

CellVM: A Homogeneous Virtual Machine Runtime System for a Heterogeneous Single-Chip Multiprocessor

Albert Noll
ETH Zurich
albert.noll@inf.ethz.ch

Andreas Gal
University of California, Irvine
gal@uci.edu

Michael Franz
University of California, Irvine
franz@uci.edu

Abstract

The Cell Broadband Engine Architecture (Cell) is a hardware platform for high performance parallel computing. Due to its architectural features and programming model, efficiently programming the Cell processor implies a detailed knowledge of the underlying hardware architecture, carefully designed communication protocols to minimize the synchronization and communication overhead between the individual processing units as well as an elaborate layout of the vector processor's local stores.

In order to take the burden from the programmer of having to deal with such low-level architecture-specific details, the Cell compiler toolchain implements the OpenMP standard [27], which allows for seamless parallelization of applications by offloading workloads to the SPEs.

In this paper we report on the design and implementation of a middleware for the Cell processor that goes one step further and completely abstracts the heterogeneity of the underlying hardware architecture and allows programmers to employ higher-level and more intuitive programming constructs than the OpenMP approach. In particular, we implemented a virtual machine (VM) that mimics the behavior of a homogeneous, shared-memory multiprocessor. Internally, our VM schedules individual threads to be executed on the vector cores which offers the same parallelism as compared to the traditional programming models of the Cell processor.

1 Introduction

Unlike the traditional processors that are commonly used in the desktop and server market, the Cell [19, 16] implements an asymmetric architecture. It consists of one general-purpose processor core (PPE) and eight identical throughput-orientated vector cores (SPE), implementing an entirely different instruction set that is geared towards computation-rich applications.

Integrating different cores onto a single chip in this manner makes sense from a hardware designer's perspective. For the intended application domain, having a number of throughput-orientated processor cores is far more efficient than trying to do the same work with general purpose cores, which would also take up more space on the die. From a software designer's perspective, on the other hand, dealing with asymmetric architectures and multiple instruction sets makes programming considerably more difficult.

The goal of our work is to provide the best of both worlds, combining the benefits of a *symmetric* multiprocessor system (which is simpler for programmers) on top of an *asymmetric* multi-processor system (which hardware architects can build more efficiently and cheaply). Our solution consists of a virtual machine layer that sits on top of the heterogeneous hardware and automatically distributes work to the different processing cores.

In particular, our virtual machine presents an interface to applications that mimics the behavior of a homogeneous, shared-memory multiprocessor. The homogeneous interface allows for a simplified programming model at a high level of abstraction, which has several benefits such as improved programmer productivity and (potentially) improved software reliability. E.g., the Java Programming Language [5] allows software development at such a high-level of abstraction, leaving lower-level implementations such as dynamic memory allocation and garbage collection to the runtime system. The disadvantage of a more abstract software development, however, is that implementations tend to incur a runtime overhead as compared to a lower-level implementation.

We have implemented a prototype of such a system that we call *CellVM*. To the application, CellVM presents the interface of a standard Java Virtual Machine (Java VM). Internally, CellVM executes Java VM instructions by co-execution between the different functional units contained on the Cell microprocessor. CellVM supports two modes of execution: a purely interpretative approach and a dynamic Java byte-code to native code compiler that translates Java code to native vector code on-the-fly.

Of particular note is our solution to the data latency problem. The vector cores in the Cell architecture do not have a hardware-based cache hierarchy. Instead, each vector core provides a software-controlled scratch pad memory in conjunction with DMA capabilities. Key to a streamlined division of labor in our cooperative execution environment is an efficient use of these scratch pads as a distributed data cache. Despite relying on a software-only cache approach, profiling shows surprisingly high hit rates above 90% for the instruction and data cache.

To summarize, the contributions of CellVM are:

- introduction of a hardware abstraction layer (HAL) that presents a homogeneous interface to the application and hence allows for software development at a high level of abstraction.
- incorporating the SPEs into the Java virtual machine to execute Java code, which current VM implementations for the Cell architecture are unable to do.
- an automatic memory management system for the SPE's local store.

The remainder of this paper is organized as follows. In Section 2 we discuss the design considerations on which we based our implementation that is described in Section 3. In Section 4, we describe the software-driven caching mechanisms that we use on the throughput-orientated processor cores. To evaluate our approach, we measured a series of benchmarks that we discuss in Section 5, followed by related work in Section 6. We discuss the benefits and drawbacks of our approach in Section 7.

2 Design Considerations

Existing Java VMs for the Cell architecture are unable to incorporate the SPEs to process Java instructions since they are not designed to operate in a heterogeneous, distributed-memory environment. Therefore, all available SPEs that provide the major computational capabilities remain idle. Incorporating the SPEs into the Java VM and using them to process Java instructions requires a significant change in the Java VM implementation (see Figure 1). In essence, our design was driven by three Cell architecture-specific features:

First, accessing the main memory from a SPE can only happen by using its DMA engine. As a consequence, accessing the Java heap (which is shared among all Java threads and hence maintained in main memory) from a SPE requires a DMA transfer that reads from the specified address in main memory and puts the data in the local store. In particular, every access to a non thread-local data (see Section 3.3) has to be implemented by a corresponding DMA command.

