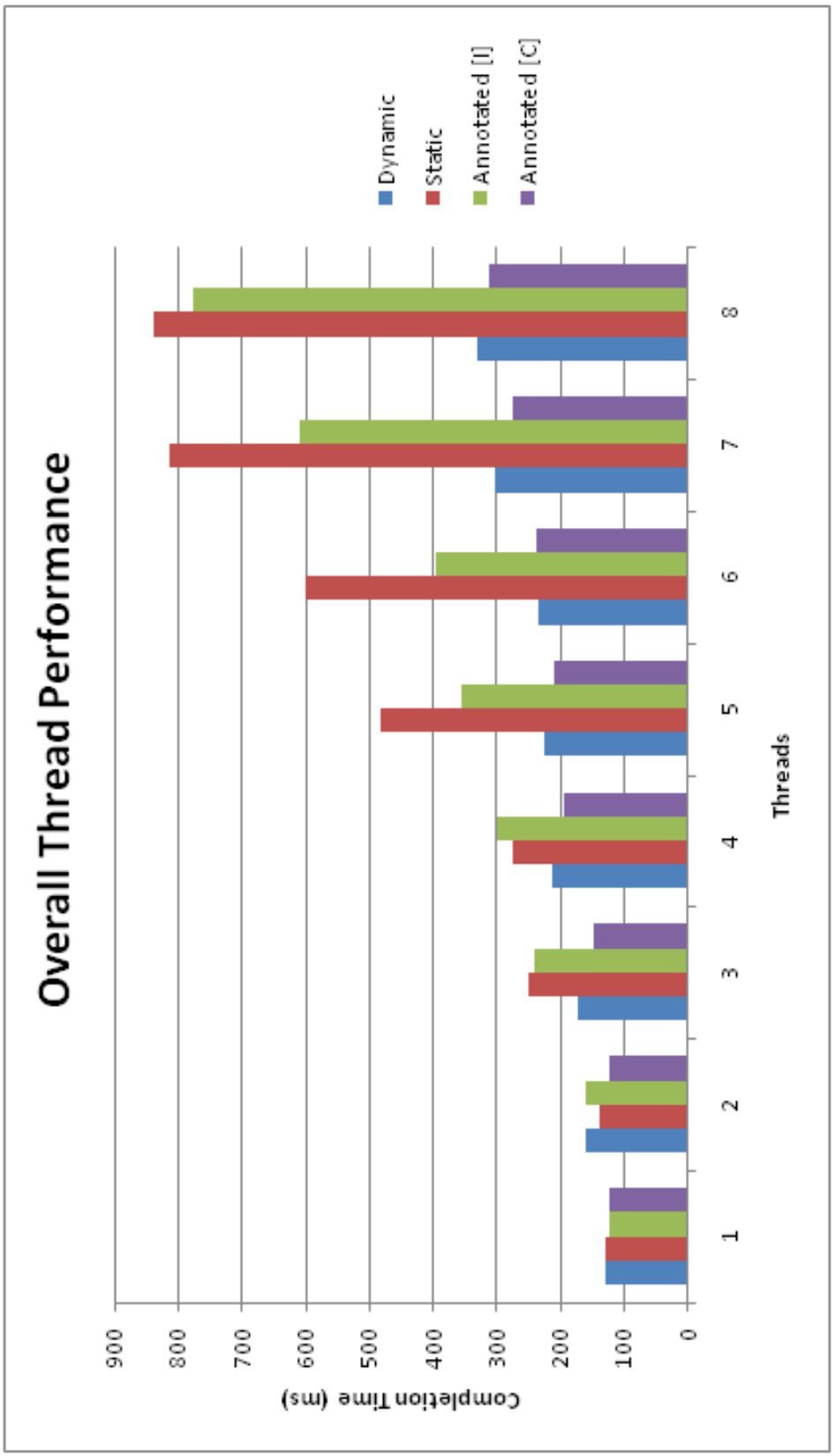

Figure 6.3: Overall Performance of each Scheduler over Multiple Threads

6.2 Evaluation of H2

H2: The performance of the dynamic affinity and annotation-based schedulers will improve for well-defined workloads with clear phase boundaries.

While it was expected that the dynamic affinity scheduler would have an ideal 'migration point density', the results of the experiment do not support this theory. Table 6.2 shows how the average completion time changes relative to the weight of the workload; this information is also presented visually in Figure 6.4.

Scheduler	Very Light	Light	Medium	Heavy	Very Heavy
Dynamic	144 ms	136 ms	137 ms	128 ms	126 ms
Static	125 ms	125 ms	127 ms	126 ms	124 ms
I-Annotated	1051 ms	506 ms	252	121 ms	97 ms
C-Annotated	133 ms	126 ms	123 ms	120 ms	119 ms

Table 6.2: Finish Times by Workload Weight

The 'weight' of the workload is described in Chapter 4. The 'heavier' the workload, the longer the time spent executing each phase. To ensure the results can be compared easily, the number of iterations over the workload are reduced as the phase length is increased. The number of migration points will decrease as the weight of the workload increases.

It was predicted that this would have an effect on the static and the dynamic schedulers; however, the results of the experiment strongly suggest that this is not the case. Performance for the dynamic workload remains consistent over all of the workloads, with the results reported in Section 5.3.2 showing a standard deviation of only 5.3% in overall completion time and 1.5% in phase completion time.

The annotation scheduler provides results that support this theory. Figure 6.4 shows that the additional migrations caused by a light workload create enormous overheads. The only time that the intuitive schedule is able to offer competitive performance is when scheduling very heavy workloads with few migration opportunities.

The counter-intuitive annotated schedule is designed to reduce the overheads significantly by placing the migration at the highest point. As such, this scheduler would not be affected by any changes in the workload weight, which is reflected in the figures presented in Table 6.2. The standard deviation between its timings is 4.5% for overall time, and 4% for phase time. These standard deviations are comparable to those for the dynamic affinity scheduler.

The evidence suggests that this hypothesis is incorrect; neither the dynamic or the affinity schedulers demonstrate a performance difference

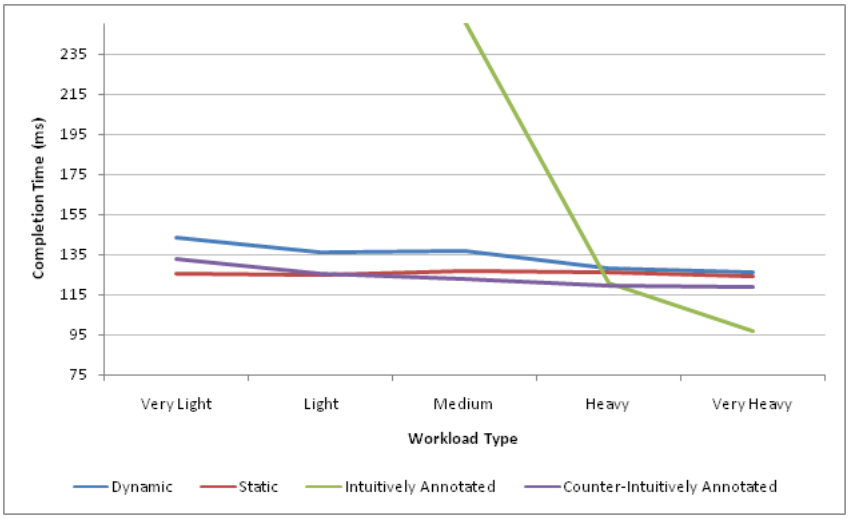

Figure 6.4: The Completion Times for each Scheduler over Various Workloads

when the weight of the workload and the number of migration opportunities is changed. While the large performance difference in the intuitively annotated schedule appears to support the hypothesis, the counter-intuitive scheduler demonstrates that it is the position of the annotations, and not the construction of the workload, that is responsible for the performance division.

6.3 Evaluation of H3

H3: The dynamic affinity scheduler and the annotation-based scheduler will have better multi-threaded performance than the static scheduler.

This hypothesis is based on the knowledge that the static scheduler does not possess multi-threading capabilities. As such, it is expected to fare poorly in the multi-threaded tests. Table 6.3 compares the the results of the multi-threaded experiment. This information is also shown in Figures 6.3 and 6.2.

These results demonstrate the the static scheduler performs worst when faced with multiple threads. With all of the threads competing for resources on the same processor, it displays an average performance that is worse than that of the intuitively annotated scheduler. In addition, it also has the longest in-phase times.

While the static scheduler offers excellent performance over a single thread, this performance scales very poorly. The evidence in this case strongly supports the hypothesis, demonstrating that the dynamic affinity and annotation-based schedulers offer far superior multi-

Overall Performance				
Threads	Dynamic	Static	Annotated [I]	Annotated [C]
1	126	126	121	120
2	159	138	158	122
3	170	250	238	145
4	210	272	299	192
5	223	481	354	207
6	232	600	393	235
7	300	814	609	273
8	330	837	775	311

In-Phase Performance				
Threads	Dynamic	Static	Annotated [I]	Annotated [C]
1	121	124	89	115
2	121	138	93	116
3	126	250	99	139
4	124	272	96	164
5	123	480	94	202
6	126	597	95	232
7	135	813	99	235
8	132	837	98	235

Table 6.3: Multi-Threaded Performance by Scheduler

threaded performance.

6.4 Evaluation of H4

`H4`: For both the dynamic affinity scheduler and the annotation-based scheduler, the benefit of affinity scheduling will be greater than the cost.

This hypothesis attempts to establish if dynamic affinity and annotation based scheduling techniques are actually beneficial. For either technique to have real-world applications, it would need to ensure a net saving when compared to the alternatives.

