

Implementation Aspects of a Persistent DSM Operating System in Java

M. Schoettner, O. Schirpf, M. Wende and P. Schulthess
Department of Distributed Systems, University of Ulm
89075 Ulm, Germany

ABSTRACT

The Java trademark encompasses the Java Virtual Machine (JVM), the Java language itself, and a large continuously growing class library. Beyond the development of applets in the context of the World Wide Web, more and more people use Java for large scaled standalone applications. This user group points out the performance disadvantage of the Java environment in comparison to sophisticated C/C++ compilers. As a result of this highly optimizing Java compilers begin to emerge which compile Java to machine code and abandon JVM-platform independence. The Plurix project goes one step further developing a native PC Operating System (OS) in Java. The central abstraction of the Plurix OS is a persistent Distributed Shared Memory (DSM). Our Plurix Java Compiler (PJC) translates Java sources into Intel protected mode code. It is itself written in Java and after bootstrapping it will become an integral part of the Plurix OS. In this paper we shortly review the persistent DSM environment of Plurix and give an overview on the architecture of PJC. We present how the basic run-time structures are modeled in Java and how PJC is built on top of it. The implementation of the Java language in the persistent Plurix DSM reveals interesting semantic issues. We suggest extending initialization rules for classes and an additional attribute clarifying semantic ambiguities of static variables. Finally, we present how a generalized type equivalence check scheme enhances flexibility in the Plurix DHS.

Keywords: Distributed Shared Memory, Persistence, Java, Operating Systems, Compiler, Run-Time Structures, Plurix..

1. INTRODUCTION

The Plurix project implements a lean distributed Operating System (OS) from scratch for the PC platform. We suggest that the distribution functionality should be moved into the OS and not be reimplemented by each distributed application. This is achieved using the well-known Distributed Shared Memory (DSM) paradigm [1], [2]. A DSM provides a virtual address space shared among tasks on loosely coupled nodes. The distribution of data on several computers is not noticed by the application programmer. Any reference can either point to local or remote memory blocks. During program execution the OS or run-time environment will detect a remote memory access and automatically fetch the desired

memory block. Numerous memory access detection schemes have been evaluated in previous research [7].

Plurix extends the traditional DSM paradigm to include shared code and shared data. Based on previous research in memory consistency models [4], [5], [6] we introduce a new transaction based memory consistency model [3], [10]. Therefore we prefer the term Distributed Heap Storage (DHS) instead of “DSM”. Plurix is the first DSM system not implemented on top of an existing OS discarding any overhead caused by commercially justified backward compatibility [11].

Language based OS development has been successfully demonstrated by native Oberon [9] e.g. The Operating System projects JavaOS and JOS are developing OSs in Java, but both rely on a Java Virtual Machine (JVM) written in C/C++ and do not offer DSM/DHS storage facilities [16], [17]. To us developing a native DHS system for the PC platform is an interesting alternative. We expect that using a popular programming language like Java will improve the acceptance for the new Plurix OS.

The system is developed using our Plurix Java compiler (PJC) which will eventually be an integral part of the OS. All fundamental run-time structures are modeled in Java and the Plurix Java Compiler (PJC) is constructed on top of them. This includes the creation of class descriptors, instances, and arrays. We were aware of other projects Cygnus and Jove working on highly optimizing Java compilers generating executables for existing OSs [18], [19]. However, to efficiently support our transactional consistency model we must ensure small working sets and short transactions [3], [11]. All operating systems classes must be compact and lean and we may not support all existing Java packages. The decision to develop our own compiler and optimized run-time structures is further explained in section 3.

The remainder of this paper is organized as follows. The persistent target environment – the Plurix OS is briefly reviewed in section 2.

Section 3 illustrates the architecture of the Plurix Java Compiler (PJC). The persistent DHS environment somewhat influences the compiler and the implementation of the Java language [12], [13], [14]. Section 4 presents how the basic Plurix run-time structures can be modeled in Java.

A minor language extension clarifying semantic problems of static variables in the Plurix DHS and an extended type checking algorithm are presented in section 5. Finally we conclude with the current status of the implementation and give an outlook on future work.

