

# Implementation of the Java language in a persistent DSM Operating System

M. Schoettner, O. Schirpf, M. Wende and P. Schulthess  
Department of Distributed Systems,  
University of Ulm, Germany

## Abstract

*Java has gained considerable attention in the IT-world. The Java language and its class library is widely used for application development in the context of the World Wide Web and elsewhere. The popularity of the Java language also makes it attractive for other areas such as Operating System development. Our Plurix project implements a Distributed Shared Memory (DSM) Operating System (OS) establishing a new memory consistency model based on restartable transactions and optimistic synchronization. The OS is developed with a proprietary Java compiler transforming Java sources into Intel protected mode code. Abandoning the hardware independence of Java eliminates performance loss and allows developing the total OS in Java. In this paper we shortly review the architecture of the Plurix Java Compiler (PJC). We discuss the compact memory layout of Plurix Java run-time structures in detail. Furthermore we present an elegant and efficient implementation of Java interfaces. Finally we show how PJC supports hardware-level programming.*

*Keywords:* Java, Compiler, Run-Time Structures, Distributed Shared Memory, Operating System

## 1 Introduction

The PJC Java compiler is an integral part of our distributed Operating System (OS) Plurix. The central abstraction in Plurix is the Distributed Shared Memory (DSM) paradigm providing a virtual address space shared among tasks on loosely coupled nodes [1], [2]. There-with the resource distribution is managed by

the OS and mustn't be reimplemented in each distributed application. DSM offers to the application programmer a transparent view on data shared over several computers. Pointers can either reference local or remote memory blocks. The OS will detect a remote memory access, fetch the desired memory block and maintain memory consistency.

The Plurix Java compiler (PJC) is used for the total development of the object-oriented Plurix OS including hardware-level programming (e.g. kernel and drivers). PJC is written in Java and after bootstrapping of the compiler from the MS Windows platform to the native hardware it will be an integral part of the Plurix OS. Language based OS development has been successfully demonstrated in systems like Oberon [3]. However, the implementation of the Java language in the orthogonal persistent Plurix DSM reveals interesting aspects regarding efficient memory layout of run-time structures. Additionally, we believe that the Plurix OS is a contribution to the DSM community as it implements a new memory consistency model and includes shared code [4], [5].

Our paper is composed of five sections. Section two briefly reviews the architecture of the PJC. Section three presents the memory layout of all necessary run-time structures for the implementation of the Java language in detail. In section four we discuss problems of the Java language when used in hardware-level programming. Finally we present the current implementation status and illustrate outlook on future work areas.

## 2 The Plurix Java Compiler

The Plurix Java Compiler (PJC) transforms Java source code into executable Intel protected mode code. As the compiler is used to develop an OS we insist on generating machine code and avoid performance penalties by a Java Virtual Machine (JVM). Furthermore we opted against existing Java JIT compilers because drivers and interrupt handlers are also written in Java requiring minor language extensions (see section four). The Java compiler itself is written in Java allowing subsequent bootstrapping within the emerging Plurix environment.

Due to limited human resources the current compiler does not provide an intermediate representation (e.g. Static Single Assignment) nor state of the art code optimizations. Parser and code generator are coalesced resulting in a small compact compiler size (<100kb bytecode). The current code generation is preliminary and uses a single stack allocation strategy. Among our research goals is evaluating integration aspects between PJC and the Plurix OS.

As Plurix implements an orthogonal persistent DSM the compiler directly writes run-time structures and code segments into the DSM and bypasses the creation of create object-, symbol-, library- and exe-files. Run-time structures are created during the compilation phase. We don't have a separate linker, instead classes are linked by the compiler. As the compiler initializes classes after successful compilation a separate class loader is also superfluous. Furthermore symbol information may be discarded and recreated on demand.

We have established a testing environment under Windows 95/NT [5]. PJC is executed on a JVM and deposits its output not into a file but into a virtual memory block (VMB). The VMB is a large virtual memory block allocated during startup of the testing environment on the same Windows 95/NT machine. PJC accesses the VMB using native method calls. The run-time structures and the code generated in the VMB can be examined with a regular debugger. After examining the code generation

we copy the image to the Plurix machine either via a serial link or a floppy disc where it will be executed.