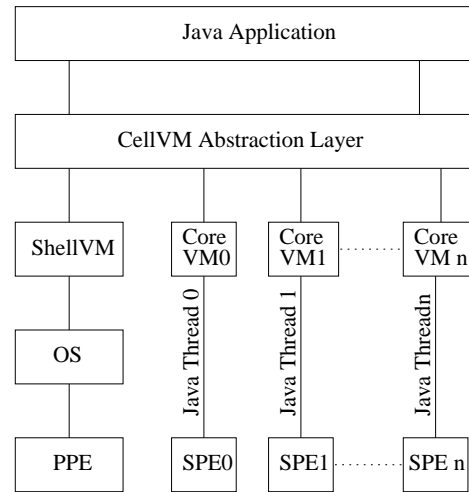


Figure 1. In order to enable the processing of Java instructions on the SPEs, the Java VM is divided into two cooperating VM implementations: While the Shell-VM operates on the PPE and primarily maintains global system resources, each SPE is equipped with a Core-VM instance, which represent the virtual Java code execution units.

A further consequence of the restricted access capabilities to main memory is that a small subset of Java opcodes cannot be implemented efficiently on the vector cores. Hence, our design restricts the SPEs to only process Java opcodes, which either require access to thread-local data structures (see Section 3.3) or require “simple” main memory interaction. Simple interactions with main memory are constricted to be read or write accesses. “Complex” memory interactions, however, are released to be processed by the PPE. A complex memory interaction is simulated by a switch of control from the executing SPE to the PPE (see Section 3.4). Complex interactions include the resolving of references and dynamic memory management which is used for allocating objects on the Java heap, for instance. Furthermore having a centralized resource manager located at the PPE can reduce expensive synchronization overhead between SPEs significantly.

Second, each SPE maintains its own, private local store that can be regarded as a software controlled cache. To mask the latency of the mandatory DMA commands, our approach implements an automated software-controlled memory management system for the SPE's local store. The main purpose of the memory management system is to copy instructions and data to the software-managed caches.

Finally, the size of the local store is limited to 256KB. As the local store is used for both, holding the SPE binary

(the Core-VM instance) and the data that are processed, a major design goal was to keep the SPE binary as small as possible, since the SPE binary subtracts from the available memory that can be used for the caching of Java instruction and data. For this reason, we decided to offload a subset of Java VM functionality (mainly complex interactions) from the SPE to the PPE.

While the offloading of functionality to the PPE seems to be an inherent performance-limiting factor of our approach (transfer of control from a SPE to the PPE is an expensive operation), Kazi et al. [21] show that the most frequent Java instructions are local variable loads (30%–47%), memory stores (11%–17%) and arithmetic operations (6%–23%), measured for several common benchmark programs. All of the instructions mentioned above can be executed local on the SPE. In fact, only an insignificant portion of the processed instructions is handled by the PPE (see Section 5.2). In particular, all opcodes that create new objects (< 1%) as well as *native methods* are exclusively handled by the PPE. Since these instructions are rare, performance is not affected significantly.

In our context a native method is a function written in a language other than Java. Calling a native method requires its compiled code to be available to the Java VM. Consequently, the compiled code would have to reside within the local store of *each* SPE. Alternatively, the address could be resolved dynamically (which can be tricky without having transparent access to main memory) and copied to the local store on the fly. Since native method invocations are usually rare (see Section 5.2) we decided to let the PPE execute native methods. Furthermore there are native methods that manipulate the Java heap (e.g. array copying) which must be handled by the Shell-VM in any case. Attempting to execute native methods on the SPEs would also be pointless because most native methods interact with the operating system, which SPEs are unable to do and would have to consult with the PPE for anyway.

While our prototype system focuses on using SPEs for the execution of Java programs, it would be conceivable to use the PPE in addition to the SPEs whenever no requests are pending from any SPE. Currently this is not implemented in our prototype. However, this is rather an implementation deficiency instead of a general limitation. Running Java code on the PPE is well understood and has been implemented by many existing Java VMs. Our work focuses on running Java code on SPEs, which has not been explored so far.

CellVM conforms to the Java Virtual Machine Specification [23], which facilitates the execution of existing software without the need for modifications to the existing Java code. In particular, multi-threaded applications are able to exploit the computational power of the Cell processor, since our model offloads Java threads to the on-chip SPEs.

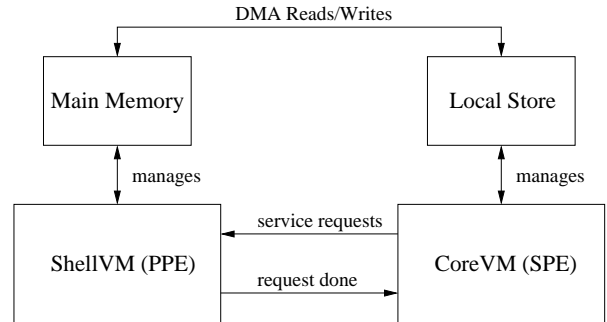


Figure 2. Execution model of CellVM. If a Core-VM faces an instruction with a “complex” memory interaction a switch from the Core-VM (SPE) to Shell-VM (PPE) is performed. The Shell-VM then executes the instruction on behalf of the Core-VM. In our model Shell-VM primarily acts as a resource manager.

The following sections provides deeper insights into the architecture and implementation of CellVM. Furthermore we describe code optimizations strategies for the interpreter and the just-in-time compilation and present a list of limitations of the current prototype implementation.

3 Implementation

CellVM effectively consists of two separate programs (Figure 2), compiled by two different compiler tool chains. The Shell-VM runs on the PPE and is primarily intended to manage *global* system resources. The Core-VM is loaded onto each SPE and manages *local* data structures like the Java stack or local caches. The Core-VMs are the virtual execution units and process the majority of Java VM instructions.