2. THE PLURIX ENVIRONMENT

Plurix is a standalone PC Operating System (min. 80486) entirely written in Java. The first prototype runs within a single LAN segment (10 Mbit/s Ethernet). The central abstraction within our design is a global address space shared by several nodes which is organized as a distributed heap storage (DHS). The DHS concept goes beyond traditional DSM systems in as much as it shares class descriptors and code segments.

Memory accesses (read or write) are monitored by the Memory Management Unit (MMU). Tasks in the OS are partitioned into restartable transactions. The DHS is orthogonally persistent – any Java object is by default persistent. Memory persistence and backup storage is initially provided by a disk-server within the Plurix cluster.

Distributed memory consistency

The Plurix DHS introduces an new memory consistency model: restartable DSM transactions and optimistic synchronization [3], [10], [11]. All transactions emanate from the idle loop in each node and memory consistency is guaranteed according to the ACID (Atomicity Consistency Isolation Durability) property.

The transactions obtain fully transparent access to the DHS. At Begin-Of-Transaction (BOT) all access privileges of present pages are set to read only. If the transaction (TA) accesses a memory location which is not present, the page-fault handler fetches the page and sets the page attributes to read-only and present. If the page-fault handler is invoked because a read-only page was accessed by a write operation, the handler stores the original page in a “before-image” and sets the privileges of the current page to read and write. The “before-image” is needed to provide undo capability. At End-Of-Transaction (EOT) the access flags in the hardware page tables are analysed to compute the read-set and the write-set. If the access flag of a page is set to “dirty”, the page will be included into the write-set. If the access flag of a page is set to “accessed” but the dirty flag is cleared, the page will be included in the read-set. If the TA is entering the commit-phase these read- and write-sets are used to detect possible collisions between participating nodes. Early measurement results are encouraging and show that a 10Mbit/s Ethernet Plurix cluster consisting of three machines achieves in excess of 600 transactions per second.

Collisions are resolved by time-proven optimistic synchronization techniques [8], [3]. If a collision between one or more transactions is detected all but one of these node must be restarted. This can be done by using the “before-images” saved earlier [3], [8], [11]. Transactions should be short to reduce collision probability. We believe a lean OS architecture tailor-made for the transactional DHS can result in short transactions with small working sets. It is optional for the application programmer to partition long transactions into transactions of a smaller granularity by explicitly inserting EOT. The OS automatically creates a BOT after a manually inserted EOT. Subtransactions may be an option if cascading aborts can be controlled.

Core OS services

Our transactional DHS supports two higher level memory management services: distributed garbage collection (DGC) and distributed object relocation (DOR). The DGC and the DOR services run as separate transactions and rely on the strong typing of Java. Both services need to know all global references to an object in the heap. For this purpose every heap object A is extended by a backchain pointer pointing to a pointer variable B which references A. Of course there can be many references to A. All these references are chained by a so called “back-chain” [3], [10]. Objects in the heap can be relocated by adjusting all pointer values in the backchain, see figure-1. If the backchain is empty the object is garbage and can be collected by the DGC.

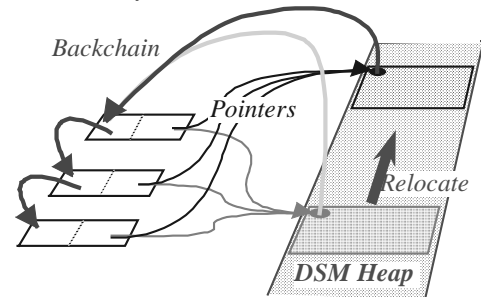


Figure-1, The backchain concept

The typical problem of false sharing [15] for paging based DSM systems might be alleviated by the DOR. The DOR can resolve a false sharing conflict between two objects in a single page by moving an affected object to another memory page. To deal effectively with the problem of false sharing we are still in need of a valid formal definition and an appropriate detection mechanism.

Memory blocks

To introduce the compiler run-time structures and basic run-time functions we shortly review the memory layout of a Plurix memory blocks.

