Hera-JVM: Abstracting Processor Heterogeneity Behind a Virtual Machine

Ross McIlroy and Joe Sventek
Department of Computing Science
University of Glasgow
{ross,joe}@dcs.gla.ac.uk

Abstract

Heterogeneous multi-core processors, such as the Cell processor, can deliver exceptional performance, however, they are notoriously difficult to program effectively. We present Hera-JVM, a runtime system which hides a processor's heterogeneity behind a homogeneous virtual machine interface. Preliminary results of three benchmarks running under Hera-JVM are presented. These results suggest a set of application behaviour characteristics that the runtime system should take into account when placing threads on different core types.

1 Introduction

Multi-core CPU architectures have become prevalent in recent years. Whilst most multi-core architectures are symmetric, a number of recent architectures incorporate different types of processing cores onto a single CPU. Chips such as the Cell processor [7] and the IXP network processor [1] employ this approach, with multiple small cores, specialised for a particular function, alongside a general purpose core for control and operating system support. Whilst these *Heterogeneous Multi-core Architectures* (HMAs) can deliver increased performance, compared to a typical symmetric multi-core architecture [4], it is notoriously difficult to exploit their potential performance in all but the most specialised applications.

This trend towards heterogeneity of processing cores is likely to increase in coming years, and move out of specialist niche markets. This will be driven by the ability to write general purpose software for Graphics Processors (GPUs) using frameworks such as CUDA [8], the emergence of GPUs based upon CPU cores (e.g. Intel Larrabee) and the promise of integrating CPUs and GPUs on a single chip (e.g. AMD Fusion). These integrated GPU/CPU chips will have similar characteristics to current HMAs, such as the IBM Cell processor, with a set of high performance processors specialised for floating point and data-parallel operations (the GPU) and the more typical control CPU. If these architectures are to be exploited by non-specialist programmers, a programming model must be developed which abstracts the difficulties involved in programming these HMAs, whilst still enabling their potential performance to be exploited.

We present Hera-JVM, a Java Virtual Machine designed to alleviate many of the difficulties involved in developing applications for HMAs. Our approach is to completely hide the architecture's heterogeneity from the developer, while allowing the developer to specify *hints* about the program's behaviour using code annotations. Hera-JVM abstracts the heterogeneity of the architecture by presenting the illusion of a homogeneous, multi-threaded virtual machine. The runtime system transparently maps application threads to the underlying heterogeneous core types, using information about each thread's behaviour (either through code annotations or runtime monitoring) and each core type's capabilities, to inform its thread placement decisions.

We have created an implementation of Hera-JVM for the Cell processor. This paper describes the challenges involved in designing a runtime system which runs concurrently on the two different core types available on the Cell processor. The Cell processor has an unusual memory architecture, in that one of the core types does not have direct access to main memory, nor a hardware cache capability. Given this constraint, we describe the provision of a relatively efficient uniform view of memory for threads running on different cores. The results of running three Java benchmarks on Hera-JVM are presented. These three benchmarks have very different characteristics and thus enable a preliminary analysis of the capabilities of each processing core type.

2 The Cell Processor

The Cell processor was developed primarily for multimedia applications, specifically the game market, where it is the main processor used by the Sony Playstation 3. It is also being actively employed in a variety of other areas, such as scientific and high performance computing and in media workloads such as video decoding.

The Cell processor consists of a single PowerPC core (PPE) and eight *Synergistic Processing Engine* (SPE) cores (Figure 1). The PPE is intended to manage the system overall and co-ordinate the SPEs. As a PowerPC based core, it can support the Linux operating system and run any applications compiled for the PowerPC. The SPEs are designed to perform the bulk of the computation on the Cell processor. They have a unique instruction-set, highly tuned for floating point, data-parallel workloads. The SPEs do not run any operating system code, relying on the PPE to perform operations such as page table up-

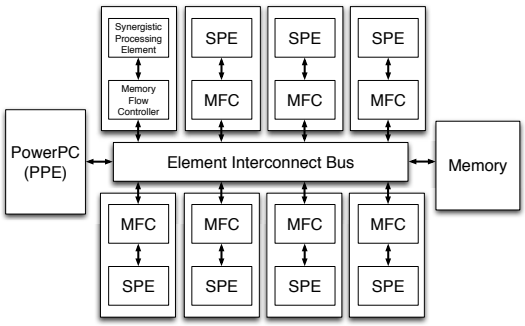

Figure 1: Cell Processor Architecture.

dates or file I/O.

