The Architecture of the DecentVM

Towards a Decentralized Virtual Machine for Many-Core Computing

Annette Bieniusa
University of Freiburg, Germany
bieniusa@informatik.uni-freiburg.de

Johannes Eickhold Thomas Fuhrmann
Technische Universität München, Germany
{jeick,fuhrmann}@so.in.tum.de

Abstract

Fully decentralized systems avoid bottlenecks and single points of failure. Thus, they can provide excellent scalability and very robust operation. The DecentVM is a fully decentralized, distributed virtual machine. Its simplified instruction set allows for a small VM code footprint. Its partitioned global address space (PGAS) memory model helps to easily create a single system image (SSI) across many processors and processor cores. Originally, the VM was designed for networks of embedded 8-bit processors. Meanwhile, it also aims at clusters of many core processors. This paper gives a brief overview of the DecentVM architecture.

Categories and Subject Descriptors C.1.4 [*Processor Architectures*]: Parallel Architectures—Distributed architectures; D.1.3 [*Programming Techniques*]: Concurrent Programming—Distributed programming; D.3.4 [*Programming Languages*]: Processors—Run-time environments

General Terms Design, Languages

Keywords distributed virtual machine, transactional memory, distributed shared memory, intermediate language

1. Introduction

The *Decent Virtual Machine (DecentVM)* is a research platform for distributed, decentralized systems. As such, it targets clusters of embedded systems [1] as well as multi- or many-core systems as found in areas of HPC [3].

It has originated from a light-weight virtual machine for 8 bit microcontrollers and distributed embedded applications. Hence, from its beginning, the following goals governed our design:

- small footprint in VM's code size;
- small memory demand, both for application code and at runtime;
- potentially large number of processing cores, each with limited per-core memory;
- bare-metal execution without any operating system;
- fault-tolerance for reliable execution; and
- transparent migration of code, data, and threads.

VMIL'10, 17-OCT-2010, Reno, USA

We have accomplished these goals by combining an off-line transcoder with a virtual machine that executes modified bytecode. The transcoder performs the verification and statically links the required classes of an application or library. Thereby, the resulting binary – a so-called *blob* – is significantly smaller than the original class files, and its execution requires a significantly less complex virtual machine. Both these effects result in the low memory footprint of DecentVM.

All libraries that are required by an application are identified via cryptographic hashes. Thus, an application's code can be deployed as if it was a statically linked executable. The software vendor can thus test the executable as it runs on the customer site without interference from different library versions. Together with the VM's lightweight native interface, this property helps the programmer to create robust and secure applications. Nevertheless, classes and libraries can also be bound dynamically as specified in the Java standard. However, because of stability and security issues, this mechanism is discouraged in the DecentVM.

Since the verification is offloaded to the transcoder, a malicious blob could jeopardize the VM's integrity. To avoid this, the VM's instruction set is designed so that it isolates the applications' heap memories from each other. This is especially important because the VM cannot rely on an OS to isolate applications. In particular, there is no way in DecentVM to forge a reference! All references are obtained by allocating new memory objects, or by reading an existing reference. Moreover, all memory access is object based. (Here, objects means both, an instance of a class and an array instance.) Each object access is checked to remain within the bounds of that object.

In general, distributed operation in a partitioned global address space (PGAS) model is ensured via references that do not only map to local memory addresses, but also to node identifiers. Any memory access then comes with a latency whose extent depends on the memory's position in the memory hierarchy. These latencies are especially important with respect to memory consistency. Thus, the DecentVM has been especially designed as a platform for *software transactional memory* (STM) systems, but it also supports traditional lock based consistency via monitors. The latter is emulated with the same VM instructions that implement the STM model: copying objects into the private memory area of a thread, and replacing existing objects with a new version once the commit has succeeded. The respective operations are emulated with library functions, which the transcoder inserts instead of the Java monitor bytecodes. The definition of the precise semantics of those library functions is still ongoing work in our group.

The rest of the paper is structured as follows: In section 2 we give a brief overview of the most relevant related work. In section 3 we describe the memory architecture upon which we have built

our VM. In section 4 we describe the main parts of the DecentVM architecture. In section 5 we briefly explain the blob concept for loading bundles of code into the DecentVM. In section 6 we present the transcoder that produces such blobs. In section 7 we briefly report on the current status of our implementation. In section 8 we conclude with a summary and an outlook to future work.

2. Related Work

Software distributed shared memory (SDSM) provides the foundation for a range of distributed Java virtual machines (dJVMs). SDSM systems are either page-based [17, 18] or object-based [26]. Object-based SDSMs avoid the false sharing problem of page-based ones. DecentVM follows the object-based approach as it fits object-oriented languages like Java more naturally. A detailed analysis by Fang et al. [10] shows the challenges and pitfalls of object-based SDSMs. Buck et al. [8] showed that the advantages of the page-based approach can outperform the object-based one.

With the rise of powerful dedicated and specialized co-processors like graphic cards, different extensions of the JVM model have been proposed to leverage their computational power. Detection and offloading of loops onto CUDA devices as an extension to the JikesVM is described in [14]. The CellVM [20] uses an interpreter for a subset of Java byte codes to leverage the VLIW co-processors of the Cell Broadband Engine Architecture [13]. Williams et al. [25] suggest further interpreter optimizations for the CellVM. Besides the improvements in performance, other aspects of the JVM, e. g. implementation of heap and concurrency, have to adapt to current directions in the evolution of multi- and many-core processors, too. Ungar and co-workers [24] explored how to implement a Java heap on a 56-core TILE-64 many-core system. DecentVM tackles the problem of hardware variety by implementing hardware-dependent kernel traps. Therefore, it can make use of dedicated computational units internally, yet provide a standardized interface for bytecode execution.

To cope with increasing diversity in support for concurrency by new hardware architectures, Marr and co-workers [19] explore the separation of abstract and concrete concurrency models. The architecture for DecentVM follows this trend by offering the programmer both monitor mutexes and software transactional memory primitives for memory consistency. The advantages of STM have been recently described by the Velox project [9].

An object in the heap of a VM represents only the last state after chain of modifications. Older versions of objects are usually overwritten and thus lost. The DecentVM keeps partial history information about older versions of objects in memory to support its STM algorithm. Lienhard and et al. [15] used a similar representation of historic object states to efficiently support back-in-time debugging in a modified Smalltalk virtual machine.