Scheduler	Avg Total (ms)	Avg Phase (ms)
Dynamic	134	122
Static	125	123
Intuitively Annotated	405	102
Counter-Intuitively Annotated	124	119

Table 6.4: Average Performance and Overheads

Table 6.4 presents the average performance over all workload weights and patterns for a single thread, which is also shown in Figure 6.5. From these, it can be clearly seen that intuitively annotating code will result in a poor schedule. While it offers the highest increase in phase time, it comes at a high price.

However, the counter-intuitive schedule presents excellent performance by actively reducing these overheads. The results of the experiment suggest that correctly annotated code will offer a considerable performance gain. While the actual overheads and savings are directly linked to the placement of the annotations, the evidence suggests that the annotated scheduler can improve performance enough to cover the costs of affinity scheduling, which supports the hypothesis.

The dynamic affinity scheduler also has high overhead costs, as it incurs the cost of both migration and measuring affinities. However, unlike the static and annotation schedulers it is able to adjust the schedule based on the affinities of each thread. As discussed in Section 6.1, this usually allows the dynamic affinity scheduler to take advantage of affinities in a way that the other schedulers cannot.

However, Table 6.4 shows that the average completion time is higher than that of the static scheduler. In a single thread, the static scheduler represents the normal performance; given that the static scheduler outperforms it on average, it suggests that the overheads are not accounted for by the performance gains.

However, this does not consider multi-threaded performance. Figure 6.3 shows that the average performance increases at a much lower rate than that other schedulers as the number of threads is increased. Also, Figure 6.2 shows that as the number of threads is increased the average time taken to complete each phase does not increase significantly.

This implies that as the number of threads increases, the performance benefit of dynamic affinity scheduling also increases. Therefore, when considering eight threads there is a greater 'saving' in which to justify the overheads. Figure 6.3 shows that the dynamic affinity and counter-intuitively annotated schedules are almost evenly matched.

Given that the dynamic affinity scheduler offers the best in-phase performance, at this point the substantial cost of affinity scheduling fits within the savings. the counter-intuitive schedule has less overheads, but also offers less savings.

As the number of threads is increased, both the dynamic and the counter-intuitive annotated scheduler show the best overall performance. The high overheads incurred by the counter-intuitive scheduler are a result of increasing the traffic on the EIB, which also manifests in the form of diminishing returns regarding in-phase savings.

The dynamic scheduler also increases the EIB traffic, but it also provides consistent in-phase performance. These facts strongly support the hypothesis that the cost of both dynamic affinity scheduling and annotation based scheduling can be justified by the performance increase.

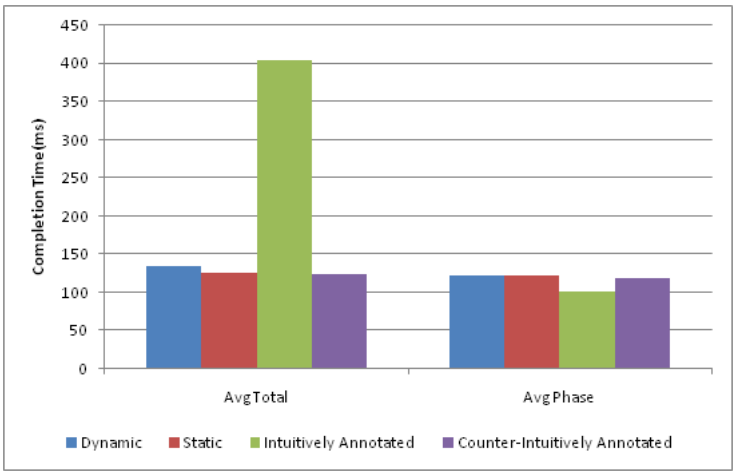

Figure 6.5: Average Time for each Scheduler to Complete the given Workload in a Single Thread

6.5 Conclusion

In this chapter the hypotheses outlined in Chapter 4.3 were evaluated. From these, it was concluded that the dynamic affinity scheduler will provide a superior affinity schedule than the annotation based scheduler; it was concluded that the dynamic affinity and annotation-based scheduler would outperform the static scheduler; and it was concluded that the dynamic affinity and annotation-based schedulers are capable of justifying their cost through performance improvements.

It was also concluded that the workload weight and the number of migration points does not affect the performance of either the dynamic affinity or the annotation-based schedulers, which refutes the second hypothesis given in Section 4.1.

This suggests that dynamic affinity scheduling techniques have considerable potential when heterogeneous processors are considered. In most cases, the dynamic affinity scheduler was able to match or outperform the other schedulers, suggesting that a higher level of heterogeneous abstraction can be achieved.

Chapter 7

Conclusion

The purpose of the research carried out here was to investigate the properties and potential of dynamic affinity scheduling techniques when applied to heterogeneous processor abstraction. Such techniques would assist in the abstraction of the heterogeneous hardware, which would make it easier to program for complex modern systems.

The literature survey carried out Chapter 2 highlighted that dynamic affinity techniques, while previously explored, have not been considered in terms of a highly heterogeneous system such as the Cell processor. Previous research was limited in this field, with research that considered hardware affinities aimed at reducing power usage or the overall completion time of the program. A knowledge gap exists when considering highly heterogeneous hardware, like the Cell processor.

An investigation was carried out into the Cell processor in Chapter 3. This involved an exploration of the Cell processor's heterogeneous nature and its code affinities. This research concluded that arithmetic code would run much faster on one of the Cell's 8 SPE cores than it would on the large PPE core; conversely, code which relies on memory access would run much slower on the SPE.

This information was used to build a prototype dynamic affinity scheduler. This scheduler was built into Hera JVM, which aims to abstract the heterogeneous nature of the processor. Hera JVM is equipped with a basic static scheduler that uses code annotations to schedule threads to cores; if the dynamic affinity scheduler developed in Section 3.4.2 could be shown to be more effective than this scheduler, then it could replace it, completing the abstraction of the heterogeneous processor.

An experiment was designed in Chapter 4.3 that would allow the prototype dynamic affinity scheduler to be compared to the annotation-based scheduler. For this experiment, the annotation-based scheduler was represented in three ways; unannotated code was used to represent a static scheduler, and two different annotation configurations were used to fully explore the properties of annotation-based scheduling.

Four hypotheses were constructed based on the information taken from the investigation, each one designed to explore a different aspect of each scheduler. The first hypothesis asserted that the dynamic affinity scheduler would be able to provide more useful affinity-based schedules than the annotation based scheduler.

The second hypothesis considered that the performance of both the annotation-based and the dynamic scheduler would depend on the number of migration opportunities and the length of each test phase.

The third hypothesis considered multi-threaded performance, and asserted that the static scheduler would not be able to provide efficient multi-threader performance compared to any of the other schedulers.

The fourth hypothesis asserted that both the annotation-based scheduler and the dynamic affinity scheduler would be able to create a large enough performance improvement to nullify the overheads of the scheduler itself.

An experiment was designed that was able to identify the costs and benefits of each scheduler, providing enough data to fully explore these hypotheses. The results of this experiment were presented in Chapter 5, and any interesting points or unusual behaviour was identified and explored.

In Chapter 6, the hypotheses were evaluated using the results of the experiment. The first hypothesis was concluded to be valid, as the evidence supported the theory that the dynamic affinity scheduler would be able to create superior affinity schedules.

The second hypothesis was refuted, as the evidence suggested that the nature of the code had little impact on either the dynamic affinity or the annotation-based scheduler. The dynamic affinity scheduler performance was not relative to either the migration opportunities or the time spent in each phase, and the annotation-based scheduler performance was relative to the location of the annotations.

The third hypothesis was considered valid, as the evidence strongly supported the theory that a multi-threaded workload would complete faster when scheduled by either a dynamic affinity scheduler or an annotation based scheduler.

The fourth hypothesis was also concluded to be valid, as both the dynamic affinity scheduler and the annotation-based scheduler were able reduce the phase time enough to cover the cost of scheduling.

These results are highly significant, as they demonstrate that dynamic affinity scheduling is able to match or beat the performance of a hand-tuned annotation based scheduler.

7.1 Implications

Two configurations of the annotation-based scheduler were used during the course of this experiment; one representing the most intuitive way to annotate the code, and another representing the code annotations that would result in the highest possible performance for the test code. The intuitive annotations fared very poorly in the tests; the overheads of scheduling in this manner were so severe that code annotated in this way would actually provide the worst general level of performance.

The alternative configuration is almost completely counter-intuitive, and was labelled as such during the course of the research. For any programmer to annotated code in this manner, it would require an in-depth understanding of both the runtime properties of the code and the Cell architecture.

The dynamic affinity scheduler easily outperformed the intuitive configuration of the annotation scheduler, and managed to come very close to the performance of the counter-intuitive scheduler in every experiment. In fact, the actual in-phase times were generally lower for the dynamic affinity scheduler than they were for the 'best performance' annotations.

This meant that the dynamic affinity scheduler was generally able to offer the greatest savings for the phase code, only being held back by the high costs of migration. The first hypothesis concluded that the dynamic scheduler would produce better affinity schedules than the annotations could provide, while the third hypothesis demonstrated that the dynamic affinity scheduler would provide efficient multi-threaded performance. The fourth hypothesis concluded that the dynamic affinity scheduler was able to increase the overall performance of the system, to a level on-par with that of the best configuration for the annotation-based scheduler.

The second hypothesis, while refuted, also helps to build a case in favour of the dynamic affinity scheduler by demonstrating that these performance gains are not closely tied to the nature of the code.

In conclusion, a dynamic affinity scheduler offers consistently high performance. It also allows for the complete abstraction of the hardware in Hera JVM, as the programmer is no longer required to annotate their code to get the best level of performance. It has been shown that dynamic affinity scheduling techniques have considerable potential in heterogeneous processor abstraction.