Unlike the PPE, the SPE cores do not have transparent hardware caches for accessing main memory; each SPE contains 256KB of non-coherent local memory. The processing elements can access only this local memory directly. To read from or write to main memory an SPE must initiate a DMA (Direct Memory Access) transfer between main memory and its local memory using an associated Memory Flow Controller. By offloading memory reads and writes, large block transfers can be performed very efficiently; however, small transfers are much less efficient due to the overhead involved in setting up a DMA transfer. This approach suits the intended target applications of the Cell processor - large blocks of data (e.g. image fragments) being loaded, processed and then streamed out to main memory. However, it is much less suited to general purpose computation where memory is seldom accessed in large chunks **and** developers expect threads on different cores to share a coherent view of memory.

3 Hera-JVM

Hera-JVM is a Java Virtual Machine (JVM) runtime system that enables the non-specialist programmer to exploit heterogeneous cores on an HMA processor, without requiring intimate knowledge of the processor's design. Unmodified, multithreaded Java applications can be run on Hera-JVM, with threads being transparently migrated between the core types on the HMA. Currently, Hera-JVM runs on the Cell Processor, supporting execution of Java threads on both the PPE and SPE cores.

To exploit a HMA effectively, the different portions of an application must be run on the core type which they would most benefit from. Rather than require the developer to partition an application for a particular HMA, our approach is to provide the developer with a set of annotations that can enhance an application with platform-neutral *hints* of its expected behaviour. These hints, alongside runtime monitoring, inform Hera-JVM's thread placement and migration decisions. At present, this set of annotations is minimal, however, it is envisioned that these annotations will describe the behaviour of portions of code, such as tagging floating-point intensive code. On

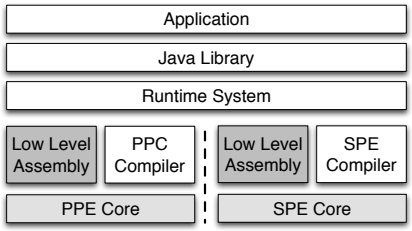

Figure 2: Much of Hera-JVMs runtime can be shared by both cores, given its Java in Java design.

encountering an annotation, the runtime system infers the best course of action (e.g. migration to a different core type) based upon its knowledge of the capabilities of the different core types and their current workloads. We intend to use the results presented in Section 4 to infer likely candidates for such annotations and deduce appropriate runtime responses to them.

3.1 Two Architectures, One JVM

Hera-JVM is based on the JikesRVM [2], a Java in Java research virtual machine. JikesRVM is a fully capable JVM with performance comparable to production JVMs. A major advantage of using JikesRVM is that it supports the PowerPC architecture, and can thus run on the PPE core of the Cell without modification. This section details the modifications necessary to support execution of Java applications on both the PPE and SPE cores.

JikesRVM (and thus Hera-JVM) is a non-interpreting JVM, with Java methods compiled to machine code before being executed. Since the SPE core has a different instruction-set to the PPE core, a Java bytecode to SPE machine code compiler is required to support the SPE cores. As a Java in Java virtual machine, almost all of the JikesRVM runtime system is written in Java. Thus, once the SPE compiler, and a small portion ($\sim$3KB) of supporting low-level assembly code is built, the rest of the runtime system (e.g. object allocation, file handling or thread scheduling) essentially comes for free (Figure 2).

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

Hera-JVM supports transparent migration of Java threads between the PPE and SPE cores. Migration occurs when invoking a method which has either been tagged by an annotation or selected by the scheduler. If required,

this method is JITed for the core type to which the thread is migrating (the method is not compiled for the core type the thread is migrating from unless it is run from that core type at a different point of execution). The parameters of the method are packaged and a marker is placed on the stack to signal this as a migration point. The thread is then placed on the ready-queue of the core to which it is being migrated, and will subsequently be scheduled by that core. The thread executes on this new core until, either it elects to migrate back, or returns to the migration marker placed on the stack.

3.2 Java on the SPE Cores

Supporting execution of Java threads on the SPE cores presents a number of challenges, not found in more typical architectures. One of the main difficulties involves the limitation of having no direct access to main memory. Instead, data must be DMAed to and from the 256KB local memory available to each SPE. Therefore, a form of software caching had to be developed for both objects and methods. These caching systems are presented in Sections 3.2.1 and 3.2.2, respectively.

Another issue is the lack of OS support on the SPEs. Section 3.2.3 details how methods which require OS support, such as file access, are handled on the SPE core.