Our current prototype implements two modes of execution. The first approach installs a byte-code interpreter on each SPE to process Java instructions. The second approach provides a runtime environment that supports the execution of native SPE instructions. The PPE dynamically compiles Java byte-codes to SPE code which is then distributed to the SPEs on demand. Since the SPE binary resides within the local store, we considered implementing a mixed-mode execution (only frequently used methods are compiled to native code) as potentially less efficient with respect to available local store space for caching, since more SPE code would consume additional space. However, providing SPEs with either an interpreter or a native execution environment and migrating the state of a “interpreter-SPE” to a “native-SPE”

could potentially prove beneficial for certain applications. Our current prototype implementation does not implement this migration process.

To implement our approach we extended JamVM [18], an open source Java Virtual Machine. We chose JamVM amongst available alternatives because of its compact size. This is important for the interpreter version, since the interpreter needs to reside in the local store of a SPE and subtracts from the local memory available for storing instruction and data.

3.1 Byte-Code Interpretation

The standard technique for Java execution is interpretation, which can be implemented using different approaches [22]. Due to the missing hardware branch prediction facilities of the SPE architecture the implementation of a “switched interpreter” would incur a significant performance penalty since at least three machine level branch instructions are executed per byte-code instruction. Direct-threaded interpreting [7] reduces the number of mandatory branch instructions to one, which is thus the most suitable interpreting technique for the SPE. Our interpreter uses the direct-threaded interpreter variant of JamVM. In [34] we show how to build a customized direct-threaded interpreter targeted towards the Cell architecture.

3.2 Just-In-Time Compilation

Even in the case of direct-threaded interpretation we still incur at least one expensive branch operation per executed Java bytecode. To address this problem many high performance VMs compile byte-codes to directly executable machine instructions at runtime (dynamic just-in-time code generation). CellVM follows the same approach, and includes a dynamic compiler that can translate Java bytecodes to directly executable SPE machine code. The dynamic compiler in CellVM implements a straight-forward instruction by instruction code generator that only performs a very limited set of optimizations. Since the main goal of the compiler is to reduce the branching penalty (and allow us to benchmark the cost of the branch), for the purpose of this prototype system this is not a significant limitation. When compiling bytecode, each bytecode is translated into a series of SPE machine instructions. To keep the translated code size at a reasonable level instructions that would consume too much space are implemented in C and compiled with GCC. If the code generator faces such an instruction, it emits a branch instruction that invokes the helper function instead of emitting machine code to implement the entire functionality of the instruction in place. This significantly reduces the size of the generated machine code. E.g., implementing the opcode `LDIV` requires 5 SPE instructions when

a branch to a C routine is issued but 40 SPE instructions if directly compiled to machine code.

It should also be noted that this branch to the helper function is much cheaper than the branch the interpreter incurs per instruction, because this branch is unconditional and direct and thus can be hinted using a special SPE instruction. Due to the code size constraints of the SPE binary, code generation is done on the PPE. Another reason is that some optimizations described in Section 3.5 require access to main memory. The generated code is copied to the instruction cache of a Core-VM on demand.

One challenge in this approach is identifying which instructions should be “exlined” (instead of “inlined” in place). Keeping the code size small has the advantage of achieving higher instruction cache hit rates, which is crucial to good performance, as described in Section 3.5. The downside, however, is that more performance limiting branches to C-equivalent implementations have to be performed. Profiling the current approach shows an increase of compiled code size by a factor of 12 as compared to the original byte-code size. This growth mainly originates in the fact that the SPE has a RISC ISA. What furthermore increases native code size is the fixed instruction length of a SPE instruction (32-bit).

CellVM’s JIT compiler is a simple pattern matching code generator. For building an optimizing compiler numerous existing optimizations [11] can be ported to CellVM. E.g., replacing the original stack-architecture of the Java VM by a register architecture [30] is particularly interesting for the SPE since it possesses 128 general purpose registers.

3.3 CellVM Memory Layout

The memory layout of CellVM is illustrated in Figure 3. Global memory regions (e.g., Java heap) are accessible by every thread and hence managed by Shell-VM. Data areas that are private to each thread are maintained locally by each SPE. In essence the Java VM-Specification describes two per-thread data areas.

The Java stack is analog to the stack of a conventional programming language such as C. It holds operands, computational results and arguments for method invocation and return values. For every method call, space for local variables and the Java frame is reserved. The operand stack is used to hold intermediate operands and pass parameters to the callee (see Figure 3).

We adapted the Java frame by adding fields that are required for returning from a function. As in traditional Java VMs, a method area pointer, program counter and previous frame pointer are used to restore the caller’s state before method invocation. The previous frame pointer points to the frame of the caller and the method area holds function

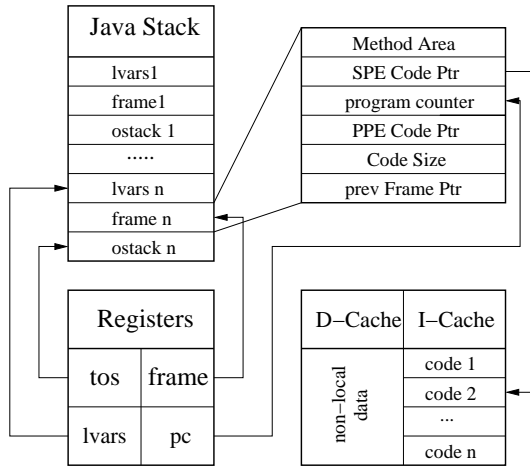


Figure 3. Core-VM memory layout. Whereas per-thread data areas (Java stack and Java frames) are managed by each instance of a Core-VM individually, global resources as the Java Heap or method-code are managed by Shell-VM. In addition, each Core-VM maintains a cache (instruction and data) to hold frequently used data and instructions on the local store.

properties like accessibility (private, public). We added the fields “PPE Code Ptr”, “SPE Code Ptr” and “code size”. If the “SPE Code Ptr” is valid (instruction cache hit), the address of the caller’s next instruction is calculated along with the program counter. Otherwise, (instruction cache miss) the parameters “code size” and “PPE Code Ptr” are used to copy instructions from main memory to the I-cache (instruction cache). Finally, a set of registers allows for fast access to local, frequently used data.