To improve Java application throughput and reduce management overhead, several projects try to remove the operating system from the classical stack of hardware – OS – JVM . Examples are the SquawkVM which runs on the bare metal, or Jnode [4] which implements an OS completely in Java. As DecentVM started out as a VM for 8 bit microcontrollors, it only requires a thin OS-like layer. Parts of the OS functionality has been integrated into the VM structure, whereas the hardware interaction layer could be moved into an extended version of a BIOS. An alternative solution could be to execute a JVM directly on a hypervisor [5, 21]. A hypervisor is a thin software layer below the OS that enables virtualization of the hardware to execute multiple guests operating systems on a single machine [6].

The concept of transcoded byte code and specialized modular packaging of deployment units in form of suite files is well-known and also used in other JVMs like the SquawkVM [23].

3. Memory Architecture

The distributed memory architecture plays a central role in the architecture of the DecentVM. The VM operates according to a model of three different levels of (shared) memory: private, local, and global memory. This model is chosen to resemble the OpenCL [12] memory model, but is adjusted for our needs. The DecentVM memory model reflects a potentially large number of processors, which all have their individual address space.

- *Private memory* is thread-local memory, i. e. it can only be accessed by a single VM thread. Typically, this memory type is mapped to the memory on a single-core micro-controller, or core-local memory on a multi-core processor.
- *Local memory* is shared between several VM (and hardware) threads that share the same address space. We assume that a hardware cache coherence system provides a test-and-set or compare-and-swap operation to implement memory consistency. Moreover, we assume mutual trust and (a limited amount of) fate-sharing.
- *Global memory* is shared by several hardware threads that do not have a common address space. Access to this memory type requires explicit inter-process communication. DecentVM uses global memory either via local object copies or via explicit writes, e. g. to acquire a lock on a remote object.

DecentVM distinguishes conceptually between these three kinds of memory. In practice, the entire memory space is comprised of the memory of all nodes. Thus, publishing a local object does not necessarily require an actual copy operation if the object remains on the same node.

The DecentVM's memory system is object based. Objects have an immutable size and type, which is determined upon the creation of the object. Besides its size and type information, each dynamic object is associated with garbage collection information, location information that allows access to remote objects, and a monitor reference for tracing which thread has unique access to the object. Object contents are stored as immutable versions. In the context of software transactional memory (cf. Sec. 4) this means that each successful commit creates a new version, which points to its direct predecessor version. Thus, the object itself points to the latest version.

The DecentVM memory layout distinguishes further between static and dynamic objects (cf. Fig. 1). A *static object* is identified by a unique name at code-level, for example, by the name of a class. Object instances that are created at run time, on the other hand, are *dynamic objects*. Access to dynamic objects is only possible when the accessing entity knows the according reference to an object.

Both, static and dynamic objects can have different visibility. Objects that are visible at the application level comprise the regular Java objects and arrays. Code and execution contexts are only visible at the VM level.

Figure 2 shows the memory architecture. The three different types of memory are shown as layers with the VM in the top layer. The object references that the VM deals with are managed via a *private reference map* (PRM). The PRM is a thread-local data structure. It contains references to objects that have just been created or modified. In a software transactional memory context, 'just' means 'within the current transaction'. In a lock based context, it means 'after the last memory access barrier'. In other words, the PRM manages objects (or modifications of objects) that are only visible to the current thread. As a consequence, private object

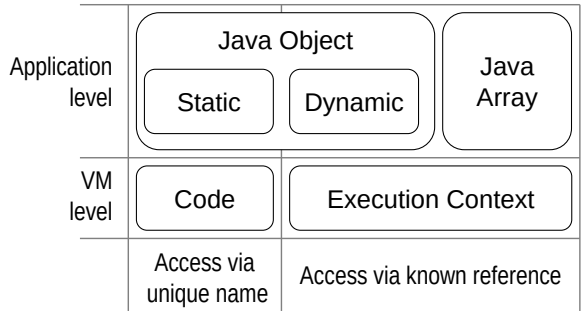

Figure 1. Object type overview.

references are only stored in the stack or in private objects. One may think of a private reference as an index in the PRM.

References to the PRM redirect within the PRM, to private objects in the local heap, or to globally accessible objects (GAOs). The latter are managed in the *global memory layer* (GML, see below).

When the VM allocates an object, it creates a PRM entry that points to the newly allocated memory slot. Figure 2 assumes that there is only one physical memory component which contains both, private and local objects. (In systems that offer a larger variety of memory types, private objects could be stored in a fast per-core memory.) If the PRM runs out of slots or the if the per-core memory is exhausted, the VM needs to offload a part of the private memory to the local heap memory. This is a rather complex process, which we are only about to specify. Thus, for simplicity, the current version of DecentVM signals an out-of-memory exception, as it would do when the heap is exhausted.

Object references that were present on the stack at the beginning of the current transactions are mapped to the GML (see below). Object references that were read from non-private objects are mapped within the PRM. Let us illustrate this with an example: Assume that we read field at offset 5 from object A, which is referenced by an indirection via the GML. This read creates an entry in the PRM that refers to (A, 5) without actually reading anything. Reading now, for example, the field at offset 4 from the same object, creates another entry in the PRM. Only when reading a numeric value from an indirection, we need to make use of the communication layer. The overhead of non-private communication is small, because the corresponding *read request message* contains the entire indirection path. The result of the read operation is stored in the private memory, so that future read operations on the same (part of the) object can be performed locally. A mask in the PRM entry indicates which parts of the object are present in the private memory and which still need to be fetched when they are requested. To further save on the number of read messages, the respond message contains as many fields (or array elements) as fit well into a frame of the underlying communication network.

When the current transaction successfully concludes, the PRM is garbage collected and then written to the GML. Assuming that we have a second core that can do that, the VM core can proceed the GC task asynchronously. In this case, the VM has to apply a copy-on-write within the private memory to preserve the atomicity of the transaction.

When writing objects to the GML, the memory system creates entries in the *local reference map* (LRM) that point to the respective objects in the heap. If the hardware does not distinguish between private and local memory, this step does not require actual copying. In that case, copy-on-write suffices.

The rest of the GML that is shown in Fig. 2 manages objects that are not present in the local address space. Please refer to our technical report [22] for an explanation of that part of the system.