3.2.1 Data Caching

Setting up a DMA operation to transfer data to and from local memory is an expensive operation (about 30-50 cycles, not including the data transfer itself). Therefore, an early design decision of the software cache was to transfer large blocks of memory wherever possible. This approach is enhanced by the high-level information still present in Java bytecodes. For example, an instance object will only be accessed from a given set of bytecodes (e.g. `getField`), and arrays with another set of bytecodes (`aaload`, etc.). The software cache can therefore specialise the access of different data types. When an object is accessed for the first time, the software cache transfers the entire object to local memory (type information in the bytecode is used to discover its size), on the assumption that it is likely that other fields in that object will be accessed. When an array element is accessed, a block of up to 1KB of neighbouring elements is also transferred, since they are likely to be accessed shortly.

This caching system means that cached elements are not equally sized, so space must be allocated as elements are cached. A simple bump-pointer scheme is used to manage allocation, with the cache simply being flushed if it is filled. A cache look-up involves hashing the object's address to index into a small, local-memory resident hashtable. If the object is cached, this hashtable entry will point to the local memory copy of the object. This simple strategy seems to work well; future work will investigate if more sophisticated strategies can improve performance.

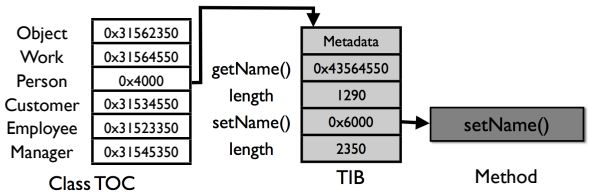

Figure 3: The code cache data structures.

In a multi-threaded application, the same object may be accessed by multiple threads. This software caching system does not support cache coherency, thus a thread running on an SPE core may not see new modifications made to an object which has been previously cached, leading to potential race conditions. In Hera-JVM, we prevent this by purging the cache before a lock or volatile field read operation and flushing any local modifications made to cached objects before an unlock or volatile field write operation. The Java Memory Model [5] allows caching of values between lock and unlock operations. Therefore, this caching scheme conforms to the Java Memory Model and any correctly synchronised multi-threaded application will run correctly under Hera-JVM.

3.2.2 Code Caching

Code must also reside on the SPE's local memory before it can be executed. Therefore, software caching of code is required on the SPE cores. In keeping with the approach of DMAing large blocks of data wherever possible, Java methods are cached in their entirety. As with the object cache, a bump pointer allocation scheme is used to manage this cache, with the cache being completely purged whenever it becomes full.

Unlike the object cache, this code cache does not use a hashtable to perform look-ups (to avoid costly hash collisions, and to enable virtual method invocation). Instead, each class has an associated *type information block* (TIB), which contains a pointer to the SPE code for each method declared by that class. These TIBs are themselves only cached in local memory when required (exploiting *class locality*). The only data-structure which permanently resides in local memory is a 2KB class *table of contents* (TOC), with an entry for each resolved class, pointing to the class's TIB. When a method is invoked, the appropriate class's TOC entry is read to locate the class's TIB. This TIB is cached, if necessary, and the method's entry is read to locate the method's code. Finally, the method is cached, if necessary, then invoked (Figure 3). This process is repeated on returning from a method, since the callee method may have been purged from the cache in the meantime. This process involves a double de-reference to find a method's location, however, in the case of a cache hit, both pointers are in the low latency (3-6 cycles) local memory, so the overhead is minimal.

The support for code caching resides in a block of in-

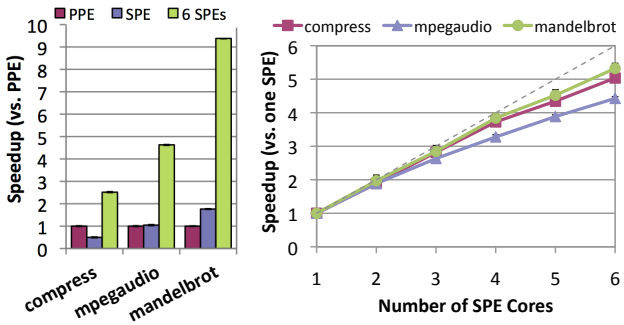

Figure 4: (a) SPE / PPE performance (b) Scalability

structions permanently held in local memory, along with low level assembly code to support SPE initialisation, object caching and a first level interrupt handler. Since the rest of the Hera-JVM runtime system is written in Java, it can be loaded as required by this caching mechanism. Thus, there is no need to specialise the SPE runtime, such that it can entirely fit in the 256KB local memory, unlike the approach taken by CellVM [6].