7.2 Further Work

There is a lot of scope for further work and research. While it has been shown that dynamic affinity scheduling techniques allow for a greater degree of heterogeneous abstraction, the experiment described here is based entirely on a single benchmark program. While this program is designed to stress the schedulers as much as possible, it could be expected that real-world applications would not behave in this way. As such, further testing and research using different benchmarks such as the SPEC JVM suite should be carried out.

This research makes no attempts to specify an implementation strategy for a dynamic affinity scheduler. Although it has been implemented in Hera JVM for the purpose of research, the migration costs incurred in Hera JVM are significant.

Therefore, further research could be carried out at both the hardware and the software level. The technique described here uses compile-time to score the code, but hardware counters could perform this task just as easily at runtime, and with a greater degree of accuracy. The possibility of hardware support for dynamic affinity scheduling should be explored, and it may even be possible to create a framework that allows different heterogeneous processors to provide a consistent interface for software schedulers.

Reducing the migration overheads would significantly increase the profitability of dynamic affinity scheduling techniques. This would also be an important avenue for exploration; the dynamic affinity scheduler offered the best-value reduction in phase time during the experiment. It is likely that these overheads can be reduced further outside of the Hera JVM environment.

This research project has sufficiently demonstrated that dynamic affinity scheduling techniques could have a wide range of applications. It has been shown that they are capable of matching the performance of even the most well-configured annotation-based scheduler, and that they are a powerful technique for heterogeneous processor abstraction. These results contribute to the heterogeneous processor and compiler research fields, and provide scope for further work in this area.

Appendix A

Cell Affinity Results

This Appendix presents graphs showing the basic results of the preliminary experiment described in section 3.3.5. Graphs come in pairs, with each pair of graphs representing a particular instruction category.

The first of these graphs shows the actual finish times for each experiment, while the second graph represents the performance difference when code of this type is run on the SPE compared to the PPE.

The *y-axis* denotes the percentage of 'special' instructions. At 0%, the test sample is made up entirely of get and put field instructions. Each iteration of the test increases the number of special codes by 1%. At 100%, the code is entirely made up of special instructions. The full details of this experiment, along with the reasoning behind the tests and an interpretation of these results, is given in chapter 3. An index of the graphs is presented in table A.1.

Instruction Type	Graphs
Integer	A.1A.2
Long	A.3A.4
Float	A.5A.6
Double	A.7A.8
Method	A.9A.10
Field	A.11A.12
New	A.13A.14
Branch Perfective	A.15A.16
Branch Destructive	A.17A.18
Branch Alternating	A.19A.20
Switch/Case	A.21A.22

Table A.1: A Comparison of the Arithmetic Abilities of each Core Type.