4. The Architecture of the DecentVM

The DecentVM architecture and instruction set was designed such that an interpreter has a small code and memory footprint. Even though the design does not prescribe a particular hardware architecture, it was inspired by the Cell Broadband architecture – or, more generally, by RISC architectures that have a limited, but fast on-chip memory and the capability to communicate with a large, local bulk memory as well as with other such processors. Since the design originally targeted embedded devices, the architecture should also fit easily into an FPGA.

In this section, we highlight several issues in DecentVM's design and illustrate their interplay. Figure 3 shows a summary of the overall architecture.

4.1 Thread model

The top layer contains the per-thread state. We assume that each DecentVM implementation maintains the state for a constant number of threads in private memory. This improves the performance of the VM, given that the hardware supports fast per-core memory. If the actual number of threads exceeds the VM's limit, the DecentVM issues an interrupt so that the kernel software can offload one of the threads to the local memory. Such an offloaded thread cannot be executed until it is loaded again into the VM's thread state memory. While offloading the thread, all private references must be transformed into local references that can then be handled by the global memory layer (GML). Hence, the GML can transfer such threads onto another processor.

4.2 Shared memory

Following the JVM specification, DecentVM implements basic mutual exclusion with per-object monitors. Using these primitives in a correct way, i.e. without data races and dead-locks, can be surprisingly difficult in trivial settings. Software Transactional Memory has emerged as a promising alternative. Computations that need to be performed in isolation from other threads and on a consistent snapshot of shared data can be wrapped in a transaction. Similar to database transactions, the VM as the middleware is then responsible to provide non-conflicting access to shared data within the scope of a transaction. All modifications to shared memory are first performed on local versions of the shared objects. It is only visible to other threads after the transaction has committed successfully. Further, for read operations a consistent copy of the object is pulled from the global memory to provide repeatable reads even in case of intermediate updates by other threads. In case of conflict, the modifications are discarded, and the transaction is restarted.

DecentVM uses a versioning approach of transactional memory in combination with a randomized consensus protocol [7]. The start and end of a transaction can be seen as memory barriers that are also used to enforce the visibility of updates when using monitors.

In contrast to most other library-based STM implementations, DecentVM integrates STM into the VM internal memory management. This reduces the complexity of using transactions in combination with other synchronization mechanisms. The interplay of STM with monitors is on-going research in the context of DecentVM, as is the integration of non-revertible I/O operations.

4.3 Stacks

Each thread in the DecentVM maintains three stacks: one for numerical values, one for references, and one for frame pointers and

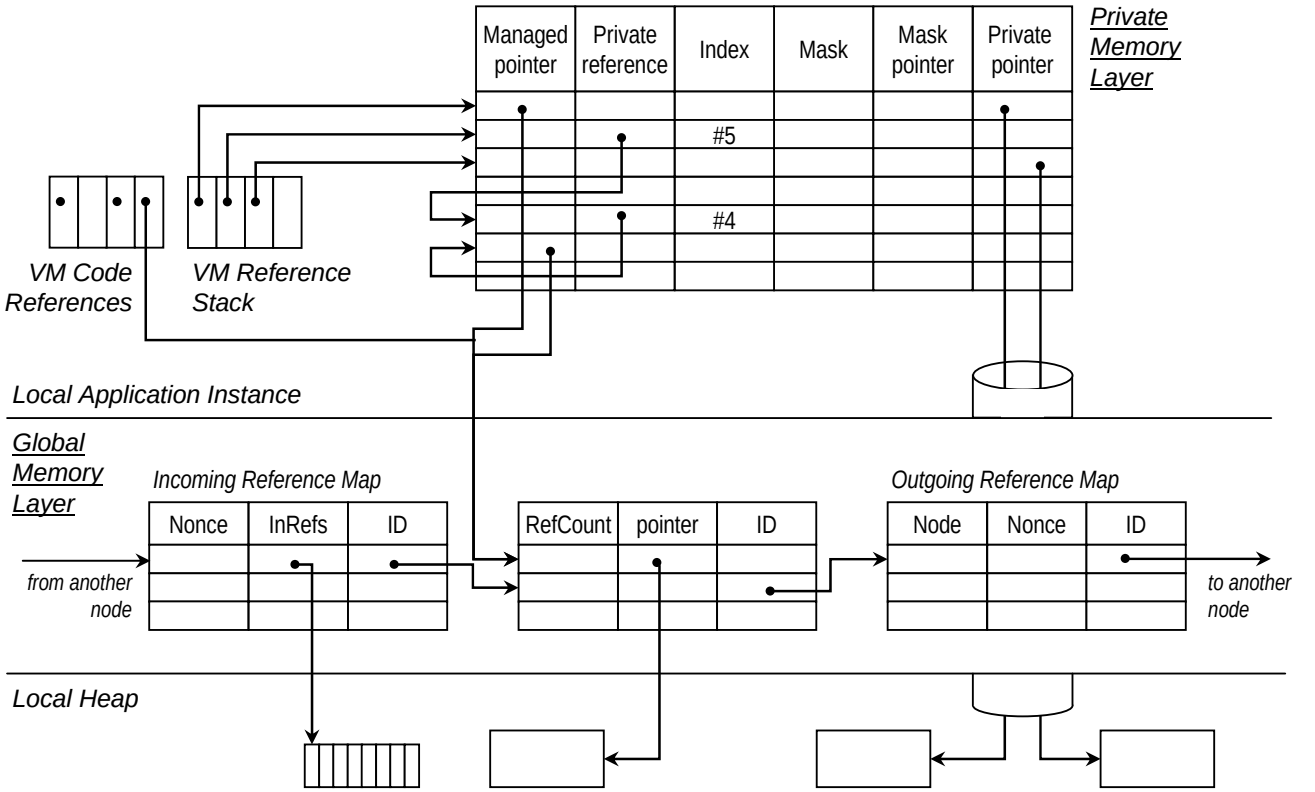

Figure 2. Overview of the memory access architecture.

return addresses. These stacks have fundamentally different roles in the machine. There are no instructions that allow general access to all stacks, which helps to protect the DecentVM from malicious code.

For example, the strict separation of numerical values and references guarantees that references cannot be forged. A reference can only be obtained when allocating an object or as a copy of an already existing reference. Moreover, the separation allows the DecentVM's implementation to choose any reference size that is suitable for the hardware at hand. There is no need to match Java's 32 bit slot granularity for references.

