Architecture of an Object-Oriented Cluster Operating System

M. Schoettner, O. Marquardt, M. Wende, and P. Schulthess
Department of Distributed Systems,
University of Ulm, Germany
[schoettner|marquardt|wende|schulthess]@informatik.uni-ulm.de

Abstract *The Plurix project implements an object-oriented Operating System (OS) for PC clusters. Network communication is implemented via the well-known Distributed Shared Memory (DSM) paradigm using restartable transactions and an optimistic synchronization scheme to guarantee memory consistency. We believe using a DSM management for a general purpose OS offers interesting perspectives, e.g. for simplified development of distributed applications. The OS (including kernel and drivers) is written in Java using our own Plurix Java Compiler (PJC) translating Java source texts directly into Intel machine instructions. PJC is an integral part of the language-based OS and tailor-made for compiling in our persistent DSM environment. Furthermore, device-level programming is supported by some minor language extensions. In this paper we illustrate the architecture of the DSM kernel and integration aspects of PJC. We present early performance measurement results and discuss experiences of using the Java language for OS development.*

Keywords: Operating System, Object-Orientation, Programming Languages

1 Introduction

Plurix is an object-oriented Operating System (OS) for the PC platform totally written in Java. We abandon the hardware independence of Java as we do not rely on a Java Virtual Machine (JVM) like JavaOS does [1]. Herewith we gain efficiency, speed, and minor language extension allow us to use the Java language for hardware programming. Language based OS development has been successfully demonstrated by systems like native Oberon [2]. Plurix is not implemented on top of an existing OS discarding any overhead caused by commercially justified backward compatibility.

Distributed Shared Memory (DSM) offers a natural solution for distributing data among several nodes [3] and Li [4]. Applications running on top of the Plurix DSM are unaware of data locations. Any reference can point to local or remote memory blocks. During program execution the MMU detects a remote memory access and automatically fetches the desired memory block.

A disk filing system can be avoided by implementing orthogonal persistence [5]. There have been efforts to add persistence to existing languages without disturbing their semantics and implementation [6]. A more promising approach is to directly support persistence in the OS [7]. One of our research goals is to provide a persistent DSM compromising the kernel and compiler.

Little is found in the literature about adding persistence to DSM systems [8]. We are aware of other object-oriented OSs implementing persistence like Grasshoper [7] and Charm [9] but to the best of our knowledge no other OS project combined the properties of Plurix: persistence, DSM, and object-orientation.

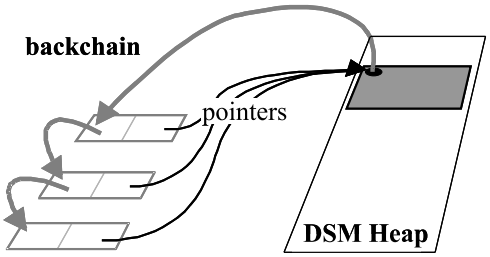

Figure 1: Backchaining references

Our system is developed using a proprietary Plurix Java Compiler (PJC) constituting an integral part of the PLurix OS. All fundamental runtime structures are modeled in Java and the kernel and the PJC are built on top of them [10]. PJC is also responsible for linking and initialization of classes [11].

The remainder of this paper is organized as follows. The properties of the Plurix environment including integration aspects of the compiler are presented in section two. The following section illustrates the object-oriented architecture of the Plurix kernel including some preliminary measurement results of our DSM system. Finally, we report experiences we gained from using the Java language for hardware-level programming and give an outlook on future work.

2 The Persistent DSM Environment

2.1 The Operating System

The current prototype runs within a single LAN segment (100 Mbit/s Ethernet). The central abstraction within our design is a Distributed Shared Memory (DSM) sharing data and code. A node runs one or several cooperative tasks which are controlled by a central loop similar to the Oberon system-loop [2]. Tasks in the OS are partitioned into restartable transactions. Together with an optimistic synchronization scheme restartable transactions implement the consistency model of Plurix [12]. Memory accesses (read or write) are monitored by the MMU. There may be many read-only replications of a memory page until a write to that page occurs.