## 3 Compact memory layout of run-time structures

In the following section we discuss the memory layout of run-time structures in detail. This includes class descriptors, instances, code segments and interfaces. Since the entire Plurix OS is written in Java all run-time structures should be addressable with Java language constructs. The run-time memory organization is designed for fast access and low memory consumption while taking into account the persistent DSM environment.

### 3.1 Class descriptors

The first naive approach modeling a class descriptor in Java entails to a Java compatible class descriptor, see fig. 1. The heavy pointer usage yields an excessive amount of heap objects which is a burden to every memory manager. Especially in the DSM context each separate heap block introduces a certain amount of overhead, e. g. during object relocation [4]. Furthermore every indirection involves a speed penalty at run-time. Therefore we adopted the concept of a monolithic memory layout for class descriptors, see fig. 1. This is a similar approach as for SUNs JVM and [6] but they used C/C++ as an implementation language and not the strong typed Java itself.

Java discards the problems linked with multiple inheritance by offering only single inheritance. Static variables are unique in the total class hierarchy and are hence not replicated if inherited. Any access from a subclass to an inherited static field is resolved as a static access via the class name of the corresponding superclass and is therewith statically bound.

Method jump tables entries of dynamic methods are replicated in the class extensions resulting in a fast invocation of inherited methods [7]. The inherited methods jump table en-

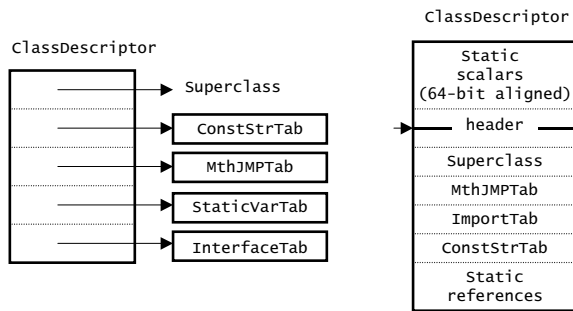


Figure 1: Memory layout of class descriptors (Java compatible/Plurix monolithic)

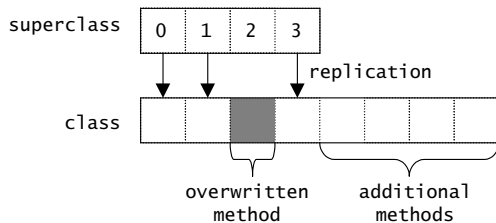


Figure 2: Replication of inherited dynamic method jump table entries and overwriting

tries are clustered together at the beginning of the method jump table of the subclass, see fig. 2.

A method is overwritten by simply changing the corresponding entry in the replicated part of the method jump table. At run-time the appropriate method within the dynamically bound class is called. This is an easy task with replicated method jump table entries. This scheme ensures a fixed index in the method jump table for an overwritten method independent of current type level in the inheritance hierarchy. By dereferencing the instance pointer we access the correct class descriptor containing the correct overwritten method entry in the method jump table.

Additionally, class descriptors contain a constant string table similar to the constant string pool of the Java class files.

### 3.2 Instances

The memory layout discussion for instances is quite similar to that of class descriptors. Java compatible run-time structures introduce over-

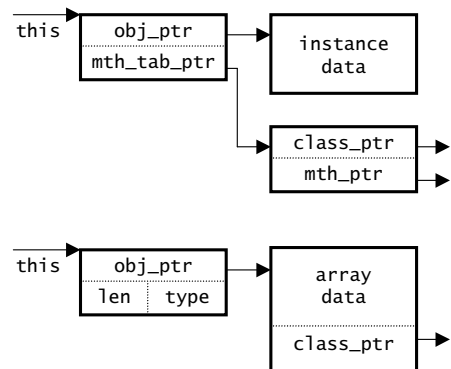


Figure 3: Run-time memory layout of SUN JVM for instances/arrays

head for memory management and decrease program performance. The SUN JVM approach, see fig. 3 is expensive.

Taivalsaari [8] describes the implementation of a JVM in Java. He uses even more indirections as Java doesn't allow mixing of primitive and non-primitive data types. Therefore Taivalsaari uses one object for every value, e.g. the integer value '3' is physically represented by an Integer(3) object. Of course performance penalties and memory overhead for such a solution are substantial.