3.4 Switching and Communication

A switch of execution from a Core-VM to the Shell-VM simulates the processing of a complex instruction on the SPE and is the most expensive communication facility between a SPE and the PPE. As each Core-VM follows its execution path independently the Shell-VM cannot track the Core-VM’s current state. Hence, as a first step, the requesting Core-VM updates its internal state to the Shell-VM and raises an interrupt. The waiting service-thread (each Core-VM has an own service-thread on the PPE) catches the signal and executes the requested instructions. Meanwhile the SPE stalls. After completion of the request the PPE notifies the SPE, which in turn adapts its internal state and continues with execution. We call this mechanism *cooperative execution* or *co-execution*. By using the co-operative ap-

proach, the application of more time-consuming optimizations for JIT would be reasonable if admitted by the number of service requests. Both version of CellVM use the same approach for switching.

Reading or writing data to and from main memory is implemented using DMA transfers, which can be issued in a *blocking* or *non-blocking* manner. Blocking DMAs suspend execution until the data to copy is in place. This mechanism is applied to transferring data from main memory to the local store. Writing data back to the PPE domain is implemented using non-blocking DMA commands. By installing a buffer that holds data to be written back, the resulting latency can be reduced to the overhead stemming from filling the buffer and programming the DMA engine.

3.5 Code Optimizations

Reasonable performance can only be achieved if the number of (blocking) DMA transfers and switches of execution are comparatively small. The latter is of major importance in that performance penalties do not only stem from switching latencies. Too much service request could overuse the PPE, which then would become the system’s performance bottleneck.

To meet the conditions mentioned above we applied *code preparation* to reduce the number of blocking DMA transfers and *opcode rewriting* or *binary rewriting* to minimize the switching activity. Code preparation is accomplished by processing the method’s byte-code and gathering as much information as possible prior to method invocation. E.g., the opcode LDC pushes a constant value (located in the constant pool) of the method onto the operand stack. Per definition the constant value must not change at any time and hence is known at compile-time. Instead of accessing the constant pool at runtime the value can be resolved in the preparation stage and coded into the operands of the LDC operation. As a consequence the method’s constant-pool does not need to reside within the local store, which saves a blocking DMA transfer, or if we would maintain the constant pool in the local store (which we do not), would make room for other data in the cache.

The simple code preparation technique is limited to statically resolvable information. Code rewriting (opcodes for the interpreter and native code for the JIT) is a dynamic mechanism used by the SPE to replace Java code on the fly with so-called “quick opcodes”. Quick opcodes include dynamically resolved information that can be re-used with every execution of the same instruction. E.g., the opcode GETSTATIC can be re-written as a quick-opcode by resolving the reference to the constant pool (which requires a switch) and storing the attached operands. The next time this particular opcode is executed the reference can be used to perform a cache lookup and gather the ac-

tual value. To avoid expensive synchronization overhead in multi-threaded applications, opcode rewriting is done locally by each SPE. Therefore a high I-cache hit-rate is absolutely necessary to keep the number of DMA transfers and switches at a minimum since dynamically resolved information is lost when the cache must be purged (see Section 4).

Another way of reducing mandatory switches is extending CellVM's instruction set by *math-ops*. CellVM uses GNU classpath version 0.91 [15] in which numerous function calls to the math library are native methods. Instead of performing a switch each time, Shell-VM detects such calls and rewrites the native function call to the corresponding math-opcode, which then can be executed locally by the SPE.

3.6 Limitations of the Current Prototype

Our approach exploits full computational capabilities of the Cell processor if the application is made up of 6 (or fewer) threads (see Section Section 5), since each SPE can process a single thread only. An implementation shortcut of the current prototype is that it cannot execute applications with more than 6 threads since each thread is physically bound to a SPE. However, a heuristic to route SPE threads vs. the PPE could be added easily. Also, garbage collection (GC) is not implemented in the current version of CellVM. We plan to add a GC implementation either in the manner of [10], which use a SPE as a GC coprocessor, or by placing a GC thread on the PPE.

The main performance limiting factors of CellVM are DMA transfers and switches as described in Section 3.4. Having such operations in the application's "hot spot" reduces performance noticeably. Also "synchronized" statements that require a switch to the Shell-VM reduce the throughput, since the PPE coordinates such synchronization points. SPEs offer hardware mechanisms for direct synchronization without involving the PPE. We plan on adding this feature in a future version of CellVM.

Also, our current prototype does not strictly follow Java's memory model. To improve performance, we do not offer a cache coherent view of the Java heap. Each SPE can operate on the entire heap, but Java threads are not allowed to synchronize via main memory. Instead, threads have to explicitly synchronize using Java's locking mechanism and synchronize keyword. Several avenues exist to remedy this limitation: multi-processor systems often implement cache coherency protocols [24] in hardware. Another common approach is to introduce explicit memory barriers that provide certain cache coherency guarantees only when needed [13]. The Cell architecture offers-hardware features in the SPUs that are well suited to implement a software-only cache coherency protocol and we plan on implementing a partially cache coherent Java heap in the future that guarantees co-

herent memory access of object fields declared as *volatile*.

