# Hera-JVM: A Runtime System for Heterogeneous Multi-Core Architectures

Ross McIlroy *
Microsoft Research Cambridge
rmcilroy@microsoft.com

Joe Sventek
University of Glasgow
joe@dcs.gla.ac.uk

## Abstract

Heterogeneous multi-core processors, such as the IBM Cell processor, can deliver high performance. However, these processors are notoriously difficult to program: different cores support different instruction set architectures, and the processor as a whole does not provide coherence between the different cores' local memories.

We present Hera-JVM, an implementation of the Java Virtual Machine which operates over the Cell processor, thereby making this platforms more readily accessible to mainstream developers. Hera-JVM supports the full Java language; threads from an unmodified Java application can be simultaneously executed on both the main PowerPC-based core and on the additional *SPE* accelerator cores. Migration of threads between these cores is transparent from the point of view of the application, requiring no modification to Java source code or bytecode. Hera-JVM supports the existing Java Memory Model, even though the underlying hardware does not provide cache coherence between the different core types.

We examine Hera-JVM's performance under a series of real-world Java benchmarks from the SpecJVM, Java Grande and Dacapo benchmark suites. These benchmarks show a wide variation in relative performance on the different core types of the Cell processor, depending upon the nature of their workload. Execution of these benchmarks on Hera-JVM can achieve speedups of up to 2.25x by using one of the Cell processor's SPE accelerator cores, compared to execution on the main PowerPC-based core. When all six SPE cores are exploited, parallel workloads can achieve speedups of up to 13x compared to execution on the single PowerPC core.

***Categories and Subject Descriptors*** C.1.3 [*Processor Architectures*]: Other Architecture Styles—Heterogeneous (hybrid) systems; D.3.4 [*Programming Languages*]: Processors—Run-time environments.

***General Terms*** Design, Languages, Performances.

* Work performed while at the University of Glasgow.



*OOPSLA/SPLASH'10,* October 17–21, 2010, Reno/Tahoe, Nevada, USA.


## 1. Introduction

Commodity microprocessors are providing increasing numbers of cores to improve their performance, as issues such as memory access latency, energy dissipation and instruction level parallelism limit the performance improvements that can be gained by a single core. Current commodity multi-core processors are symmetric, with each processing core being identical. This kind of architecture provides a simple platform on which to build applications, however, a *Heterogeneous Multi-core Architecture* (HMA), consisting of different types of processing cores, has the potential to provide greater performance and efficiency [1, 6].

There are two primary ways in which an HMA can improve performance. First, heterogeneous cores allow specialisation of some cores to improve the performance of particular application types, while other cores can remain more general purpose, such that the performance of other applications does not suffer. Second, an HMA can also enable programs to scale better in the presence of serial sections of a parallel workload. Amdahl's law [4] shows that even a relatively small fraction of sequential code can severely limit the overall scalability of an algorithm. A HMA can devote silicon area towards a complex core, on which sequential code can be executed quickly, and use the rest of its silicon area for a large number of simple cores, across which parallel workloads can be scaled. This enables an HMA to provide better potential speedups compared with an equivalent symmetric architecture when Amdahl's law is taken into account [8].

However, this potential for higher performance comes at the cost of program complexity. In order to exploit an HMA, programmers must take into account: the different strengths and weaknesses of each of the available processing cores; the lack of functionality on certain cores (e.g., floating point hardware or operating system support); potentially different instruction sets and programming environments on each of the core types; and (often) a non-coherent shared memory system between cores of different types.

If mainstream application developers are to exploit HMAs, they must be made simpler to program. High level virtual machine based languages, such as Java, present an opportunity to hide the details of a heterogeneous architecture from the developer, behind a homogeneous virtual machine interface.

This paper introduces Hera-JVM, a *Java Virtual Machine* (JVM) which hides the heterogeneous nature of the Cell multi-core processor behind a homogeneous virtual machine interface. The Cell multi-core processor is a particularly challenging environment on which to develop applications, due to cores with different instruction set architectures and a non-coherent memory subsystem.

Hera-JVM supports the full Java language[1]; unmodified Java applications can be executed across both the Cell processor's main PowerPC-based core and the additional *SPE* accelerator cores. Migration of threads between core types is handled transparently from the point of view of the application and does not require application source code to be modified. Hera-JVM uses a *Just-In-Time* (JIT) compiler to generate machine code for the disparate instruction sets of these two core types on-demand. Threads running on either core type can invoke native methods, dynamically allocate memory, have it recovered by GC, and synchronize using shared-memory data structures (consistent with the Java Memory Model [10]), even though the hardware does not provide hardware cache coherency.

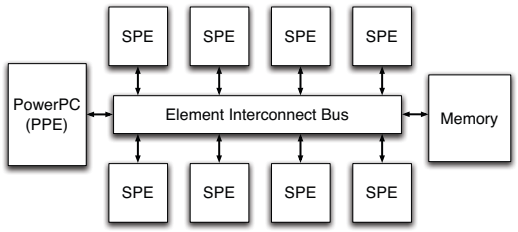
This paper builds upon the work presented in [13], but describes a much more complete runtime system that can support real-world Java applications as well as providing a much more thorough evaluation of this runtime system.

The main contributions of this work are:

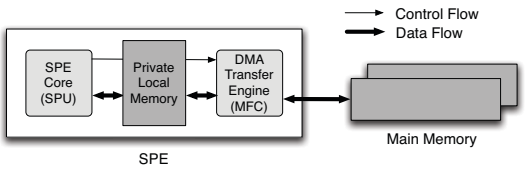
- The creation of the first JVM implementation to support execution of real-world Java applications across heterogeneous processing core types with different instruction sets architectures (ISAs) and provide transparent migration of threads between these core types.
- A software caching mechanism that provides efficient access to the non-coherent memory subsystem of the Cell processor by employing high-level type information embedded in Java bytecode.
- Demonstration of real-world Java workloads that exhibit up to a 2.25x speedup when executed on one of the Cell processor's SPE accelerator cores, compared to execution on its main PPE core, and up to a 13x speedup if scaled across all 6 SPE cores.

Section 2 introduces the Cell processor in more detail, outlining the main features of its architecture that make application development difficult. Section 3 presents the design principles around which Hera-JVM is based and discusses the problems that the Cell processor's unusual architecture presents in achieving these principles. Section 4 describes the implementation of these design principles in Hera-JVM for the Cell processor. Section 5 expands upon this implementation overview to provide more in-depth details of the features which are required for Hera-JVM to support real-world Java applications on the Cell processor. Hera-JVM's performance under both synthetic and real-world Java benchmarks is presented in Section 6. Section 7 contrasts Hera-JVM with relevant related work. Finally, Section 8 concludes and discusses possible future directions for this work.

[1] The only deviation from the Java Runtime Specification is that it uses a different floating point rounding mode (*rounding towards zero* instead of *rounding to nearest*). This is due to lack of hardware support on one of the Cell processor's cores types. It only affects the least significant bit of single precision flotation point calculations; the more commonly used double precision format is unaffected.




(a) The architecture of the Cell processor.




(b) An SPE core's memory subsystem.

**Figure 1.** The Cell Processor.

## 2. Background: The Cell Processor

The Cell processor [6, 9, 17] 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.

The Cell processor contains two different processing core types: a single *Power Processing Element* (PPE) core; and eight *Synergistic Processing Engine* (SPE) cores (Figure 1(a)). Both core types are dual issue, in-order architectures, running at 3.2 GHz, however, they have substantially different architectures. The PPE is a conventional 64-bit PowerPC-based core, supporting the Linux operating system and any applications compiled for the PowerPC architecture. The SPEs are designed to perform the bulk of the computation on the Cell processor. They have a unique instruction-set, highly tuned for floating point, data-parallel workloads. The SPEs do not run any operating system code, relying on the PPE to perform operations such as page table updates or file I/O.

The processing cores share access to external DRAM memory through a circular ring-based *Element Interconnect Bus* [2]. The PPE core has a two-level cache to reduce data access latencies, with a 64KB L1 cache (split evenly between data and instruction caches) and a 512KB L2 cache.

Unlike the PPE, the SPE cores do not have transparent hardware caches for accessing main memory; instead, each SPE contains 256KB of non-coherent, private, local memory. The processing elements of the SPEs can access only

this local memory directly. To access data in main memory, an SPE must initiate a Direct Memory Access (DMA) transfer to copy the data from main memory to its local memory. It can then modify this data in local memory, but must initiate another DMA transfer to write the results back into main memory. Each SPE has an associated DMA engine, called a Memory Flow Controller (MFC), that performs these data transfers (Figure 1(b)). These DMA engines have virtual memory support, therefore different threads of a single process share a consistent view of the process's virtual memory address space whether running on the SPE or PPE cores. However, data which has been copied to an SPE's local memory is not automatically kept consistent with the original copy in main memory or any copies made by other SPE cores, meaning cores do not automatically share a coherent view of data.