Plurix implements the concept of ‘Two Headed Memory Blocks’ (THMB), see figure-2. The strong typing of the Java language guarantees that references can only point to the center of a memory block – to the header. The main purpose of the THMB concept is the separation of references from scalars simplifying garbage collection and relocation of memory blocks. References are addressed by negative offsets and scalars by positive ones.

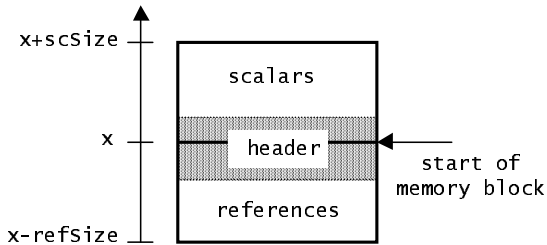


Figure-2, Plurix Two Headed Memory Block (THMB)

3. THE PLURIX JAVA COMPILER

The Plurix Java Compiler (PJC) abandons platform independence and translates Java sources into executable Intel protected mode code. As it is used for Operating System development we must generate machine code and can not afford to loose performance by interpreting code in a Java Virtual Machine (JVM). A Just-In-Time (JIT) compiler is not an alternative either because the kernel, drivers and interrupt handlers are written in Java using hardware oriented language extensions (see section 4) [20]. Modifying an existing full-size Java compiler for our purposes would certainly be complex and we would have to provide the substantial run-time support required by this compiler.

Currently the PJC doesn't use 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 byte-code). The current code generation is preliminary and uses a single stack allocation strategy. One of our research goals is to evaluate integration aspects between PJC and the Plurix OS after the bootstrapping of the compiler on the target machine.

PJC is tailored for the persistent DHS and directly writes run-time structures and code segments into the DSM and avoids to create object-, symbol-, library- and exe-files. It avoids complex serialization and deserialization operations typical for disk based filesystems.

Classes are statically bound by the compiler and no explicit linking and loading is required. Most initialisations can already be done by the compiler. Symbol information may be discarded and recreated on demand. PJC doesn't support separate compilation of classes at this time. A Java program must be stored together with all necessary run-time classes in a single source file.

In future we will support separate compilation and retain part of the symbol information for a class because transitive reconstruction of symbol information would be to expensive [9], [25].

Currently software is developed under Windows 95/NT inside a testing environment [10]. PJC is executed on a regular JVM and deposits its output not in a file but into a large virtual memory block (VMB). The VMB is allocated during process startup of the testing environment on the same Windows 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 standard machine-level debugger. After verification of the code generation we copy the image either via a serial link or a floppy disc to the Plurix machine where it can be executed.

4. BASIC PLURIX RUN-TIME STRUCTURES IN JAVA

The run-time structures required by Java are class descriptors, instances, arrays and code-segments. A detailed discussion of the memory layout of all PJC run-time structures is described in [20]. In the following section we discuss the bootstrapping strategy of PJC and how the basic Plurix run-time structures are modeled in the Java language.

The ancestor of all classes in the Plurix system is the class ‘‘PObject’’ describing the private memory management information stored in every THMB, see figure-3. A detailed publication on the distributed memory management and distributed garbage collection is planned. We shortly describe the content of the THMB header: ‘‘Flags’’ is for internal use of the memory management. The ‘‘Backchain’’ (see also section 2) entry points to the first *PObject* reference pointing to this memory block. We distinguish from regular pointer references (64-Bit) by representing the special ‘‘Backchain’’ pointer by a 32-Bit integer variable. The variables ‘‘Len’’ and ‘‘RelocLen’’ describe the bidirectional THMB sizes. The ‘‘Stopper’’ variable is no real pointer but a special bit pattern used to identify the beginning of the THMB. The pointers ‘‘Previous’’ and ‘‘Next’’ chain all memory blocks in the DHS. All Plurix instances have a type reference ‘‘Type’’ to the associated class. Finally, every Plurix class descriptor has an ‘‘Parent’’ pointer to its superclass. The ‘‘MetaParent’’ pointer will be described in section 5.