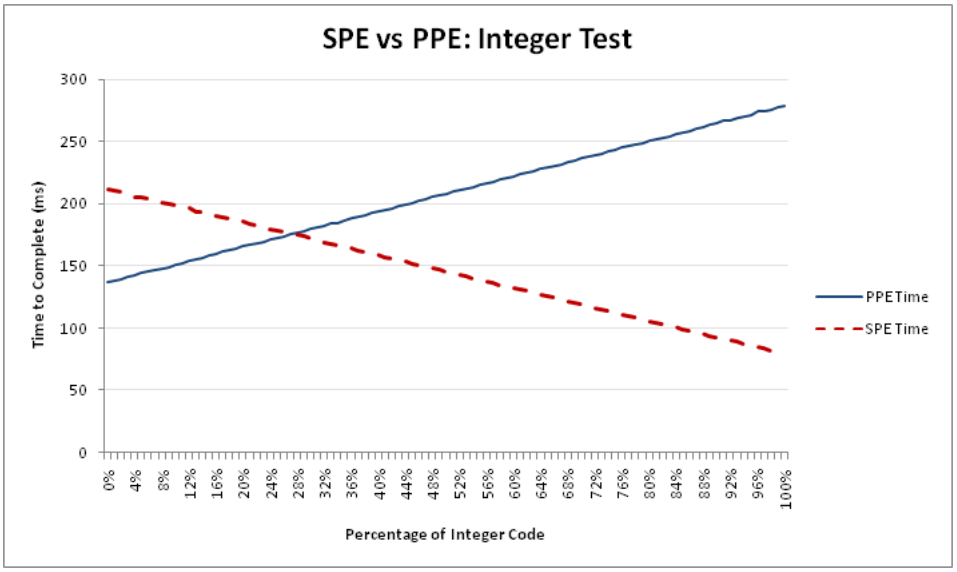

Figure A.1: Integer Code Timings

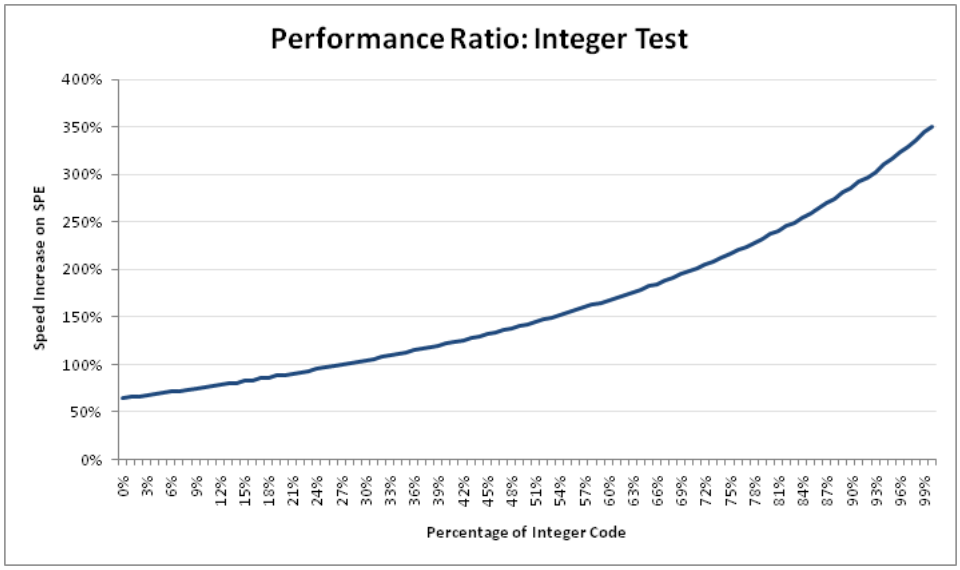

Figure A.2: Integer Speed Difference

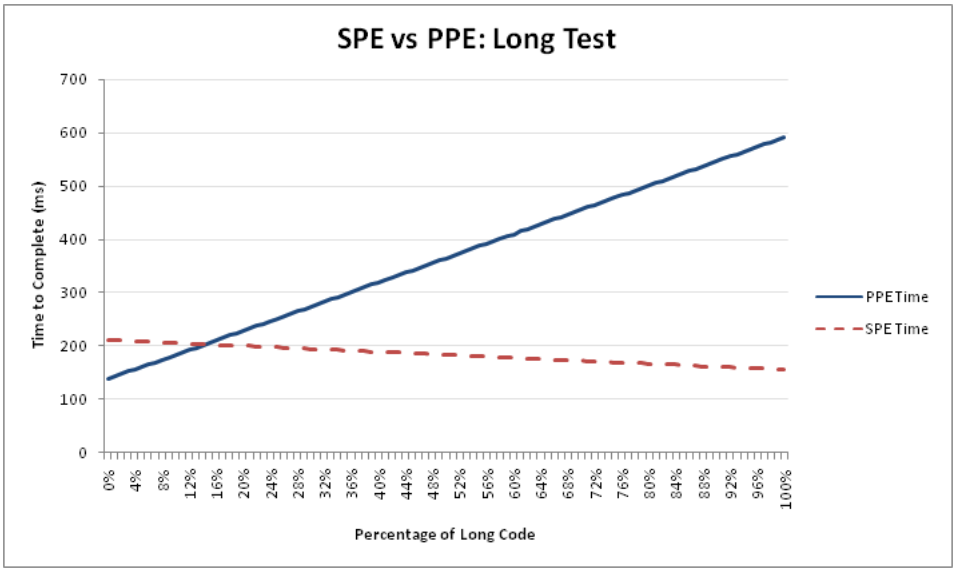

Figure A.3: Long Code Timings

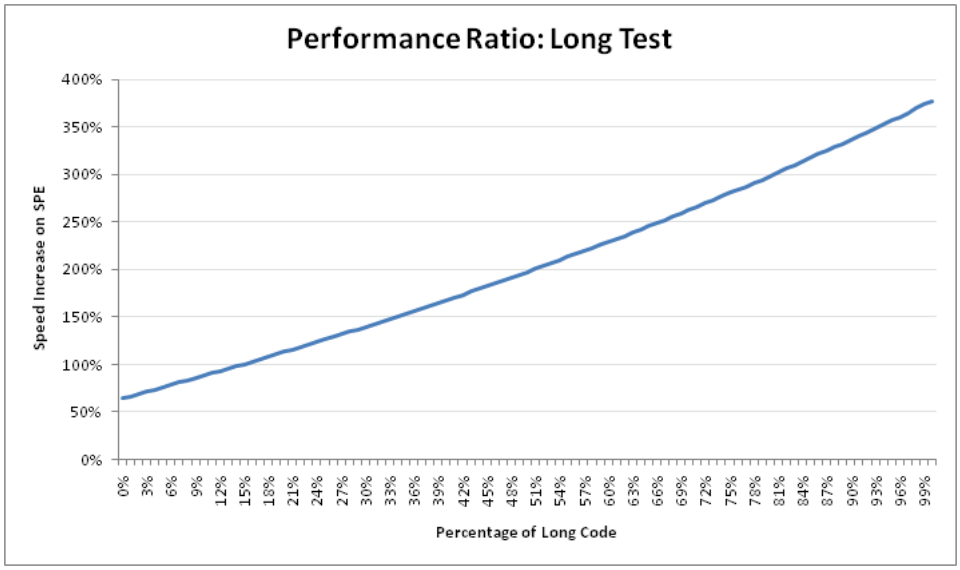

Figure A.4: Long Speed Difference

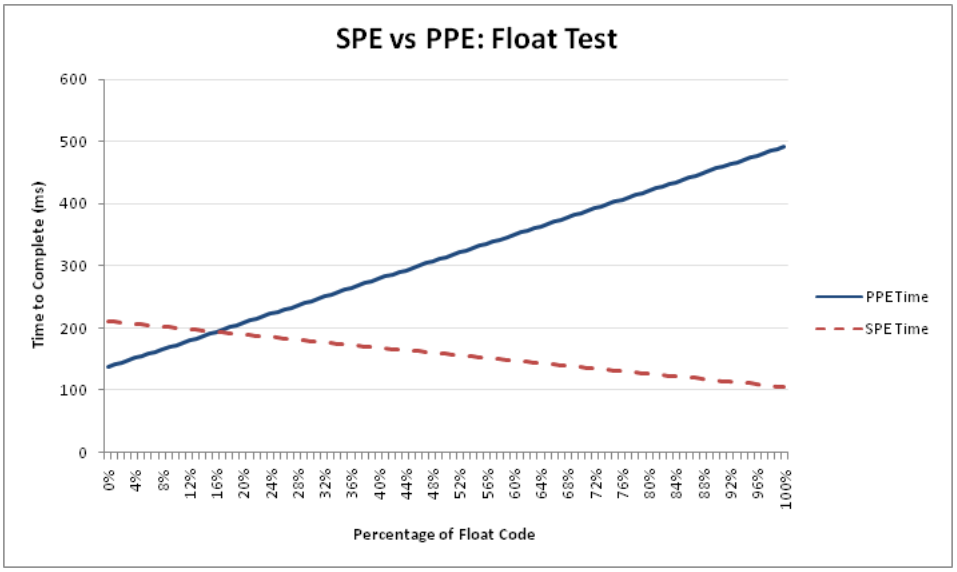

Figure A.5: Float Code Timings

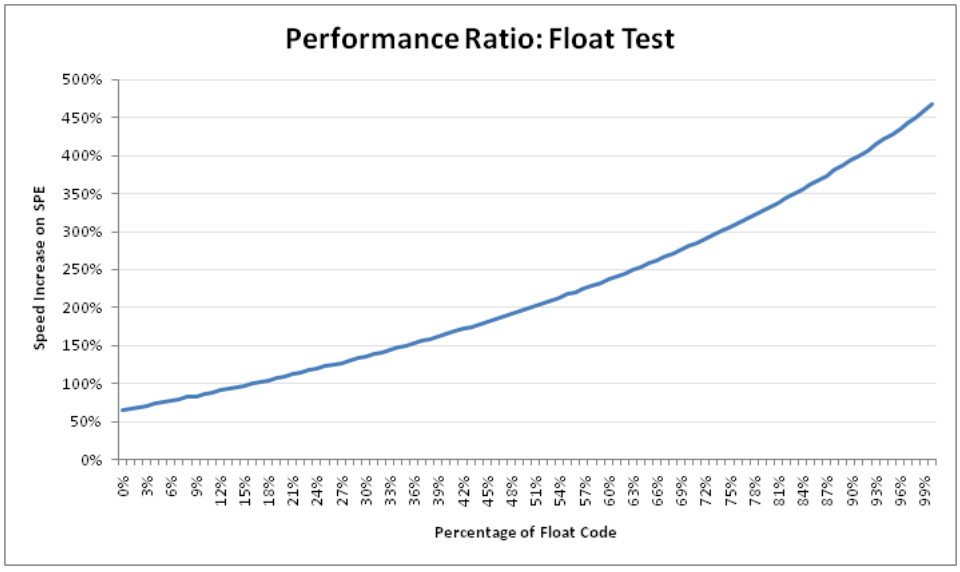

Figure A.6: Float Speed Difference

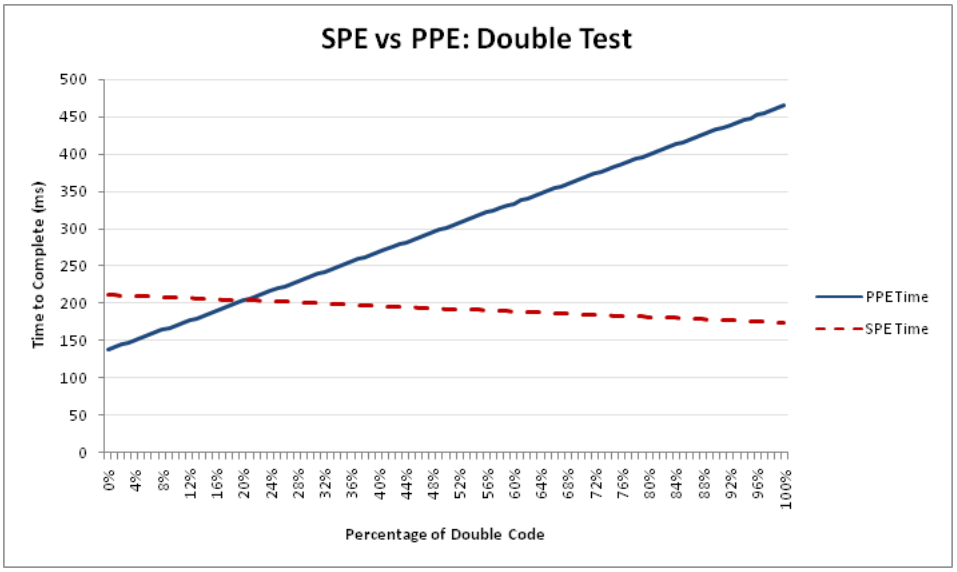

Figure A.7: Double Code Timings

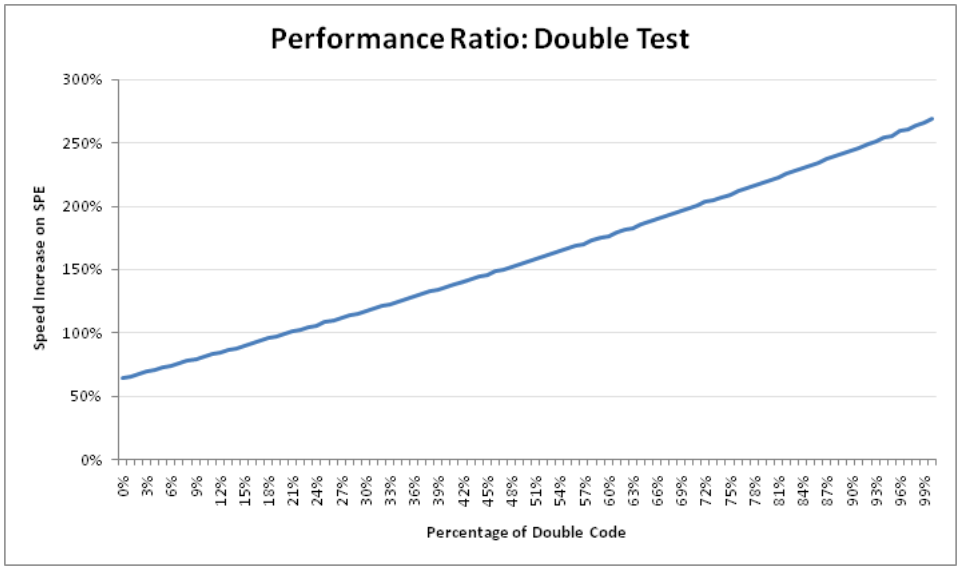

Figure A.8: Double Speed Difference

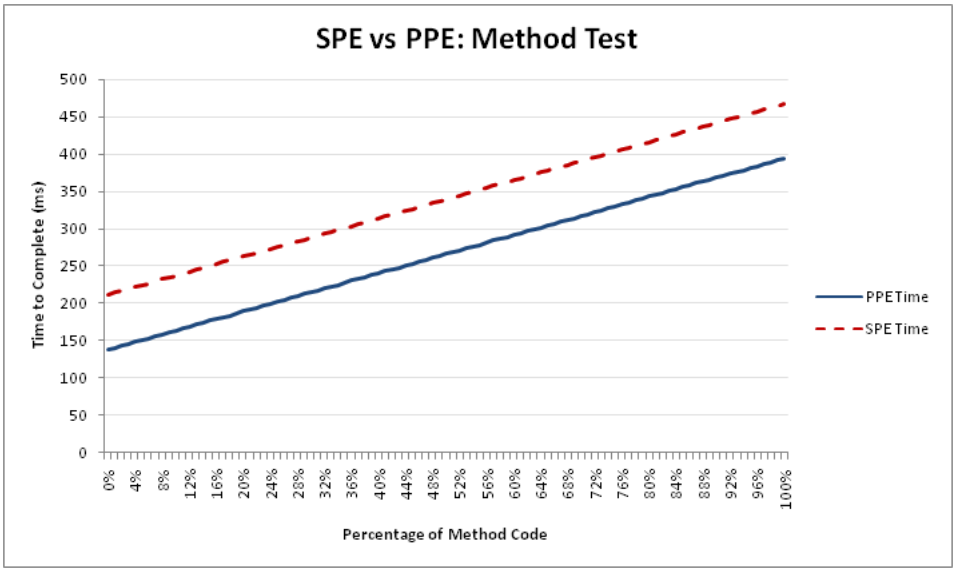

Figure A.9: Method Code Timings

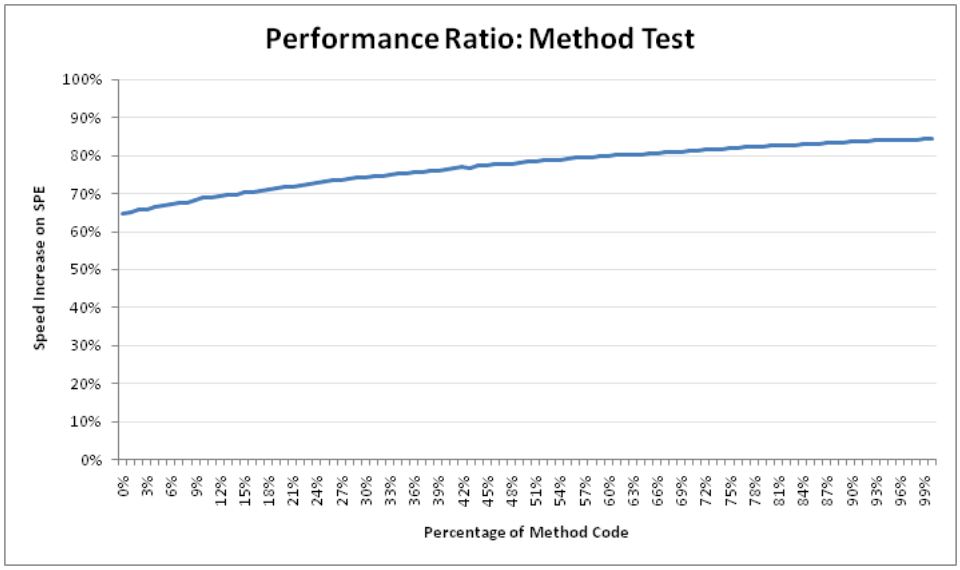

Figure A.10: Method Speed Difference

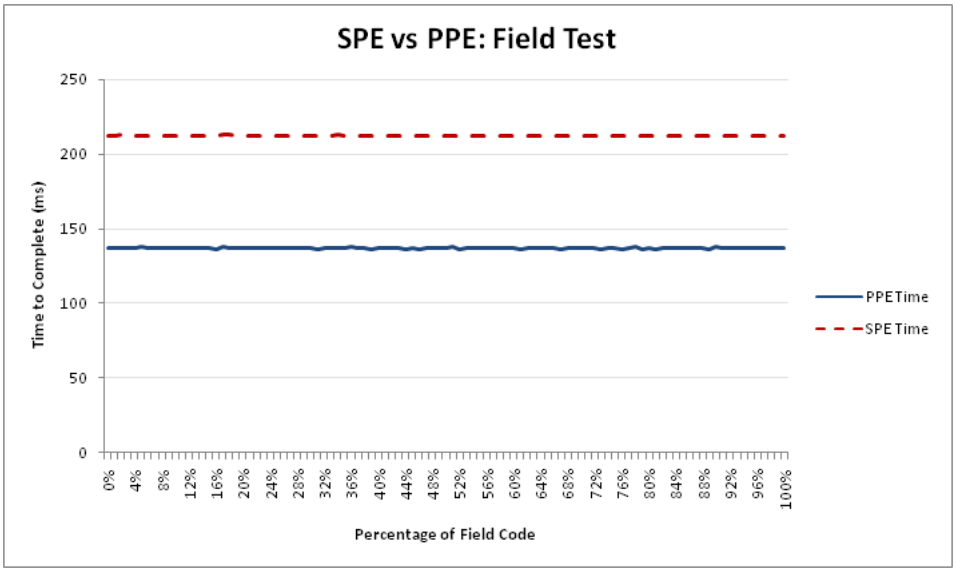

Figure A.11: Field Code Timings

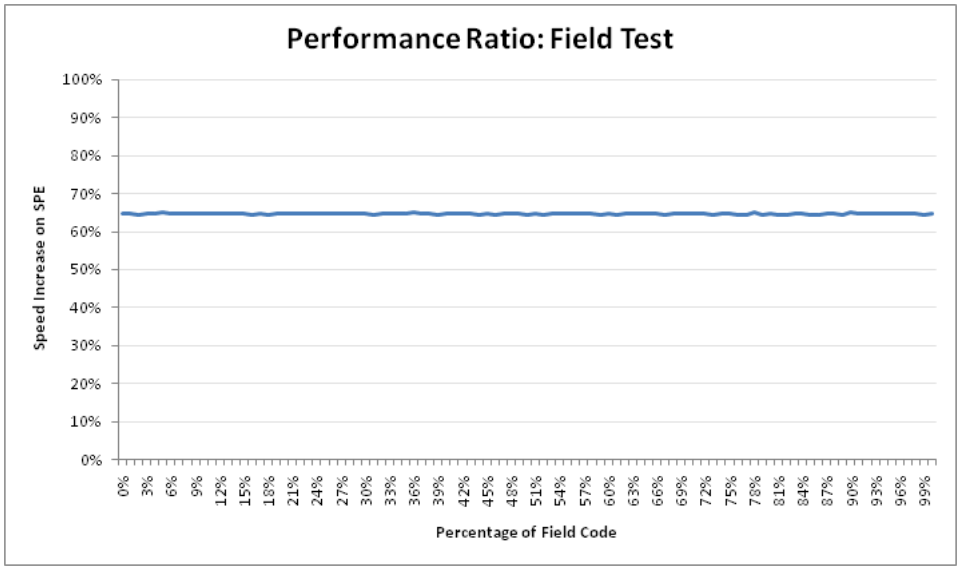

Figure A.12: Field Speed Difference

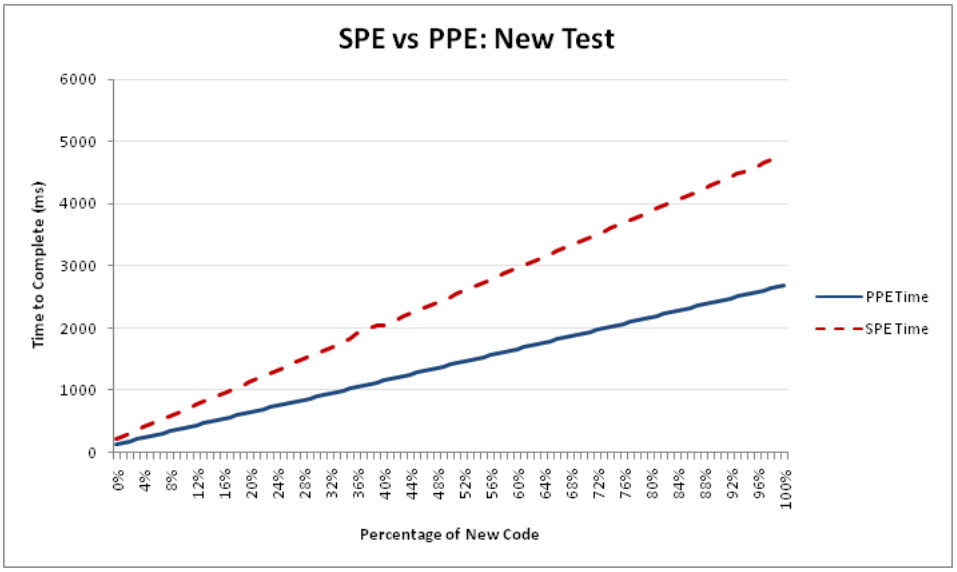

Figure A.13: New Code Timings

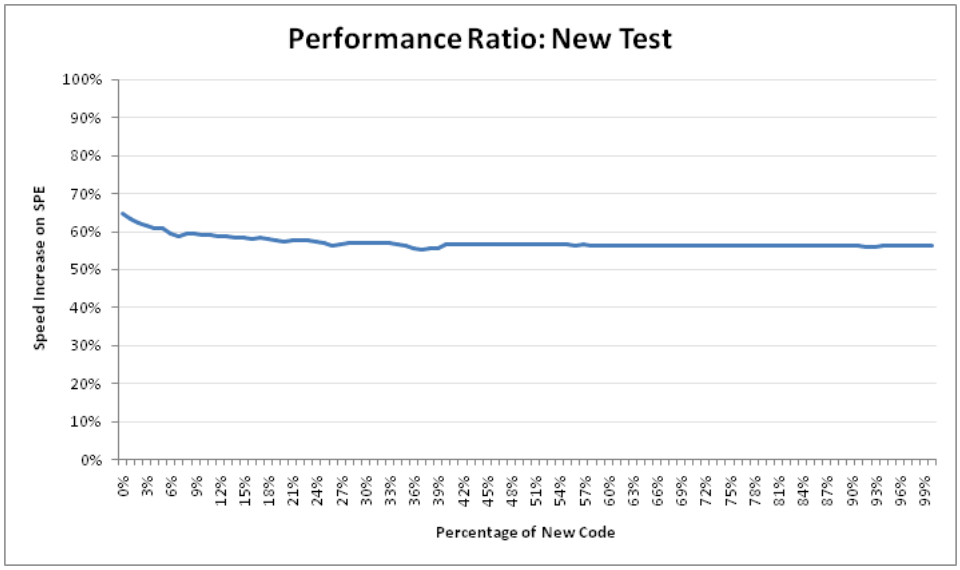

Figure A.14: New Speed Difference

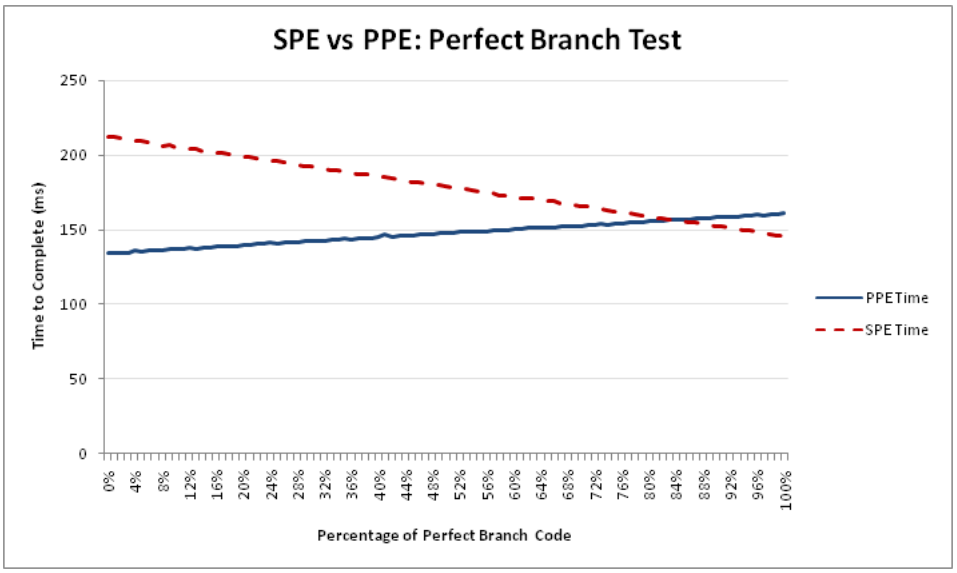

Figure A.15: Branch (Perfective) Code Timings

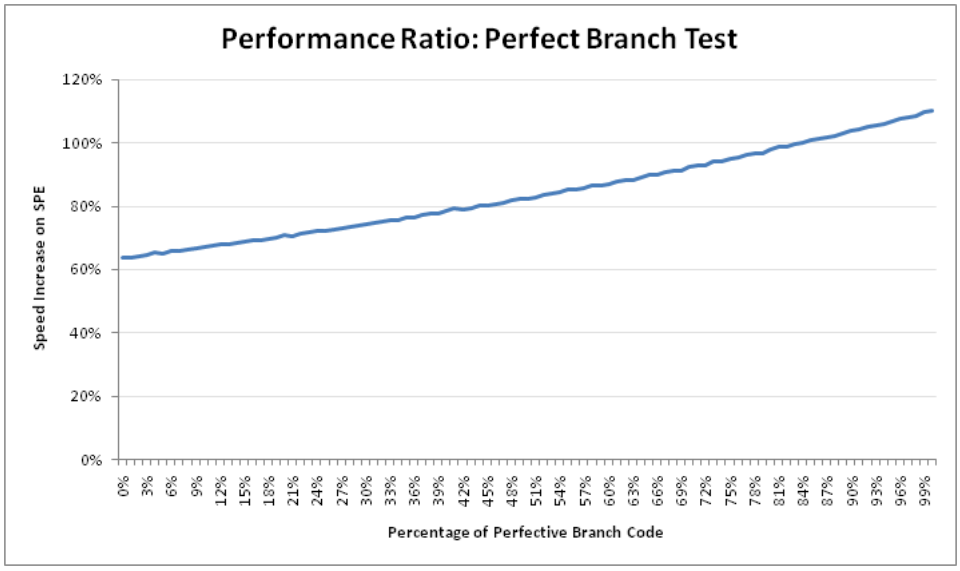