3.2.3 Native Method Support

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

However, if a thread is running on a SPE core, there is no underlying OS to support native methods. SPE cores must rely on the PPE core to perform native methods. In the case of a JNI method, the thread is migrated to the PPE core for the duration of the native method. For fast syscall methods, the SPE core signals a dedicated thread on the PPE core with an appropriate message. This dedicated thread performs the required syscall on the SPE thread's behalf, then signals the SPE with the result.

4 Preliminary Results

We present results for three applications running on Hera-JVM. Two applications (*compress* and *mpegaudio*) are unmodified, multi-threaded benchmarks taken from SPECjvm 2008. The other (*mandelbrot*) calculates the mandelbrot set for an 800x600 pixel image. All experiments use the baseline (non-optimising) compiler for both PPE and SPE code. Hera-JVM is configured with a mark-and-sweep, stop-the-world garbage collector, which only runs on the PPE core. These experiments were performed on a Playstation 3 running Fedora 8 Linux.

Figure 4(a) shows the performance of each benchmark when run on one or six SPE cores, relative to the PPE

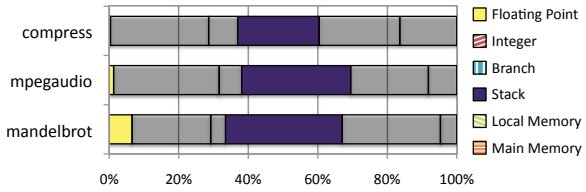

Figure 5: Proportion of cycles per operation type.

core. A large disparity between PPE and SPE performance is observed. When running on a single SPE core, *compress* runs much slower than on the PPE core, *mandelbrot* runs significantly faster and *mpegaudio* is roughly equivalent. However, there are 6 SPE cores available on the PS3's Cell processor and only one PPE core. When using all 6 SPE cores, the speedup compared to the PPE core is about 2.5x for *compress*, 4.6x for *mpegaudio* and 9.4x for *mandelbrot*. Figure 4(b) shows the scalability of each benchmarks when run on multiple SPE cores, relative to a single SPE core.

To ascertain the most important program characteristics that Hera-JVM should consider when making thread and data placement decisions, we investigated the reasons behind benchmarks running better on one core type than another. Using a simulator we calculated the proportion of processor cycles spent executing different operations when the benchmarks were running on the SPE cores (Figure 5). The *mandelbrot* benchmark performs significantly more floating point calculations than the other benchmarks. Given the SPE core's strong floating point performance, this goes some way to explaining *mandelbrot's* superior performance on the SPE core. It is also noticeable that *compress* spends more of its execution accessing main memory than the other benchmarks, likely leading to its poor performance on the SPE.

This motivated an investigation into the effectiveness of the software caches on the SPE core. Figures 6 and 7 show the effect of reducing the data and code caches on hit-rate and performance. The *compress* benchmark has a consistently lower data hit-rate than the other benchmarks and responds very poorly to a reduction in the data-cache size. On the other hand, *mpegaudio* is relatively insensitive to data-cache size, but is very susceptible to a reduction in the code-cache. These results show the importance of effective caching on the SPE architecture, and suggests that adaptive sizing of the code and data caches would likely benefit many applications.

5 Related Work

The approach of hiding the Cell processor's heterogeneity behind a Java virtual machine is also employed by CellVM [6]. CellVM supports execution of unmodified Java applications on both Cell core types, however, a single thread is bound to each SPE core, thus threads cannot be migrated transparently as in Hera-JVM. CellVM also

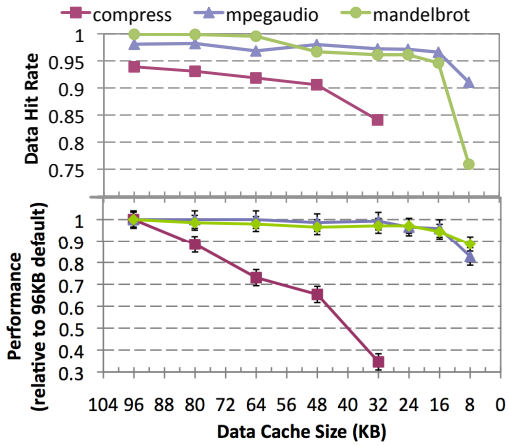

Figure 6: Changing the size of the data cache.

