

Compiling in a Persistent Distributed Shared Memory Environment

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

Abstract

Plurix is a general purpose Operating System (OS) developed for the PC platform. Network communication is implemented by a Distributed Shared Memory (DSM). Restartable transactions and optimistic synchronization guarantee memory consistency. Making the DSM persistent allows us to abandon traditional file systems. The OS is developed using our own Plurix Java Compiler (PJC) translating source texts into machine instructions. PJC is an integral part of the OS and is used for program development. The persistent environment allows direct runtime structure generation. Symbol table entries are automatically registered by PJC in a cluster-wide name service and therewith survive the compilation process. After a short review of the persistent DSM environment we present integration aspects of PJC. Furthermore, we describe the implementation of our separate compilation scheme benefitting from the integration of OS and compiler. Finally, we present measurement results.

Keywords: Compiler, Separate Compilation, Persistence, Distributed Shared Memory

1 Introduction

Network communication in the Plurix Operating System (OS) is performed using the Distributed Shared Memory paradigm introduced by Keedy [1] and Li [2]. Plurix implements its own memory consistency model using restartable transactions combined with an optimistic synchronization scheme [3]. The aim of our transactional consistency model is not as much the achievement of highly paral-

lel processing rather one of our main research goals is to investigate the DSM as general purpose communication medium e.g. to simplify the development of distributed applications. Message-passing over TCP/IP is used for accessing the Internet.

Plurix implements orthogonal persistence [4] for DSM in the sense that any object reachable from a cluster-wide name service root can persist independently of its type. Our system supports persistence at the OS level like the Grasshopper project [5]. Even the OS and the compiler reside in the persistent DSM. We release programmers from the burden of writing complex serialization and deserialization operations necessary to load and store objects in file-based systems.

Our Plurix Java Compiler (PJC) is used for application and OS development, directly translating Java source texts into Intel machine instructions [6]. We hope to increase acceptance for a new OS by using a popular language like Java although some minor language extensions were necessary to support hardware-level programming. PJC is written in Java and bootstrapped to the Plurix world of which it becomes an integral part. Language based OS development has been successfully demonstrated in systems like Oberon [7].

In the first section, we shortly review the properties of the Plurix system. Subsequently, we present the compiler architecture tailored to the persistent DSM environment. This includes integration of symbol table entries into the name service, direct runtime structure generation, adaptive binding, and initialization.

In the fourth section, we present the design and implementation of our separate compilation scheme. We evaluate our design by performance measurements. Finally, we illustrate future work areas.

2 The Plurix DSM

The Plurix DSM is organized as a distributed heap storage (DHS). The DHS concept goes beyond traditional DSM systems in as much as it shares not only data but also class descriptors and code segments. The current setup runs within a single LAN segment (100 Mbps Ethernet).

Each node in a Plurix PC-cluster is controlled by a central loop; threading through a number of cooperating tasks [7]. The tasks are not separate processes because they share the single DSM address space. Furthermore there is also no distinction between kernel-mode and user-mode memory. Having one global address space is no limitation for future 64-Bit CPUs [8]. Alternatively, memory protection can also be realized using segment descriptors or a strongly typed languages like Java.

Memory accesses (read or write) are monitored by the Memory Management Unit (MMU). There may be many read-only replications of a memory page until a write to that page occurs. Tasks in the OS are partitioned into restartable transactions. The Plurix memory consistency model uses an optimistic synchronization scheme. During a running transaction, the DSM seems to implement a sequential consistency model. However, from the application point of view it is rather a strict consistency model ensured by the ACID property of transactions and the optimistic synchronization scheme. The memory consistency model has been described in [3].

The DHS is *orthogonally persistent* in the sense that any Java object may persist [9]. Memory persistence and backup storage is initially provided by a central disk server within the Plurix cluster. A distributed solution will be developed to improve fault tolerance.