Some instructions inherently operate on numeric values (e. g. arithmetic and logic instructions) or on references (e. g. object and array access). Others can apply to either of the stacks, e. g. local variable load and store instructions. When the transcoder compiles Java byte code for execution on the DecentVM, it assigns such instructions to either the numeric or the reference stack whichever is correct in the given context.

4.4 Instruction set

Altogether, the DecentVM instruction set covers the entire Java bytecode functionality, except for method-local subroutines, which have been deprecated in Java 6, because they are difficult to handle in a light-weight VM [11].

We give a quick overview of the semantics of the instructions. A brief discussion of the transformation from Java bytecode to DecentVM code can be found in section 6. Details on the instruction set are described in the accompanying technical report.

Table 1 gives an overview of the DecentVM instructions. Each instruction is one octet wide and can have up to two operands. In the table, they are denoted by x and y. Depending on the instruction, the instruction code holds two or four bits of the x operand. The remaining bits follow the instruction code as indicated.

Three `WIDE` instructions can encode further 8 bit, 16 bit, and 32 bit of the operands. The transcoder prepends them to an instruction when the operand does not fit into respective instruction's slots.

Memory transfer within and onto the stack A method's stack frame contains the method's parameters, its local variables, and some ephemeral values, like the frame pointer, that are not visible at source code level. Three operations operate on the stack frame: *`LOAD`* reads a slot anywhere in the method's frame and pushes the contained value on top of the stack. Since both, the numerical stack and the reference stack do not contain the frame boundaries, such reads are not confined to the current method's frame. Thereby, an optimizer could, e. g. , avoid copying a constant parameter into the current frame. *`LOAD_IMM`* pushes a constant 32 bit value onto the stack. The value is an operand of the load operation, i.e. it is part of the code. *`STORE`* pops a value from the top of the stack and writes it into a slot of the stack frame. `LOAD` and `STORE` come in two flavors that differentiate between the numeric and the reference stack. But there is no instruction that mixes both stacks.

Arithmetic and logic instructions The DecentVM supports only the most basic of arithmetic and logic operations natively as displayed in Table 1. It provides them only for 32 bit integers. All other operations are reflected by trap instructions (see below). Thus,

0000 xxxx	WIDE	x	Extend next instruction's operand
0001 xxxx XX	WIDE	x	
0010 xxxx XX XX XX	WIDE	x	
0011 xxxx	LOAD	IMM x	Push immediate numeric
0100 xxxx	LOAD	Rx	Read reference onto the stack
0101 xxxx	LOAD	Nx	Read numeric onto the stack
0110 xxxx	STORE	Rx	Pop reference into a variable
0111 xxxx	STORE	Nx	Pop numeric into a variable
1000 xxxx YY	INC	Nx y	Increment numeric variable by y
1001 0000	LOAD	NULL	Push null reference
1001 0001	TEST	NULL	Test reference for null
1001 0010	TEST	EQUAL	Test references for equality
1001 0011	ARITH	ADD	Arithmetic instructions
1001 0100		ADD_WITH_CARRY	
1001 0101		SUB	
1001 0110		SUB_WITH_CARRY	
1001 0111		NEG	
1001 1000	LOGIC	AND	Logic instructions
1001 1001		OR	
1001 1010		XOR	
1001 1011		SHL	
1001 1100		SHL_WITH_CARRY	
1001 1101		SHR	
1001 1110		SHR_WITH_CARRY	
1001 1111		USHR	
1010 0000	BRANCH ZERO		Conditional branch within method
1010 0001	BRANCH NON_ZERO		
1010 0010	BRANCH LESS_THAN		
1010 0011	BRANCH LESS_EQUAL		
1010 0100	BRANCH GREATER_THAN		
1010 0101	BRANCH GREATER_EQUAL		
1010 0110	BRANCH ALWAYS		Unconditional branch within method
1010 0111 XX	SWITCH	x	Switch among x given branch targets
1010 1000 XX	STATIC	x	Get reference to static object x
1010 1001 XX	INSTANCE	x	Check instance for type x
1010 1010 XX YY	INSTANCE	x y	Check for array with element type x and dimension y
1010 1011 XX YY	INSTANCE	x y	Check for array with primimitve elements x and dimension y
1010 1100	LOAD NULL		Push null reference
1010 1100 XX			
1010 1101 XX	NEW	x	Allocate object of class x
1010 1110 XX YY	NEWARRAY	x y	Allocate array with element type x and dimension y
1010 1111 XX YY	NEWARRAY	x y	Allocate y dim. array of prim. elements
1011 xxxx	GET	Rx	Read reference from an object
1100 xxxx	GET	Nx	Read numeric value from an object
1101 xxxx	PUT	Rx	Write reference into an object
1110 xxxx	PUT	Nx	Write numeric value into an object
1111 0000	GETARRAY		Read reference from an object
1111 00xx	GETARRAY	x	Read numeric of size x from array
1111 0100	PUTARRAY		Write reference into an object
1111 01xx	PUTARRAY	x	Write numeric of size x into array
1111 1000	ARRAYLEN		Read number of elements of an array
1111 10xx	ARRAYLEN	x	
1111 1100 XX	INVOKE	x	Invoke given method on an object
1111 1101 XX	TRAP	x	Kernel trap
1111 1110 XX YY	RETURN	Rx Ny	Return from a method
1111 1111	TRHOW		Throw an exception
.... XX YY	FRAME	Rx Ny	Frame shift depth needed to create a frame on the stack

Table 1. Overview on the DecentVM Instruction Set (DIS).