4 Software Controlled Caches of SPEs' Local Stores

The local store of a SPE can be roughly divided into three sections: the SPE binary covering 51%, the caches using 25% and 24% are reserved for the Java runtime stack. We use static pool sizes because dynamic cache pools require utilizing dynamic memory allocation on the SPEs which we found to be very slow. Performance measurements on IBM's cycle-accurate and memory-timing accurate Cell simulator indicate that both malloc and free of 1-KB of local store consumes 1400 cycles. It is unlikely that this overhead could be reduced significantly merely through a more efficient implementation.

The instruction cache is implemented as a *fully associative cache* with variable cache block length. The cache block length equals the code size of a method and its corresponding address in main memory is used as the tag. A cache miss triggers a DMA transfer that copies the entire method instructions to the local store. The data field of the cache is filled with instructions until the pre-defined limit is reached. If insufficient space is available to cache the next method *all* cached instructions are purged. By using this replacement strategy we accept latencies coming from the lookup, since tags are added to the cache in sequence of their appearance. On the other hand, we can except conflict misses (as they can appear in direct-mapped caches) and can furthermore enforce that recently used instructions are kept inside the cache at a greater probability.

Providing a fast lookup for the data cache is more important than for the instruction cache since it is accessed more frequently. We decided to implement a *direct mapped cache* since a software implementation can be done efficiently. The data cache configuration (number of cache lines and cache block size) can be configured at compile time. A cache line is written back to main memory in case of a conflict miss. One more reason to update the data cache to main memory and purge it is a switch of execution from a Core-VM to the Shell-VM. On the one hand, writing the SPE cache content back to main memory guarantees coherency for the PPE (read accesses) and on the other hand by purging the local cache on the SPE it is assured that PPE manipulated data is updated correctly to the SPE with the next DMA transfer.

5 Performance Evaluation

In this section we describe the experimental evaluation of our prototype implementations. We use Java Grande [31] to benchmark the performance of both implementations, the

interpreter and the JIT. The baseline of our benchmarks is JamVM running on the PPE of the Cell. That baseline is compared to running 1 to 6 Java threads in parallel using the 6 available SPEs of the Cell processor. The benchmarks were conducted on a Playstation 3 running a 2.6.16 Linux kernel populated with eight SPEs. For the programmer, only 6 SPEs are available since one SPE is reserved for internal uses and one is disabled to improve manufacturing yields. The PPE as well as the SPEs run at 3.2 GHz.

To analyze the performance characteristics and expose possible performance bottlenecks we ran a series of micro benchmarks using Java Grande Section 1 benchmark suite. Section 1 is composed of small programs measuring low-level VM operations such as loops, casting or arithmetic operations.

Following the micro benchmarks we used Section 2 of the Java Grande benchmark suite to evaluate the performance for a series of real-world computation intensive kernels. Each benchmark was scaled up from running a single SPE to running in parallel all 6 available SPEs of the Cell. CellVM currently does not offer a fully cache-coherent view of main memory. Since the multi-threaded benchmarks use shared-memory for thread synchronization, a software-based cache coherence protocol is required to ensure data integrity. Hence, we modified the benchmarks that worker-threads only operate on non-shared data. This benchmarking methodology does not take synchronization overhead of truly multi-threaded applications into account, however, it allows us to evaluate our co-operative execution model. Furthermore, we are able to measure our caching strategy and investigate whether the EIB (Element Interconnect Bus) could become a bottleneck.

The cache configuration for both versions of CellVM is the same. 48-KB are reserved for the instruction cache and 16-KB for the data cache. The data cache was configured with a cache block length of 128 byte and 128 sets.

5.1 Microbenchmarks

The results of Java Grande micro benchmarks are shown in Figure 4. Performance was evaluated by pitting a single SPE against the PPE of the Cell processor. Note that for these benchmarks DMA latencies only carry a minor penalty since the data cache is large enough to hold the entire data operated on. Values in Figure 4 are normalized to the JamVM running on the PPE (1 is JamVM’s performance).

The interpreter version is able to slightly outperform the PPE in the loop benchmark. For the arithmetic, cast and math benchmark, the performance slightly varies from 0.77 to 0.60. These benchmarks can run without intervention from the PPE, which explains the solid performance results. The assign and method benchmark, on the other

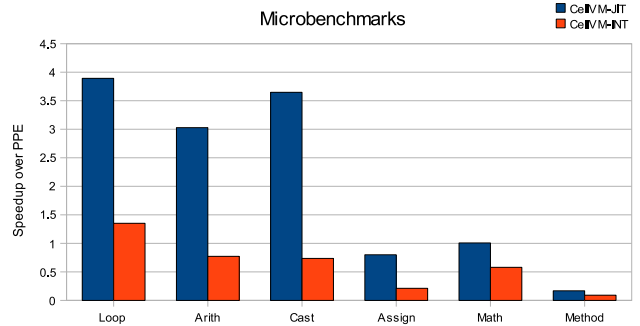


Figure 4. Performance evaluation of low-level VM operations. Values are normalized to JamVM running on the PPE.

hand, include CellVM’s worst case scenario: synchronized methods and data structures. Since the SPEs release synchronization issues to the PPE, the SPE has to yield for object lock and lock release. This overhead could be reduced by exploring direct SPE-to-SPE synchronization in favor of performing all synchronization exclusively by the PPE.