The Plurix DSM is *orthogonally persistent* - any Java object reachable from a global name service root persists including classes and code. Memory persistence is initially provided by a central disk-server within the Plurix cluster which will be substituted by a distributed solution.

A distributed Garbage Collection (GC) relieves programmers of the burden of memory management. The GC keeps track of all references to an object by linking them in a so called *backchain* [12]. Every heap object includes a backchain pointer locating the first reference pointing to it, see figure 1. If there are several references to an object they are chained together. If the backchain is empty an object is garbage.

The current Plurix user interface (figure 2) continues the Oberon tradition [2]. The viewer concept is adopted including executable text elements. Event handling is modified to mirror the event listener model of Java [1].

Commercial OSs typically use multiple name services implemented in different components (e.g. file system, address book, ...). The persistent DSM concept lends itself to the implementation of a single cluster-wide namespace. Each instance can be registered in the name service by creating a named entry storing a reference to the object. Packages and classes are automatically registered by the compiler.

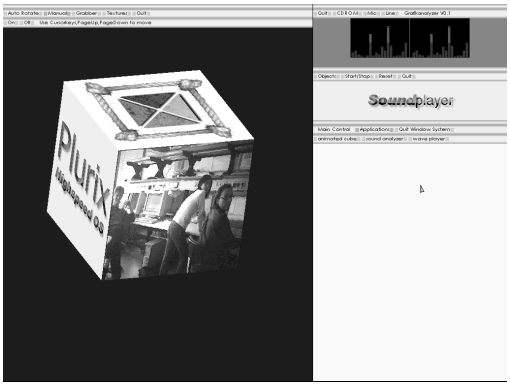

Figure 2: CeBIT 2001 snapshot

2.2 The Plurix Java Compiler

Plurix development uses a proprietary Java compiler translating source texts directly into machine instructions. The first version of the Plurix Java Compiler (PJC) doesnt use an intermediate representation nor state of the art code optimizations. Parser and code generator are coalesced resulting in a compact compiler size (150 Kb byte-code).

The PJC is written in Java and bootstrapped to the Plurix world where it is used for device-driver and application development. Additionally, we have a cross compilation facility running under Micosoft Windows or Linux.

The compiler is tailored for the persistent DSM and automatically registers parts of the symbol table in the name service by melting the scope concept with the directory concept. Storing symbol information in the name service lays the foundation for a textual user interface like known in the Oberon system [2]. Java reflection functionality can be offered with no additional efforts. Runtime structures and code segments are directly generated avoiding traditional object-, symbol-, library- and exe-files. If a source text is successfully compiled the symbol class descriptor (SyCD) is registered by the compiler in the cluster-wide name service. The corresponding runtime class descriptor (RtCD) is constructed and attached to the SyCD. The RtCDs are linked via an import table if necessary. Finally the RtCDs are initialized by the compiler and are ready for execution from any node.

Persistent object systems need to cope with the problem of type evolution [5]. Types are modified due to different reasons and existing class relationships and instances must not be invalidated. We intend to use the backchain and the full persistent type information to design a reasonable strategy.

3 The Plurix Kernel

The Plurix kernel is very lean and currently consists of 31 classes totalling to 4238 lines of source code. This package implements the heap management, the automatic garbage collection, and the transactional memory consistency model, see figure 3. The DSM network protocols run on top of IP allowing integration of nodes outside the LAN in future versions.

All memory blocks are typed using compiler-known base classes e.g. PObject which are automatically attached by the compiler. Even the memory management works with typed memory blocks. Hence the heap only stores instances, arrays, and type-descriptors (class- or interface-descriptors). The PJC uses the class PMemory to allocate new objects, supporting the backchain, and for runtime type checks.