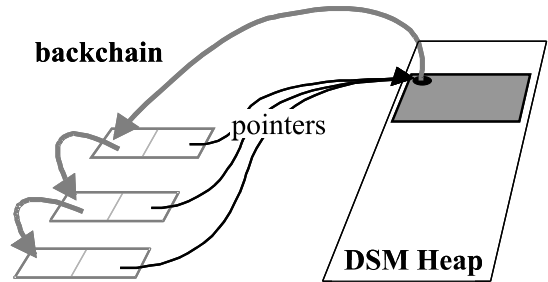


Figure 1: Backchaining references

An automatic distributed Garbage Collection (GC) relieves programmers of the burden and of the risks of explicit memory management. The GC keeps track of all references to an object by using the *backchain concept* [3]. Every heap object includes a backchain pointer locating the first reference pointing to it, see figure 1. All references to a single object they are chained together. If the backchain of an object is empty, no more references to that object exist and it is garbage. To simplify the backchain implementation references must always point to the header of a heap object, not into the body. The compiler directly supports the backchain by inserting a call to a runtime function if an assignment to a heap pointer is performed. References on the stack are not inserted into the backchain because GC only runs if the stack is empty. Hence, we avoid performance penalties for temporary stack references.

Typically, commercial Operating Systems implement numerous name services in different components (e.g. file-system, addressbook, ...). A persistent DSM environment facilitates the implementation of a single cluster-wide name service. All memory blocks reachable from the global name service root persist.

3 Compiler Architecture

The Plurix Java Compiler (PJC) transforms Java source texts directly into machine instructions. As PJC is used for OS development we are forced to abandon the hardware independence of Java to avoid performance penalties

by a Java Virtual Machine (JVM). The compiler itself is written in Java allowing to bootstrap itself into the emerging Plurix environment. We are aware of other Java compiler projects like Marmot [10] and GNU Java Compiler [11] but we needed an integrated solution and were not willing to provide the runtime environment required by those compilers.

The current compiler prototype does not provide an intermediate representation (e.g. Static Single Assignment form) nor code optimizations. Parser and code generator are coalesced resulting in a small compiler size (< 150 kb byte-code). One of our present research goals is evaluating integration aspects between compiler and the persistent DSM environment. An optimizing compiler using SSA is currently being developed.

3.1 Persistent Symbol Tables

During the semantic analysis phase the compiler registers symbol table entries in the cluster-wide name service. This is achieved by melting compiler scopes with the directory concept. Directories define a scope for their content; classes do the same for their methods and instance variables. Users can browse through the name space and transparently access data, classes, and methods.

The compiler directly operates on the name service resolving names according to the scoping rules of the Java language. Debugging information is naturally available without distinction of release and debug versions like in current commercial systems. The task of separate compilation is simplified by the persistent symbol tables, discussed in section four.

Persistent symbol table entries also lay the foundation for a textual user interface similar to the one used in the Oberon system [7]. Texts can contain user commands of the form *classname.methodname* which are resolved on mouse click using the name service to execute the desired method.

3.2 Runtime Structure Generation

As Plurix implements an orthogonal persistent DSM the compiler creates runtime structures and code segments directly in the DSM avoiding the traditional generation of object-, symbol-, library-, and exe-files. Even no separate linker nor a loader is required in our persistent DSM world. After successful compilation the runtime structures are bound and initialized by the compiler and are ready for execution, even from other nodes in the cluster. The memory layout of the runtime structures has been described in [6].

Although binding is performed statically by the compiler we do not sacrifice extensibility. We are able to alter or upgrade existing types using the bookkeeping of references realized by the backchain concept and a cluster-wide name service. We call this extended binding scheme *adaptive linking* described previously in [12].

4 Separate Compilation

Traditionally, compilers generate object- and symbol files to support separate compilation. The binder subsequently reads in symbol files to resolve inter-module or inter-class references. The linking task is of recursive nature because linking one class requires linking all imported classes first. It is the task of the binder to check consistency of inter-module relationships to prevent runtime errors.