These features, of heterogeneous core types and an unusual memory architecture, make developing efficient, or even correct, applications for the Cell processor a challenge.

## 3. Two Architectures, One JVM

The aim of Hera-JVM is to abstract the Cell processor's challenging architectural features behind a more typical symmetric multi-core virtual machine abstraction, whilst still preserving the performance benefits provided by the Cell processor's heterogeneous cores. Hera-JVM provides a conventional JVM interface to applications: the runtime system can then schedule and migrate Java threads across the heterogeneous core types of the Cell processor in a manner that is completely transparent from the point of view of the application.

The two core types provided by the Cell processor have different instruction set architectures, and therefore require different compiled machine code to execute the same Java code. Hera-JVM exploits the fact that Java code is distributed in architecturally-neutral Java bytecode. This bytecode is just-in-time compiled to a particular core-type's machine code only if it is to be executed on that core type.

Hera-JVM allows transparent migration of Java threads between these different core types. Whenever a method is invoked, Hera-JVM can migrate the current thread's execution to another core type for the duration of this method and any methods which it calls. By only allowing migrations at method invocations, there is a well defined point at which the thread's execution transfers from one core type to another. This enables Hera-JVM to tailor a thread's execution for the core type on which it is executing, by for example, structuring stack-frames differently for each of the core types or performing inter-bytecode optimizations. Consequently, Hera-JVM can exploit heterogeneous cores without having to execute in a sub-optimal manner on one or more of the core types; however, this does mean that a thread must migrate back to its original core type once a migrated method returns, since the method to which it will return is part-way through its execution on the original core type.

To enable this form of seamless migration of threads between the different core types of the Cell processor, Hera-JVM supports the full Java specification on both core types. If an operation cannot be supported by a particular core type, this limitation is hidden from the application by automatically migrating the thread to a more capable core type for the duration of the operation. Since migration is a relatively expensive operation, Hera-JVM supports all of the core Java operations (e.g., arithmetic, method invocation, object allocation, thread synchronization and reflection) natively on both core types. Only heavy-weight operations, such as file opening and process creation, are limited to a particular core type.

Hera-JVM must also hide the fact that the SPE cores can only directly access their 256KB of private local memory, and must perform DMA transfers to access any data in main memory. An SPE core can only execute machine code that is resident in its local memory. Since most applications are likely to require more than 256KB of machine code, Hera-JVM provides an efficient mechanism to automatically *cache* code in an SPE core's local memory as it is required.

Similarly, data must reside in local memory before it can be accessed by a thread executing on an SPE core. Hera-JVM provides a software cache of recently accessed data in each SPE's local memory to limit the overhead of DMA transfers of data between main memory and local memory. However, since this local memory is private to the SPE core, any changes made to this cached data will not be visible to threads running on other cores. For Hera-JVM to support multi-threaded Java applications correctly, this software cache must conform to the Java Memory Model [10], by performing coherency operations at thread synchronization points.

Section 4 describes how these design decisions were implemented in Hera-JVM for the Cell processor. Section 5 expands upon this overview to provide more in-depth implementation details of features which are required for Hera-JVM to support real-world Java applications on the Cell processor.

## 4. Hera-JVM

Hera-JVM is based upon Version 3.0 of the JikesRVM [3] JVM. JikesRVM is a full implementation of the JVM with performance comparable to production JVMs (however, Hera-JVM only supports the slower, non-optimizing baseline compiler backend). It supports execution on the PowerPC processor architecture, and can therefore execute Java code on the PowerPC-based PPE core of the Cell processor without any modification. Hera-JVM extends JikesRVM in three main ways: (i) runtime system and compiler support for the SPE cores; (ii) changes to the overall runtime system to support simultaneous execution of a Java application across two different architectures; (iii) support for migration between the different core types of the Cell processor.

JikesRVM (and thus Hera-JVM) is a *Java in Java* virtual machine, with the majority of the runtime system written in Java. This confers a number of advantages in the design of Hera-JVM. Given Java's *write once, run anywhere* philoso-

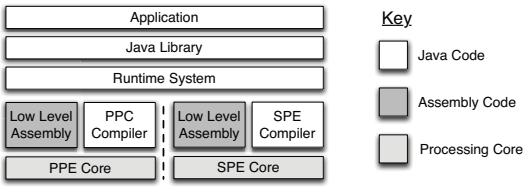



**Figure 2.** The structure of Hera-JVM. Much of Hera-JVM's runtime can be shared by both cores, given its Java in Java design.

phy, this code is largely portable. Thus, other than a small number of architecture-specific routines, the same runtime system code is shared by both core types (Figure 2). This approach extends the philosophy of hiding the architecture's heterogeneity right through application code, the Java Library code and the majority of the runtime system's code, simplifying the runtime system's design. This also improves the runtime system's maintainability; the fact that the same code is shared by both core types reduces the likelihood of introducing integration bugs and inconsistencies in shared data structures.

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

### 4.1 Compiling Java Bytecode for the SPE Cores

To execute Java code on the SPE cores of the Cell processor, Hera-JVM requires a Java bytecode to SPE machine code compiler and some low-level runtime system support code. The low-level runtime system support code is the only part of the Java runtime system which is kept permanently resident in the SPE's local memory (taking up less than 4KB of each SPE's 256KB of local memory). This low-level support code deals with caching of data and code, and the lowest levels of inter-thread synchronization and interrupt handling. The rest of the Hera-JVM runtime system is written in Java and can be cached into the SPE's local memory as required like any other Java method.

The remainder of this section describes the process by which this compiler and runtime system support code enables the SPE cores to execute Java code. A running example of a simple Java method "`sum()`", that calculates the total of all the elements in a linked list, will be used to illustrate

```
int sum(ListNode n) {
  int total = 0;
  while (n != null) {
    total += n.val;
    n = n.next();
  }
  return total;
}
```

(a) Java Code

```
0:  iconst_0
1:  istore_1
2:  aload_0
3:  ifnull <21>
6:  iload_1
7:  aload_0
8:  getfield <val>
11: iadd
12: istore_1
13: aload_0
14: invokevirt <next>
17: astore_0
18: goto <2>
21: iload_1
22: ireturn
```

(b) Resulting Bytecode

**Figure 3.** Example Java method - summing a linked list.

different aspects of this process. Figure 3 shows the source code for this method (left), and the resulting Java bytecode (right). Hera-JVM does not require any changes to the Java source-to-bytecode compiler or to the bytecode format.

A Java method, such as `sum()` in Figure 3, is compiled into a block of machine code that can be executed natively by an SPE core. Fundamental bytecodes, such as arithmetic and branch operations, can be translated directly into one or more SPE machine instructions by the compiler. More complex bytecodes, such as the `new` bytecode used for object allocation, are translated into *calls* to special runtime system entry points. These runtime system entry points are special Java methods that perform the required operation, then return execution to the original method. Since this runtime system code is shared by both the PPE and SPE cores, these complex bytecode operations can essentially be leveraged from the existing JikesRVM implementation. Similarly, complex runtime system components, such as file handling, class loading or thread scheduling, can be supported on either core type with little modification.

As a stack-oriented language, Java bytecodes implicitly operate on variables located on an *operand stack*. For example, the `iadd` bytecode in Figure 3 pops two integer values off the operand stack, adds them, and pushes the result back onto the operand stack. Since almost every bytecode pushes or pops values from the stack, it is important that these operations are efficient.

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

Stack overflow checks are required when a method is invoked to ensure that this method's stack frame will not cross the 16KB limit of the block held in SPE local memory. If the stack does grow beyond this limit, the stack overflow routine *pages* this block out to main memory, and *pages* in the next 16KB of the thread's stack.

Whilst accessing local memory on the SPE cores is much more efficient than accessing main memory, it is complicated by the SPE's unusual instruction set. SPE registers are 128 bits wide, with instructions treating these 128 bits as a vector of sixteen 8-bit, eight 16-bit, four 32-bit or two 64-bit values, depending upon the operation. Hera-JVM only ever uses the first vector slot, since Java has no in-built vectorization support. However, the SPE's instruction set requires that loads and stores from local memory are 128bit aligned. Therefore, either each stack variable must be 128bit aligned, wasting a considerable proportion of the valuable local memory, or stack push and pop operations must shuffle variables between the first vector slot and the variable's original alignment, making stack access inefficient. Hera-JVM uses the second approach, of shuffling variables, however, an optimization is employed to reduce its overhead. One of the SPE's registers is reserved to hold the 128-bits currently at the top of the stack. Variables are shuffled in and out of this register as required, but the values are only written to the local memory stack when the stack pointer passes a 128-bit boundary. This optimization reduces the overhead of pop operations from two machine instructions to one instruction, and the overhead of push operations from four instructions to two.