Figure A.16: Branch (Perfective) Speed Difference

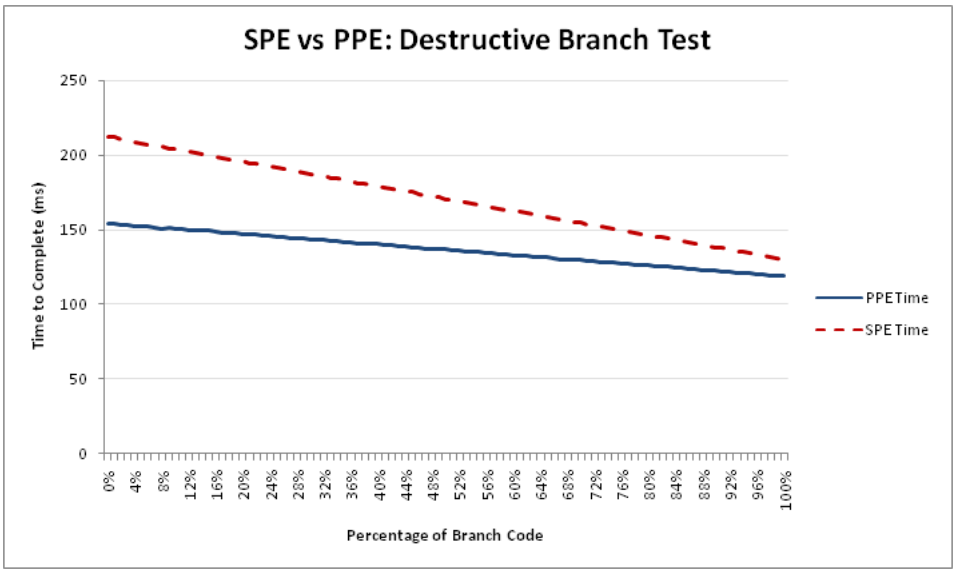

Figure A.17: Branch (Destructive) Code Timings

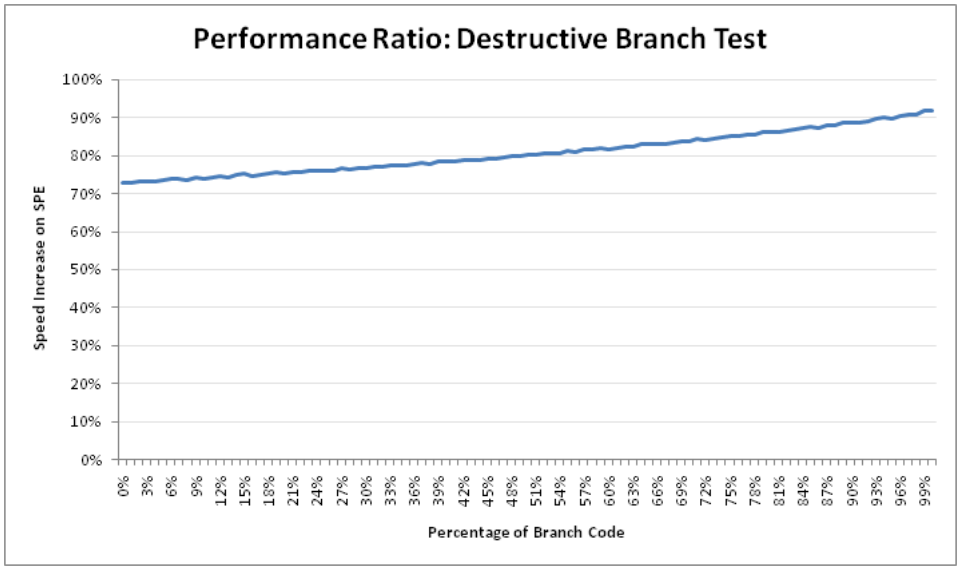

Figure A.18: Branch (Destructive) Speed Difference

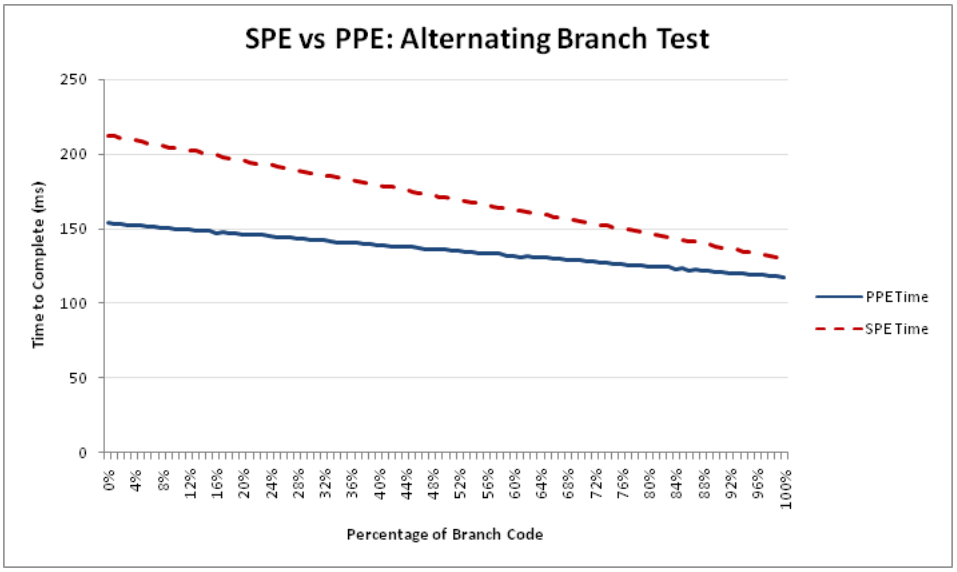

Figure A.19: Alternating Branch Code Timings

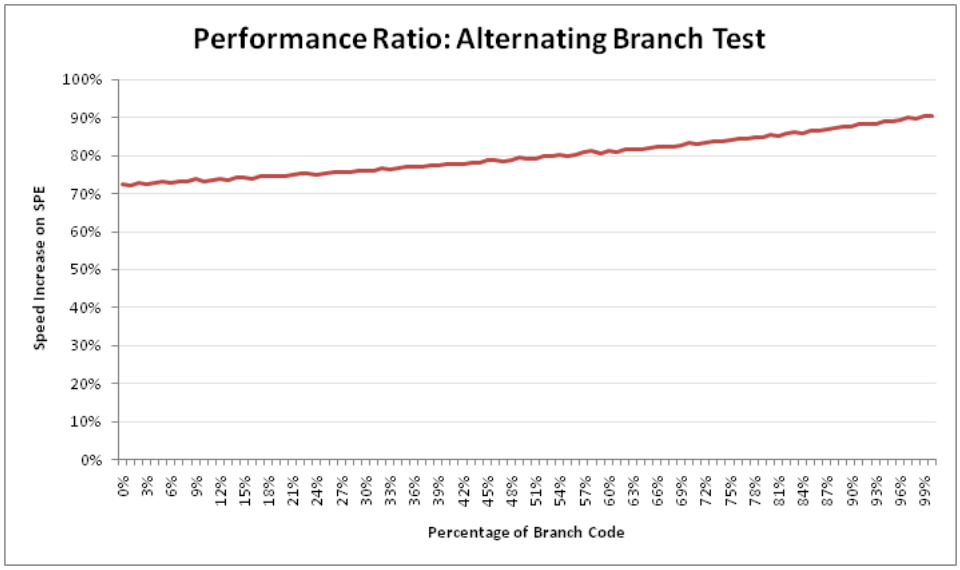

Figure A.20: Alternating Branch Speed Difference

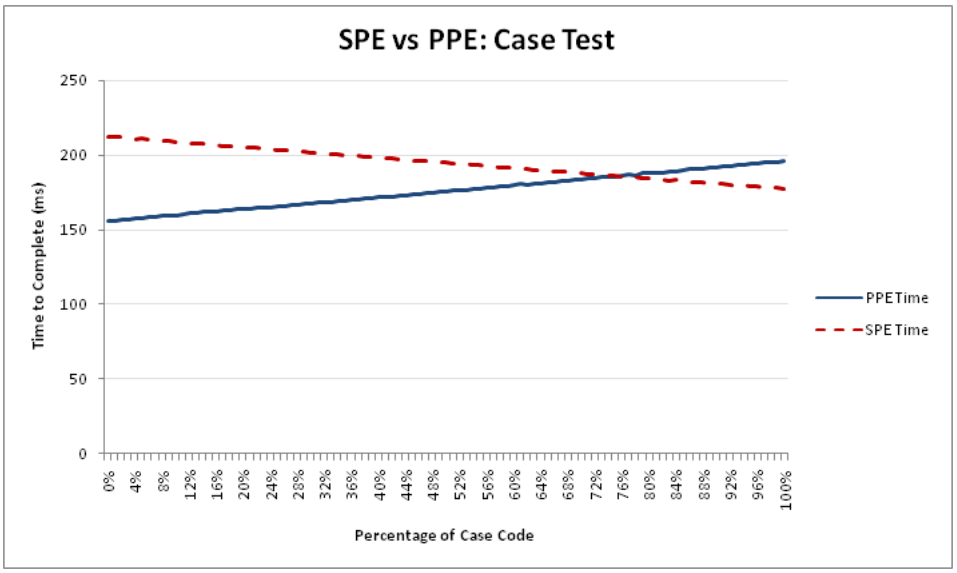

Figure A.21: Case/Switch Code Timings

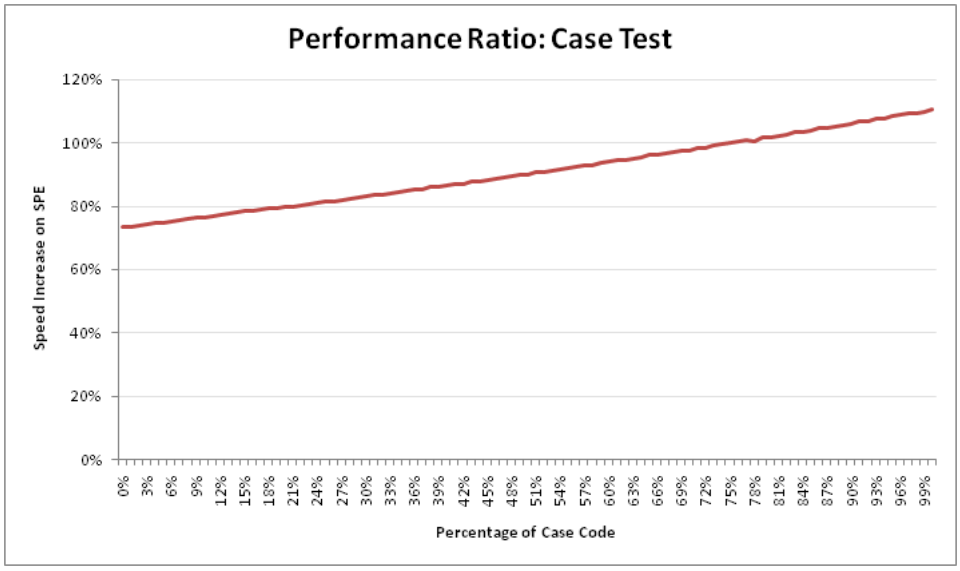

Figure A.22: Case/Switch Speed Difference

Appendix B

Full Results of Workload Experiment

In this Appendix, the full results of the workload experiment are presented. This provides additional information to accompany the results presented in Chapter 5. There are five tables in this section; an index is presented in Table B.1.

Results	Graphs
Very Heavy	B.2
Heavy	B.3
Medium	B.4
Light	B.5
Very Light	B.6

Table B.1: Index of Workload Experiment Result Tables

Very Heavy Workload								
	Dynamic		Static		Static I-Ann		Static CI-Ann	
Pattern	Total	Phase	Total	Phase	Total	Phase	Total	Phase
AAAA	120	107	120	120	66	53	57	52
AAAB	111	107	125	125	81	75	86	83
AABA	153	133	121	121	80	73	87	84