It might be confusing why the class descriptor is modeled as a subclass of instance. In the subsequent discussion the meaning of the terms *class descriptor* and *instance* depends on the point of view – we distinguish two perspectives:

- Compiler perspective
- Application perspective

```

class PObject {
    private int Flags;
    private int Backchain;
    private int Len, ReLocLen;

    private PObject Stopper;
    private PObject Previous, Next;
}

class PInstance extends PObject {
    PInstance Type;
}

class PClassDescriptor extends PInstance {
    PClassDescriptor Parent;
    PClassDescriptor MetaParent;
}

```

Figure-3, Basic Plurix run-time structures modelled with common Java classes

From the compiler perspective the class descriptors are instances of class “PClassDescriptor” and therewith have a “Type” pointer to this class. From the application view they are not instances but class descriptors representing a type for instances. For a better understanding we show the different views in a simple run-time snapshot, see figure-4.

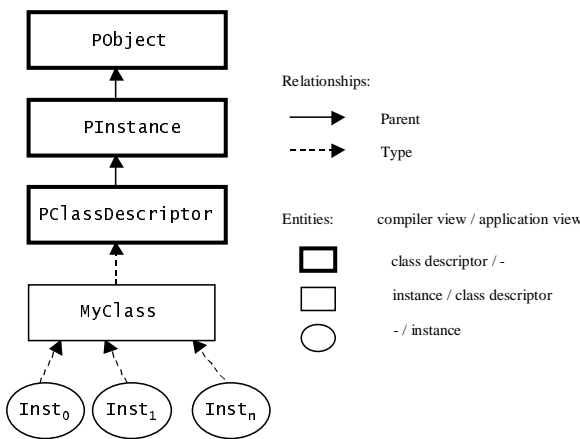


Figure-4, Basic Plurix run-time structures: entities and relationships from compiler & application point of view

Depending on the view the entities have different meanings. We do not define terms for basic class descriptors like “PObject” from application view and for instances of “MyClass” from the compiler view. The representation of the basic classes “PObject”, “PInstance” and “PClassDescriptor” is obvious. The breakpoint of the two different views is “MyClass” which is once an instance and then a class descriptor. Note the “Parent” pointer of “MyClass” is null and doesn’t point to “PClassDescriptor”. Generally, the “Parent” pointer of topmost classes in a inheritance hierarchy is always set to null.

Putting all together we present the resulting memory block headers for class descriptors and instances in figure-5.

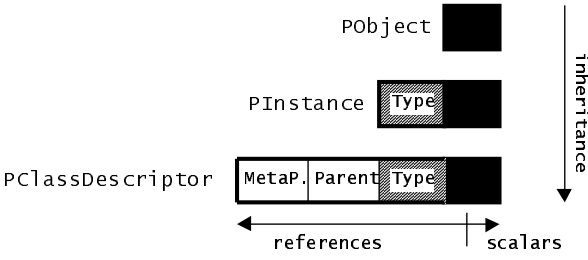


Figure-5, Basic memory block headers

Before the compiler is bootstrapped the classes “PObject”, “PInstance” and “PClassDescriptor” must be translated together with their Java programs. After pass-1 the compiler automatically extends all topmost classes in inheritance hierarchies by inserting “extends PClassDescriptor”. This causes the variables to be inherited. The compiler maintains two offset counters “staticOff” and “dynamicOff” for offset determination of static class variables and dynamic instance variables. Since only dynamic variables are replicated in their respective subclasses only the dynamic offset counters are propagated to the subclasses. Without modifications the “staticOff” would be erroneous zero for all program classes – and would destroy the THMB header. During the recursive offset computation the compiler knows when running through basic classes. The offsets of “PInstance” are used to set “dynamicOff” and the offsets computed after “PClassDescriptor” are used as “staticOff”, see figure-6.

```

class PObject { ... }

class PInstance extends PObject { ... }
    ———> dynamicoff

class PClassDescriptor extends PInstance { ... }
    ———> staticoff

class MyClass extends PClassDescriptor { ... }

```

Figure-6, Computation of run-time offsets for class descriptors and instances

Again the compiler view comes into play. Because class descriptors are instances of “PClassDescriptor” the dynamic offsets of “PClassDescriptor” are assigned to the “staticOff” of program .