A Java method can also store method arguments and intermediate values in an array of variables known as *locals*. For example, in Figure 3, the `total` variable is stored in local number 1, and is accessed using the `iload_1` and `istore_1` bytecodes.

Hera-JVM exploits the large register file provided by the SPEs (each SPE core has 128 registers) to hold each local variable in its own register. If a method has too many local variables to fit in the available SPE registers, the additional local variables are spilled to the method's stack frame. The registers holding local variables must be non-volatile across method invocations, therefore code in each method's prologue saves the previous value of any local variable registers the method might overwrite and then restores these values when it returns.

### 4.2 Software Caching of Heap Objects

In addition to stack and local variables, which are private to a method, Java bytecode can also access data in a shared heap of object instances and arrays. In Figure 3 the `sum` method accesses the `val` field of a `ListNode` object in the heap, using the `getfield` bytecode.

Since the heap is shared between cores, it is located in main memory. Therefore, before accessing data from the heap (e.g., the `getfield` bytecode in Figure 3), an SPE core must perform a DMA transfer of the data it wishes to access from main memory into its private local memory. To reduce heap data access latencies and limit the number of DMA transfers required, Hera-JVM provides a software data cache for SPE heap access.

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 data cache was to transfer large blocks of memory wherever possible. However, to limit cache pollution, only data which is likely to be used in the future should be cached. The high-level type information preserved in Java bytecode provides the opportunity to meet these two conflicting demands.

The `getfield` and `setfield` bytecodes access a single field of an object. However, rather than caching just the field that is being accessed, or a fixed size block around that field, Hera-JVM exploits the fact that high level type information is encoded in these bytecodes to cache the full object instance. Subsequent accesses to any of the fields of this object instance will result in a cache hit, and can be read directly from the cached local memory copy without requiring further DMA transfers. This approach exploits *object-based* locality of reference - i.e., the thread is likely to access other fields in the same object instance.

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

When the SPE compiler encounters a bytecode that requires accesses to data in the heap (e.g., `getfield`, `iaload`, etc.) it generates *inline* machine code that checks if this access *hits* the local memory cache and then performs the operation on this cached copy. In the case of a cache miss, execution *traps* to a cache miss handler, which is part of the SPE's permanently resident runtime system support code. Thus, the fast path cache hit code is performed inline to reduce performance overheads, whilst the more complex, but less used cache miss code is kept out-of-line to reduce code bloat.

The data cache is structured as a small 1024-entry hashtable resident in the core's private memory (see Figure 4). Each entry is either blank, or holds the main memory address of an object instance or array block as a key, and the local memory address of its cached copy as a value. A cache lookup involves hashing the main memory address of the object or array block which is being accessed using a simple XOR folding hash (chosen due to its simplicity), which provides an index into this hash table. If the entry at this index is the same as the main memory address requested, this access has *hit* the cache and the bytecode operation is performed directly on the cached copy pointed to by this entry. A cache lookup that results in a cache hit requires

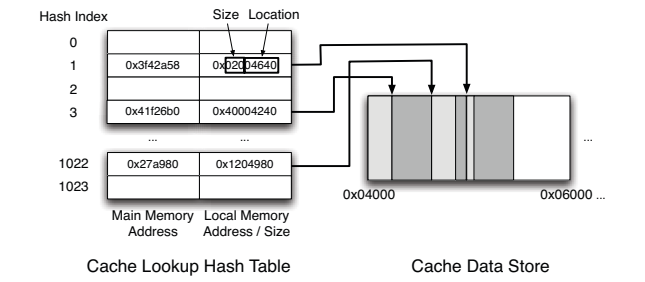



**Figure 4.** Outline of the SPE data cache.

only 12 fast SPE machine instructions. Otherwise, a cache miss has occurred and the data must first be pulled into local memory.

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

Operations which write to the heap (`setfield`, `iastore`, etc.) must update both the cached data in local memory and the original copy in main memory. Write operations update the data in the local memory cache directly, then immediately initiate a DMA transfer to copy the object field or array element which was modified to its original main memory location (i.e., this software cache uses a *write-through* policy). Unlike the cache read operations, the DMA transfers initiated by write operations are non-blocking; the thread can continue executing while the DMA engine performs this write to main memory concurrently. Threads do, however, block on these write operations before reading data from main memory to service a cache miss and when performing thread synchronization operations, to maintain memory consistency. The process of maintaining a coherent heap across multiple SPE's software caches, as required by the Java Memory Model [10], is discussed in Section 5.3.

### 4.3 Invocation and Caching of Methods

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

In keeping with Hera-JVM's approach of using DMA to transfer large blocks of data wherever possible, Java methods are cached in their entirety. When the SPE compiler encounters a method invocation bytecode, such as the `invokevirtual` bytecode which calls the `next()` method in Figure 3, this will be compiled into SPE machine code which checks if this method is cached in local memory and, if so, jump to the address of this cached copy. As with the data cache, a cache miss will result in the thread's execution *trapping* to a cache miss handler, located in the SPE's low-level runtime system support code.

Unlike the data cache, the code cache does not use a hash-table to perform look-ups. A hash-table is not suitable as a means of looking up a method because of the need to support virtual method invocation for Java instance methods.

When an instance (as opposed to a static) method is called, the actual method which is invoked depends upon the type of the instance object upon which this method was called. For example, if the object `n` in Figure 3 is a sub-class of ListNode, then `n.next()` must invoke the sub-classes implemention of the `next()` method, not ListNode's implementation. Therefore, the actual code that should be invoked by the `invokevirtual` bytecode is unknown at compile time; it must be inferred at runtime, based upon the object instance's type.

The standard method of supporting virtual method invocation in a JVM is to include a pointer to a *type information block* (TIB) in each object instance's header. A single TIB exists for every class loaded into the runtime system. Each TIB contains an entry for each method declared by the class, with each entry pointing to the code that implements the method. The TIB is laid out such that inherited methods are located at the same index in the sub-class's TIB as in the super-class's TIB. By looking up the index of the virtual method being invoked in the TIB of the object upon which it is being invoked, the runtime system can find the actual instance method which it should run for this virtual method invocation.

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

Hera-JVM exploits the fact that only a limited number of classes will be resolved for the SPE core to simplify TIB and method caching. A small (4KB) class *table of contents* (TOC) resides in SPE local memory, with an entry for each class that has been resolved for the SPE. Rather than a direct pointer to the SPE TIB's main memory address, each object instance has an index to its class's entry of this TOC in its header (see Figure 5). Each TOC entry initially points to the location of that class's SPE TIB in main memory. When a method is invoked on an SPE core, the appropriate class's TOC entry is read to locate the class's TIB, which is cached if necessary. When a TIB is cached, its TOC entry is updated to point to this local memory copy, thus, subsequent look-ups immediately know the location of the cached copy. The required method is then looked up in the TIB, and if necessary, cached in local memory, with the TIB entry being updated to point to the cached method's address.

An added benefit of this approach is that, while a direct pointer to a class's SPE TIB would require a full word to

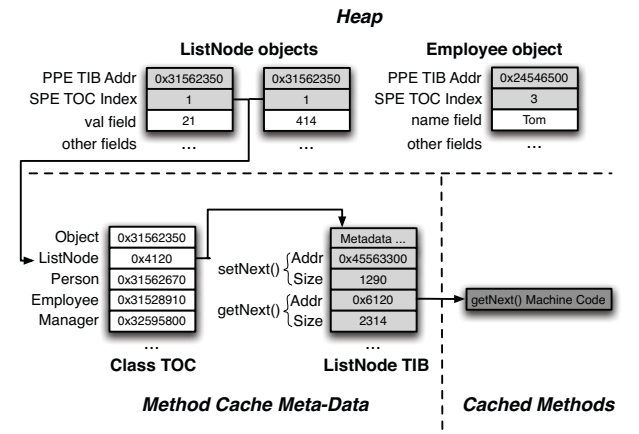



**Figure 5.** The code cache data structures.

specify, a class's TOC index only requires 10 bits in Hera-JVM. Therefore, it can be accommodated in spare bits of the object instance's header, rather than having to reserve an additional word in every object instance's header. The object instance headers do still contain a pointer to their PPE TIB, so that the PPE code can perform virtual method invocation in the usual manner; however, since the TOC index is hidden in spare bits of the header, object instances are the same size in Hera-JVM as they are in JikesRVM.