AABB	142	136	125	125	96	91	119	116
ABAA	108	104	122	122	79	71	87	84
ABAB	110	107	125	125	96	91	119	115
ABBA	140	135	125	125	95	91	119	116
ABBB	142	138	128	128	112	110	151	147
BAAA	120	120	121	121	82	72	87	84
BAAB	124	124	125	125	106	102	119	115
BABA	124	123	124	124	96	90	119	116
BABB	126	126	128	128	112	110	151	147
BBAA	122	122	124	124	95	91	119	115
BBAB	126	126	127	127	115	110	151	147
BBBA	126	126	127	127	115	110	151	147
BBBB	128	128	129	129	129	129	183	179
Average	126	122	124	124	97	91	119	115
Std. Dev	12.5	11.0	2.7	2.7	16.9	19.6	32.9	32.7
Avg. Overheads	3.17%		0.00%		6.19%		3.36%	

Table B.2: Full Results of the Very Heavy Workload Experiment.

Heavy Workload								
	Dynamic		Static		Static I-Ann		Static CI-Ann	
Pattern	Total	Phase	Total	Phase	Total	Phase	Total	Phase
AAAA	129	91	120	119	91	49	61	53
AAAB	158	119	154	117	128	70	86	85
AABA	127	115	122	122	101	66	87	85
AABB	138	132	124	124	116	92	123	116
ABAA	125	116	121	121	122	67	87	84
ABAB	129	128	124	124	133	86	119	116
ABBA	128	128	125	125	105	94	119	116
ABBB	129	129	127	127	126	107	151	148
BAAA	123	112	122	122	112	65	87	83
BAAB	129	129	125	125	156	109	123	116
BABA	127	127	125	124	131	84	123	116
BABB	131	131	127	127	132	108	151	147