During the bootstrapp process the compiler must create class descriptors manually. After it is bootstrapped it can use run-time functions to create class descriptors, see figure-7. The statements “Magic.Cast” and “Magic.Val” are hardware-level language extensions [20]. “Magic.Cast” allows arbitrary assignments without type checking. “Magic.Val” retrieves the address of run-time structures in the method “newDe-

descriptor” it is used to get the address of the class descriptor of “PClassDescriptor”.

```

public static PClassDescriptor NewDescriptor(
    int NonRelocSize,
    int RelocSize,
    PClassDescriptor parent
)
{
    PClassDescriptor newCD=null;
    int Address;

    // memory management stuff ...
    newCD = Magic.Cast(PClassDescriptor, Address);
    newCD.Parent = parent;
    newCD.MetaParent = null;
    newCD.Type = Magic.Val(PClassDescriptor);
    return newCD;
}

```

Figure-7, Creating new Plurix class descriptors

We conclude the discussion of basic run-time structure with a full-size sample showing entities and relationships including inheritance in program classes, see figure-8.

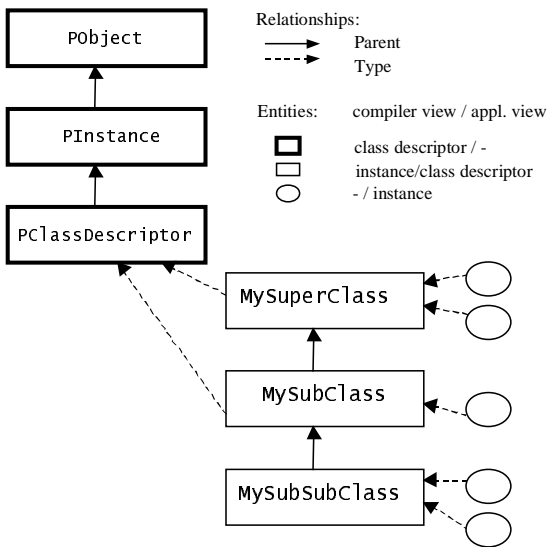


Figure-8, Plurix run-time structures

If the THMB header content needs to be modified we simply edit the Java sources of the basic class “PObject” and recompile the OS. In this way we achieve a tight integration between compiler and memory management. Static methods are not replicated in the method jump table of subclasses hence even basic classes can contain static methods without increasing memory consumption overhead in all run-time structures [20].

The keyword “new” is mapped by the compiler to one of the methods: “newInstance”, “newArray” and “newMultiArray” depending on the dimension requested by the call. These run-time functions are all implemented quite similar like “newDescriptor”.

5. AN EXTENDED JAVA TYPE CHECKING SCHEME

In this section we focus on the semantic ambiguity of static variables and inherited problems in the persistent Plurix DHS. Other semantic obscurities of the plain Java language (not in distributed systems) have been revealed in [22].

We shortly review the memory layout of the Plurix class descriptor. Modeling the class descriptor in a Java compatible manner would create many heap objects [20]. Abundance of heap blocks implies heavy pointer usage which is a burden to the memory management. This would be especially painful during object relocation situations in a DHS (see section 2). Furthermore every indirection involves a speed penalty at run-time. As a cure Plurix adopts a monolithic memory layout for class descriptors [20]. This approach is similar to SUNs JVM and [21] but both used C/C++ as an implementation language and not strongly typed Java.

Following the THMB memory block structure the class descriptor has the bidirectional memory layout shown in figure-9. If a class is publicly available ever task can read and write the static variables in common DHS storage.

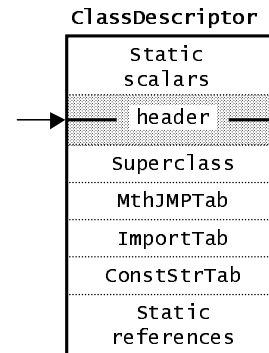


Figure-9, The Plurix bidirectional monolithic class descriptor

This distributed environment changes the semantics of static variables in shared classes. The central question in the persistent Plurix environment is: when or how often is a class initialized.