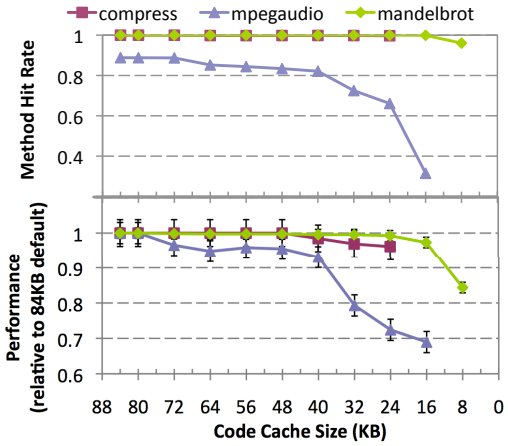

Figure 7: Changing the size of the code cache.

relies on the PPE core to perform thread synchronisation operations and the current prototype does not strictly follow the Java memory model. These limitations present scalability and correctness issues.

An alternative approach is taken by the BarrelFish Operating System [10]. Rather than abstracting the processor's heterogeneity, BarrelFish exposes this information to applications and provides mechanisms for them to deal with it. This is effectively the reverse of Hera-JVM's approach, where the applications expose information about their expected behaviour to the runtime system. It will be interesting to see how these different approaches affect performance and usability.

Other work has investigated different aspects of processor heterogeneity, most notably OS support for NUMA (non-uniform memory access) architectures [3]. Cashmere [11] provides software based, pseudo-coherent shared memory similar to Hera-JVM, but on a different scale (multi-node clusters instead of multi-core processors). Intel's manycore runtime McRT [9] is currently targeted at symmetric multi-core architectures, however, their sequestered mode, where application threads run "bare metal" on processing cores, is similar to our execution of Java threads on the SPE cores and provides a basis for McRT support of heterogeneous processors.

6 Conclusion

Hera-JVM abstracts processor heterogeneity by enabling unmodified Java applications to run across the different core types of the Cell processor. Preliminary results show that the performance of applications can be significantly affected by the core type on which it is run. This suggests that if an application can provide information about its behaviour to the runtime system, it could be scheduled more effectively. Future versions of Hera-JVM will take this approach by employing code annotation *hints* and runtime monitoring. In the long term, we expect a combination of application hints, static code analysis and runtime monitoring to enable effective binding of threads to core types in heterogeneous architectures.

7 Acknowledgements

We thank the Carnegie Trust for the Universities of Scotland for funding this work and MSR Cambridge for funding associated equipment. We also thank the JikesRVM team for open sourcing their work.

References

[1] Adiletta, M, Rosenbluth, M, Bernstein, D, Wolrich, G, and Wilkinson, H The Next Generation of Intel IXP Network Processors. *Intel Tech. Journal 6*, 3 (2002).

[2] Alpern, B, Augart, S, Blackburn, S, Butrico, M, Cocchi, A, Cheng, P, Dolby, J, Fink, S, Grove, et al. The Jikes Research Virtual Machine project: building an open-source research community. *IBM Systems Journal 44*, 2 (2005)

[3] Cox, A, and Fowler, R The implementation of a coherent memory abstraction on a NUMA multiprocessor: experiences with platinum. *ACM SIGOPS Operating Systems Review 23*, 5 (1989)

[4] Hill, M, and Marty, M Amdahl's Law in the Multicore Era. *Computer 41*, 7 (2008)

[5] Manson, J, Pugh, W, and Adve, S The Java Memory Model. In *The Symp. on Principles of Programming Languages* (2005)

[6] Noll, A, Gal, A, Franz, M CellVM: A homogeneous virtual machine runtime system for a heterogeneous single-chip multiprocessor. In *W'kshop on Cell Systems and Applications (2008)*.

[7] Pham, D, Asano, S, Bolliger, M, Day, M, Hofstee, et al. The design and implementation of a first-generation CELL processor. *IEEE Solid-State Circuits Conference* (2005).

[8] Ryoo, S, Rodrigues, C, Baghsorkhi, S, Stone, S, Kirk, D, and Wen-mei, W Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Symp. on Principles and practice of parallel programming* (2008)

[9] Saha, B, Adl-Tabatabai, A, Ghuloum, A, Rajagopalan, et al. Enabling scalability and performance in a large scale CMP environment. In *Proc. of EuroSys* (2007)

[10] Schupbach, A, Peter, S, Baumann, A, Roscoe, T, Barham, P, Harris, T, and Isaacs, R Embracing diversity in the Barrelfish manycore operating system. In *Proc. of the Workshop on Managed Many-Core Systems (MMCS)* (2008).

[11] Stets, R, Dwarkadas, S, Hardavellas, N, Hunt, G, Kontothanassis, L, Parthasarathy, S, and Scott, M Cashmere-2L: software coherent shared memory on a clustered remote-write network. In *Proc. of the Symp. on Operating Systems Principles* (1997)