CellVM performs significantly better in the JIT-version. The greater speedup essentially results from the reduced number of branches when executing native instructions in comparison to interpreting code. As illustrated in Figure 4, executing native code on the SPE results in a speedup of 3.8 to 2.6 for the loop, cast and arith benchmark. The loop benchmark performs best since the generated instructions do not include branches to C routines. The arith, cast and math benchmarks, on the other hand, contain opcodes that are implemented in a C routine to keep the code size reasonable small (e.g. LDIV for the math benchmark). Performance limiting factors for the assign and method benchmark are identical to the interpreter version, namely synchronized operations.

5.2 Application Performance

We explored CellVM’s behavior in more complex applications using the modified version of the sequential Java Grande benchmark suite. When running multi-threaded applications three types of performance bottlenecks might occur: first, if the applications requires too many switches from a Core-VM to the Shell-VM, the PPE can become congested by too many requests. Second, the number of blocking DMA transfers can affect throughput significantly since SPE execution must be suspended until the data is in place. Finally, if the total number of DMA requests exceeds the DMA queue size (16 commands per DMA engine) the SPE must stall until a slot in the queue becomes available.

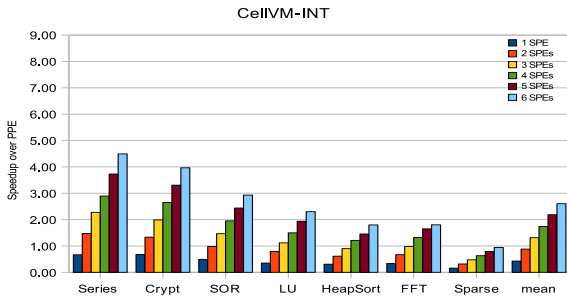


Figure 5. Speedup for the interpreter version of CellVM for Java Grande Section 2 benchmarks relative to the performance of JamVM when using 1 to 6 parallel SPE threads.

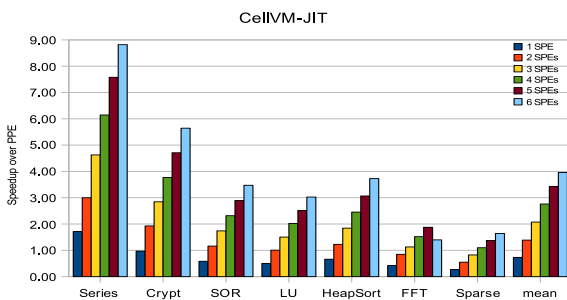


Figure 6. Speedup for the JIT version for Java Grande Section 2 benchmarks relative to the performance of JamVM when using 1 to 6 parallel SPE threads.

Our benchmarks were performed using 1–6 threads, each executing its own instance of the benchmark concurrently. Figure 5 shows the results for the interpreter version and Figure 6 illustrates the speedup for the JIT-version. Figure 7 and Figure 8 show the corresponding cache statistics and provide information about the the DMA transfers of the JIT-version. While the interpreter-version shows a slightly higher I-Cache hit rate (since byte-codes are more compact in size) the data cache numbers are identical. The number of the I-Cache hit rate are based on method-level, which means that if a requested method code resides within the cache we count a hit and a miss otherwise.

For both version, the number of switches from a SPE to the PPE is negligibly small for all tested applications. Over 99.99% of all opcodes are executed by Core-VM, which can be traced back to the high instruction cache hit rate and to our VM's extraordinary efficiency in rewriting opcodes to quick-opcodes. Figure 8 shows that performance sig-

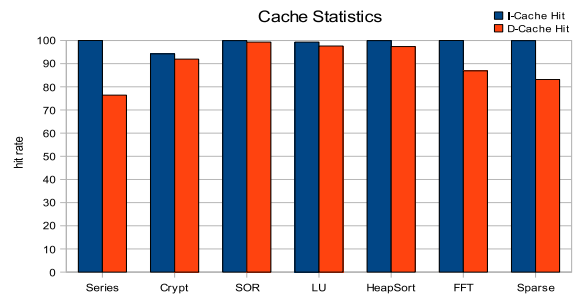


Figure 7. Cache hit rates for the instruction and data cache for each benchmark

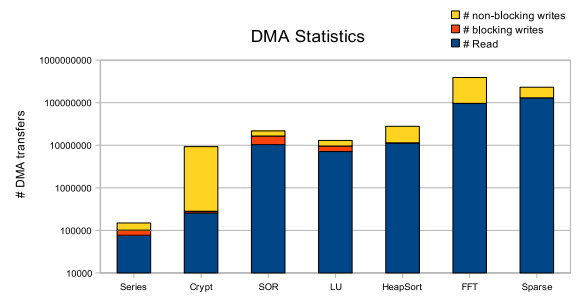


Figure 8. Number of performed DMA transfers for a benchmarks using a single SPE.

nificantly depends on the number of DMA transfers, with blocking DMAs causing a higher latency than non-blocking transfers. It can be seen that the sparse matrix multiplication benchmark has the lowest data cache hit rate, which results in a higher number of DMA transfers.

The Series benchmarks is characterized by an extensive use of transcendental and trigonometric functions. The low data cache hit rate barely affects performance since the data cache is accessed infrequently. The high throughput basically results from the low number of DMA transfers and the application of the math opcodes. The second best performance results were achieved by the IDEA (International Data Encryption Algorithm) encryption benchmark. The benchmark requires a comparatively high number of DMA transfers. As a large fraction of them are non-blocking, the capability of the SPE of moving data in parallel with program execution reduces the latencies considerably.

The SOR, LU and the HeapSort benchmark show similar cache, DMA and performance characteristics. The FFT benchmark in the JIT-version, however, seems to saturate the DMA capabilities of the EIB. The highest speedup can be achieved with 5 SPEs. The Sparse Matrix Multiplication

benchmarks perform worst due to its poor data locality. This explains the low data cache hit rate and the high number of DMA transfers.