BBAA	151	151	126	126	107	93	119	116
BBAB	130	130	128	128	125	108	151	148
BBBA	128	128	127	127	129	108	155	148
BBBB	133	133	129	129	127	127	183	178
Average	132	124	126	124	121	89	120	115
Std. Dev	9.4	12.8	7.7	3.3	15.6	21.4	32.7	32.5
Avg Overheads	6.06%		1.59%		26.45%		4.17%	

Table B.3: Full Results of the Heavy Workload Experiment.

Medium Workload								
	Dynamic		Static		Static I-Ann		Static CI-Ann	
Pattern	Total	Phase	Total	Phase	Total	Phase	Total	Phase
AAAA	176	113	120	119	402	49	61	55
AAAB	151	106	161	125	304	59	90	87
AABA	151	117	122	122	302	61	92	87
AABB	131	130	124	124	266	110	123	119
ABAA	150	125	124	124	301	99	91	87
ABAB	127	127	124	124	200	87	123	119
ABBA	130	130	124	124	290	113	123	120
ABBB	130	130	127	127	199	113	155	150
BAAA	151	130	122	122	299	46	91	87
BAAB	127	127	125	124	297	113	123	117
BABA	130	130	124	123	202	82	123	119
BABB	129	129	128	127	198	116	155	151
BBAA	150	150	125	125	252	110	127	117
BBAB	130	130	127	127	200	118	155	151
BBBA	131	131	127	126	198	114	159	151
BBBB	133	133	129	129	129	129	183	181
Average	139	127	127	124	252	94	123	118
Std. Dev	13.9	9.6	9.4	2.4	67.2	27.2	32.6	32.7
Avg Overheads	8.63%		2.36%		62.70%		4.07%	

Table B.4: Full Results of the Medium Workload Experiment.