Plurix adapts the monolithic layout scheme of class descriptors to represent instances as well. We distinguish instance variables of two categories: primitive data types (byte, short, int,...) and non-primitive data types (references). Primitive data types (PDT) are stored in a 64-Bit aligned container and non-primitive data types (NPDT) are stored in a separate container, see fig. 4.

Dynamic variables are replicated in the case of inheritance. Unfortunately this can cause offset variations for the NPDT variables. We solve this problem by a bidirectional memory block layout - PDT variables are addressed by positive offsets and NPDT variables by negative ones. Thus we can ensure class/superclass compatibility for two different containers by using a bidirectional memory layout, see fig. 4.

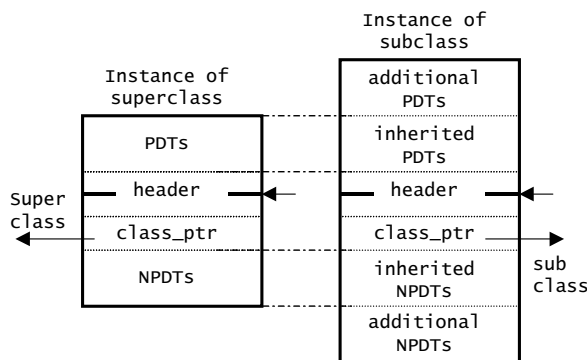


Figure 4: Bidirectional instance memory layout for class/superclass type compatibility

### 3.3 Code segments

In the following subsection we use the term code segment for memory blocks containing the code of Java methods. Every method has its own code segment to allow the building of fine grained working sets. Every method call is a jump to a given address in memory.

Hard coded addresses in the code segment are not reasonable because this would introduce overhead during relocation of code segments. Every method invocation is processed indirectly via the associated class descriptor. There are three call types:

1. Dynamic calls within the current name space (including all superclasses)
2. Dynamic calls into an explicit name space (via instance reference)
3. Static calls into any name space (via import table)

Method calls of category 1) use the method jump table of the current class descriptor. Dynamic calls of category 2) dereference the given instance pointer to the class descriptor and the associated destination name space and proceed like 1). Static calls of type 3) are always processed via an import table (see later) stored within the current class descriptor. Static variable access is resolved equal.

The associated class descriptor of a code segment is called the *Owner Class Descriptor*

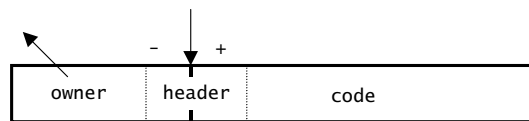


Figure 5: The memory layout of a Plurix method code segment

(OCD). Dynamic methods inherited from superclasses are also found in the method jump table of subclasses, but the OCD may be any superclass descriptor above in the parent chain.

There are situations where the OCD can only be dynamically determined (see the listing below). If the method 'Use' in the class 'Parent' is executed the OCD is 'Parent'. Which overwritten variant of 'OvMth' is called and therewith wich OCD is loaded depends on the instance type of 'p'.

```
class Parent {
    static void OvMth(int a) { ... }
    static void Use (Parent p) {
        p.OvMth(1);
    }
}

class Child extends Parent {
    static void OvMth(int a) {}
}
```

(Dynamic OCD determination)

It is essential always knowing the appropriate OCD during run-time because all static method calls and static variable accesses are processed indirect via the OCD. Therefore Plurix stores the OCD within each code segment. If the address of the callee is computed the caller can retrieve the OCD from the destination code segment.

The performance loss for static calls and variable access, introduced by the additional indirection, can be alleviated by permanently holding the OCD in a reserved register called *Owner Class Register* (OCR).

### 3.4 Import Table

Class descriptors have also an import table supporting static method calls and access to static

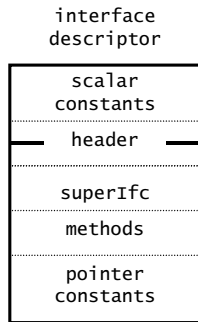


Figure 6: Memory layout of an interface descriptor

class variables. Import tables are not replicated in subclasses. This is superfluous as the OCD is stored within every code segment and therefore the correct OCR is always loaded and the right import table is used.