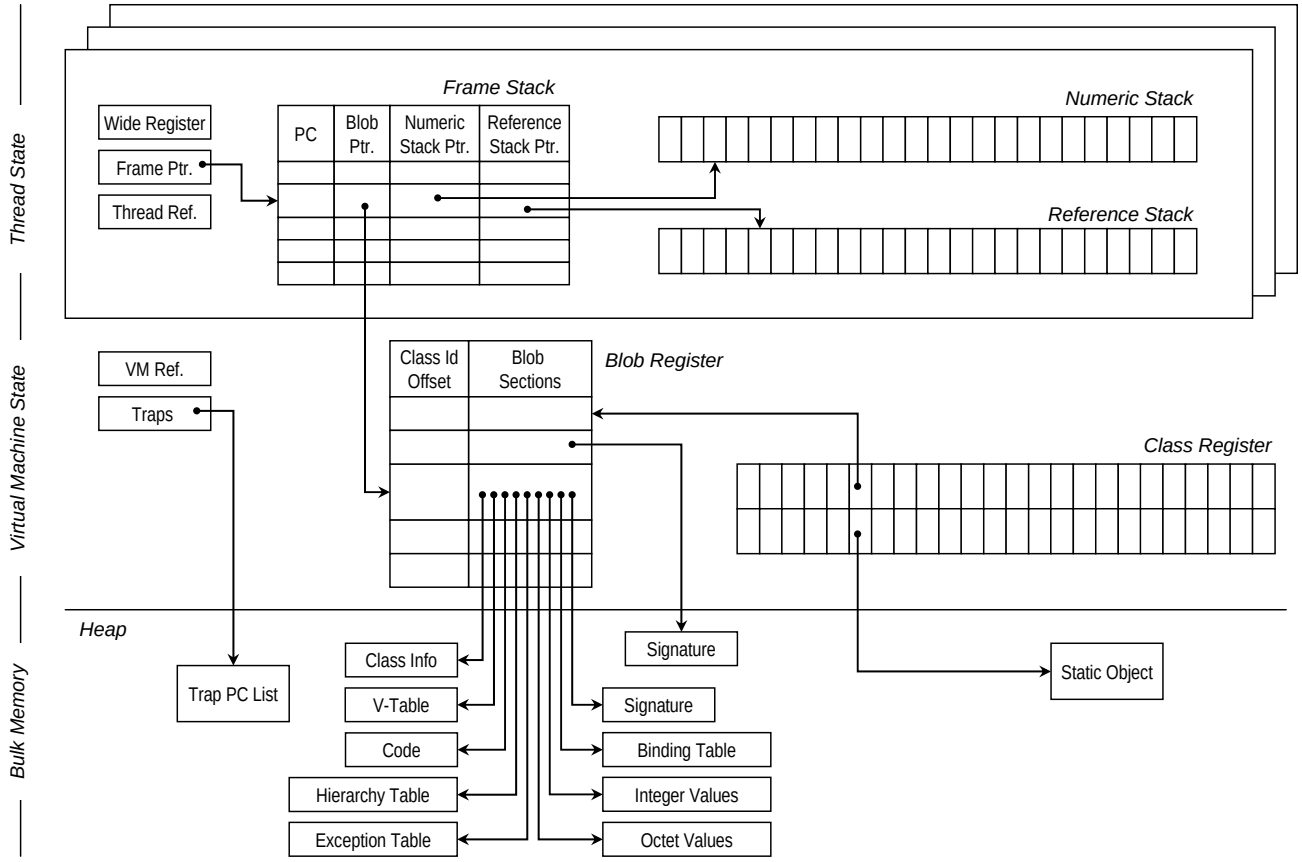

Figure 3. Overview on the VM architecture.

DecentVM implementors can decide if they provide these more complex operations natively or by means of library methods.

Object reference comparisons Comparisons between numerical data types can be emulated by a subtraction and interpretation of the result's sign. Comparisons between object references, however, cannot be emulated that way, because arithmetic operations cannot be applied to elements of the reference stack. Thus, the DecentVM instruction set contains two reference comparison operations: one pushes a numerical zero if and only if the two topmost references point to the same object, the other one pushes a numerical zero if and only if topmost reference is the null reference.

Branching and switching As said above, the transcoder maps all comparisons between two numerical values (or references) to produce one numerical value, so that the topmost stack value suffices to decide on the (conditional) branch. There are seven branch conditions: `ZERO`, `NON_ZERO`, `LESS_THAN`, `LESS_EQUAL`, `GREATER_THAN`, `GREATER_EQUAL`, and `ALWAYS`. Except for the latter, they are based on the comparison of numerical values.

Furthermore, the DecentVM ISA contains a `SWITCH` instruction. The transcoder expands Java `SWITCH` instructions, so that the underlying value range is always $0 \ldots n$. If the topmost value on the stack is zero, the VM branches to the first entry in the target list. If it is one, it branches to the second entry, etc. If the value is n, larger than n, or negative, it branches to the last entry in the list.

Operations on classes There are three different instructions, `NEW`, `STATIC`, and `INSTANCE` that take a class ID as argument. The class IDs are blob-local IDs, which have to be mapped into the VM's ID-space. (See sec. 5 for an overview of the class loading mechanism.)

The `NEW` operation allocates an object instance of the given class on the (private) heap. It looks up the object's size, allocates an appropriate memory chunk, and pushes the reference onto the stack.

The `STATIC` operation pushes a reference to the class' static object onto the stack. (The static object contains the static fields of a class, cf. sec 5.) If the static object does not yet exist for the class of interest, the DecentVM allocates an according object for the static fields, and executes the static initializer before the regular execution continues. If another thread performs a `STATIC` operation while the initialization has not yet completed, that thread is suspended during the initialization.

The `INSTANCE` operation checks if the topmost reference is an instance of the given class (or any of its sub-classes).

The Java specification also describes a *checkcast* instruction that does not consume the reference, but that throws an exception if the check fails. The transcoder emulates this variant by duplicating the reference before checking, and by adding a trap instruction after the check that throws the exception if necessary.

Heap access for objects and arrays Objects and arrays are always segmented into 32 bit slots, which are accessed atomically. Thus, long and double values have to be written by two separate access instructions, which the scheduler may interrupt. This definition is in accordance with the Java specification, even though the transcoder actually splits the instructions. The application programmer must ensure that the application is thread-safe.

Arrays of booleans, bytes, or shorts may exploit a finer grained internal structure in which case the DecentVM must ensure that array access is thread-safe. Thus, the `PUTARRAY`, `GETARRAY`, and `ARRAYLEN` instructions encode the size of the elements that are stored in the array so that the DecentVM can atomically perform the respective access.

Objects and arrays have an internal header structure contains the size and type of the object, garbage collection information, and a monitor reference for the respective object or array.

Accordingly, the `ARRAYLEN` instruction is merely a read instruction that reads the array's header. Further kernel trap instructions give access to the other fields of the header.