Light Workload								
	Dynamic		Static		Static I-Ann		Static CI-Ann	
Pattern	Total	Phase	Total	Phase	Total	Phase	Total	Phase
AAAA	185	119	119	118	802	93	62	59
AAAB	159	126	155	122	603	112	94	90
AABA	159	123	120	119	598	98	95	91
AABB	128	127	122	122	401	95	127	121
ABAA	168	133	120	120	602	92	95	90
ABAB	128	128	122	122	400	140	123	121
ABBA	127	127	122	122	399	93	127	121
ABBB	131	131	125	124	201	128	155	153
BAAA	155	120	120	119	601	121	95	90
BAAB	127	127	122	122	400	147	127	119
BABA	128	128	122	122	402	140	127	123
BABB	132	132	125	125	1701	129	155	153
BBAA	127	127	123	123	401	134	127	117
BBAB	130	130	125	125	212	118	159	154
BBBA	130	130	125	125	210	117	156	152
BBBB	136	136	127	127	155	155	191	185
Average	140	127	124	122	505	119	125	121
Std. Dev	18.3	4.5	8.4	2.5	366.0	20.8	32.6	32.5
Avg Overheads	9.29%		1.61%		76.44%		3.20%	

Table B.5: Full Results of the Light Workload Experiment.

Very Light Workload								
	Dynamic		Static		Static I-Ann		Static CI-Ann	
Pattern	Total	Phase	Total	Phase	Total	Phase	Total	Phase
AAAA	194	121	119	118	1886	177	69	65
AAAB	177	124	159	121	1402	105	102	93
AABA	177	125	120	120	1419	103	99	97
AABB	126	126	122	122	940	144	131	126
ABAA	170	125	121	121	1383	103	99	96
ABAB	127	127	122	122	2435	31	131	127
ABBA	126	126	123	123	950	147	135	125
ABBB	129	129	125	125	524	101	163	158
BAAA	170	122	121	121	1494	128	103	95
BAAB	126	125	123	123	919	157	135	126
BABA	126	126	123	123	907	74	131	128
BABB	128	127	125	125	464	100	167	158
BBAA	127	126	124	124	966	169	131	126
BBAB	128	128	126	125	471	102	167	158
BBBA	129	127	125	124	499	100	167	160
BBBB	132	132	128	128	153	153	199	186
Average	143	126	125	122	1050	118	133	126
Std. Deviation	24.5	2.6	9.3	2.4	595.3	38.1	33.7	32.0
Avg Overheads	11.89%		2.40%		88.76%		5.26%	

Table B.6: Full Results of the Very Light Workload Experiment.

Appendix C

Full Results of Thread Experiment

In this Appendix, the full results of the thread experiment are presented. This provides additional information to accompany the results presented in Chapter 5. This information is presented in four tables, one for each workload; an index is presented in Table C.1.

Results	Graphs
Dynamic Scheduler	C.2
Static Scheduler	C.3
Annotated Scheduler (Intuitive)	C.4
Annotated Scheduler (Unintuitive)	C.5

Table C.1: Index of Thread Test Tables

	Thread 1		Thread 2		Thread 3		Thread 4		Thread 5		Thread 6		Thread 7		Thread 8	
Threads	T	P	T	P	T	P	T	P	T	P	T	P	T	P	T	P
1	126	121														
2	170	121	148	120												
3	170	123	166	129	173	126										
4	195	128	202	123	196	122	247	123								
5	221	122	242	125	213	124	216	121	223	124						
6	265	126	201	127	259	124	217	133	234	124	213	124				
7	285	130	277	138	292	131	382	134	301	141	302	135	261	139		
8	359	136	318	127	336	134	345	125	429	136	316	127	258	139	277	132

Table C.2: Full Dynamic Thread Test Results

	Thread 1		Thread 2		Thread 3		Thread 4		Thread 5		Thread 6		Thread 7		Thread 8	
Threads	T	P	T	P	T	P	T	P	T	P	T	P	T	P	T	P
1	126	124														
2	136	136	140	139												
3	250	250			249	249										
4			277	277			267	267								
5	486	486			478	478	479	477	482	479						
6			604	603	598	595	602	599	595	592	602	594				
7	814	814	820	820	814	810	811	811	809	809	819	819	809	809		
8	847	847			838	838	850	850	824	824	836	836	827	827	835	835

Table C.3: Full Results of the Heavy Workload Experiment.

	Thread 1		Thread 2		Thread 3		Thread 4		Thread 5		Thread 6		Thread 7		Thread 8	
Threads	T	P	T	P	T	P	T	P	T	P	T	P	T	P	T	P
1	121	89														
2	153	94	162	92												
3	226	99	238	98	251	99										
4	281	94	295	95	301	99	320	96								
5	332	95	343	95	356	94	365	93	374	95						
6	370	94	396	95	390	95	398	97.6	402	94	400	94				
7	496	100	612	99	621	100	623	102	631	98	638	100	645	95		
8	816	96	757	100	767	100	775	98	783	96	791	97	704	98	808	100

Table C.4: Full Results of the Medium Workload Experiment.

	Thread 1		Thread 2		Thread 3		Thread 4		Thread 5		Thread 6		Thread 7		Thread 8	
Threads	T	P	T	P	T	P	T	P	T	P	T	P	T	P	T	P
1	120	115														
2	121	116	122	116												
3	145	139	146	139	145	139										
4	264	165	167	162	167	164	168	164								
5	205	201	206	200	206	201	206	201	210	206						
6	245	242	233	230	233	230	234	231	233	230	234	231				
7	249	230	265	235	266	236	274	239	285	238	293	240	276	224		
8	297	236	307	237	297	240	322	237	323	235	318	239	312	225	314	232

Table C.5: Full Results of the Light Workload Experiment.

Bibliography

[1] Bala, V., Duesterwald, E., and Banerjia, S. Dynamo: a transparent dynamic optimization system. *SIGPLAN Not. 35*, 5 (2000), 1–12.

[2] Becchi, M., and Crowley, P. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *CF '06: Proceedings of the 3rd conference on Computing frontiers* (New York, NY, USA, 2006), ACM, pp. 29–40.

[3] Bellens, P., Perez, J. M., Badia, R. M., and Labarta, J. Cellss: a programming model for the cell be architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 2006), ACM, p. 86.

[4] Blagojevic, F., Nikolopoulos, D. S., Stamatakis, A., and Antonopoulos, C. D. Dynamic multigrain parallelization on the cell broadband engine. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2007), ACM, pp. 90–100.

[5] Blagojevic, F., Nikolopoulos, D. S., Stamatakis, A., Antonopoulos, C. D., and Curtis-Maury, M. Runtime scheduling of dynamic parallelism on accelerator-based multi-core systems. *Parallel Comput. 33*, 10-11 (2007), 700–719.

[6] Burke, M. G., Choi, J.-D., Fink, S., Grove, D., Hind, M., Sarkar, V., Serrano, M. J., Sreedhar, V. C., Srinivasan, H., and Whaley, J. The jalapeno dynamic optimizing compiler for java. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande* (New York, NY, USA, 1999), ACM, pp. 129–141.

[7] Dhodapkar, A. S., and Smith, J. E. Comparing program phase detection techniques. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2003), IEEE Computer Society, p. 217.

[8] Duesterwald, E., and Bala, V. Software profiling for hot path prediction: less is more. *SIGPLAN Not. 35*, 11 (2000), 202–211.

[9] Duesterwald, E., Cascaval, C., and Dwarkadas, S. Characterizing and predicting program behavior and its variability. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, Sept.-1 Oct. 2003), IEEE Computer Society, pp. 220–231.

[10] Flachs, B., Asano, S., Dhong, S., Hofstee, H., Gervais, G., Kim, R., Le, T., Liu, P., Leenstra, J., Liberty, J., Michael, B., Oh, H.-J., Mueller, S., Takahashi, O., Hatakeyama, A., Watanabe, Y., Yano, N., Brokenshire, D., Peyravian, M., To, V., and Iwata, E. The microarchitecture of the synergistic processor for a cell processor. *Solid-State Circuits, IEEE Journal of 41*, 1 (Jan. 2006), 63–70.

[11] Kahle, J. A., Day, M. N., Hofstee, H. P., Johns, C. R., Maeurer, T. R., and Shippy, D. Introduction to the cell multiprocessor. *IBM J. Res. Dev. 49*, 4/5 (2005), 589–604.

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

[13] Kumar, R., Farkas, K. I., Jouppi, N. P., Ranganathan, P., and Tullsen, D. M. 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* (Washington, DC, USA, 2003), IEEE Computer Society, p. 81.

[14] Kumar, R., Tullsen, D. M., Ranganathan, P., Jouppi, N. P., and Farkas, K. I. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture* (Washington, DC, USA, 2004), IEEE Computer Society, p. 64.

[15] Li, T., Baumberger, D., Koufaty, D. A., and Hahn, S. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 2007), ACM, pp. 1–11.

[16] McIlroy, R., and Sventek, J. Hera-jvm: Abstracting processor heterogeneity behind a virtual machine. In *HotOS '09: 12th Workshop on Hot Topics in Operating Systems* (2009).

[17] Perez, J. P., Bellens, P., Badia, R. M., and Labarta, J. Cellss: making it easier to program the cell broadband engine processor. *IBM J. Res. Dev. 51*, 5 (2007), 593–604.

[18] Sherwood, T., Perelman, E., and Calder, B. Basic block distribution analysis to find periodic behavior and simulation

points in applications. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 2001), IEEE Computer Society, pp. 3–14.

[19] Sherwood, T., Perelman, E., Hamerly, G., and Calder, B. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2002), ACM, pp. 45–57.

[20] Sherwood, T., Sair, S., and Calder, B. Phase tracking and prediction. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture* (New York, NY, USA, 2003), ACM, pp. 336–349.

[21] Sondag, T., Krishnamurthy, V., and Rajan, H. Predictive thread-to-core assignment on a heterogeneous multi-core processor. In *PLOS '07: Proceedings of the 4th workshop on Programming languages and operating systems* (New York, NY, USA, 2007), ACM, pp. 1–5.

[22] Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., and Yelick, K. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers* (New York, NY, USA, 2006), ACM, pp. 9–20.