### 3.5 Java Interfaces

Java supports multiple subtyping by allowing a class to implement several interfaces. An interface exhibits method declarations and constants but no method implementation. Constants can either be static or dynamic and scalars or references. All of these constants are stored within the interface descriptor including the dynamic ones, see fig. 6. A class implementing a certain interface  $\tau$  must implement all methods of  $\tau$  and can use all associated interface constants. Every interface can inherit once from a superinterface.

An instance of a class implementing a certain interface  $\tau$  can be assigned to an interface pointer  $\phi$  of type  $\tau$ . All methods and constants of  $\tau$  can be accessed via  $\phi$  including possible inherited methods and constants from superinterfaces of  $\tau$ .

The challenge of interfaces is finding interface methods in the method jump table of the class descriptor fast. Krall [9] used a bidirectional class descriptor layout for separating the interface methods table from the method function table. Additionally interface methods are also stored in the method jump. Fixed offsets for interface methods are ensured by a system-wide unique number for every interface

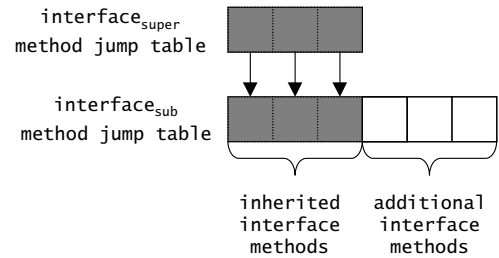


Figure 7: Interface methods jump tables (inheritance & clustering)

method. Therefore the memory consumption for the interface table is quite large:

$$\text{number of classes which implement interfaces} * \text{number of interface methods.}$$

The resulting empty slots in the interface table are reduced by coloring algorithms [10].

The unique numbering of interface methods is not acceptable in a DSM. There might be node private and global public classes. Assume a situation where a private class is published to the global DSM. All interface numbers on the publishing node and in the global class pool have to be adjusted. Certainly this would result in an unacceptable overhead.

We decided to allow interface method offsets to be variable. We store interface methods within the normal method jump table. All methods belonging to one interface are clustered in the method jump table. Interface methods inherited from superinterface are always stored together at the beginning of the subinterface, see fig. 7.

Assume that different classes implement the same interface  $\tau$  and every class implements further different interfaces. Of course this common interface  $\tau$  might not be located at the same offset in the method jump tables of these class descriptors. The compiler copes with this problem by maintaining an interface offset table in the symbol table. An entry of this interface offset table contains the offset to the first method of an implemented interface in the method jump table. Fortunately the entries in the interface offset table can be statically determined by the compiler.

```

Java source
...
MyClass pInst = new MyClass();
IFC    pIfc = (IFC)pInst;

pIfc.mth0();
...

assembly code for „ifc.mth0()“
mov eax, pIfc;      // load ifc ptr
mov eax, [eax-30];  // load ptr. Of class descr.
add eax, [off];     // add ifc offset
add eax, mth0_Off; // add offset of mth0

```

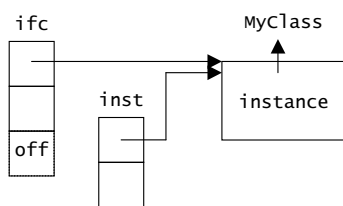


Figure 8: Extended interface pointers

Interface pointers are extended by four bytes for storing the beginning of the associated interface in the method table, see fig. 8. If the compiler generates code for the assignment of an instance reference to an interface reference variable it generates an additional assignment storing the appropriate interface offset (found in the symbol table) in the offset field of the interface pointer.

How interface method calls are processed is also shown in fig. 8. The invocation of the method "pIfc.mth0()" is performed by simply dereferencing the interface pointer twice to load the address of the associated class descriptor. The offset "off" is loaded from the extension field of the interface pointer and index of the interface method is retrieved from the interface descriptor. The index and offset are added to the retrieved start address of the class descriptor and the correct method jump table entry can be loaded.

The subtyping solution presented here is obviously more suitable for a DSM by avoiding global numbering. Furthermore it consumes less memory for the jump tables than other approaches and introduces no additional indirections for interface method invocations.