Method invocation Executing a method invocation instruction requires actions both of caller and callee. At the caller's side, the destination address is calculated using the virtual method table of the respective class. After resolving the callee, the frame shift depth (FSD) is read from the callee's code, and the stack pointer is advanced accordingly. The FSD is determined by the number of local variables excluding parameters. The FSD comprises two values: one for the numerical stack and one for the reference stack. Then the former program counter and the new stack pointers are pushed onto the frame stack. Finally, the program counter is set to the first instruction, i. e. the position just behind the FSD.

Upon execution of a `RETURN` instruction, the DecentVM resets the stack pointer to the position stored on the call stack, pops the remaining local variables from the stack (excluding potential return value slots), and resets the program counter to the value that it pops from the call stack. (The number of numerical and reference stack slots that need to be popped are parameters of the `RETURN` instruction.)

Interface method invocation Interface method execution is implemented via a special dispatcher method. To this end, all interface methods obtain an identifier that must be unique among the entire code base of an application. (I. e. it must be unique across the application blob and all the library blobs that the application depends on.) The transcoder equips each class that implements an interface with a *dispatch* method, which contains a switch instruction that dispatches interface method calls. The transcoder maps each *invoke interface* instruction to a `LOAD_IMM` of the interface method ID followed by an `INVOKE` of *Object.dispatch*.

Exception handling Exceptions can be thrown explicitly, by means of the `TRHOW` instruction, or implicitly, for example, when trying to access an object via a null reference. When an instruction (explicitly or implicitly) throws an exception, the VM looks up all exception handlers that match both, the current program counter and the object type of the exception. If multiple such handlers exist, the VM takes the first matching handler, i. e. it sets the program counter to the value determined by the matching handler, and it sets the stack pointer to the value peeked at the called stacked. If no exception handler matches, the VM pops one entry from the frame stack. Again, the VM checks if any exception handler matches the new program counter value and proceeds accordingly. If the call stack has become empty without any handler matching, the VM kills the respective thread.

Trap instructions *Kernel traps* are special instructions, which can be implemented either natively or as part of the system libraries. They provide functionality from Java bytecode that the DecentVM ISA does not cover:

- monitor enter and exit,
- look-up table conversion (for switch operation),
- long, float, and double arithmetics and comparisons,
- native type casting, and
- throwing class cast exceptions.

Kernel traps also serve as general trap into the system library, for example, for providing input and output functionality.

Further traps provide access to the internal data structures of the DecentVM, e. g. the object headers. Such traps (*virtual machine traps*) can only be called from the kernel code. This restriction corresponds to ring 0 in other processor architectures.

Class initialization Class initialization takes place upon the first access to the static object of that class. The transcoder generates such an access operation for put and get access to static fields, static method invocation, and the so-called special method invocation. Class initialization upon dynamic object allocation, which the Java standard requires, is also guaranteed, because dynamic object allocation is always followed by an invocation of the class' constructor, i. e. a special method invocation to the respective class.

5. Blobs

The transcoder (cf. Sec. 6) compiles multiple Java classes into one blob. A blob may either be self-contained or bind to other blobs. In the latter case, these other blobs must be identified at compile time. There are, however, methods for dynamic class loading, which bypass the mechanisms described in this paper.

Typically, each application blob binds to one so-called *kernel blob* that contains the essential API classes. These classes constitute a kind of operating system. The other blobs extend the kernel blob with further libraries. They also contain the actual application code.

Small applications may combine all OS and library code into one self-contained blob. This would allow for more compile-time optimization, but it prevents the DecentVM from sharing the OS and libraries among different applications. Moreover, it comes with a reliability risk because in this case all code runs at kernel level, so that faulty or malicious code can compromise the virtual machine.

Each blob contains several sections (cf. Table 2). Each section begins with a 32 bit length field and a 32 bit section type field. In the following, we briefly explain these sections.

Blob information section The information section contains ASCII encoded clear text meta information about the blob, e. g. the blob name, the authors' names, a brief description, and the copyright information. The DecentVM does not process this information, but when loading a blob, it checks for the correct section type number to ensure that the blob matches the VM version. Thus, future versions of the VM with incompatible blob formats must have a different section type number.

Binding section The binding section contains the class ID offsets together with the blob SHA1 signatures. These signatures identify the blobs and ensure that the library that the DecentVM is about to load has not been tampered with. Classes at offset zero are the classes that are provided by the respective blob file. Hence, the according signature is the blob's own signature.

Note that all class and interface IDs in the blob are blob-local. Upon blob loading, the DecentVM maps them into the VM's internal class ID space (cf. Fig. 4).

Class information section The class information section includes the basic information about the classes and interfaces that are provided in the blob, for example, the number of fields. It also gives the offset of the virtual method table in the method table section, as well as the offset into the class hierarchy section, which contains the information about the classes' super class and interfaces. (For details cf. Table 3.)

Section name	Description
Blob information	Clear text info and version number
Binding section	Blob signature and list of required blobs
Class information	Basic info about the provided classes
Method tables	Virtual method tables for all provided classes
Exception tables	Code ranges, handler code position, and handled class IDs
Hierarchy tables	Implemented interface IDs and super class IDs
Method code	Start-up code and bytecode for all provided methods
Kernel traps	Code positions for trap implementations (only for kernel blobs)
Unicode data	Octet data, e. g. for UTF-8 encoded strings
Integer data	Integer value data, e. g. for handling switch instructions

Table 2. Blob section types.

Size	Class information	Interface information
16 bit	Class Hierarchy Table offset	ditto
16 bit	Virtual Method Table offset	Static Initializer offset
8 bit	Number of slots for static numerical fields	ditto
8 bit	Number of slots for static reference fields	ditto
8 bit	Number of slots for dynamic numerical fields	0xFF
8 bit	Number of slots for dynamic reference fields	0xFF

Table 3. Class information section of a blob.

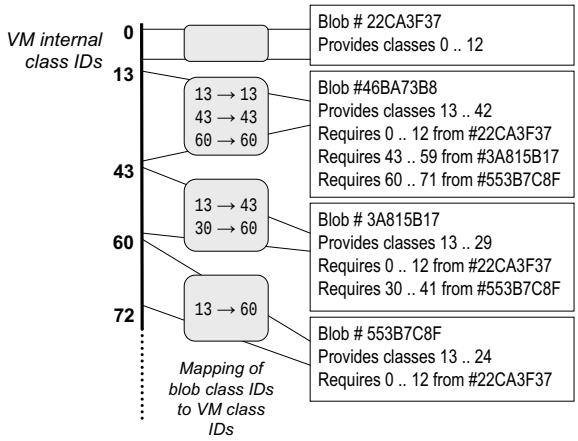

Figure 4. Illustration of blob binding.