6 Related Work

The Java Programming Language offers features such as platform independence for portability and support for multi-threading or distributed programming. These features make Java particularly attractive for parallel and distributed program developers. Al Jaroodi et al. [3] survey current parallel Java projects. The authors break existing approaches down into three categories:

The first category of systems replaces the standard Java VM by building a new system. Among others this category includes Titanium [33], which compiles a Java dialect to C, and ParaWeb [8], which aims at utilizing the Internet as a computing resource. While this approach gives developers more freedom in applying parallel methodologies its acceptance among users might be limited, since users are forced to apply a new dialect or port code to a new platform. In contrast to these works, our implementation fully conforms to the Java VM specification and completely hides the heterogeneous multi-core architecture beneath a VM layer. Hence, existing Java code can be executed without the need for modifications.

The second category enhances the Java runtime environment by additional class libraries that provide explicit parallelization functions. Examples are Agents [17] and JPVM [14]. These approaches provide a good support for heterogeneity, however they tend to be less efficient due to the overhead of implementing remote objects and message passing at the language level instead of implementing them as virtual machine primitives. The CellVM project pursues a different approach: by implementing a new Java VM, which matches the capabilities of the hardware, parallel execution on multiple cores is managed by the VM, without the need for additional class libraries.

The third category aims at providing seamless parallelization for multi-threaded applications. This approach does not require programmers to address the parallelization process. Any Java multi-threaded application can run in parallel on a *distributed system* without any changes. CellVM closely resembles this category, however differs in that it explicitly distributes workloads to execution units on a single die and thus reduces communication and synchronization overheads greatly. Other examples of this class of Java parallelization approaches include cJVM [4] or JavaParty [28].

The Java Programming Language has also gained ground in the domain of high performance computing although Java offers a high level of hardware abstraction that entails performance penalties. The inherent performance limiting fac-

tors of the Java language [26] can be mitigated by several approaches: for example, accessing arrays in Java is expensive (as compared to C) since a compulsory reference check has to be performed and a corresponding exception to be thrown if the check fails. Midkiff et al. [25] describe a collection of transformations that can significantly reduce this overhead while remaining fully compliant with the Java language semantics. Another additional cost of Java arrays in comparison to unchecked C arrays are dynamic out-of-bounds check that assure that the index lies within the specified array size with each access. Techniques that eliminate the expensive runtime check are described in [29].

In addition to optimizations targeted towards eliminating language-based performance constrains, dynamic just-in-time compilers [6] aims at reducing performance penalties originating in byte-code interpretation [1]. Moreover, the application of compiler optimization techniques [2] (which are commonly used in static compilers) have proven to be efficient for dynamic compilation too. Among others, these include common subexpression elimination (CSE) or (global) register allocation. The design and implementation of such a just-in-time compilation framework is described in [32].

On the hardware side, the Cell processor is evolving into a serious alternative to traditional processors in the domain of high-performance computing. Williams et al. [35] present quantitative performance data for scientific kernels (dense matrix multiply, 1D and 2D FFTs) that compares Cell performance to leading superscalar (AMD Opteron), VLIW (Intel Itanium 2) and vector (Cray X1E) architectures. They conclude that the Cell processor shows a tremendous potential for scientific computations and demonstrate that it can outperform the processor cores mentioned above. However, exploiting the full capabilities of the Cell architecture includes a reasonable load balancing over the available throughput-orientated processor cores, efficient software based management of the scratch-pad memories [20] in conjunction with the appropriate DMA buffering scheme [9], as well as the usage of low-level assembly intrinsics or architecture dependant compiler optimizations [12].

The CellVM project aims at opening up the full computational capabilities of the heterogeneous Cell multiprocessor to Java technology. Due to the architecture and programming model of the Cell existing virtual machines are limited to use Cell's main processor only. To the best of our knowledge we are the first to incorporate the throughput-orientated processors into a Java Virtual Machine.

7 Conclusion

We have presented CellVM, a Java virtual machine that executes Java programs cooperatively on the different cores

of the Cell multiprocessing platform. In our design, most common Java byte-code instructions are executed directly on the throughput-orientated SPEs, while instructions that require complex main memory interaction are processed by the main PPE core. To mask expensive DMA transfers between main memory and the SPEs, we have devised a software-based caching strategy that maintains local copies of frequently accessed data structures such as method byte-codes or arrays in the local store of the SPE.

Experiments show that cooperative execution is able to distribute most of the activity to the SPEs so that even six simultaneous Java threads running on six SPEs do not saturate the PPE core with service requests. Furthermore, for the tested applications, only one benchmark program saturated the EIB when increasing the number of parallel threads to six. This shows that our software-based caching strategy is successful for 6 of 7 applications. Using all six SPEs in parallel our system was able to achieve an average speedup of 3.5 for the JIT- and an average speedup of 2.5 for the interpreter version.

Our approach relies on several properties an application has to satisfy to achieve good performance: First, performance significantly depends on the switching activity from a Core-VM to the Shell-VM. Having a transfer of control from the SPE to the PPE (which also includes synchronization operations in the current implementation) makes CellVM intolerably slow. Second, the DMA transfers that are required to move instructions and data to and from the local store add an additional overhead. Hence, our approach relies on a reasonable data-locality. While an application implemented in C allows for an efficient overlapping of a data-transfer with computation, this overlapping is not inherently transparent to a Core-VM. An efficient prefetching of data would either require an analysis stage in the preparation phase (which is performed at runtime) or require monitoring of e.g. array accesses at runtime. It remains an open question, whether this analysis/monitoring can be performed effectively.