## 4 Hardware-level programming in Java

Some minor language extensions became necessary to support hardware-level programming. We have implemented interrupt handlers as normal static Java methods. Unfortunately the stack frame of an interrupt handler is defined by the Intel architecture. To support this special stack frame by the compiler we have introduced a new method modifier "interrupt". Since interrupt handlers are invoked by the hardware they do not obtain the OCD parameter. As denoted earlier the OCD is essential for any method calls and variable access. Fortunately the OCD is stored within the code segment of every method (see previous section). As the Intel CPU doesn't allow program counter (pc) relative addressing we use a trick to access this pointer.

```

call    0x805
0x805: pop    eax      // get pc
        mov   esi, [eax-off] // load OCR = esi

```

Low-level functions of the kernel and device drivers require special CPU instructions (e.g. port access). Therefore PJC allows inlining of machine instructions. Furthermore the memory management needs to compute addresses without type checking. Therefore we introduce the not typesafe cast operation "Magic.Cast". As the compact Plurix run-time structures presented in section 3 cannot be modeled with the Java language we allow the compiler to construct them manually by directly accessing memory with a special language construct "Magic.Mem32[address]". This language construct permits access to the total virtual memory (0-4 GB) with an array operator.

These low-level language features are reserved for OS development and will not be available for application developers as this might jeopardize system security and stability.

## 5 Status and Outlook

We have presented the implementation of the Java language for the Plurix OS. We discussed the memory layout of run-time structures customized for the underlying DSM of the Plurix OS. We presented a method jump table construction scheme efficiently supporting inheritance, overwriting of methods and interfaces.

Compact monolithic run-time structures violate strict object-oriented modeling rules, but result in less heap objects and less indirections - less overhead for the memory manager and a performance boost for applications. Anyway the memory layout should be hidden from the application programmer and therefore we don't believe it is crucial to use monolithic run-time structures.

It is obvious that Java is well suited for hardware-level programming only if minor language extensions are present.

We have demonstrated at the CeBIT99 a first prototype of the Plurix OS running on three Pentium PCs connected via an 10 Mbit Ethernet. The bootstrapping of the Java compiler is imminent. After the compiler is bootstrapped and available on the Plurix machine we will study integration aspects between the compiler and the DSM. Furthermore we will study version management strategies to cope with the evolution of classes. We believe that our integrated design of compiler and OS will result in synergies and simplifications.

### Acknowledgment

We express our thanks to the contributions of Dr. S. Traub and A. Boehm.

## References

- [1] J.L. Keedy and D.A. Abramson. Implementing a large virtual memory in a Distributed Computing System. In *Proceedings of the Hawaii International Conference on System Sciences*, 1985.
- [2] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Pro-*

*ceedings of the International Conference on Parallel Processing*, 1988.

- [3] N. Wirth and J. Guteknecht. *Project Oberon - The Design of an Operating System and Compiler*. Addison-Wesley, 1992.
- [4] S. Traub. *Speicherverwaltung und Kollisionsbehandlung in transaktionsbasierten verteilten Betriebssystemen*. PhD dissertation, University of Ulm, Germany, Distributed Systems Department, 1996.
- [5] M.Schoettner S. Traub and P. Schulthess. A transactional DSM Operating System in Java. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1998.
- [6] J.C. Gylenhaal C.-H. Hsieh and W. Hwu. Java bytecode to native code translation: The caffeine prototype and preliminary results. In *Proceedings of the International Symposium on Microarchitecture*, 1996.
- [7] K. Driesen. *Method Lookup strategies in Dynamically Typed Object-Oriented Programming Languages*. PhD dissertation, University of Brussels, Belgium, Department of Computer Science, 1993.
- [8] A. Taivalsaari. Implementing a java virtual machine in the java programming language. Technical report, SUN Microsystems, 1998.
- [9] A. Krall and R. Grafl. CACAO - A 64 bit JavaVM Just-in-Time Compiler. In *Workshop on Java for Science and Engineering Computation*, 1997.
- [10] J. Vitek and N. Horspool. Compact dispatch tables for dynamically typed object oriented languages. In *Proceedings of the International Conference on Compiler Construction*, 1996.