Previous publications have identified the problem but did not propose a solution, see [12], [13], [14]. The Java language specification defines: a class is initialized once, during the first access [23]. Adopting this rule for the Plurix DHS would mean only the first user accessing a class descriptor will see the initial values of static class variables. Due to the persistent DHS subsequent accesses in different transactions would not return the initial values and might not even return the last value written by this user. This distributed interpretation of the semantics of static class variables might confuse a typical application programmer.

Therefore we decided to introduce an additional attribute “*shared*” for clarifying the distributed semantic of static variables. Due to the monolithic memory layout of class descriptors it’s not possible to mix *shared* static variables and *non-shared* ones within a single class. Therefore “*shared*” is a new class attribute denoting that all static variables of that class might have been modified by other users and may not have their initial values. The implementation of shared classes is trivial as this is exactly the behaviour implemented by the DHS. The implementation of non-shared classes containing initialized static variables needs closer attention. A trivial option is to prohibit the publishing of non-shared classes. However this is undesirable because we agreed that the default behaviour of classes is non-shared and to prohibit class sharing in a DHS would be contradictory to the original goal of distributed operation. In the following text we assume non-shared class descriptors contain initialized static variables.

Assume user A created a non-shared class descriptor CD_A and makes it public available. If user B imports this class descriptor it must be replicated to CD_B ensuring that every user has its own set of static class variables. The replicated class descriptor CD_B is initialized ensuring user B sees the initial values. Thus we denote the extended initialization rules for Plurix classes:

- 1) shared classes are initialized once when created
- 2) non-shared classes are initialized once per user

Continuing the described scenario we assume both users A and B created several instances from their non-shared descriptors CD_A and CD_B . Again user A wants to publish an instance I_A of CD_A . This should be legal as the DHS offers data sharing. If user B imports I_A he can indirectly access the class descriptor CD_A . Nevertheless he can never create an instances I_B connected to CD_A . We believe it is acceptable that a published instance of a non-shared class can access the associated class descriptor.

Unfortunately the instances I_A of user A and I_B of user B are no longer type-compatible. The type equivalence check “*instanceof*” fails because they are assigned to different class descriptors CD_A and CD_B .

The type equivalence check of Java bases on the name equivalence. This check can be efficiently carried out by reducing the string matching to a pointer comparisons.

PJC maps the “*instanceof*” keyword to the runtime method “InstOf” implementing the type check, see figure-10. The parameter “inst” is dereferenced to get the associated pointer to the class descriptor. The class descriptor pointer is compared to the given type “kl” (also a pointer to a class descriptor). The inheritance chain is searched in upward direction for a match.

```
public static boolean InstOf(
    PInstance inst,
    PClassDescriptor kl
)
{
    PClassDescriptor instClass;

    if (inst==null) return false;
    instClass = inst.Type;
    do {
        if (instClass == kl) return true;
        instClass = instClass.Parent;
    } while (instClass!=null);
    return false;
}
```

Figure-10, The type equivalence check of Java

A *Meta Class Descriptor* (MCD) can solve the problems of non-shared classes, see figure-12. MCDs are automatically generated by the compiler during the creation of non-shared class descriptors. Every standard class descriptor is extended by an *meta-parent* pointer (see section 4) additional to the normal parent pointer which is set to *null*. The *meta-parent* pointer is only set for non-shared classes pointing to their associated MCD. The type equivalence check “*instanceof*” is modified preferring valid present *meta-parent* pointers resulting in successful type checks, see figure-11.

```
public static boolean ExtendedInstOf(
    PInstance inst,
    PClassDescriptor kl
)
{
    PClassDescriptor instClass;

    if (inst==null) return false;
    instClass = inst.Type;
    do {
        if (instClass == kl) return true;
        if (instClass.MetaParent!=null)
            instClass = instClass.MetaParent;
        else
            instClass = instClass.Parent;
    } while (instClass!=null);
    return false;
}
```

Figure-11, The extended type equivalence check of Plurix

The MCDs are fully handled by the compiler and are invisible to the programmer. He can publish non-shared classes without worrying about other users modifying his private set of static variables and instances. At the same time all instances of non-shared classes of different users can be published and remain compatible.

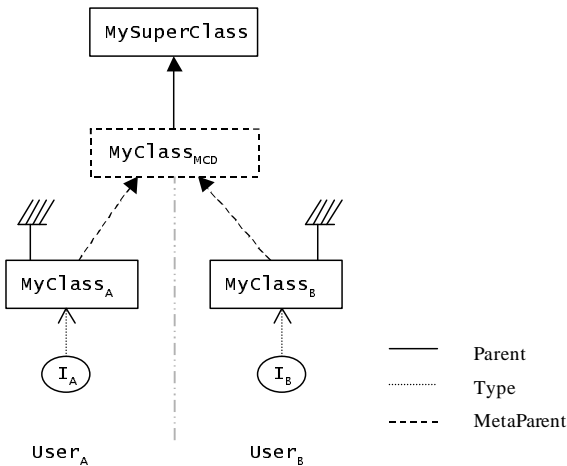


Figure-12, The Plurix Meta Class Descriptor (MCD)

If a class C^{NS} is denoted *non-shared* all subclasses of C^{NS} must also be non-shared. Static variables of parent classes are accessed via the import table as they are statically bound [20]. Which parent class descriptor should a shared subclass of C^{NS} use in its the import table (there might be several parents)? Importing the MCD is not possible as it doesn't contain static variables and is only used for type checking. There might be none or many "shared" class descriptors from the top of the inheritance hierarchy but after the first occurrence of a non-shared class descriptor below only non-shared ones can follow, see figure-13.

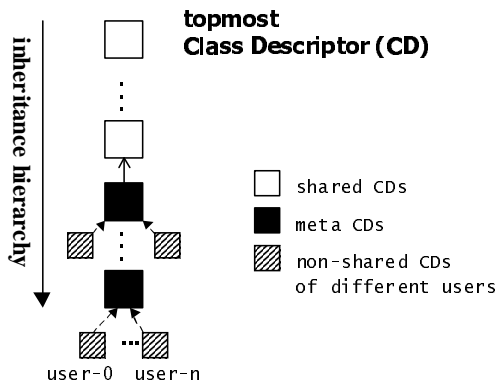


Figure-13, Valid usage of the attribute "shared" within one inheritance hierarchy

The constraints discussed above must be enforced by the semantic analysis of the compiler. Furthermore the compiler must support the transition from shared to non-shared classes by setting the parent pointer of the first meta class descriptor to the last shared class descriptor.

Cutts [24] exacts a structural type equivalence check for persistent programming environments. But he also admits that the equivalence check costs are substantially higher.

Structural equivalence would also solve the problem of incompatible instances for private class descriptors but it would totally change the type system of Java and certainly cause a lot of new semantic ambiguities and undesired side-effects.

6. CONCLUSIONS

We have presented implementation aspects of the Java language in the Plurix environment. We briefly reviewed the persistent DHS environment of Plurix and the architecture of the Plurix Java Compiler (PJC). We discussed how basic run-time structures for class descriptors and instances are modeled in Java and how PJC is build on top of them.

Furthermore we discussed the arising semantic problems of static class variables in a persistent DHS in detail. A new class attribute "shared" clarifies this point. Shared classes are initialized once whereas non-shared ones are initialized once per user during replication. Replicating non-shared class descriptors introduces the problem of incompatible instances. We proposed a meta class descriptor and an extended type checking scheme as a solution to this problem.

We plan to investigate the side effects produced by the introduction of meta class descriptors and the Plurix initialization rules. We intend to optimize memory layout of run-time structures with respect to type evolution. The main goal is to avoid transitive trickle-down recompilations which are typical for persistent programming environments [25].

Currently we have developed a first prototype of the Plurix OS running on three ore more Pentium PCs connected via an 10 Mbit Ethernet. A 360 degree panorama image is kept in DHS storage and horizontally scrolled by permanently transferring memory pages over the network. Initially the image is read from a local disc and each station is responsible for obtaining its section of the total image. The achievable frame rate exceeds 100 fps. We presented this prototype during the CeBIT99 fair.