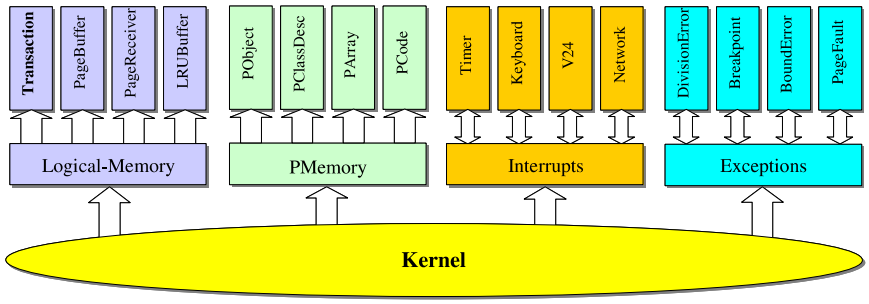

Figure 3: The kernel classes

The Plurix kernel does not implement the typical user-kernel-mode separation to protect the kernel because this would be a contradiction to the DSM paradigm. Access to kernel routines is controlled by hiding parts of the name service. Hence, the kernel interface can be hidden form applications and is only visible for driver developers. The strong typing of the Java language lays the foundation for our security model and only code from a trusted compiler is allowed to execute. We are aware of the need to investigate sophisticated access control mechanisms in future work.

3.1 Device Driver Model

For each hardware bus in the system a dedicated bus driver is responsible to collect information about attached devices during configuration time - similar to the Microsoft Windows Driver Model. Scanning the PCI bus yields different configuration information including memory, interrupt, and I/O-Port requirements for each device. Furthermore each PC-device is uniquely identified by a hardware and vendor ID. These strings are used to query the name service for finding the proper device driver. Drivers are represented by instances which may be persistent and need only be created during the initial device detection. Plurix device drivers need to be aware of the persistent environment and must be able to reinitialize using previous persistent instances. All drivers extend the abstract class "device" defining the minimal driver interface.

Drivers offering interrupt service must register their driver instance at the kernel. The kernel uses a two-level interrupt handling scheme supporting shared interrupts by chaining the devices sharing an interrupt, see figure 4. Regular methods retrieve their class and instance context from their stack frame which is not possible for interrupt handlers as they are called directly from the hardware.

Their context is set by the first level interrupt handler instead together with sending the EOI to the interrupt controller. The IRQ number from the state register of the 8359 chip is used to retrieve the instance from the level two dispatch table. The instance can be used to properly call the interrupt handler of the device driver.

3.2 Programming the Hardware in Java

The Java language uses the sandbox approach to prevent applets from accessing the hardware. In our case, however, direct and fast access to the hardware was crucial to develop the kernel and drivers. Hence, the PJC includes minor language extensions to cope with these requirements. In place of the privileged module "SYSTEM" for device oriented programming in Oberon we provide a pseudo class "Magic" which hosts conversion operators and the in/out instructions required for Intel CPUs. The special invocation and return sequence for interrupt handlers is indicated to the compiler by an additional method attribute "interrupt".

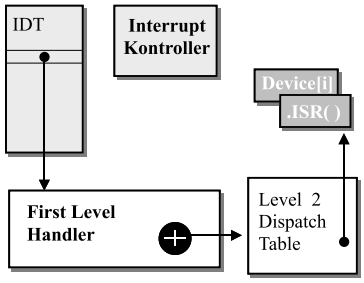

Figure 4: Interrupt Handling in Plurix

In Oberon it is very convenient to mimic memory mapped device registers by defining a record structure with suitable fields and basing it over the memory area which is occupied by the device. Unlike Oberon, however, the structure mapping of Java is not capable of simulating memory based device registers. We implement such a feature by defining a base class called "PMapper" indicating the compiler to use this class for device register mapping purposes only. The class can not be instantiated by using "new" instead a reference is created by calling the method "PMapper.MapAt(int address)" which can subsequently be used for accessing device registers in a structured way.

3.3 Preliminary Measurement Results

Although we are currently working with a proprietary code generation scheme, the kernel performance is promisingly. Page requests over the network can be processed in less than 1ms (100 Mbps). The local shadow page creation exceeds 50 Mbytes per second. Compiling with the native version of PJC (running as a transaction in the DSM) is able to translate a source text of 2250 lines in 210ms including runtime structure generation, binding, and initialization on a 1 GHz Athlon machine. This is a compilation speed of 10,000 lines per second.

4 Experiences and Future Work

Our experience with object-oriented programming in the kernel is generally positive. We have developed the kernel and several drivers: S3, ATI Radeon, V24, ATAPI, RLT8029, and keyboard. Objects provide a cleaner design because of the encapsulation of data structures and the enforcement of well-defined interfaces. Furthermore, using a strongly typed language like Java eliminates several typical error classes (e.g. pointer arithmetic, type errors, index range checking) well-known from programming drivers in C. After some prejudices most people were convinced by programming device drivers in Java.

The code quality of our non-optimizing compiler can not compete with sophisticated code from aggressive optimizing C-compilers. Nevertheless, several compiler projects (e.g. Marmot [13], Jove [14], and Swift [15]) are applying regularv optimization strategies to the Java language to improve the performance. We are currently designing a next generation compiler using a SSA-based intermediate representation together with an optimizing backend for the IA64-architecture.

The kernel is quite stable and we have shown a prototype of the Plurix OS running on three Pentium PCs connected via an 100 Mbit/s Ethernet at the CeBIT 2001 fair. In future we will investigate different memory consistency models, attack false-sharing, and design a distributed page server.

References

[1] T. Saulpaugh and C. Mirho. *Inside the JavaOS Operating System.* Addison-Wesley, 1999.

[2] N. Wirth and J. Guteknecht. *Project Oberon - The Design of an Operating System and Compiler.* Addison-Wesley, 1992.

[3] J.L. Keedy and D.A. Abramson. Implementing a large virtual memory in a Distributed Computing System. In *Proc. of the Hawaii International Conference on System Sciences*, Hawaii, USA, 1985.

[4] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the International Conference on Parallel Processing*, 1988.

[5] M.P. Atkinson and R. Morrison. Orthogonal Persistent Object Systems. In *Very Large Data Base Journal 4(3)*, 1995.

[6] K.J. Chisholm M.P. Atkinson and R.M. Marshall. Algorithms for a Persistent Heap. In *IEEE Software, Practice and Engineering, 13(3)*, 1983.

[7] R. di Bona A. Dearle S. Norris J. Rosenberg A. Lindstroem and F. Vaughan. Persistence in the Grasshopper Kernel. In *Australasian Computer Science Conference*, Australia, 1995.

[8] C. Morin and I. Puaut. A survey of recoverable Distributed Shared Memory Systems. In *Technical Report Nr. 975*, IRISA, France, 1995.

[9] A. Dearle and D. Hulse. Implementing Self-Managing Protection Domains in Charm. In *Proc. of 3rd Workshop on Object-Orientation and Operating Systems*, Sophia Antipolis, France, 2000.

[10] M. Schoettner O. Schirpf M.Wende and P. Schulthess. Implementation Aspects of a Persistent DSM Operating System in Java. In *Proc. of the International Conference on Information Systems Analysis and Synthesis*, Orlando, USA, 1999.

[11] M. Schoettner O. Marquardt M. Wende and P. Schulthess. Multiple Subtyping in a Persistent Distributed Shared Memory Operating System. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA, 2000.

[12] S. Traub. *Speicherverwaltung und Kollisionsbehandlung in transaktionsbasierten verteilten Betriebssystemen.* PhD thesis, University of Ulm, Germany, Distributed Systems Department, 1996.

[13] T.B. Knoblock R. Fitzgerald and E. Ruf. Marmot: An optimizing compiler for java. Technical Report MSR-TR-99-33, Microsoft Research, USA, 1999.

[14] Jove. Technical Report Technical Report Instantiations, Instantiations, USA, 1998.

[15] K.H. Randall S. Ghemawat D.J. Scales and J. Dean. The swift java compiler: Design and implementation. Technical Report Technical Report 2000/2, Compaq Western Research Laboratory, USA, 2000.