Method tables section The method tables section contains the virtual method tables for all provided methods.

If a class does not override an inherited method, the entry in the method table points to the implementation in the respective super class. If that super class is not provided by the same blob, the transcoder enters a trampoline method.

Entires for abstract methods point to a dummy piece of code that throws an abstract class exception.

Exception tables section The exception tables section encodes the position of all defined exception handlers. For each handler it holds the following information:

- The method position gives the offset of the respective method in the code section.
- Start, end, and handler offsets are unsigned offsets relative to that position.

Exception tables must occur in increasing order of their method position. The catch type is the blob-local class ID that an exception handler catches. The root class type indicates that the handler catches any exception.

Class hierarchy and interface tables section The class hierarchy section encodes the class and type hierarchy of the application. It contains the lists of interfaces that the classes implement, as well as the classes' respective super classes. For interfaces, it contains the lists of extended other interfaces, as well as the root object as formal super class. Each entry in the list is a 16 bit blob-local class identifier.

Code section The code section contains a sequence of variably sized bytecode instructions (cf. Sec. 4.4). Upon start-up of a blob, the DecentVM starts execution at the beginning of the blob. This code should create the required thread state, e. g. a thread object, store it in the VM's thread state, and then call the entry point of the application, e. g. the main function of the application.

Trap method tables section If a blob contains a trap offset section, it is a kernel blob. This means that the blob provides certain built-in classes such as the root of the class hierarchy (e. g. *java.lang.Object*), the VM exception classes, and further kernel classes. The latter contain implementations of the fundamental instructions of the VM, e. g. floating point arithmetics and monitor handling.

An instance of the DecentVM may or may not implement these trap methods natively. Thus, a kernel blob should contain code for all the trap methods so that the blob can run on any DecentVM implementation.

Unicode and integer data sections Strings are encoded as 16 bit unicode characters in the unicode data section of the blob. Integer constants, e. g. those that the transcoder produces when handling a lookup switch instruction, are stored in the integer data section.

If the local endianess does not match the blob endianess, the DecentVM must convert the endianess, e. g. upon blob loading.

6. Transcoder

The *transcoder* is an important element in the DecentVM toolchain. It compiles Java bytecode into the DecentVM instruction set (DIS) and packs the DIS code together with all the other required class file information into a blob file that the DecentVM can execute. The transcoder reads Java class files, JAR archives, and other blobs. The latter serves only as means of binding, i. e. the transcoder does not extract code from other blobs, but refers to those blobs via a secure hash key.

Transcoding is a two-stage process: In the first stage, the transcoder reads the class information from the given input files. Thereby, it fills its *class store*. In particular, it reads the class hierarchy, including the respectively implemented interfaces, as well as the classes' fields and method declarations.

In the second stage, the transcoder reads the methods' code. Because the class store contains the full information about the entire code base, it can directly assign method and field IDs and thus generate the transcoded output on the fly.

During the second stage, the transcoder builds up the *virtual method tables*, which contain the methods' start positions in the code section. It also builds up the exception tables, which relate the code parts that are covered by try-catch-blocks to the respective handler code positions.

6.1 Compiling Java byte code to DIS

A few instructions in the Java ISA do not directly correspond to an instruction in the DecentVM ISA, for example, the *monitor enter* and *exit* instructions, the *load string* instruction, native type casts, as well as long, float, and double arithmetics. The transcoder maps these instructions to so-called kernel trap instructions so that they can can be executed from the system library rather if a DecentVM implementation does not provide them natively.

Several different Java bytecode instructions map to the DecentVM's transfer instructions (cf. section 4.4). The Java VM specification [16] describes nine stack instructions: `DUP`, `DUP_X1`, `DUP_X2`, `DUP2`, `DUP2_X1`, `DUP2_X2`, `POP`, `POP2`, and `SWAP`. `DUP`, is mapped to `LOAD 0`, and `POP` is mapped to `STORE 0`. All other operations are handled similarly.

The Java VM specification describes 36 different arithmetic and logic operations: There are 6 different arithmetic operations (addition, subtraction, negation, multiplication, division, and remainder), each for the 4 different native numerical data types (int, long, float, and double), and there are 6 logic operations (and, or, xor, shift left, shift right, and unsigned shift right), each for the 2 different integer data types.

The transcoder maps the corresponding 64 bit instructions either to sequences of 32 bit instructions or to kernel traps. Float and double instructions as well as multiplication and division instructions are among the kernel traps. If the hardware does not support them natively, they need to be covered by the DecentVM's system library.

The `SWITCH` instruction is equivalent to Java's *tableswitch* instruction. The transcoder supports Java's *lookupswitch* instruction by inserting a kernel trap that maps the numerical targets to a number between 0 and n.

Java's JSR and RET operations are not supported, because recent Java compilers should not produce these operations any more [2].

According to the Java VM specification, there are 8 operations that perform casts between the native types. Furthermore, there are 3 cast operations that narrow (and potentially sign-extend) an integer value within a 32bit slot, namely to *short*, *unsigned short* (aka Unicode character), and *byte*. All eleven casts are provided as kernel traps, i. e. they are not necessarily provided natively in the DecentVM.

6.2 Synchronized methods and code blocks

Java compilers generate monitor enter and monitor exit operations for synchronized code blocks. Both expect a reference to the corresponding object on the stack.

The transcoder extends this approach to synchronized methods. To this end, it places additional code at the beginning and end of a synchronized method:

```
load <this>
monitor enter
L1: ...
L2: load <this>
    monitor exit
    return
```

In case of a static method, the transcoder loads the respective static object reference. To protect against unhandled exceptions, the transcoder also puts an according handler into the exception table:

```
L1:L2 catch any at L2
```

6.3 Array access instructions

The DecentVM inherently assumes a 32 bit memory layout. Hence, 64 bit instructions need to be transcoded into two 32 bit instructions. This applies also to array access.

Moreover, 8 bit and 16 bit array access requires special treatment. Finally, the DecentVM fundamentally separates references from numeric values.

Hence, all array instructions have one of four different flavors, depending on whether they act on an array of references, 8 bit, 16 bit, or 32 bit numeric values.

The Java bytecode contains type information for `GET` and `PUT` instructions, but not for `ARRAYLEN` instructions. Hence, the transcoder has to perform a static type analysis.

7. Implementation

The implementation of the DecentVM is ongoing work[1]. In this section, we give a brief overview on the current status of the project.

The DecentVM system comprises the blob transcoder, the virtual machine, and a platform-dependent BIOS. The latter provides a uniform interface for system calls and implements system services such as memory virtualization directly, depending on platform and OS. The BIOS is available for Linux both, on x86-style PCs and ARM A8 embedded devices. It is also available for the bare metal modules that resulted from the Ambicomp project [1].

The DecentVM itself consists of approximately 2.000 lines of C code. The basic features, for example, the interpretation of the blob bytecode have already been fully implemented; but some kernel traps, like floating point arithmetic, are not yet available.

As for now, applications run multi-threaded and with monitor synchronization on a single node. Our experience with multi-versioned STM is still limited to simulations [7], because the heap access has not yet been adapted to the new memory model describe in this paper. As part of the restructuring process, the former mark-and-sweep garbage collector still needs to be substituted by a distributed variant. In particular, the garbage collector also has to take object versioning into account. The details of this issue are still ongoing work in our group.

Owing to the early stage of the implementation of our distributed memory system, we cannot yet give performance measurement results.

8. Conclusion and Future Work

DecentVM is a fully decentralized Java virtual machine designed for running on large clusters of many-core processors. Its sophisticated memory layout aims at minimizing communication overhead while guaranteeing consistent memory snapshots. Its simplified instruction set allows for a small VM code footprint as does the consise blob format.

Several parts of the VM's layout open up interesting questions and require further research. For example, it is not clear yet what impact the versioning of objects and other meta-information that

[1] The code and documentation is available at `www.j-cell.org`

is needed for the tentative transactional execution will have on the VM's footprint. Futher, the semantics of the interplay of transactional operations with monitors and volatile fields also requires a non-standard implementation of memory-barriers. Other open issues involve the migration of threads between different computational nodes to optimize data locality, or the integration of a JIT to speed-up execution.

Acknowledgments

This work was partially funded by the German Federal Ministry of Education and Research under grant numbers 01ISF05, 01IH08011 and 01IS09040.

References

[1] Ambicomp project. URL `http://ambicomp.org`.

[2] New Java SE 6 Feature: Type Checking Verifier. URL `https://jdk.dev.java.net/verifier.html`.

[3] J-cell project. URL `http://j-cell.org`.

[4] JNode. URL `http://www.jnode.org`.

[5] G. Ammons, J. Appavoo, M. Butrico, D. Da Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. Van Hensbergen, and R. W. Wisniewski. Libra: a library operating system for a JVM in a virtualized execution environment. In *VEE '07: Proceedings of the 3rd International Conference on Virtual Execution Environments*, pages 44–54, New York, NY, USA, 2007.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003.

[7] A. Bieniusa and T. Fuhrmann. Consistency in hindsight: A fully decentralized STM algorithm. In *IPDPS 2010: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium. Atlanta, Georgia, USA.*, April 2010.

[8] B. Buck and P. Keleher. Locality and performance of page- and object-based DSMs. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International ... and Symposium on Parallel and Distributed Processing 1998*, pages 687–693, Mar. 1998.

[9] A. Dragojevic, P. Felber, V. Gramoli, and R. Guerraoui. Why STM can be more than a research toy. Technical Report LPD-REPORT-2009-003, EPFL, Oct. 2009.

[10] W. Fang, C.-L. Wang, and F. C. M. Lau. On the design of global object space for efficient multi-threading Java computing on clusters. *Parallel Comput.*, 29(11-12):1563–1587, 2003.

[11] T. Fuhrmann. The ACVM/DecentVM Architecture. Technical report, Technische Universität München, 2010. URL `http://www.j-cell.org`.

[12] K. O. W. Group. *The OpenCL Specification, Version 1.1*, June 2010.

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

[14] A. Leung, O. Lhoták, and G. Lashari. Automatic parallelization for graphics processing units. In *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 91–100, New York, NY, USA, 2009.

[15] A. Lienhard, T. Girba, and O. Nierstrasz. Practical object-oriented back-in-time debugging. In *ECOOP 2008: Proceedings of the 22nd European Conference on Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 592–615. Springer Berlin / Heidelberg, 2008.

[16] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.

[17] M. Lobosco, A. Silva, O. Loques, and C. L. d. Amorim. A new distributed jvm for cluster computing. In *Euro-Par 2003 Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pages 1207–1215. Springer Berlin / Heidelberg, 2004.

[18] M. Lobosco, O. Loques, and C. L. de Amorim. On the effectiveness of runtime techniques to reduce memory sharing overheads in distributed Java implementations. *Concurrency and Computation: Practice and Experience*, 20(13):1509–1538, 2008.

[19] S. Marr, M. Haupt, S. Timbermont, B. Adams, T. D'Hondt, P. Costanza, and W. D. Meuter. Virtual machine support for many-core architectures: Decoupling abstract from concrete concurrency models. In *Proceedings Second International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*, volume 17, 2010.

[20] A. Noll, A. Gal, and M. Franz. CellVM: a homogeneous virtual machine runtime system for a heterogeneous Single-Chip multiprocessor. In *Workshop on Cell Systems and Applications*, Beijing, China, June 2008.

[21] Oracle. Project Guest VM. URL `http://labs.oracle.com/projects/guestvm`.

[22] B. Saballus and T. Fuhrmann. A decentralized object location and retrieval algorithm for distributed runtime environments. Technical report, Technische Universität München, 2010. URL `http://www.j-cell.org`.

[23] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java™ on the bare metal of wireless sensor devices: the squawk java virtual machine. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 78–88, New York, NY, USA, 2006.

[24] D. Ungar and S. S. Adams. Hosting an object heap on manycore hardware: an exploration. In *DLS '09: Proceedings of the 5th Symposium on Dynamic Languages*, pages 99–110, New York, NY, USA, 2009.

[25] 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 conference on Computing Frontiers*, pages 189–198, New York, NY, USA, 2008.

[26] W. Zhu, C.-L. Wang, and F. C. M. Lau. JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support. In *IEEE Fourth International Conference on Cluster Computing*, Chicago, USA, September 2002.