Modules may be extended or modified over the time and lead to invalidations of client references. An invalidated client needs to be recompiled and in the case of an incompatibility edited by the user. This procedure always allows to restore module consistency by recompiling all modules in the system which might be time consuming. In today's typically distributed world client invalidations are still more crucial because simple changes can not rely on recompilation of distributed components. The main goal must be to avoid client invalidations in a distributed system.

In file-based systems this problem is known as module extensibility and the granularity of

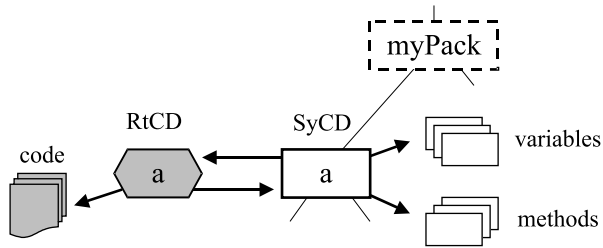


Figure 2: Symbol information in the name service

consistency checks at linktime needs to be chosen carefully. Tests basing on coarse granularity can be performed fast but easily result in trickle down compilations of many modules. Two fine grained consistency checking schemes were proposed by Crelier for the Oberon language avoiding unnecessary invalidations of modules [13]. He computes fingerprints for every exported object (variable, procedure, or record). Importing an object is done by storing the corresponding fingerprint in the object file. This approach is appropriate for a file-based environment like the Oberon System but persistent environments have further requirements. Modifications do not only affect code or classes but persistent instances may also be invalidated. To support type evolution we have to handle three tasks: providing enough type information, possibility for upgrading classes & instances, and managing different versions.

Type information can be provided by making parts of the symbol table persistent. Here-with we avoid creation and maintenance of symbol files. By integrating symbol table entries in our cluster-wide name service separate compilation is simplified. Symbol class descriptors (SyCD) are automatically registered in the name service by the compiler, see figure 2. A SyCD points to method, variable descriptors, and to the corresponding runtime descriptor. The native code segments are attached to the runtime class descriptors (RtCD).

The compiler transparently accesses new and already existing classes. Even the location is transparent because symbol table entries currently resident on other nodes are automatically fetched by the DSM during the

first access in a compilation run. If a class of the current compilation package is already stored in the name service a temporary symbol table entry is allocated.

Before binding starts the compiler checks upgrading candidates for *binary compatibility* with their ancestors. The Java language specification 2.0 defines the term binary compatibility "A change to a type is binary compatible with ... pre-existing binaries if pre-existing binaries that previously linked without error will continue to link without error". Java uses lazy binding which might cause binding errors at runtime. Binary compatibility rules may also cause runtime errors e.g. ErrorNoSuchMethod.

As we compile the Java language directly into machine instructions and use static binding we avoid such runtime errors. On the other side we accept reduced flexibility regarding binary compatible modifications. For example class hierarchy modifications are allowed by the Java language specification but in our case will invalidate subclasses since offsets in the method jump tables will be affected.

If an upgrading candidate has not been extended nor been shrunk, compatibility is obvious and the new code segments can be attached to the old version and the classes of the current compilation are bound to the old class descriptor version. In that case the instances of the upgrading candidate are not affected and will automatically use the new code versions.

If a candidate is extended in a binary compatible way all clients of the old class descriptor are bound to the new version including the rest of the current compilation package. Old classes can be rebound by using the backchain bookkeeping (see section two). Furthermore, instances of the ancestor class will also be attached to the updated versions using the backchain. If old methods were modified in a way that they access added instance variables a runtime error would occur. This situation can be treated like an incompatible modification (see following text) or instances are upgraded. Upgrading of instances can be done by instantiating new objects and copying the old instance data to the new copy.

If the new version has been modified in an incompatible way - e.g. a method required by some client was deleted - both type descriptors are required. Old clients and instances continue to use the old version whereas new types and those of all subsequent compilations will be bound to the new version. Old clients may be rebound on user request and may require recompilation with user supported source code modifications.

Fortunately, if modifications yield different type versions old unused versions will automatically be deleted by the garbage collection if no more references to them exist. Hence, the typical problem like in Microsoft (MS) Windows when to delete a shared Dynamic Link Library is avoided.

5 Evaluation

In this section present early measurements investigating compilation speed of our Plurix Java Compiler (PJC) and performance of the DSM system.

We compare our native Plurix compiler (nPJC) to a cross compilation version (cPJC), to the GNU Java Compiler (GCJ) and to the SUN JDK 1.2.2 Javac. Javac, nPJC, and cPJC are written in Java and nPJC is generated by bootstrapping itself to cPJC. The latter is executed on a JVM running under MS Windows or Linux. The result is written to floppy disc and loaded on a PC. The native nPJC version runs as a restartable transaction within our Plurix system allocating memory in the DSM.

GCJ is a front-end for the well-known back-end of the GNU C compiler. Java source texts or byte-code class files are translated to assembly code which is processed by the GNU assembler to generate machine instructions which are bound by the GNU linker (10,000 lines). GCJ is written in C (38,500 lines) and directly benefits from its optimizing back-end.

Javac is the Java compiler provided by SUN with the JDK. It is written in Java and executed on the JVM translating Java source texts into byte-code class files.

package	source	code-size
name	lines	bytes
Kernel	4,100	76,800
Drivers	10,400	250,050
Runtime	1,500	19,600
nPJC	10,000	319,571

Table 1: Source Text and Machine Code

The first table shows the size of selected packages of our system. The source size includes comments and the code-size column contains only the machine instructions of all generated methods. The kernel package includes the DSM memory management, interrupt handling, and BIOS support. The driver package contains graphic (ATI Radeon, S3, Vesa), sound, keyboard, serial, PCMCIA, network (10/100 MBps Ethernet), and frame grabber drivers. The runtime package listed in table 1 is a subset of the *java.lang* and *java.io* base packages. The size of the compiler package includes runtime structure and code generation, binding, and initialization which is comparable small.

The second table compares compilation speed of cPJC, Javac, GCJ, and nPJC for translating 29 classes with 2,242 lines of source text. The compilation time includes generation of machine instructions and allocation of runtime structures or object files. Binding and completing runtime structure construction is shown in column "bind". Since GCJ and Javac perform a lazy binding strategie no numbers are available. All tests were performed on AMD-Duron 700 MHz PCs with 160MB main memory. The cross compilation, GCJ, and Javac was executed on the Microsoft JVM on MS Windows 2000 SP1.

The startup time for Javac on the JVM is quite long and results in a compilation time of 3830ms. The bare compilation consumes only 1152ms.

nPJC allocates about 2.15 MB of memory during compilation (without shadow pages). The statement *New* was called 14,500 times

compiler	mem	compile	bind
	MB	ms	ms
cPJC	5.8	900	NA
nPJC	4.3	323	0.9
Javac	9.6	3830/1152	NA
GCJ	8.1	1050	NA

Table 2: Compilation Speed

compiler	mem	compile	bind
	MB	ms	ms
nPJC	4.36	323	0.9
no shadowing	2.18	320	0.9
no backchain	4.36	305	0.9

Table 3: Memory Management Overhead

with an average object size of 158 bytes. The result of our measurements show a compilation speed of 7,000 lines per second which is encouraging with respect to the fact that compilation is performed as a restartable transaction and memory allocation is not as cheap as in traditional systems. Furthermore, the OS and the compiler are currently generated with a stack-based code generation scheme with much potential for speed-up.

To assess the overhead of the DSM memory management we compare compilation speed of nPJC with single features turned off: allocating no shadow-copies and not supporting the backchain, see table 3. The DSM cluster was composed of three nodes connected via 100 Mbps Ethernet hub. Page access is monitored by the CPU and during the first access (within a transaction) a shadow-copy is created to ensure restartability. The backchain concept requires the compiler to insert a runtime function call for every heap pointer assignment.

The overhead added by creating shadow pages and supporting the backchain adds less than 10 measurements results are encouraging for using a persistent DSM for a general purpose OS, including compiling programs.

6 Conclusions

We have presented a preliminary evaluation of a general purpose DSM system and our Plurix Java Compiler. Traditionally, DSM systems are built to support specific parallel algorithms written for multiprocessor machines. The Plurix project investigates a the DSM concept as a general purpose communication facility together with persistence.

In this paper we presented the integration of our Java compiler into the DSM system. The compiler is aware of the persistent environment and directly generates runtime structures which are bound and initialized after successful compilation. Persistence and a cluster-wide name service lay the foundation for a simple and efficient separate compilation scheme. Classes are automatically registered by the compiler during compilation and can be accessed by any node. Type evolution is supported by the backchain concept (bookkeeping of references) which allows old clients of a type to be updated to a newer version (including the instances) when type compatibility rules are not violated.

Although our compiler runs in a transactional environment and allocates memory in a DSM the compilation speed of about 7,000 lines per second is encouraging. This compilation speed is sufficient to ensure short compilation transactions to minimize collision probability of our optimistic synchronization scheme. A next generation compiler will deploy a SSA intermediate representation together with different optimization strategies. Possibly, the compilation phases will be distributed over several transactions.

We have presented our DSM system at the CeBIT 2001 fair. The Plurix demonstrator showed a preliminary user interface, 3D graphics, a sound-, and a video-application, see figure 3. The coordinates of a rotating cube were shared through the transactional DSM allowing a synchronous rotation.

Currently, we are finalizing a first version of our page server to investigate issues related to persistence and recovery.

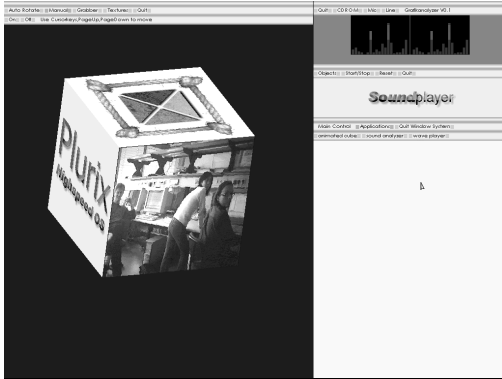


Figure 3: The CeBIT 2001 Demonstrator

References

- [1] 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.
- [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, Germany, Distributed Systems Department, 1996.
- [4] M.P. Atkinson and R. Morrison. Orthogonal Persistent Object Systems. In *Very Large Data Bases Journal*, 4(3), 1995.
- [5] 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.
- [6] M.Schoettner O. Schirpf M. Wende and P. Schulthess. Implementation of the Java language in a persistent DSM Operating System. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA, 1999.
- [7] N. Wirth and J. Guteknecht. *Project Oberon - The Design of an Operating System and Compiler*. Addison-Wesley, 1992.
- [8] M. Baker-Harvey J. Chase, H. Levy and E. Lazowska. Opal: A single Address Space System for 64/Bit Architectures. In *IEEE Workshop on Workstation Operating Systems*, 1992.
- [9] M. J. Jordan T. Prinzezis M.P. Atkinson, L. Daynes and S. Spence. An Orthogonally Persistent Java. In *SIGMOD Record, Volume 25*, 1996.
- [10] T.B. Knoblock R. Fitzgerald and E. Ruf. Marmot: An Optimizing Compiler for Java. In *Technical Report MSR-TR-99-33*, Microsoft Research, USA, 1999.
- [11] Per Bothner. Compiling for Java Embedded Systems. In *Embedded Systems Conference*, San Jose, USA, 1997.
- [12] M. Schoettner O. Marquardt M. Wende and P. Schulthess. Linking and Loading in a persistent DSM Operating System. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, Las Vegas, USA, 2000.
- [13] R. B. J. Crelier. *Separate Compilation and Module Extension*. PhD dissertation, ETH Zurich, Swiss Federal Institute of Technology, 1994.