Static method invocations always invoke the same class's method (they are statically associated with the class, not a particular object instance). These method invocations are cached in the same manner as instance methods, the only difference being that the class TOC index is supplied statically by the compiler, rather than being read dynamically from an object instance's header at runtime.

## 5. Implementation Details

A number of complexities arise when real-world applications are executed on the Cell Processor under Hera-JVM. This section details some of the implementation choices made in the creation of Hera-JVM to deal with these challenges.

### 5.1 Variable Sized Cache Elements

Both the data and code software caching schemes used by the SPE cores are unusual in that the elements being cached are not of a fixed size. In order to service a cache miss, Hera-JVM's runtime system must be made aware of the size of the cache element which is to be cached, so that it can transfer it in its entirety the SPE's local memory. For object instance accesses, type information embedded in the `getField` and `setField` bytecodes enables the runtime system to infer, at compile time, the type, and thus the size of the object instance being accessed. Hera-JVM embeds this information into the machine code that it compiled for the SPE cores, such that the object's size can be passed automatically to the cache miss handling routing. Array accesses cache either a full block, of 1KB in size, or, if accessing the last block in the array, a block sized to fit the remainder of the array. The length of an array is held in its header. The cache miss handling code checks this length when caching a block to discover if this is the last block of the array and, if so, what size this last block should be. Finally, the size of a method's machine code is known once it has been compiled. This size is included in the class's TIB, next to the pointer to the method's machine code (see Figure 5), where it can be easily accessed by the code cache miss handler.

One issue with this approach relates to Java's subtyping inheritance abilities. An object access bytecode (e.g. `getfield` or `setfield`) has a particular type associated with the operation. However, the actual instance object accessed at runtime may be a subtype of the type associated with the bytecode. This subtype may have more fields than the supertype referred to by the bytecode, resulting in its instance objects being larger. Since the caching system uses the type associated with the bytecode to infer the size of the object being cached, it will not cache the full subtype object instance, only those fields associated with its supertype. This is not a problem for the execution of this specific bytecode, since it is accessing one of the fields associated with the supertype. However, subsequent accesses to this object instance will *hit* this cached copy directly. If they are trying to access one of the subtype fields, invalid data will result from reading from this cached copy, since it does not contain any of the subclass data. To avoid this, Hera-JVM stores the size of the data it has cached alongside the local memory address of the cached object in the data cache's hash table. Since the local memory has a small address space (18 bit address width), the object's size and cached local memory address can fit in a single 32 bit word entry of the hash table. When a cache hit occurs, the size of the cached data is compared to the expected size of the object type being accessed. If the cached data is not large enough, the object is re-cached.

### 5.2 Returning from a Method

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

Instead, Hera-JVM places a *return offset* on the stack when code running on an SPE core invokes another method. The value of this return offset is the distance of the invoking instruction from the start of the caller method. Alongside this return offset, the caller's stack-frame also contains a method ID, which specifies both the TOC and TIB indexes necessary to look up this method in the code cache. When the callee method returns, it ensures the caller method is

cached by performing the same look up process as if it were invoking the method, using the indices specified in the method ID in its caller's stack-frame. Adding the return offset on the caller's stack-frame to the start address of this cached method provides the callee method with an absolute return address, to which it can jump in order to resume execution of the caller method.

### 5.3 Synchronization and Coherence

In a multi-threaded Java application, the same object could be accessed by multiple threads simultaneously. Consequently, as well as residing in the main memory heap, multiple copies of this object could reside in different SPE local memory caches. In order to correctly execute multi-threaded Java applications, some form of synchronization is required to keep the data in these cached copies consistent with main memory. This is usually performed by a hardware cache coherence system, however, the Cell processor does not provide hardware cache coherency for SPE local memory, and providing a software coherence protocol which broadcasts every object update to all copies of the object's data would cripple the SPEs' performance. Fortunately, it is not necessary to keep all of the replicas of a given object consistent all the time. This is because the Java Memory Model [10] allows data to be cached in a non-coherent manner, as long as these inconsistencies are resolved before thread synchronization points.

Java's memory model is based upon a notion of happens-before relationships. Synchronization operations, such as lock acquire/release operations and volatile field accesses, impose a *happens-before* order on program execution. A read operation is not allowed to observe a write which happens after this read in the *happens-before* partial order (i.e., it should not see any writes which happen after a subsequent synchronization point). Similarly, the read can observe a write $w$, as long as there was no intervening write $w'$, where $w$ happened before $w'$ in the *happens-before* partial order (i.e., the thread does not see a value which was overwritten before the previous synchronization point).

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

In a traditional JVM implementation, this model requires that (for instance) writes to a field do not remain in a processor's registers when releasing a lock: the writes must be made back to memory and, on some architectures, a low-level memory fence operation must be performed before releasing the lock. In the case of the SPE core this means fully propagating all that thread's writes to main memory before releasing the lock, due to the lack of cache coherence on the SPE cores. This state is also flushed to main memory whenever a thread is context switched or migrated off of the SPE core, to ensure the changes it has made are visible to the core on which it will next be executed.

Hera-JVM ensures this by completely purging an SPE's data cache whenever the thread it is executing locks an object or reads a volatile field. Before unlocking an object or writing to a volatile field, the thread is blocked until all of its DMA write transfers have been completed.

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

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

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

By conforming to the Java Memory Model, any correctly synchronized multi-threaded application will exhibit sequentially consistent behaviour and run correctly under Hera-JVM. Hera-JVM sizes DMA write operations such that only the single field or data element being operated upon is overwritten. Therefore, even if neighbouring elements are protected by different locks, Hera-JVM will provide sequentially consistent behaviour. Finally, since all writes are performed in-order by code on the SPE cores, Hera-JVM conforms to the *no out of thin air values* property in the presence of data races, as required by the Java Memory Model.

### 5.4 Scheduling and Thread Switching

To support multi-threaded Java applications, Hera-JVM must schedule the execution of multiple Java threads onto each of the available processing cores. The version of

JikesRVM upon which Hera-JVM is based (version 3.0) uses a *green thread model* to schedule Java Threads. This model maps multiple Java threads to a single OS thread, with the runtime system performing user level scheduling of the Java threads, rather than the underlying operating system.

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

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

#### 5.4.1 SPE Virtual Processor Initialisation

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

The Java entry function performs some additional initialisation to set-up the SPE core's virtual processor data-structures. It then invokes the scheduling code to find a Java thread which it can execute. Initially, only a pre-built idle thread will be runnable on this SPE virtual processor. The idle thread does nothing other than yield, to enable the scheduling of a useful thread. After a given number of yields, it puts the core to sleep, by performing a blocking read on an inter-core signalling channel. To wake this core, another thread will send a signal through this channel to wake the core, after placing a thread in its run-queue or (in the case of the PPE core) migrating a thread to the SPE core.

#### 5.4.2 Scheduling Mechanism

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

When a virtual processor is making a scheduling decision, it checks whether it has more threads in its run queue than the other virtual processors. If so, it will perform load balancing by transferring some threads to another virtual processor's run-queue. This load balancing is only performed by virtual processors running on the same core type (i.e., SPE to SPE or PPE to PPE, but not SPE to PPE or PPE to SPE). The thread migration mechanism, described in Section 5.4.5, must be used to transfer a thread to a different core type.

#### 5.4.3 Context Switching Mechanism

The process of context switching a virtual processor's execution to a different Java thread is highly architecture dependent. The scheduling code calls a *magic* method to perform a context switch. This magic method is compiled directly into inline context switching machine code, specific to the core type to which it is being compiled.

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

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

#### 5.4.4 Timer Interrupts

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

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

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

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

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

#### 5.4.5 Migration between Core Types

Hera-JVM supports migration of Java threads between the PPE and SPE cores to enable an application to exploit both the core types available on the Cell processor. This migration process is transparent from the point of view of the application; no changes in application code are required to enable the application to be migrated between core types. The invocation of any Java method (other than those marked as `@Uninterruptable`) can act as a migration point. A migration can be triggered either dynamically by Hera-JVM's scheduler or explicitly by invocation of a method that is annotated with a `@RunOnSPECore` or a `@RunOnPPECore` annotation. The experiments presented in this work use explicit annotations to trigger migration; future work will examine automatic migration triggered by the scheduler guided by runtime monitoring of a program's behaviour.