We conclude that the current prototype implementation of CellVM performs well for computation intensive programs that only operate on small set of data. However, the current implementation of CellVM leaves much room for improvements and we strongly believe that if the data prefetching and the synchronization/communication between the SPEs can be implemented efficiently, a Java VM for the Cell processor can perform very well.

8 Acknowledgments

This research effort was partially funded by the State of California and Microsoft Research under the Microelectronics Innovation and Computer Research Opportunities (MICRO) program, the Bavaria California Technology Center,

and the National Science Foundation (NSF) under grants TC-0209163 and ITR-0205712. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation or any other agency of the U.S. Government.

The authors would like to acknowledge Christian Steger, Technical University of Graz, and Thomas Gross, ETH Zurich, for their support of this research project.

References

- [1] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a just-in-time java compiler. *SIGPLAN Not.*, 33(5):280–290, 1998.
- [2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
- [3] J. Al-Jaroodi, N. Mohamed, H. Jiang, and D. Swanson. A comparative study of parallel and distributed java projects for heterogeneous systems. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2002.
- [4] Y. Aridor, M. Factor, A. Teperman, T. Eilam, and A. Schuster. A high performance cluster jvm presenting a pure single system image. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 168–177, New York, NY, USA, 2000. ACM Press.
- [5] K. Arnold and J. Gosling. *The Java programming language (2nd ed.)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.
- [6] J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003.
- [7] J. R. Bell. Threaded code. *Commun. ACM*, 16(6):370–372, 1973.
- [8] T. Brecht, H. Sandhu, M. Shan, and J. Talbot. ParaWeb: Towards World-Wide Supercomputing. In *EW 7: Proceedings of the 7th workshop on ACM SIGOPS European workshop*, pages 181–188, New York, NY, USA, 1996. ACM Press.
- [9] T. Chen, Z. Sura, K. O'Brien, and K. O'Brien. Optimizing the use of static buffers for DMA on a CELL chip. In *The 19th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2006)*, November 2006.
- [10] C.-Y. Cher and M. Gschwind. Cell GC: using the cell synergistic processor as a garbage collection coprocessor. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 141–150, New York, NY, USA, 2008. ACM.
- [11] B. Davis and J. Waldron. A survey of optimizations for the java virtual machine. In *PPPJ '03: Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 181–183, New York, NY, USA, 2003. Computer Science Press, Inc.

- [12] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing Compiler for the CELL Processor. In *PACT '05: Proc. 14th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.
- [13] A. Feldmann, T. Gross, D. O'Hallaron, and T. M. Stricker. Subset barrier synchronization on a private-memory parallel system. In *SPAA '92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 209–218, New York, NY, USA, 1992. ACM.
- [14] A. J. Ferrari. JPVM: Network Parallel Computing in Java. Technical report, Charlottesville, VA, USA, 1997.
- [15] GNU Classpath, <http://www.gnu.org/software/classpath/>.
- [16] M. Gschwind. Chip multiprocessing and the cell broadband engine. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 1–8, New York, NY, USA, 2006. ACM.
- [17] M. Izatt, P. Chan, and T. Brecht. Ajents: towards an environment for parallel, distributed and mobile java applications. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 15–24, New York, NY, USA, 1999. ACM.
- [18] JamVM, A Compact Virtual Machine, <http://jamvm.sourceforge.net/>.
- [19] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research & Development*, 49(4/5):589–604, July/September 2005.
- [20] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratchpad memory space. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 690–695, New York, NY, USA, 2001. ACM.
- [21] I. H. Kazi, H. H. Chen, B. Stanley, and D. J. Lilja. Techniques for obtaining high performance in java programs. *ACM Comput. Surv.*, 32(3):213–240, 2000.
- [22] P. Klint. Interpretation techniques. *Software — Practice & Experience*, 11(9):963–973, September 1981.
- [23] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
- [24] D. E. Marquardt and H. S. Alkhatib. C2mp: a cache-coherent, distributed memory multiprocessor-system. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 466–475, New York, NY, USA, 1989. ACM.
- [25] S. P. Midkiff, J. E. Moreira, and M. Snir. Optimizing array reference checking in java programs. *IBM Syst. J.*, 37(3):409–453, 1998.
- [26] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. Java programming for high-performance numerical computing. *IBM Syst. J.*, 39(1):21–56, 2000.
- [27] OpenMP, The OpenMP specification for parallel computing, <http://www.openmp.org/blog/>.
- [28] M. Philippsen and M. Zenger. Javaparty: Transparent remote objects in java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.
- [29] F. Qian, L. J. Hendren, and C. Verbrugge. A comprehensive approach to array bounds check elimination for java. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 325–342, London, UK, 2002. Springer-Verlag.
- [30] Y. Shi, D. Gregg, A. Beatty, and M. A. Ertl. Virtual machine showdown: stack versus registers. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 153–163, New York, NY, USA, 2005. ACM.
- [31] L. A. Smith, J. M. Bull, and J. Obdržálek. A parallel java grande benchmark suite. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 8–8, New York, NY, USA, 2001. ACM.
- [32] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the ibm java just-in-time compiler. *IBM Syst. J.*, 39(1):175–193, 2000.
- [33] Titanium, <http://titanium.cs.berkeley.edu/>.
- [34] K. Williams, A. Noll, A. Gal, and D. Gregg. Optimization strategies for a java virtual machine interpreter on the cell broadband engine. In *CF '08: Proceedings of the 5th international conference on Computing frontiers*, New York, NY, USA, 2008. ACM.
- [35] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM.