

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

Albert Noll

Technische Universität Graz
nolla@uci.edu

Andreas Gal

University of California, Irvine
gal@uci.edu

Michael Franz

University of California, Irvine
franz@uci.edu

Abstract

We present a runtime environment that implements a Java Virtual Machine by co-execution between the different functional units of the heterogeneous Cell multiprocessing platform. Our approach uses the Cell's scratchpad memory as a software-controlled cache and contains an automatic software-based memory management system for instruction and data caching. Profiling shows a cache hit rate of above 90% for the instruction cache and above 74% for the data cache.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—Incremental compilers; optimization; runtime environments; D.1.3 [Concurrent Programming]: Parallel programming

General Terms Design, Experimentation, Performance

Keywords Java Virtual Machine, Cell Broadband Engine, just-in-time compilation, dynamic parallelization

1. Introduction

The Cell Broadband Engine, jointly developed by IBM, Sony, and Toshiba, is an emerging hardware platform for multimedia devices. Unlike the symmetrical multi-core architectures that are being developed for the desktop computing market, Cell implements an asymmetric architecture in which there is a general-purpose processor core derived from the PowerPC, and eight identical throughput-oriented cores implementing an entirely different architecture and instruction set that is targeted toward multimedia applications.

Integrating different cores onto a single chip in this manner makes sense from a hardware designer's perspective. For the intended application domain of multimedia applications, having a number of throughput-oriented processor cores is far more effective 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 the asymmetrical architecture and multiple instruction sets of a heterogeneous multiprocessor makes programming considerably more difficult.

The goal of our work is to provide the best of both worlds, combining the benefits of a *symmetric* multi-processor 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 multiprocessing cores.

We have implemented a prototype of such a system that we call *CellVM*. To the programmer, CellVM presents the interface of a standard Java Virtual Machine (JVM). Internally, CellVM interprets JVM bytecodes by co-execution between the different functional units contained on the Cell microprocessor. CellVM utilizes an interpretation approach that we call *cooperative interpretation*, executing simple instructions directly on specialized processor cores, but requesting assistance from a general purpose core for executing complex (and often rarely used) instructions.

Of particular note is our solution to the data latency problem. In the Cell architecture, the throughput-oriented processors do not have hardware-based data caches but provide software-controlled scratchpad memories in conjunction with DMA capabilities. Key to a streamlined division of labor in our cooperative interpretation environment is an efficient use of these scratchpad memories as a distributed data cache. Our software-controlled data caching algorithm is surprisingly effective, achieving hit rates above 90% for the instruction cache and above 74% for the data cache for the benchmarks that we tested.

The remainder of this paper is organized as follows. Section 2 gives a brief introduction to the Cell Broadband Engine Architecture. Section 3 discusses the design rationale and implementation details of CellVM, our Java Virtual Machine for the Cell architecture. In Section 4, we describe the software-driven caching mechanism that we use on the throughput-oriented processor cores. To evaluate our approach, we have performed a series of benchmarks that we discuss in Section 5. Section 6 contains related work, followed by our conclusions and outlook to future work in Section 7.

2. The Cell Broadband Engine Architecture

The Cell Broadband Engine Architecture (CBEA) [14] is a single-chip multiprocessor designed with multimedia applications in mind. The design goals [10] of the CBEA were chosen carefully to address the fundamental challenges facing modern microprocessor development: high memory latencies, on-core power dissipation, and physical processor frequency limitations. Past generations of microprocessors achieved performance gains mostly through higher clock frequencies, deeper pipelines and wider issue width. However, since processor throughput has increased at much quicker pace than memory access time could be reduced, modern processors often spend significant amounts of time waiting out processor stalls resulting from code or data cache misses. On a modern symmetric multiprocessor, for example, main memory has a latency of upward of 1000 processor cycles. If data and code are

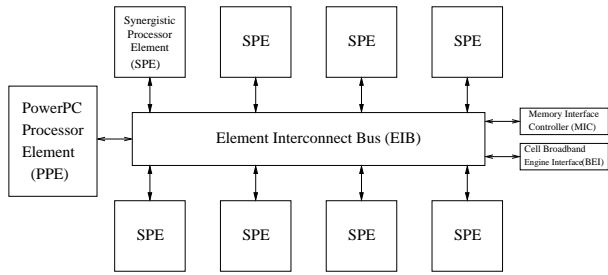


Figure 1. The heterogeneous Cell Architecture consists a PowerPC Processor Element (PPE) and eight Synergistic Processing Elements (SPE), which are SIMD RISC cores optimized for computation-intensive tasks. Communication between PPE, SPE, main memory and external interfaces is realized through the Element Interconnect Bus (EIB).

not cached efficiently in near-processor caches, processor throughput is limited to the bandwidth of main memory rather than being able to take advantage of the processor’s true computational power, forming an insurmountable performance limit also referred to as the *memory wall* [18].

Figure 1 shows the basic components of the Cell Broadband Engine Architecture. It consists of a PowerPC Processor Element (PPE), which is primarily intended to manage global resources and distributes work to on-chip Synergistic Processing Elements (SPE). Each SPE is a SIMD RISC core optimized for computation intensive tasks. Communication between the PPE, the SPEs, main memory, and external devices is realized through an Element Interconnect Bus (EIB).

2.1 The Power Processor Element

The PPE, which can be regarded as the Cell’s main processor, is optimized for high clock frequencies (up to 4 GHz) and power efficiency. It is a dual-issue design that does not dynamically reorder instructions at issue time (i.e., it provides in-order execution of instructions). The core interleaves instructions from two computational threads at the same time to optimize the use of issue slots, maintain maximum efficiency, and reduce pipeline depth. The PPE is capable of executing two threads simultaneously and can be viewed as a two-way multiprocessor with shared data flow. The illusion to the software is that two independent processing units are in place.

Its 64-bit Reduced Instruction Set Computing (RISC) core is extended by a vector multimedia extension unit (VMX) and offers branch prediction and a hardware-based transparent memory hierarchy. Hence, it is well suited for executing general purpose programs and computational un-intensive tasks, such as resource management and input/output services. A traditional cache hierarchy with a 32-KB first-level (L1) instruction and data cache and a 512-KB second level (L2) cache is used to hide memory latencies. The PPE also runs the operating system and is intended primarily for work distribution to SPEs and resource management.

The processor can be divided into three sub-units: the instruction unit (IU), a fixed-point execution unit (XU) and a vector scalar unit (VSU). The IU is responsible for instruction fetch, decode, branch, issue and completion. It fetches four instructions per cycle per thread into a buffer and dispatches the instructions from this buffer. After decoding and dependency checking the instructions are dual-issued to an execution unit. Up to two instructions per cycle can be executed, if the instructions are assigned to different units. The XU consists of a 64-bit wide, 32 entry large general-purpose register file per thread, a fixed-point execution unit and a

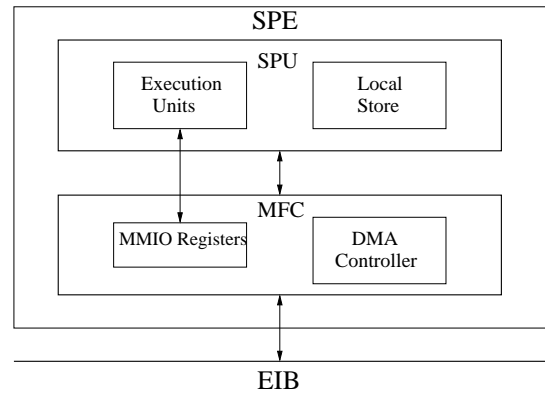


Figure 2. An SPE consists of a Synergistic Processing Unit (SPU) and a Memory Flow Controller (MFC). Data and Instruction are loaded to local store (LS) via direct memory access (DMA) through the MFC, which can be done in parallel to execution. Furthermore a special set of registers allows the exchange of 32-bit messages between an SPE and other elements.

load/store unit with a 16-entry store queue. The VSU also maintains a 64-bit wide, 32 entry large register file per thread as well as a ten-stage double precision pipeline. Additionally, there is a 32-entry by 128-bit vector register file per thread, and all instructions are 128-bit (Single Instruction Multiple Data) SIMD with varying element width (2 x 64-bit, 4 x 32-bit, 8 x 16-bit, 16 x 8-bit, and 128 x 1-bit).

To support this overall design of executing dedicated tasks on designated execution units [11], the Cell architecture offers a number of unique hardware features, such as fast context switching between SPEs and the PPE to call operating system functions (that are executed by the PPE) from SPEs.

2.2 The Synergistic Processing Element

While the PPE uses traditional hardware-based *transparent* caching to hide the access time necessary to load and store data from and to main memory, SPEs use a different mechanism: SPEs have a three level memory structure (main memory, local stores and large unified register files). The large unified register file allows to hold the majority of operands directly inside the CPU core without having to spill values onto the stack. A medium-sized local store (256k in current Cell processors) is used to store SPE code and temporary program results until the final results are written back to main memory. SPEs can perform asynchronous, coherent DMA transfers between their local stores and main memory and therefore do not need to stall while data moves back and forth on the interconnect bus.

SPEs are designed to execute regular computationally intensive programs rather than general purpose software. This allows to hide memory latencies without having to employ complex hardware mechanisms such as branch-prediction, out-of-order execution, and deep pipelining that are used for this purpose in general-purpose processor cores. By focusing on regular and computationally intensive programs, SPEs can be built with much reduced hardware complexity and yet achieve a good utilization of processor resources. This in turn reduces the overall core size of SPEs and allows to pack more SPEs at a higher density onto a single multi-core die.

Each SPE [8] is an independent execution unit with it’s own program counter, registers and 256KB local store (single port SRAM with 6 cycles access time), which can be regarded as a software controlled cache. Instructions and data are fetched from main mem-

ory to the local store via direct memory access (DMA). The SPE can be divided into two sub-units, the Synergistic Processing Unit (SPU) and the Memory Flow Controller (MFC) (Figure 2). The SPU core is a 128-bit single instruction multiple data (SIMD) RISC architecture.

Execution of instructions is strictly in order and branch-prediction units and scheduling logics are not present. Instead, an 128-entry 128-bit wide unified register file (integer, floating point, etc. data types use the same register) is used to store values, return addresses, results of computation and so forth, which replaces expensive speculative hardware techniques. Each SPU owns two pipelines, into which it can issue and complete up to two instructions per cycle. Whether an instruction goes to the “even” or the “odd” pipeline depends on the instruction type. For example, load and store instructions are assigned to the odd pipeline and single-precision floating point instructions to the even pipeline.

2.3 The Element Interconnect Bus

Communication between an SPU and the Element Interconnect Bus (EIB) is realized through the the SPU’s Memory Flow Controller (MFC). The MFC of each SPE can enqueue up to 16 DMA commands, which implies that the whole system can process more than 100 DMAs simultaneously. DMA transfers are coherent with respect to system storage, which means that a pointer from the PPE’s main storage, for example, can be used as a starting point to issue a DMA command. Furthermore, the SPU is capable of continuing execution in parallel with its MFC’s DMA transfers. It also provides memory mapped I/O registers (MMIO) and channels to monitor DMA commands, SPU events and facilitate interprocess-communication via mailboxes and signal-notification. Mailboxes are a set of queues that support exchanges of 32-bit messages between an SPE and other devices. Two one-entry mailbox queues are provided for sending messages from the SPE (SPU Write Outbound Mailbox, SPU Write Outbound Interrupt Mailbox), and one 4-entry to send messages to SPE (SPU Read Inbound Mailbox). Mailboxes are used in our implementation of the JVM to synchronize SPE and PPE threads.

The PPE and the SPEs communicate coherently with each other through the EIB. The EIB is a 4-ring structure (2 clockwise, 2 counterwise) for data, and a tree structure for commands with an internal bandwidth of 96 Bytes per cycle. The EIB has two external interfaces, the MIC, which is the interface between main memory and EIB and the BEI, which allows data transfer between EIB and I/O devices.

3. CellVM: A Java Virtual Machine

The goal of CellVM is to create a homogeneous virtual machine runtime environment on top of a heterogeneous processor architecture (Figure 3). The architectural differences between SPEs and the PPE are hidden from the programmer. Instead of having to program specifically for an SPE with all its architectural limitations, such as the limited size of the local storage, programmers can write regular Java programs that are then executed by CellVM. In particular, CellVM distributes Java threads to the on-chip cores.

Our current prototype implementation is able to execute eight Java threads in parallel by distributing them over eight SPEs. Each SPE runs a core interpreter (CoreVM), which is able to execute the majority of bytecodes without intervention and assistance from the PPE, which serves as the outer shell for all core interpreters (ShellVM). If the CoreVM encounters a bytecode instruction that cannot be executed locally, the SPE requests assistance from the ShellVM. The PPE will then execute the bytecode instruction of behalf of the SPE. We call this mechanism cooperative interpretation, or *co-interpretation*.

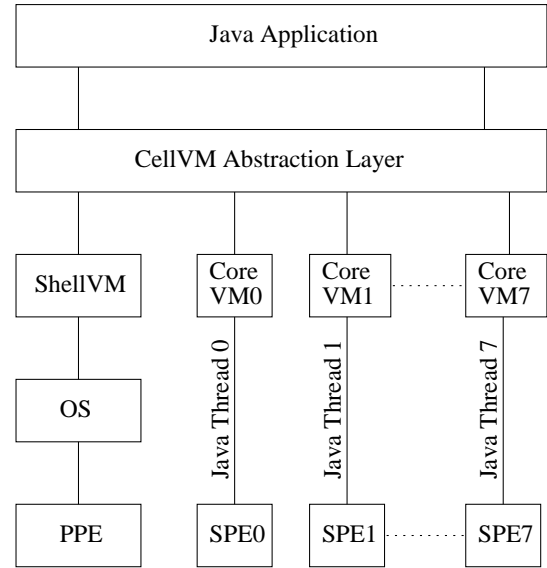


Figure 3. The basic design goal of CellVM is to abstract away the heterogeneity of the underlying hardware architecture. Regular Java programs can be executed on the Cell processor by distributing Java threads to the SPEs. The PPE services the small number of bytecode instructions that cannot be handled locally on an SPE, such as object creation.

While our prototype system focuses on using SPEs for the execution of Java code, it would be conceivable to use the PPE in addition to the SPEs to run Java threads whenever no co-interpretation requests are pending from any SPE. Our prototype also currently does not handle more than 8 concurrent Java threads. This is not an inherent limitation but merely an implementation shortcut. A mechanism to multiplex SPEs between Java threads could be added with relative ease, in particular because the actual CoreVM state that would have to be stored and retrieved during a context switch is very small in comparison to the actual hardware context of SPEs (which is several kilobytes in size).

Only a small percentage of opcodes cannot be handled by CoreVM. All opcodes that create new objects, including arrays, have to be executed by ShellVM since the entire Java heap (where the new objects are stored) resides in main memory. For this reason also object locking and unlocking is performed by ShellVM. Also, native method invocations are exclusively handled by ShellVM for good reasons: First, native method invocations are usually rare in regular Java applications. Executing them locally on CoreVM would require that the compiled code resides in local storage, which in turn reduces the available space on local store for instruction and data caching. Second, if the native method uses or manipulates the Java heap (e.g. array copying), a switch from CoreVM to ShellVM cannot be avoided anyway. ShellVM’s tasks also includes the resolving of references that are not known at compile time. Finally, code-preparation, which will be explained in Section 3.2 is also done on the PPE.

3.1 Overall Architecture

To implement our approach, we extended JamVM [13], an open source Java Virtual Machine. We chose JamVM amongst available alternatives because of its very compact size, which is important since a copy of the JamVM interpreter needs to reside in the 256kB local store of each SPE and subtracts from the local memory available for storing data.

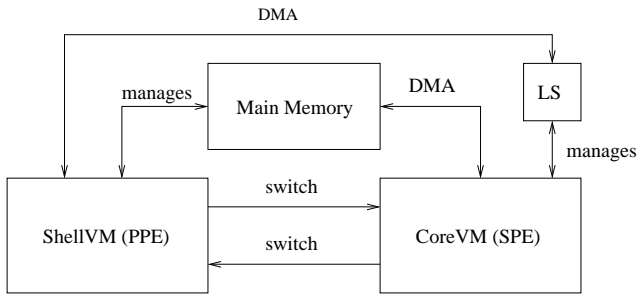


Figure 4. Co-execution between the ShellVM (on the PPE) and a CoreVM (on an SPE).

JamVM includes several different interpreter types. A “switch interpreter” dispatches instructions using a loop-embedded `switch` statement at which each opcode is represented by one case label. This approach entails large performance drawbacks [9] particularly for the SPE architecture since it does not possess a branch prediction unit and at least three machine-level branch instructions are executed per bytecode instruction. For this reason we decided to use JamVM’s *direct threaded* interpreter. In this implementation, each bytecode instruction is replaced by the address of the corresponding code block implementing the corresponding bytecode effect. Consequently, the number of machine-level branches per opcode can be reduced to one. Furthermore, JamVM uses *opcode rewriting*, which can be used to store resolved information in the operand of an instruction (see Section 3.2).

CellVM effectively consists of two separate programs (Figure 4), compiled by two different compiler tool chains. The ShellVM is intended to run on PPE and the CoreVM is loaded onto each SPE. The ShellVM manages most of CellVM’s data structures, in particular those that require large storage area such as the Java heap, constant pool and method area. CoreVM is the main bytecode interpreter and responsible for managing the Java stack, JVM registers and local caches, all of which are private to each SPE.

A major design goal of CellVM was to use the limited size of the local store efficiently because transferring data from main memory to local store via DMA incurs a latency of several hundred cycles, depending on message size. Hence not all runtime data structures of the Java Virtual Machine can be held in the local store as illustrated in Figure 5. The Java Virtual Machine Stack is analogous to a stack of a conventional programming language such as C. It holds operands, computational results and arguments for method invocation and method return values. Each CoreVM maintains a set of internal registers describing the current state of the VM, including a program counter, a pointer to the current local variable set, a pointer to the top of the Java stack and a reference to the current Java frame.

Each time a new method is invoked by CoreVM a new Java frame is created. It holds all necessary information required for executing that particular method. It holds a pointer to the method code, which resides in main memory, and the corresponding code size. This information is used to initiate a DMA transfer, which copies the entire method code to the bytecode cache (see Section 4). Additionally, a pointer to the code in the cache is stored, which is used when returning from a method in case the code is still present in the cache. Also the access flags of the current method are stored in the CoreVM frame, since in case of a synchronized or native method invocation a switch of control to ShellVM has to be initialized. Finally, a pointer to the previous frame is used when returning from a method.

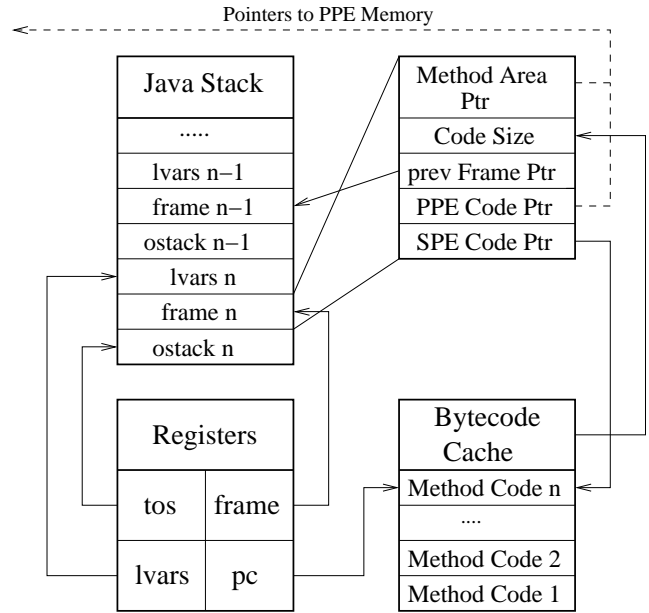


Figure 5. Since the size of the local store is limited to 256K, small and frequently used data structures are kept in main memory. The Java stack resembles a stack of a conventional programming language and is entirely kept inside the local store. Each CoreVM also has a dedicated set of VM-internal registers, which are used to save the current state of the VM in case execution is transferred to the PPE to execute a complex bytecode instruction.

Starting up CellVM can be divided into two phases: First, the ShellVM initializing process has to be completed, which includes initializing the ShellVM data areas (Java heap, native methods stack, etc.), and preparation of the CoreVM interpreter. Before starting up a CoreVM, all required information is stored in a data structure and the current method code is being prepared (see Section 3.2). Then, an SPE-thread is started and the initializing process of CoreVM, which represents the second phase can be performed. The second phase includes initialization of the caches and the transfer of the Java Frame information and the bytecodes. Once the two mandatory DMA transfers have completed, CoreVM starts executing bytecodes until a switch to ShellVM is necessary.

Communication between CoreVM and ShellVM is realized through four mechanisms that have various latencies. The most expensive way of communication is a switch of execution from CoreVM to ShellVM. In case CoreVM has to ask for assistance for executing an opcode, a snapshot of the internal state (program counter, current method, etc.) of CoreVM has to be updated into a data structure called *mailbox*. The mailbox is transferred via a single DMA to main memory. This information is used to restore the current state of the requesting CoreVM on the PPE. After the DMA transfer is completed, an interrupt is raised on the PPE (by writing a 32-bit message to a special register) and the requested instruction is executed by ShellVM. Meanwhile, the CoreVM waits for the ShellVM to complete the request. After completion, ShellVM notifies the appropriate CoreVM by sending a message to a memory-mapped I/O register. The CoreVM again initializes a DMA transfer, adapts its internal state and continues with execution. Summing up, we need two DMA transfers, two notification messages and the time used for executing the request before CoreVM can continue executing the next instruction.

The second mechanism, used for locking and unlocking of objects, only needs three communication messages. We need to send two messages to ShellVM, using two different destination registers, whereas one message contains the instruction to either lock or unlock the object and the other the 32-bit address of the object. After executing the request, the third message (sent by ShellVM) notifies CoreVM of its' completion. Although we also switch execution from CoreVM to ShellVM, we can save two DMA transfers and instead send one more message.

The third, still less expensive mechanism is a blocking DMA transfer, which is mainly used for copying data and instruction from main memory to local store. As described in Section 2 the Cell architecture is capable of issuing DMA transfers in parallel to execution. However, this mechanism is not applicable when reading data from main memory, since it must be guaranteed that requested data is already in place before execution can be continued. In theory, pre-fetching of data or instructions, realized by a non blocking DMA transfer would be possible. However, implementing this mechanism is very hard to realize since sophisticated knowledge of future program behavior is required.

Issuing a non-blocking DMA transfer allowing copying data from or to local storage in parallel to execution can be realized without significant latency. The MFC of each SPE is able to manage up to 16 DMA transfers in a queue. We have implemented a buffer that temporarily holds data that will be copied back to main memory (write DMA transfer). The current version of CellVM does not use non-blocking DMA transfers for copying data from main memory to local store (read DMA transfer) since the capacity of 16kB is sufficient for all read DMA transfers in all applications tested so far.

3.2 Code Optimization Strategies

In order to achieve reasonable performance, two requirements have to be fulfilled: First, the number of DMA transfers has to be kept at a minimum. Second, switching from SPE to PPE should be avoided whenever possible because latencies due to context switching, mandatory DMA transfers and mailbox communication occur. In addition to avoiding these latencies, it is also important to offload as much of the workload to the SPEs as possible, since all SPEs compete for the attention of the PPE when issuing requests for assistance. If there were too many parallel requests, the PPE could become a performance bottleneck.

In order to meet these design requirements, we use software-managed caches (see Section 4) to cache frequently used data structures and instructions in the SPEs' local stores. Furthermore, strategies such as code preparation and opcode rewriting are adopted to reduce the number of DMA transfers and switches.

Before a method is invoked on CoreVM, the bytecode of the corresponding method is prepared by ShellVM. The opcode LDC (load constant), for example, loads a constant value from the constant pool and puts it on the Java operand stack, which is maintained locally in the local store of an SPE. Before code preparation, as specified in the Java Virtual Machine Specification [16], the LDC bytecode is followed by an index that specifies an offset into the runtime constant pool of the method being currently executed. Instead of having to perform this lookup every time the LDC instruction is executed, it is possible to replace the index by the actual lookup result, which is a direct pointer to the data item, or even the data item itself. This brings significant performance gains for CellVM, because each time the prepared bytecode LDC is executed, we can suppress a blocking read DMA transfer to the constant pool.

Code preparation can be performed for various opcodes, which also saves numerous switches from SPE to PPE. References that do not change during the entire runtime of the application can be resolved in the preparation phase and stored into the operand.

In our specific implementation of the JVM, we reserve for each bytecode 8 bytes to store dynamically resolved information. While this does inflate the size of the bytecode program significantly, and thus limits the number of methods we can cache in the local store before we have to evict a method from the code cache, it does have the advantage that the number of DMA transfers and context switches can be reduced dramatically. Furthermore, the operands can be used to save a cache lookup. The opcode ARRAYLENGTH, for example, determines the length of an array and puts it onto the operand stack. We have installed a cache that holds the lengths of the arrays that are local to the SPE's private storage area.

In our implementation, code preparation is split into two parts, a pre-preparation phase performed by the PPE, and a post-preparation phase performed by the SPE after the method bytecode has been cached. The post-preparation phase enters the local address of the machine code sequence implementing the bytecode into the instruction stream. This address is known only to the SPE and thus cannot be filled in by the PPE.

Opcode rewriting is another method of minimizing communication between CoreVM and ShellVM. A dynamic opcode rewriting mechanism is used by the SPE to replace opcodes on the fly with so-called "quick opcodes" which update instructions with dynamically derived information. The opcode GETSTATIC, for example, can be rewritten by resolving the reference to the constant pool and storing it in the attached operand. The next time this particular opcode is executed, the reference can be used to initiate a DMA transfer and gather the actual value from main memory. Opcode rewriting is entirely done locally on each SPE with the instructions stored in the bytecode cache. As a consequence, no synchronization overhead in a multithreaded application is required. The drawback however is that resolved information is lost in case the cache has to be purged. This drawback is kept at a minimum in our implementation because the cache miss rate is negligibly small (see Section 5.2).

Another way of reducing mandatory switches from CoreVM to ShellVM is by extension of the instruction set of the JVM through the introduction of new opcodes. As CellVM uses GNU classpath version 0.91 and numerous function calls to the "math" library are "native" methods (for example $\sin(x)$, \sqrt{x} , etc.) we decided to introduce new *math opcodes*. ShellVM is able to detect a function call to the math library and tells CoreVM to rewrite the native function call into a corresponding math opcode, which can be executed locally on an SPE.

3.3 Limitations of CellVM

Since CellVM is not able to distribute workload of a single threaded application to multiple execution units, maximum performance can only be achieved if the executing Java program actually uses eight parallel threads. Since every Java thread is physically bound to one SPE, a Java program that has more than eight parallel threads will have 8 such threads running on SPEs and the remaining threads need to execute on the PPE. Our current implementation does not have any heuristic to route threads to SPEs vs. the PPE in such cases. Also using native or synchronized methods in applications' "hot spots" makes CellVM intolerably slow, since switches from CoreVM to ShellVM have to be issued.

Our current prototype does not offer a cache coherent view of memory for threads running on SPEs. Each SPE caches parts of the main memory in its local store. When SPEs update the same memory location concurrently, there is no guarantee which SPE will prevail with its memory write. Since SPE caches are speculative, collisions can also occur if SPEs operate on main memory locations near to each other since cached but actually unchanged values might be written back into main memory, potentially overwriting changes made by another SPE.

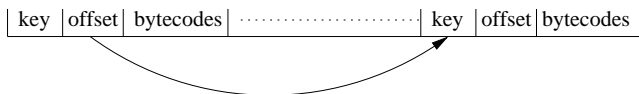


Figure 6. The instruction cache is organized as an array. The key of a method and the offset to the next key can be read using the *d-form* addressing mode without the need for explicit address arithmetic.

Several avenues exist to remedy this issue. In hardware multiprocessors, cache coherency is often implemented through cache coherency protocols. Another common approach is to introduce explicit memory barriers that provide certain coherency guarantees only when needed. We are currently investigating how these and similar mechanisms can be implemented in software on the SPEs.

4. Automated Software-Controlled Memory Management of SPEs' Local Stores

The local store of an SPE is used to hold data and instructions required for program execution. Organizing the limited local store efficiently is therefore very important in terms of avoiding processor stalls due to missing instructions or data. The main difficulty of caching main memory regions in the local store is that its very difficult to predict which memory regions a program is going to access next. Moreover, even a fairly trivial and imprecise prediction algorithm requires significant memory access monitoring, which we have to provide in software. Instead of implementing a general software-driven main memory cache, we decided to only cache specific data structure for which we have a solid understanding of their access pattern in the local store.

The local store is pre-allocated to dedicated cache pools at compile time. We use static pool sizes because dynamic cache pools require utilizing dynamic memory allocation on the SPEs, which we found to be very slow on the SPE. Performance measurements on IBM's cycle-accurate and memory-timing accurate Cell simulator [12] indicate that both malloc and free are very expensive on the SPE. Allocating and freeing 1KB of local store memory takes 1428 cycles, 546 cycles alone due to branch misses and 426 due to data dependencies. It is unlikely that this overhead could be reduced significantly merely through a more efficient implementation. The SPE is simply not designed for the kind of pointer-chasing required for implementing a dynamic memory pool.

The *instruction cache* stores bytecodes of multiple methods in local storage. The cache is organized as a big array, as illustrated in Figure 6. Index zero contains the key of the first method, which is given by the address of the corresponding method in main memory. Index one, which represents the code size, gives an offset to the next key in the instruction cache array. Index two contains the first bytecode. A lookup in the instruction cache is performed by reading the first key at index zero and comparing it with the address of the new method. In case of a cache hit, a pointer to the first bytecode is returned. Otherwise, the offset at position one is used to find the next key in the instruction cache array. Organizing the cache as a big array has the advantage, that array elements can be accessed using the *d-form* addressing mode. At this, the load-address is determined from the sum of a register and an immediate offset (e.g.: `lqd(load-addr,offset)`). Using the *d-form* to access array elements saves the overhead stemming from incrementing the array pointer. The disadvantage however is that if the cache has reached its limit, the whole cache has to be purged, since it would become fragmented. A high instruction cache hit rate is very important,

since all dynamically resolved information is lost in case the cache has to be purged.

The data cache is divided into three different types: the array cache, single-key caches and dual-key-caches. The array cache consists of a previously defined number of elements. Each element maintains three keys. One key is given by the starting address of the array in main memory, the other two represent the start index and the stop index of the cached array fraction. An array cache lookup is performed by comparing the corresponding address with key1 and checking, whether the index lies in between key2 and key3. Furthermore, each element also stores the array length, which is used to perform a bounds check of the entire cached region at once. This results in performance gains especially for the SPE architecture, since a cache hit guarantees, that the requested index lies within the array. Consequently an *if statement* can be saved, which in case of miss-prediction results in a penalty of 18-19 cycles. The data field of cache element is used to hold the array data. If the data field is not large enough to hold the entire array, only a fraction of the array is loaded onto local store. We also implemented an algorithm, which is able to detect, whether the elements are accessed by incrementing or decrementing the index. However, this strategy fails, if the number of frequently accessed arrays exceeds the number of array cache elements, since with each cache miss one cache element is replaced by a new one. Also if array accesses are performed randomly (and only a fraction of the array can be cached) a successful caching strategy can hardly be implemented.

Single-key caches are used for the JVMIL opcodes GETSTATIC, PUTSTATIC, GETFIELD, PUTFIELD and ARRAYLENGTH. A key, which is uniquely defined by the address in main memory, is assigned to the corresponding value. If the opcode GETSTATIC, for example is executed, a lookup into the cache is performed. In case of a cache hit, a read DMA-transfer can be saved. In case of a cache miss, an existing cache element is replaced by a new element, by writing the content of the element to be replaced back to main memory and obtaining the new value from main memory. Although two DMA transfers have to be performed, the writing back can be done in parallel with execution. Dual-key caches are used for the opcodes TABLESWITCH and INVOKEVIRTUAL. The caching of a virtual method invocation, for example, is done by assigning the address of the class to key one and the index into the method table to key two. The value in this case is a data structure called method-block, which holds all necessary information for method invocation, such as access flags or code address.

The caches are optimized for the SPE-architecture in terms of accessing keys and data (except bytecodes and array data) with qwords only. Scalar (subquadword) loads and stores require several instructions to format the data for the use on the SIMD architecture of the SPE. Scalar loads must be placed into the preferred slot. Scalar stores require a read, scalar insert and write operation. These extra formatting instructions reduce performance.

5. Benchmarks

To evaluate the performance of our prototype implementation, we conducted a series of benchmarks using the Java Grande Benchmark Suite [15] (sequential version). The baseline for our benchmarks is JamVM running on the PPE of the Cell, and that baseline is compared to running 1 to 8 Java threads in parallel using the 8 SPEs of the Cell processor. The benchmarks were conducted on a Cell blade server courtesy of IBM, running a 2.6.14 Linux kernel populated with eight SPEs.

To get a basic understand of the performance characteristics of CellVM, we ran a series of micro benchmarks using the Java Grande Section 1 benchmark suite. Section 1 contains small bench-

Benchmark	JamVM	CellVM	Difference [%]
loop	9.002.10	10.180.71	112.92%
arithmetic	8.063.43	6.633.55	82.23%
method invocation	3.786.27	383.66	10.13%
math opcodes	863.578	1.574.535	182.33%

Figure 7. Basic performance evaluation of CellVM using Java Grande Section 1 benchmarks. The values are performed operations per second.

mark programs measuring low-level VM operations such as casting, arithmetic operations, and method calls.

Following the micro benchmarks we used Section 2 of the Java Grande benchmark suite to evaluate the performance of CellVM for a series of real-world computation intensive kernels. Each benchmark was scaled up from running on a single SPE to running in parallel on all 8 SPEs of the Cell. Since CellVM currently does not offer a cache-coherent view of main memory, we executed 8 instances of the serial version of the Java Grande benchmark in parallel instead of using the multi-threaded version. This was necessary because the multithreaded version of Java Grande uses shared memory for thread synchronization, which fails due to the weak consistency model of CellVM.

5.1 Basic Performance Evaluation

The results for the Java Grande micro benchmarks (Section 1) are shown in Figure 7. In this benchmark, a single SPE was pitted against the PPE of the Cell processor. The results are noteworthy, considering that the SPE occupies significantly less chip space on the die and runs at a lower clock rate than the PPE core.

In case of the loop benchmark the Java thread running on an SPE outperforms JamVM’s performance on the PPE. For the arithmetic micro benchmark, which performs a series of integer and floating point additions, subtractions, multiplications, and divisions, CellVM’s performance is near the performance of JamVM (82%). As in case of the loop benchmark, the CoreVM can run the entire benchmark on the SPE without intervention from the PPE, which explains the solid performance results.

The method invocation benchmark is much less favorable for CellVM. Since each method invocation requires either consultation with the ShellVM or a DMA transfer, performance is degraded significantly. Fortunately, in actual application code, method invocations are usually not predominant since they are expensive in for a simple Java VM such JamVM as well. While certainly not as expensive as for CellVM, this additional cost is apparently sufficient to discourage programmers from excessive use of method invocations without performing any worth-while calculations inside the method body.

It is also important to note that the method invocation benchmark measures the worst case scenario for CellVM: invoking synchronized methods. In case of first time (method code is not resident in instruction cache) regular method invocations CellVM has to consult PPE in order to dynamically resolve information. When the same instruction is executed a second time, this information can be used to transfer the required information for method invocation via DMA. Returning from that method is entirely handled by SPE. In case of a synchronized method, the SPE has to yield back to PPE for both, method invocation (object lock) and method return (lock release). This overhead could be reduced by exploring direct SPE-to-SPE synchronization in favor of performing all synchronization exclusively on the PPE (which is the case in our current prototype).

Since method invocations are so expensive, performance can be improved dramatically by inlining method calls in such a way the SPE can execute the called method without intervention from the

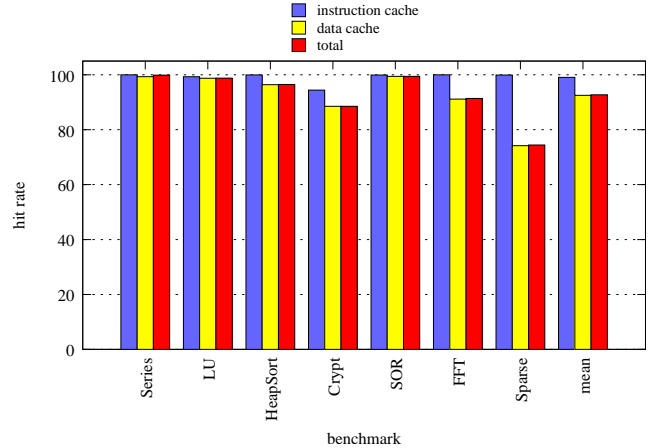


Figure 9. Instruction and data cache hit rate for each benchmark program.

PPE. CellVM is able to detect and rewrite the bytecode for 23 frequently used mathematical operations such as `sin`, `cos`, `tan`, `sqrt` and `pow`. In the standard JamVM interpreter, these operations are implemented as calls to native methods in the Java library. This is of course sub-optimal for CellVM, which would have to consult with the PPE for every mathematical operation. Instead, CellVM detects these method calls and rewrites them to internal opcodes which directly implement the mathematical operation. This yields a significant performance boost, allowing CellVM to outperform the PPE/JamVM by 82.33%. Without this optimization, the PPE/JamVM performed the mathematical micro benchmark 134 times faster than CellVM.

5.2 Application Performance Evaluation

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 a large amount of opcodes has to be handled by ShellVM, the PPE can become congested by too many requests, since all CoreVMs are competing for the ShellVM’s attention. Second, if the amount of blocking read DMA transfers is comparably high, long latencies occur, since CoreVM cannot continue with execution until the DMA transfer has completed. Finally, if the total number of pending DMA requests exceeds the MFC’s DMA queue size, the SPE has to stall until a slot in the queue becomes available.

Our Benchmarks were performed using 1–8 threads, each of them executing its own instance of the benchmark concurrently. CellVM was configured with a Java Stack Size of 20KByte, an instruction cache of 40Kbyte and a data cache of 10 Kbyte. The performance, when using the multi-threaded version, was calculated by adding up the results of each thread. Figure 8 shows that the performance of each benchmark increases linearly with the number of used threads, which is a strong indicator that neither the PPE nor the EIB are congested.

Figure 10 gives an overview of the issued DMA transfers of all tested applications. While blocking read DMA transfers and blocking write DMA transfers have long latencies, the non-blocking write DMA transfers can be issued in parallel with execution. The number of blocking write DMA transfers is negligibly small and mainly used for writing CoreVM specific internal data to main memory, which is used to restore the actual state on the ShellVM. This is implemented as a blocking DMA transfer, since it has to be

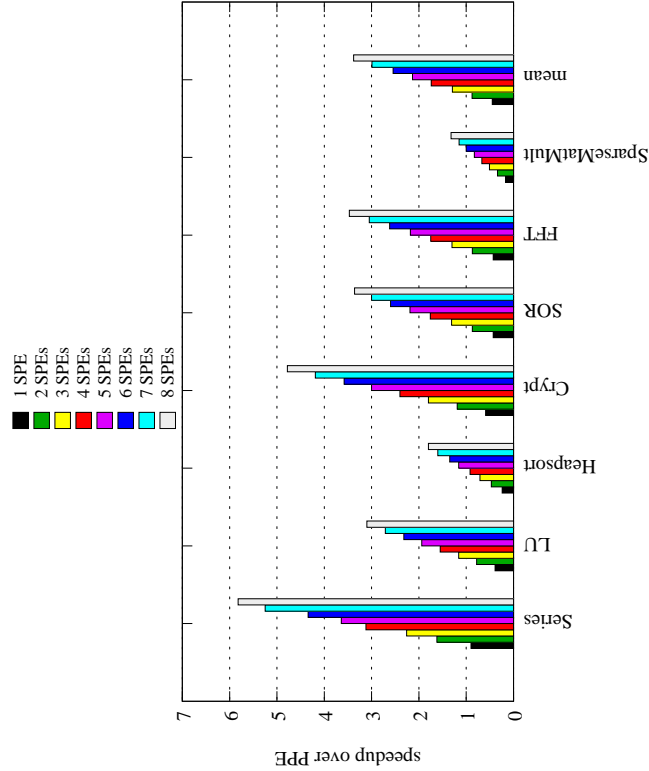


Figure 8. Speedup for Java Grande Section 2 benchmarks relative to the performance of JamVM on the PPE when using 1 to 8 parallel SPE threads.

Benchmark	reads	blocking writes	non-blocking writes	total [million]
Series	99.83%	0.04%	0.13%	480
LU	7.99%	0.03%	91.98%	752
HeapSort	61.03%	0.02%	38.95%	1.120
Crypt	61.95%	0.10%	37.95%	184
SOR	8.18%	0.01%	91.81%	1.752
FFT	38.40%	0.0042%	61.60%	5.008
Sparse	84.40%	0.0037%	15.60%	9.128

Figure 10. Number of DMA reads, and blocking and non blocking writes for each benchmark program. Almost all writes are non-blocking and can complete while the SPE continues with program execution.

ensured that data is already in place before the ShellVM can execute the request. In most benchmark programs, such non-blocking DMA writes constitute a large fraction of bus transfers (Figure 11).

Figure 9 gives an overview of instruction and data cache hit rates of every single benchmark.

The number of switches from CoreVM to ShellVM is negligibly small for all tested applications. Over 99.99% of all opcodes are executed by CoreVM. This is can be traced back to the high instruction cache hit rate and our VM's extraordinary efficiency in rewriting opcodes to quick-opcodes.

The Series benchmarks, which calculates the first 10k coefficients of the function $f(x) = (x+1)^x$ is characterized by an extensive use of transcendental and trigonometric functions. Although this benchmark has the third most DMA transfers, it performs best. This is mainly because it profits from the frequent use of the new math opcodes. In comparison to JamVM running on the PPE only, where most of the calls into the math-library are native function calls, CellVM is able to execute these instructions directly.

The second best performance results were achieved by the IDEA encryption benchmark. Although the cache hit rate is only 88%, this benchmark has the lowest number of DMA transfers. The low cache hit rate does not have a great impact on total performance, since in comparison to other benchmarks, the caches are used infrequently.

The sparse matrix multiplication benchmark shows the worst performance results. This is mainly because array accesses are performed randomly, which makes efficient data caching impossible. Enlarging the array cache results in an even worse performance. Due to the fact that available space in the local store is not large enough to hold the entire array, enlarging the number of cache elements simply increases the overhead when performing a cache lookup. Also, increasing the data field of a cache element reduces performance since the latency that comes with a DMA transfer depends on the message size.

In order to demonstrate the efficiency of our optimization strategies, we ran the LUFact benchmark with different caching set-

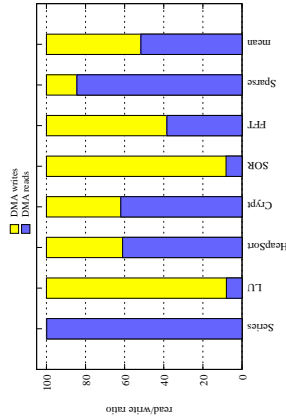


Figure 11. Ratio of DMA reads to DMA writes. DMA writes can complete concurrently while program execution continues, whereas DMA reads stall the execution.

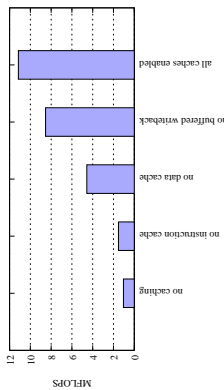


Figure 12. Performance of the LUFact benchmark for different caching settings.

tings. Figure 12 shows the resulting performance. Deactivating the instruction cache results in a performance loss by a factor of 7. While the number of read DMA transfers due to missing instructions only slightly increases by 0.6%, the overhead from switching back and forth between CoreVM and ShellVM to re-fetch method code (which is now not cached) reduces performance dramatically. When performing the benchmark with instruction cache, only 190×10^3 switches have to be performed. Disabling the instruction cache results in 2.2×10^6 switches, which is an increase by a factor of 12.

Running the benchmark without data cache slows the application down by a factor of 2.4. This is mainly caused by the high array cache hit rate of 97.8%. When conducting the benchmark with data cache, we only have 8.6 read DMA transfers per 1000 executed opcodes. This number increases to 450 read DMA transfers per 1000 executed opcodes.

Finally, disabling the buffered write-back mechanism results in an performance decrease by 24% since the majority of all DMA transfers (92%) can be issued in parallel with execution. The overall performance loss is a factor of 11.

6. Related Work

Using Java to develop parallel and distributed applications has become increasingly popular over the past years. Jameela Al Jaroodi et al. [1] give a survey of current parallel Java projects. Existing approaches can be roughly divided into three categories.

The first category of systems replaces the standard JVM by building a new system or using the current available infrastructure for parallelism. Titanium [17], for example, developed at the University of California, Berkley is a Java dialect used for large scale scientific applications. It is a compiler based approach, which compiles Titanium programs into C code. It has the advantage, that programs written for a shared memory model can be executed on distributed systems without modification. In contrast to our system, Titanium is not compatible with JVM, and cannot execute standard Java programs.

The Java Parallel Virtual Machine [7] (JPVM) is entirely written in java and provides explicit message passing based distributed memory MIMD parallel programming. Benchmarking a (512×512) matrix multiplication resulted in a speed-up of 6.7 using 9 processors. Code written for the JPVM can not be ported to the JVM. Being not compatible to JVM gives developers more freedom to exploit parallelism of the underlying system, but on the other hand entails the drawback, that portability is lost.

The second category of parallel Java systems extends Java with classes and libraries to provide explicit parallelization functions instead of replacing the entire JVM. ParaWeb [3], for example, permits the utilization of computational resources of the internet. It allows users to upload programs and execute them on multiple machines in a heterogeneous environment. The drawback of this approach is of course, that a large overhead arises caused by synchronization between the executing nodes. ParaWeb is related to our work to the extend that it enables parallel programming with Java while still maintaining compatibility with the original JVM.

The third category of parallel Java systems aims at providing seamless parallelization for multi-threaded applications. This approach does not require programmers to take care of the parallelization process. Any Java multi-threaded application can be run in parallel on a distributed system. The advantage of this approach is, that conventional Java applications can be run without any changes. CellVM falls into this third category. Other examples include cJVM [2]. cJVM is a clustered Java virtual machine that distributes multiple java threads to nodes of a cluster. The disadvantage however is, that optimizations in communication, synchronization and locality are difficult. CellVM is likely able to outperform cJVM, because its execution units are all located on the same die and thus communication and synchronization cost are greatly reduced.

Investigations on automated optimizations techniques on the compiler level for the Cell architecture were investigated by Eichenberger et al. [6]. They present optimization techniques in compiler based branch predication, instruction fetch, and generation of scalar codes on SIMD units. The penalty for a miss-predicted branch on the SPE is given by 18-19 cycles, because branches are detected at a time where already multiple fall-instructions are in flight. One way of reducing miss-predictions is the conversion of a *if-then-else* statement into a select instruction. An alternative is given by the determination of the likely outcome of a branch. This can be achieved either by doing compiler analysis or by user-directives. Furthermore the SPE provides a branch hint instruction, which specifies the location of a branch at it's likely target. Likely branch outcomes can be determined by branch profiling, which is done statically, or manually by the programmer.

Instruction fetch on the SPE is divided into two subtasks: instruction scheduling and bundling. The main task of the scheduler is to determine instructions on a critical path and give them high-

est priority. While the constraints of ordinary instruction schedulers are latencies and the use of resources the SPE additionally takes the number of instructions into consideration. For example a branch-hint instruction must not be too far away (> 256 instructions) from the branch target and also not too close (< 8 instructions). The challenge of the instruction bundling subtask is to ensure, that a pair of instructions, which are expected to be dual-issued satisfies the SPE's instruction issue constraints, since the SPE is only able to execute two instructions in parallel, if they are assigned either to the even or to the odd pipeline, as described in Section 2.2. For best performance, the authors propose a unified scheduling and bundling phase.

Due to the nature of SPE's SIMD architecture executing scalar code is not efficient. For example loading and storing of sub-quadword data types is affected with several dependent instructions [4]. The authors propose several steps to overcome this problem: First, they allocate all scalar variables in the primary slot of their own 128-bit chunk of memory. Then, they perform aggressive register allocation to make good use of the large unified register file. As a result, most variables reside in the primary slot of their registers. Consequently, the expensive load and store instructions are not needed. Besides that, they attempt the automatically simdize the code.

Another important performance issue on the Cell architecture is the use of single, double and triple buffering schemes, which hide the latency stemmed by DMA transfers by overlapping the computation with communication. Tong Chen et al. [5] and Yuan Zhao et al. [20] present various techniques to overcome this problem. CellVM uses a buffer when writing from local store to main memory.

Also work independent from the Cell architecture can be found in literature, which addresses software controlled caching. Chia-Lin Yang et al. [19] for example propose a software controlled cache architecture, which distributes data types to different cache regions. When a specific data type is needed, only the designated cache region needs to be activated. Measurements of appropriate applications, such as MPEG-2 decoder show energy savings up to 40%.

7. Conclusion and Outlook

We have presented CellVM, a Java virtual machine that executes Java programs cooperatively on different cores of a heterogeneous multi-processor. In our design, the most common Java bytecode instructions are executed directly on the throughput-oriented SPU cores, while more complex instructions and those that require main memory interaction are processed by the main PPU core.

In order to mask expensive data transfers between main memory and the throughput-oriented SPUs, we have devised a software-based caching strategy that maintains local copies of frequently accessed data structures such as method bytecodes and arrays in SPU-local memory. Our experiments show that this caching strategy is effective.

Experiments also show that cooperative interpretation is able to distribute most of the activity to SPUs, so that even 8 simultaneous Java threads running on 8 SPUs do not saturate the PPU core with service requests. Using all 8 SPUs in parallel, our system was able to significantly outperform pure PPU-bound interpretation.

We plan to continue working on CellVM. The current SPU-interpreter (CoreVM) is a close derivate of JamVM and implemented in C. Its performance is well below the theoretical peak performance, which is mostly due to alignment constraints. SPUs are designed to operate on vectors, and not scalar values. Since the JamVM interpreter was not written with this fact in mind, every stack access (and thus every operand access) currently incurs a significant overhead. We plan to eliminate this overhead by organizing

the Java stack as a stack of vectors, instead of as a stack of words, and using SPU-specific intrinsics instead of regular C-code.

Another area of improvement is dynamic compilation. CellVM is currently a pure interpreter. To compete with compiling systems (such as IBM's PowerPC JVM), we will have to add a dynamic compiler to the system that emits SPU code. It is an open research question by how much the reduced overhead due to eliminating interpretation will be outweighed by the negative impact of additional DMA transfers during compilation.

References

- [1] J. Al-Jaroodi, N. Mohamed, H. Jiang, and D. Swanson. A comparative study of parallel and distributed java projecys for heterogeneous systems. In *Proceedings of the International Parallel and Distributing Processing Symposium*, 2002.
- [2] 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.
- [3] 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.
- [4] D. A. Brokenshire. Maximizing the power of the Cell Broadband Engine processor: 25 tips to optimal application performance. Technical report, 2006.
- [5] 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.
- [6] 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: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.
- [7] A. J. Ferrari. JPVM: Network Parallel Computing in Java. Technical report, Charlottesville, VA, USA, 1997.
- [8] B. Flachs, S. Asano, H. Dhong, S.H Hofstee, G. Gervais, R. K., T. Le, L. Peichun, J. Leenstra, J. Liberty, B. Michael, O. Hwa-Joon, S. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, N. Yano, D. Brokenshire, M. Peyravian, T. Vandung, and E. Iwata. The Microarchitecture of the Synergistic Processor for a Cell Processor. *IEEE Journal of Solid-State Circuits*, Vol. 41, No. 1, January 2006.
- [9] E. Gagnon and L. Hendren. Effective Inline-Threaded Interpretation of Java Bytecode Using Preparation Sequences.
- [10] 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 Press.
- [11] H. P. Hofstee. Power Efficient Processor Architecture and the Cell Processor. *Proceedings of the 11th Intl Symposium on High-Performance Computer Architecture*, pages 258–262, 2005.
- [12] Ibm full-system simulator for the cell broadband engine processor ibm alpha works, 2006.
- [13] JamVM, A Compact Virtual Machine, <http://jamvm.sourceforge.net/>.
- [14] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Mauerer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research & Development*, 49(4/5):589–604, July/September 2005.
- [15] L. Smith and J. Bull. A Parallel Java Grande Benchmark Suite. *Supercomputing, ACM/IEEE 2001 Conference*, November 2001.
- [16] F. Y. Tim Lindholm. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Addison-Wesley, 1999.
- [17] Titanium, <http://titanium.cs.berkeley.edu/>.

- [18] W. Wulf and S. McKee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23:20–24, March 1995.
- [19] C. Yang, H. Tseng, C. Ho, and J. Wu. Software-controlled Cache Architecture for Energy Efficiency. *Circuits and Systems for Video Technology, IEEE Transactions on*, 15(5):634–644, 2005.
- [20] Y. Zhao and K. Kennedy. Dependence-based code generation for a cell processor. In *Proceedings of the 19th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2006)*. Springer-Verlag, Lecture Notes in Computer Science, 2006 to appear.