The bootstrapping of the Java compiler is going into its hot phase. After the compiler is bootstrapped and available on the Plurix machine we will study integration aspects of compiler and DHS. We believe that an integrated design of compiler and OS will result in synergies for both entities.

7. ACKNOWLEDGMENTS

We express our thanks to the contributions of Dr. Stefan Traub and Andreas Boehm (Diploma thesis: "A Java bytecode translator for Plurix").

8. REFERENCES

- [1] J.L. Keedy and D. A. Abramson, "Implementing a large virtual memory in a Distributed Computing System", In *Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences*, 1985
- [2] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing", In *Proceedings of the International Conference on Parallel Processing*, 1988.
- [3] S. Traub, "Speicherverwaltung und Kollisionsbehandlung in transaktionsbasierten verteilten Betriebssystemen", PhD thesis, University of Ulm, 1996.
- [4] *A comprehensive bibliography of distributed shared memory*, Technical Report TR96-17, Department of Computing Science, University of Alberta, 1996
- [5] P. Keleher et al., "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems", In *USENIX*, Winter 1994
- [6] W. Zwaenepoel J. B. Carter, J. K. Benet, "Implementation and Performance of Munin", In *Proceedings of the 13th ACM Symposium Operating System Principles*, 1991
- [7] Zekauskas M. J., Sawdon W. A. and Bershad B.N., "Software Write Detection for a Distributed Shared Memory", *Operating Systems Design and Implementation*, 1994
- [8] H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control", In *ACM Transactions on Database Systems*, 1981.
- [9] N. Wirth and J. Gutknecht, *Project Oberon – The Design of an Operating System and Compiler*, Addison-Wesley, 1992
- [10] S. Traub, "The Design of a Distributed Oberon System", In *Proceedings of the Joint Modular Languages Conf.*, Ulm, Germany, 1994
- [11] 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*, Las Vegas, USA, 1998.
- [12] A. Atkinson, M. Jordan, L. Daynes and S. Spence, "Design Issues for Persistent Java: a type-safe, object-oriented orthogonally persistent system", In *Proceedings of the International Workshop on Persistent Object Systems*, 1996.
- [13] A. Malhotra, "Persistent Java Objects: A Proposal", In *Proceedings of the International Workshop on Persistence and Java*, 1996.
- [14] S. Spence, "Distribution Strategies for Persistent Java", In *Proceedings of the International Workshop on Persistence and Java*, 1996.
- [15] M. L. Scott and W. Bolosky, False Sharing and its effect on shared memory performance, Technical Report MSR-TR-93-01, Microsoft Research, One Microsoft Way, Redmond, WA 98052, 1993
- [16] "A free Java based Operating System", 1998, <http://jos.org>.
- [17] "JavaOS", <http://www.sun.com/javaos>, 1997.
- [18] Per Bothner, "A Gcc-based Java Implementation", IEEE Comcon, 1997
- [19] "Jove", Technical Report, Instantiations, Inc., 1998
- [20] M. Schoettner, O. Schirpf, M. Wende and P. Schulthess, "Implementation of the Java language in a persistent DSM Operating System", to appear in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA, 1999.
- [21] C.-H. A. Hsieh, J. C. Gylenhaal and W. m. W. Hwu, "Java bytecode to native code translation: The caffeine prototype and preliminary results", In *Annual IEEE/ACM International Symposium on Microarchitecture*, 1996
- [22] E. Boerger and W. Schulte, "A Programmer Friendly Modular Definition of the Semantics of Java", In: J. Alves-Foss (Hrsg.): *Formal Syntax and Semantics of Java™*. LNCS, Springer, Frühling / Sommer 1998.
- [23] J. Gosling, B. Joy and G. Steele, "The Java Language Specification", Addison-Wesley, 1996
- [24] Q. I. Cutts, "Delivering the Benefits of Persistence to System Construction and Execution", Ph. D. Theses, University of St. Andrews, 1992
- [25] R. B. J. Crelier, "Separate Compilation and Module Extension", Ph. D. Thesis, ETH No 10650, ETH Zurich, 1994