If a method is to trigger a migration, it is invoked in the normal manner, however, code in its prologue causes the thread to trap to migration support code. The migration support code (executing on the original core type), will package the arguments of the migrating method and, if necessary, JIT compile this method for the core type to which the thread is being migrated. It then places the migrating thread on a per-core type migration queue when performing a context swap, rather than inserting it back into its own virtual processor's run queue. During scheduling operations, each virtual processor will periodically check the migration queue associated with its core type. Any threads it finds will be removed from the migration queue to be added to its own run queue, thus migrating the thread to a virtual processor running on the appropriate core type.

When a virtual processor pulls a thread off the migration queue, the migrating thread's current stack-frame is laid out for the other core type, from which it migrated. Therefore, before this thread is added to the virtual processor's run queue, a stack-frame for this core type is added to the end of the thread's stack. This synthetic stack-frame causes the thread to start executing at a migration entry-point method when it is scheduled. The migration entry-point method will unpack the parameters which were passed to the migrating method, then invoke the appropriate method using Java's reflection mechanism.

The thread continues to execute on this new core for the duration of this migrated method and the whole tree of methods which it calls (i.e., to migrate a thread for the entire duration of its execution, the thread's `run()` method can be migrated). Of course, a subsequent method invoked by this thread could cause it to migrate back to the previous core type using the same mechanism.

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

### 5.5 Stack Scanning

A number of runtime system processes must scan a thread's stack for information. For example, the garbage collector must scan every thread's stack for references to act as roots in its tracing algorithm. Similarly, exception handling code must also scan the stack to find the location of an appropriate catch block to handle a thrown exception. If a thread has been migrated to another core type, its stack will consist of a mix of PPE and SPE stack-frames, which could confuse such stack scanning code. For example, while the vast majority of the garbage collector stack scanning code is architecture neutral, the actual code which retrieves an object reference from a stack-frame is necessarily architecture dependent, since stack-frame layout varies between the core types.

The synthetic stack-frame, placed on a thread's stack as part of the migration process, acts as a marker to signal the transition from stack-frames of one core type to those of another. This enables these stack scanning algorithms to transition between PPE and SPE stack-frame scanning code as required. The garbage collector stack scanning code uses these markers to switch between PPE and SPE stack-frame

walkers. The exception handling code, on the other hand, is scanning the stack to find a suitable catch block in which to resume the thread's execution. Therefore, if it encounters a migration marker it immediately migrates the thread to the other core type, such that it is already on the correct core type on which to resume the thread's execution when it finds a suitable catch block.

### 5.6 System Calls and Native Methods

The final implementation consideration is to provide support for Java threads to call non-Java *native* code. Occasionally, a method in the runtime system, the Java Library or a Java application requires access to native code (e.g. to write to a file or start an external process). Application and library code can execute non-Java native code using the the JNI (Java Native Interface). JikesRVM / Hera-JVM also provides a fast system call mechanism for trusted code within the runtime system to perform native system calls.

However, if a thread is running on an SPE core, there is no underlying OS to support native methods. SPE cores must rely on the PPE core to execute native code. In the case of a JNI method, the thread is migrated to the PPE core for the duration of the native method, using the process described in Section 5.4.5. For fast system call methods, the SPE core uses an inter-core mailbox channel to signal a dedicated thread on the PPE core with an appropriate message. This dedicated thread performs the required system call on the SPE thread's behalf, then signals the SPE with the result.

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

## 6. Experimental Analysis

This section presents an experimental evaluation of Hera-JVM under both synthetic micro-benchmarks and real-world Java benchmarks. The aims of this evaluation are: to investigate the effectiveness of the mechanisms used to hide the Cell processor's heterogeneous architecture; to ensure that unmodified real-world Java applications can be executed correctly under Hera-JVM under the non-coherent memory subsystem of the Cell Processor; and to characterise the performance of each of the core types under different application behaviours.

### 6.1 Experimental Setup

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

Hera-JVM currently supports only non-optimizing compilation for the SPE cores, therefore, the baseline, non-optimizing compiler was used to compile both PPE and SPE machine code for these experiments[2]. Hera-JVM was built with a stop-the-world, mark and sweep garbage collector. This collector only runs on the PPE core and thus becomes a scalability limitation if it runs for a considerable proportion of a benchmark. There is no fundamental reason the garbage collector cannot also execute on the SPE cores (it is written in Java like the rest of the runtime system), however, this support was not implemented in Hera-JVM for time reasons.

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

### 6.2 Micro-Benchmarks

The micro-benchmarks provided by the Java Grande benchmark suite [11] were employed to characterise the performance of the various fundamental Java operations on both core types under Hera-JVM[3].

Figure 6 shows the difference in performance between the core types for each of the micro-benchmarks included in *section one* of the Java Grande Suite. There is clearly a wide variation in capability between the PPE and SPE cores depending upon the type of Java operation being performed.

Basic operations, such as arithmetic, primitive casting and looping code, perform much better on the SPE core than on the PPE core. Some of these operations, such as floating point arithmetic and casting operations, are more than five times faster on the SPE core. This was expected, given that the SPE is highly tuned for floating point performance, however, even integer operations are significantly faster on the SPE core. The fact that looping code performs better on the SPE core, compared to the PPE core, was surprising. The PPE core has branch prediction hardware that is not found in the SPE cores. This should reduce pipeline stalls on the PPE, thus increasing the performance of looping code. The

[2] While the use of an optimizing compiler would significantly change the absolute performance of the benchmarks presented in this section (using the optimizing PowerPC backend of JikesRVM can yield up to an 8x improvement in performance for some of the benchmarks on the PPE core), we do not believe that it would significantly change the relative difference in performance between the SPE and PPE cores. An initial investigation, where three of the methods in a mandelbrot benchmark were manually inlined, showed that this optimization could improved the performance of code running on both the PPE and SPE cores by a similar margin (2.5x and 3.1x respectivly).

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

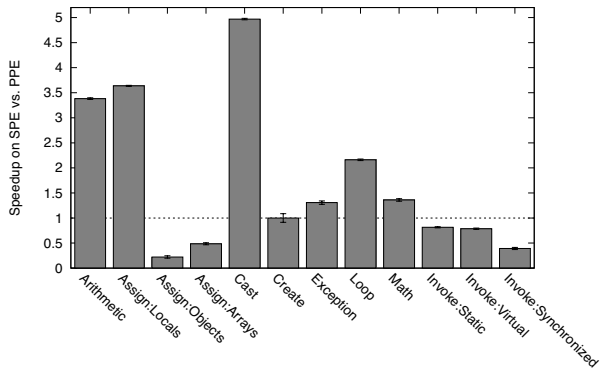



**Figure 6.** Relative performance of SPE and PPE cores under the different Java Grande micro-benchmarks.

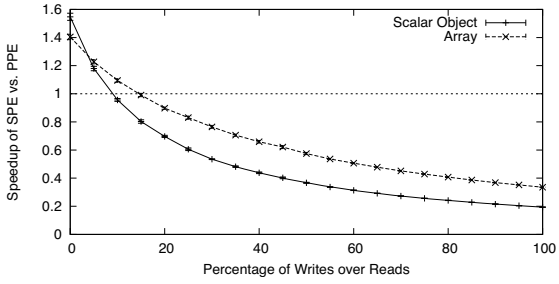



**Figure 7.** Performance as ratio of reads to writes is varied.

fact that the loop benchmark performs worse on the PPE core may be explained by the shorter pipeline in the SPE core, which reduces the impact on performance incurred by pipeline stalls.

More complex operations, such as object creation, exception handling, mathematical calculations and method invocation have roughly equivalent performance on both core types. The benchmarks that perform worst on the SPE core type are those which exercise the SPE's software data and code caches. Access to local variables (e.g. method parameters or variables on the thread's stack) is very fast on the SPE cores; however, accessing scalar objects or arrays on the heap is considerably slower due to the costs involved in setting up DMA transfers from main memory. Invocation of synchronized methods also has a large overhead on the SPE core, due to the SPE core's software data cache having to be purged before entering the synchronized method. This has a high cost on the SPE core for two reasons: it will cause cache misses for future heap accesses, which are much more expensive on the SPE core than the PPE core; and the software cache on the SPE must manually purge the cache by overwriting entries in the cache's hash-table, whereas the PPE does this in hardware.

### 6.2.1 Writing to the Heap

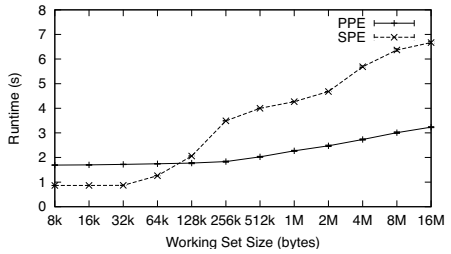
The tests in the assign benchmark of the Java Grande Suite read from, and write to, memory in equal measure. To investigate whether reading from objects and arrays on the SPE is equally as costly as writing to them, the benchmark was modified so that the ratio of reads to writes could be varied. Figure 7 shows the difference in performance between the SPE and PPE cores, as this ratio is varied. This benchmark has a small enough working set that reads always hit the cache, thus it only exercises the fast-path of the software cache. At small write ratios, the SPE core actually outperforms the PPE. The SPE is almost 55% faster than the PPE when reading from scalar objects and 40% faster when reading from array elements. Thus, even though the SPE must perform a software cache look-up operation for each object or array access, the SPE core's simple design and the software cache's lightweight implementation make these accesses faster than the hardware cache on the PPE core.

However, the SPE's performance falls significantly as the write ratio increases. Above a write ratio of between 10% and 15%, the PPE core's performance outstrips that of the SPE. Writes are expensive on the SPE core because: a write-through policy is used, meaning every write must be propagated to main memory; and each write to main memory requires a DMA transfer, which is relatively expensive to set up. Informal experiments show that the use of a write-back caching policy to enable batching of DMA transfers significantly improve write performance on the SPE core, however, implementing this write-back policy significantly complicates the design of the cache. A complete implementation of write-back caching is left as future work.

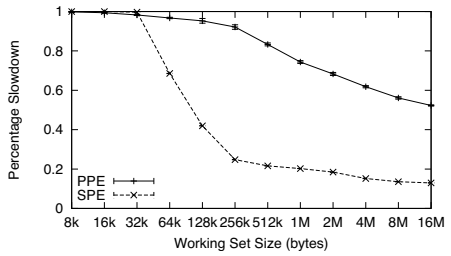
### 6.2.2 Data Caching Overheads

Each of the micro-benchmarks presented above has a small enough working set that the data it accesses always fits in the cache. To investigate the overhead of data caching, a micro-benchmark was devised in which the size of the program's data working set could be varied. This benchmark reads from, and writes to, randomly selected elements of an array. The size of the array can be varied to alter the program's working set size and affect its cache hit rate. This benchmark represents the worst case in performance for a particular working set size, since access is entirely random and no real work is done between heap accesses. Figure 8 shows the performance of the SPE and PPE cores for this benchmark.

The SPE's performance initially surpasses that of the PPE. However, as expected, cache misses on the SPE core are more expensive than on the PPE core, due to the caching being performed in software, rather than being under hardware control. Once the size of the working set grows larger than the amount of local memory reserved for the SPE's data cache (96KB), its performance degrades severely. The PPE core has a larger data cache (256KB in its L2 cache). Its performance does suffer after the working set size increases above this cache size, however, not so severely as on the SPE core. For the maximum working set size of 16MB, the overhead due to cache misses reduces the SPE core's performance to about an eighth of its original value, while the PPE core's performance drops by a half.

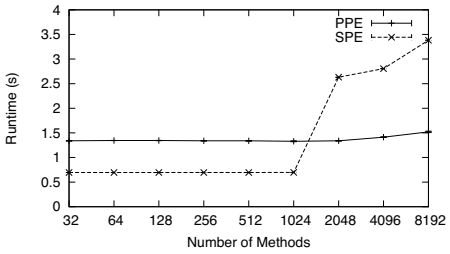



(a) Absolute performance.

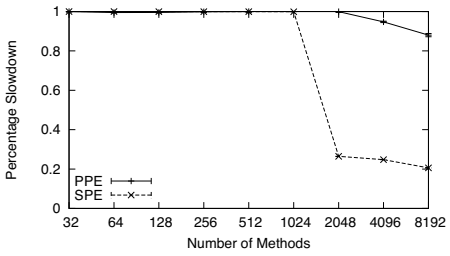



(b) Slowdown relative to 8k working set.

**Figure 8.** The effect of a thread's data working set on performance.




(a) Absolute performance.




(b) Slowdown relative to 32 methods.

**Figure 9.** The effect of a thread's code working set on performance.

### 6.2.3 Code Caching Overheads

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

Figure 9 shows the performance of this benchmark on both the SPE and PPE cores. Once the working set of methods that this benchmark invokes grows beyond 1024, it can no longer fit in the SPE's local memory cache. Performance on the SPE core drops to about one fifth of its original value, since almost every method invocation will need to re-cache the method's code, as it is likely to have been evicted since the method was last called. The benchmark's performance does not suffer as severely on the PPE core, again due to its dedicated caching hardware.

## 6.3 Real World Benchmarks

In this section, a selection of benchmarks from three real world benchmark suites are used to evaluate Hera-JVM in a realistic setting. To provide a range of applications with different types of behaviour, benchmarks were selected from: SpecJVM 2008 [21], a suite which mimics a variety of general purpose applications; the Java Grande Parallel benchmark suite [22], which aims to replicate high performance computing workloads, such as scientific, engineering or financial applications; and the Dacapo 2006 benchmark suite [5], which focuses on memory hungry benchmarks. The following benchmarks were run under Hera-JVM:

**mandelbrot** generates an 800x600 pixel image of the mandelbrot set (this benchmark is not part of any of the benchmark suites).

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

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

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

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

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

**Spec: monte_carlo** approximates the value of Pi by computing the integral of the quarter circle y = $\sqrt{1-x^2}$.

**Spec: sor** simulates the Jacobi successive over-relaxation algorithm for a 250x250 grid data set.

**Spec: sparse** performs matrix multiplication on an unstructured sparse matrix in compressed-row format with a prescribed sparsity structure.

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

**Spec: mpegaudio** decodes six MP3 files which range in size from 20KB to 3MB.

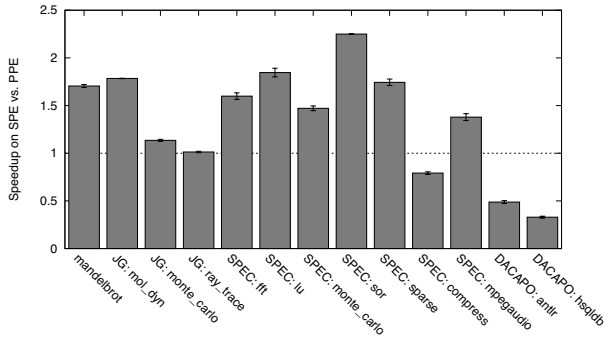



**Figure 10.** Performance comparison between benchmarks running on a single SPE core, and running on the single PPE core.

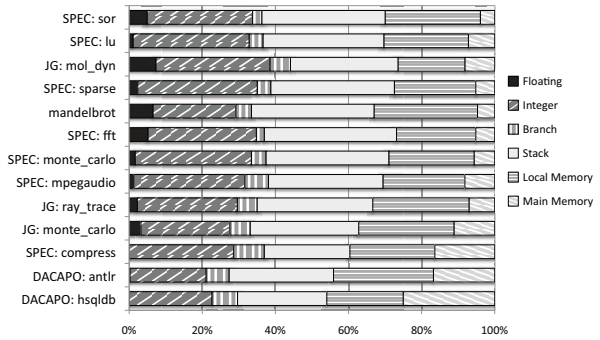



**Figure 11.** Percentage of cycles spent executing different classes of machine instructions on SPE. Benchmarks are ordered upwards by increasing SPE performance.

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

**Dacapo: hsqldb** uses JDBC to invoke an in-memory, SQL relational database, modelling a banking application.

In future work, Hera-JVM will be augmented so that it can automatically choose the most appropriate core type on which to execute blocks of code, based upon program behaviour; however, before doing so, it is necessary to uncover the relative performance of each core type under different workloads. Therefore, in the following experiments, the timed portion of each of these benchmarks was executed on either the PPE core or the SPE core in its entirety (other than forced migrations due to invocation of native code), to compare their relative performance.

The only modification required to execute these benchmarks on the SPE cores was to split a small number of exceptionally long methods into multiple smaller methods, so that they could fit in the SPE's code cache in their entirety. Developing a code caching scheme which splits a method into multiple cacheable blocks would remove the need for these modifications.

### 6.3.1 Single Threaded Performance

Figure 10 shows the difference in the performance of these benchmarks when they are run on a single SPE core, versus the PPE core. The error bars represent the standard deviation between ten runs on each core type. There is a wide variation in the performance of the benchmarks between core types, from a 2.25x increase in SPEC: sor on the SPE core, to a 3x slowdown for DACAPO: hsqldb.

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

To further investigate how a program's behaviour affects its performance on the different core types, a simulator was used to calculate the percentage of time the SPE core spends executing different classes of machine instructions. Figure 11 shows this breakout by instruction type for each benchmark. The benchmarks are ordered by their performance on the SPE core, relative to the PPE core, with the best performing benchmark at the top.

Benchmarks which perform well on the SPE core also generally spend more of their time executing floating point or integer-based calculations. Benchmarks which spend a large proportion of time accessing data elements in the heap (the local memory and main memory categories) perform more poorly on the SPE core than the PPE core. This is especially prominent when those accesses result in cache misses or write operations which require DMA operations to main memory.

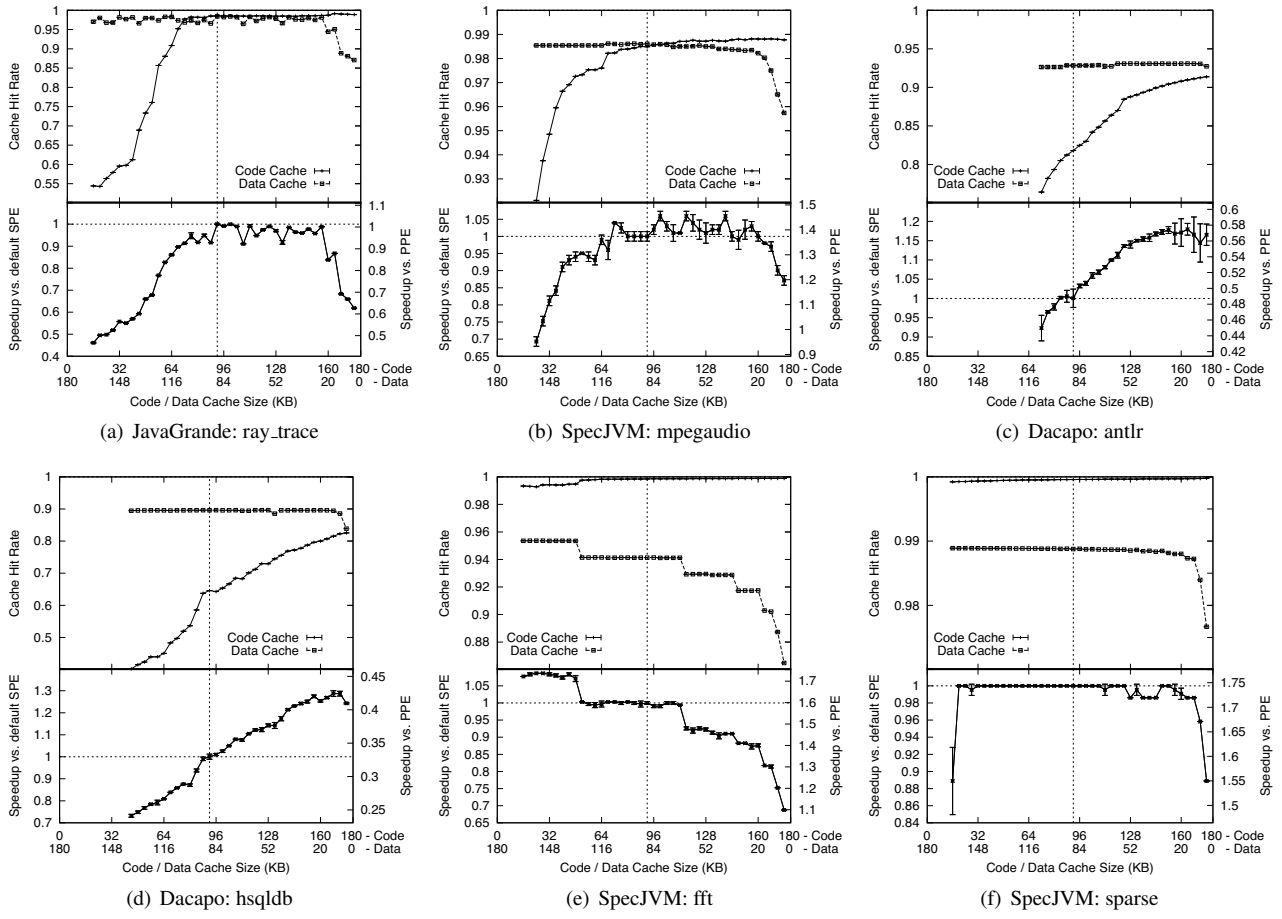
### 6.3.2 The Effect of Cache Size

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

Figure 12 show how the performance is affected as this ratio is altered for an interesting subset of the benchmarks (results for all benchmarks are presented in [12]). The effect of cache size on hit rate for both data and code accesses is also shown in these figures. The cross formed by the dotted lines in these figures shows the performance at the default cache sizes.

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

**Peaked:** The ray_trace and mpegaudio benchmarks show a peak in performance, when roughly an equal percentage of memory is reserved for each cache. These benchmarks are equally affected by small code or data caches. How-




(a) JavaGrande: ray_trace (b) SpecJVM: mpegaudio (c) Dacapo: antlr

(d) Dacapo: hsqldb (e) SpecJVM: fft (f) SpecJVM: sparse

**Figure 12.** The effect of varying the proportion of local memory reserved for use by the data and code caches.

ever, the plateau shape of these two graphs suggests that the working set of both code and data for both benchmarks fits comfortably in the local memory provided by the SPE core.

**Rising:** The performance of the JavaGrande: monte_carlo and Dacapo benchmarks rises as the proportion of memory provided to cached code increases. The working set of code required by these benchmarks is clearly too large to completely fit into the SPE's local memory, leading to this behaviour.

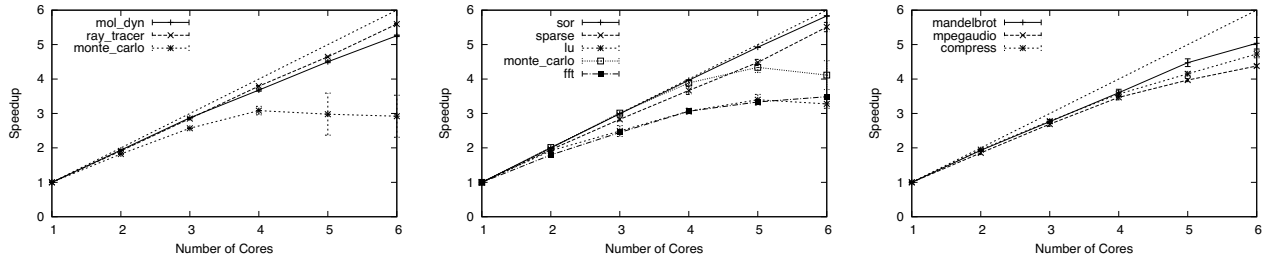
**Falling:** The performance of the mol_dyn, fft, lu and compress benchmarks are more heavily affected by the size of the data cache. These benchmarks would seem to benefit from an even larger data cache size than can be provided by the SPE's local memory.

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

Given the different behaviours of these benchmarks, it is clear that no fixed segregation of the code and data caches will provide the best performance for all applications. One possible solution to this would be to mix code and data in a single larger cache. The problem with this approach is that the data cache must be purged on thread synchronization operations, whereas the code cache need not. With a single shared cache, either the code must be needlessly purged alongside data, or a more complex cache allocation and purging scheme must be employed. Another approach would be to provide Hera-JVM with the capability to dynamically alter the code / data cache ratio, based upon runtime monitoring of a program's cache hit rates. The provision of such a system for Hera-JVM is left for future work; the default code / data cache ratio of 88KB / 92KB is used in all subsequent experiments.

### 6.3.3 Scalability

The preceding experiments evaluated the performance of the benchmarks when run on a single SPE core compared with single PPE core. However, the Cell processor contains eight SPE cores and can provide significantly more computing power if an application can be parallelised. Other




(a) Java Grande Parallel benchmarks (b) SpecJVM scimark benchmarks (c) Remaining benchmarks

**Figure 13.** Scalability of benchmarks running on SPE cores.

than the two Dacapo benchmarks, these benchmarks have all been developed with scalability in mind. The SpecJVM benchmarks scale by running multiple instances of the same benchmark simultaneously, and therefore only evaluate the scalability of the runtime system itself. However, the Mandlebrot and JavaGrande benchmarks scale by having multiple threads coordinate on a single instance of the benchmark, therefore these benchmarks also evaluate interaction and synchronization between application threads running under Hera-JVM.

The Cell processor in the Playstation 3 used for these experiments only provides six SPE cores for user applications (one core is disabled, due to manufacturing defects, and the other runs a secure hypervisor). Figure 13 shows the speedup obtained by each of the benchmarks as they are scaled from one to six SPE cores.

Most of the benchmarks scale well as the number of SPE cores increase. However, the lu, fft and both monte_carlo benchmarks stop scaling after four cores. The reason these benchmarks stop scaling is due to garbage collection. These benchmarks allocate a large amount of data during their execution. Since the garbage collector currently runs on only the PPE core, this leads to a scaling bottleneck.

Figure 14 provides an overview of the performance of running each benchmark on all six SPE cores, compared with running on the single PPE core. For those benchmarks which scale, running on all six SPE cores provides from a 3x to a 13x speedup, compared to running on the single PPE core.

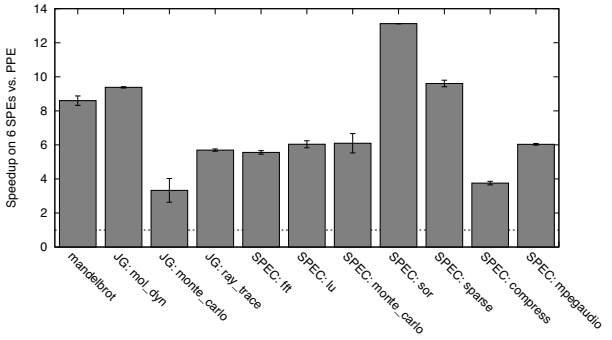



**Figure 14.** Performance of each benchmark running on all 6 SPE cores compared to a single PPE core.

## 7. Related Work

The abstraction of heterogeneous processing resources has been examined by a large body of related work over the years. One of the most widely available types of heterogeneous processing resource is the *Graphics Processing Unit* (GPU) present in most commodity computer systems. As GPUs have become more capable, a number of systems, such as Cuda [18], Sieve C++ [7] and OpenCL [14], have been developed to enable general purpose applications to exploit a GPUs potential processing performance. While these systems abstract many of the difficulties involved in writing general purpose code for GPUs, they are relatively inflexible; a program's threads of execution cannot easily be migrated between the CPU and GPU cores as workload or resource requirements change. Data transfers must also, typically, be controlled explicitly by the program.

A number of projects have investigated techniques to aid programing of the Cell processor. The Cell Superscalar (CellSs) framework [16] enables the developer to identify blocks of code that would benefit from execution on the SPE cores and then automatically schedules task blocks on the appropriate core type, by employing a task dependency graph. However, the developer must decide ahead of time which core type to use for a given piece of code, reducing flexibility in moving a thread's execution to a different core type dynamically at runtime as workload requirements change. The developer is also expected to have in-depth knowledge of the Cell processor's unusual architecture to exploit it effectively, reducing the appeal of this model for mainstream developers.

The CellVM [15] takes a similar approach to Hera-JVM, by supporting the execution of applications written in Java on both of the Cell processor's core types. However, Hera-JVM provides a number of features which are not found in CellVM: in CellVM each Java thread is bound to a single SPE core and cannot be transparently migrated between core types as in Hera-JVM; CellVM is structured as two different runtime systems (one for each core type), complicating maintenance and integration; CellVM's SPE core type runtime system supports only a limited subset of the Java language specification, relying on the PPE core to perform

operations such as thread synchronization and object allocation, which limits scalability; and CellVM does not provide the coherence guarantees that are required by the Java memory model. These limitations mean that CellVM does not support significant real-world Java applications and can only run parallel applications where the threads do not share access to data objects.

Some proposed multi-core architectures provide cores that have symmetric instruction set architectures but are asymmetric in their performance. Saez et al. [19] describe the CAMP scheduler, which uses a utility function to optimize system-wide performance, by placing CPU intensive workloads on *fast* cores and memory intensive workloads on the *slower* cores.

Other work has investigated similar approaches to those used in Hera-JVM, but for different reasons. Cashmere [23] 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 many-core runtime McRT [20] currently targets 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.

## 8. Discussion and Future Work

The Cell processor is a challenging architecture upon which to develop general purpose software. However, by abstracting the details of this architecture behind a JVM, Hera-JVM hides the heterogeneous nature of the Cell processor and the unusual features of the SPE core. This enables developers to write application code for this challenging architecture in the same manner as they would for more conventional architectures, while still gaining a performance benefit from the architectures heterogeneity.

Of course, this abstraction does not come for free. The wide impedance mismatch between a JVM abstraction and the SPE core's architecture means that applications running under Hera-JVM cannot take take full advantage of the SPE core's potential performance. However, the techniques described in this work, such as efficient stack management and software caching using high level type information, enable the SPE core to provide better performance than the PPE core for the majority of the benchmarks with which Hera-JVM was evaluated. Given that Hera-JVM is targeting mainstream application developers, who would otherwise be unlikely to exploit the Cell processor's SPE cores at all, the cost of providing this abstraction would appear to be justified.

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

Finally, the ability of Hera-JVM to transparently migrate a thread between the PPE and SPE core types provides the runtime system with the flexibility to fully utilise the heterogeneous core types of the Cell processor under varying conditions and workloads. Future work will investigate techniques which enable the runtime system to automatically select the most appropriate core type on which to schedule different threads and phases of a given application's execution, based upon runtime monitoring of the application's execution behaviour.

## Acknowledgments

This work was funded by the Carnegie Trust for the Universities of Scotland. We would also like to thank Microsoft Research for providing funding for some of the equipment used in this work. Many thanks to Tim Harris, Simon Peyton Jones and Rachel Lo for their invaluable advice and feedback on early drafts of this paper, which greatly improved the presentation of this work.

## References

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

[2] T. Ainsworth and T. Pinkston. Characterizing the Cell EIB On-Chip Network. *IEEE Micro*, 27(5):6–14, 2007.

[3] B. Alpern, S. Augart, S. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, et al. The Jikes Research Virtual Machine project: building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005.

[4] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference*, pages 483–485, 1967.

[5] S. Blackburn, R. Garner, C. Hoffmann, A. Khang, K. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Guyer, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*, pages 169–190, 2006.

[6] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell Broadband Engine Architecture and its First Implementation: A Performance View. *IBM Journal of Research and Development*, 51(5):559–572, 2007.

[7] A. Donaldson, C. Riley, A. Lokhmotov, and A. Cook. Auto-parallelisation of Sieve C++ programs. *Lecture Notes in Computer Science*, 4854:18, 2008.

[8] M. Hill and M. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, 2008.

[9] H. Hofstee. Power efficient processor architecture and the cell processor. *11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, pages 258–262, 2005.

[10] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *Proceedings of the 32nd Symposium on Principles of Programming Languages (POPL'05)*, pages 378–391, 2005.

[11] J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and development of Java Grande benchmarks. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 72–80. ACM, 1999.

[12] R. McIlroy. *Using Program Behaviour to Exploit Heterogeneous Multi-Core Processors*. PhD thesis, Department of Computing Science, The University of Glasgow, 2010.

[13] R. McIlroy and J. Sventek. Hera-JVM: Abstracting Processor Heterogeneity Behind a Virtual Machine. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2009.

[14] A. Munshi. The OpenCL Specification. *Khronos OpenCL Working Group*, 2009.

[15] A. Noll, A. Gal, and M. Franz. CellVM: A Homogeneous Virtual Machine Runtime System for a Heterogeneous Single-Chip Multiprocessor. In *Workshop on Cell Systems and Applications*, June 2008.

[16] J. Perez, P. Bellens, R. Badia, and J. Labarta. CellSs: Making it easier to program the Cell Broadband Engine processor. *IBM Journal of Research and Development*, 51(5):593–604, 2007.

[17] D. Pham, S. Asano, M. Bolliger, M. Day, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, et al. The design and implementation of a first-generation CELL processor. *IEEE Solid-State Circuits Conference*, 2005.

[18] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, and W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, pages 73–82, 2008.

[19] J. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A Comprehensive Scheduler for Asymmetric Multicore Processors. In *Proceedings of EuroSys'10*, 2010.

[20] B. Saha, A. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, et al. Enabling scalability and performance in a large scale CMP environment. In *Proceedings of EuroSys'07*, pages 73–86, 2007.

[21] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. SPECjvm2008 Performance Characterization. In *Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, pages 17–35. Springer, 2009.

[22] L. Smith, J. Bull, and J. Obdrizalek. A parallel Java Grande benchmark suite. In *Proceedings of the Conference on Supercomputing (SC'01)*, 2001.

[23] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: software coherent shared memory on a clustered remote-write network. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP'97)*, pages 170–183, 1997.