

A transactional DSM Operating System in Java

M. Schoettner, S. Traub and P. Schulthess
University of Ulm
Germany

Abstract

Distributed Shared Memory (DSM) is a well known interconnection paradigm in the scientific community. The development of distributed applications basing on message passing is more complex than on a DSM. The notorious performance drawbacks of DSM Systems are often caused by overly expensive distributed locking mechanisms. In response to this our Plurix Operating System (OS) implements a transaction based DSM. Memory consistency is maintained by atomic transactions and optimistic synchronization mechanisms which have been used in database technology in the past. Such a transaction based DSM would perform poorly if placed on top of an existing OS. Therefore Plurix is developed as a standalone PC-based OS taking integral account of the transaction concept. Plurix is written in Java using a tailor made new Java Compiler which will be an integral part of the OS. The Compiler generates Intel machine code and avoids performance loss caused by an interpreter like the Java Virtual Machine (JVM). Memory management in Plurix goes beyond traditional DSM Systems as it implements a distributed heap storage (DHS) with location independent views on data and code. Additionally a distributed garbage collection is implemented. An object relocater relying on the strong typing of Java alleviates the "false sharing" problems and avoids excessive memory and page table fragmentation.

Keywords: Java, Distributed Shared Memory, Operating Systems, Transactions

1 Introduction

The commercial Systems: Windows NT, Unix and MacOS offer network services based on message passing. Traditional socket interfaces force developers to superpose their own network protocols, to handle a plethora of er-

ror conditions and increase software complexity and costs of maintenance. Higher level interfaces are offered by RPC and object oriented approaches like RMI, CORBA and DCOM. These higher level APIs offer a rich set of functions but fail to simplify the design of distributed software systems. Rather the development effort is spread across several software layers and development tools.

The functions center on providing distributed database functionality and messaging. Computer Supported Collaborative Work (CSCW) services are not provided in spite of a growing demand in future teleworking environments. In a CSCW conference several participants concurrently work on shared documents and media, raising a fundamental problem of memory consistency which is difficult to solve at the application level. Special distributed applications or middleware groupwork toolkits (e.g. NetMeeting) offer CSCW solutions but will not effectively compete with sophisticated single station application programs (e.g. MS-Word or a special group text editor).

Moving the distributed functionality into the OS is an interesting alternative. This can be achieved by a Distributed Shared Memory (DSM) mechanism providing a virtual address space shared among tasks on loosely coupled processors. The application programmer is offered a transparent view on data shared over several computers connected via a network. He uses regular pointers for both local and remote memory accesses. It is the task of the OS to detect a remote memory access, fetch the desired memory block and maintain memory consistency.

The main effort of developing CSCW applications is the distributed document consistency. In Plurix, however each program inherits the potential to maintain consistent data structures from the DHS.

Our paper consists of five sections. Section two gives an overview of existing page based DSM implementations. Section three describes the architecture of the Plurix operating system. Section four gives an overview of the Java compiler and the bootstrapping process. The deployment areas of Plurix are discussed in section five. Finally we will give an outlook to future work.

2 Overview of existing page based DSM Systems

An early idea of DSM was presented by L. Keedy in 1985 [1]. In the following years the research interest in DSM Systems has grown steadily. A multitude of software and hardware level systems and hybrid architectures have been developed [2]. We do not attempt to give a comprehensive perspective of the state of DSM systems in this section. However, because Plurix is a page based system we shortly review some representative paged based systems: IVY [3], Mirage [4] and TreadMarks [5].

Page based DSM systems detect memory accesses to pages by using the protection features of the MMU. MMU hardware support can substantially speed up program execution in comparison to software based implementations but it is afflicted by the false sharing problem. The term "false sharing" is only vaguely defined in the literature [6]. False sharing is a characteristic performance penalty of page based DSM systems and occurs when two semantically independent variables reside in the same page and are alternately accessed by different processors. As a result the page is exchanged again and again between the processors. It is crucial to choose the right page size. Larger pages can speed up memory access due to the locality of data [7].

On the other side the probability of false sharing increases when larger memory pages are used.

2.1 IVY

IVY [3] was one of the first proposed DSM systems. It is a user-level implementation running on a group of network processors, the Apollo Domain system. It implements a page based DSM allowing multiple readers but only one writer per page. Memory consistency is guaranteed by an invalidation protocol, which requires that all read-only copies of a page are invalidated before a processor writes to a page. Sequential memory consistency is enforced similar as in highly coupled multiprocessors. This is the main reason why the performance of IVY is not convincing. Additional overhead is introduced by the user-level approach.

2.2 Mirage

In contrast to IVY, Mirage is implemented in the kernel of an existing operating system [4]. The main idea is that a writer of a page should retain access to that page for a fixed period of time Δ . This can improve the exploitation of processor locality and avoid trashing. The value of Δ may be dynamically tuned. Mirage handles memory segments which are partitioned into pages. A process creates a segment by defining its size, name, and access protection. All other processes locate and access the segment by name. Requests are sent to the creator of the segment, where they are sequentially processed. The performance of the whole system is highly sensitive to the proper choice of the parameter Δ value.

2.3 TreadMarks

TreadMarks [5] is a user-level implementation on top of common available Unix systems. It applies the lazy release consistency model, together with a page invalidation protocol, that allows multiple concurrent writers to modify the page. On the first write to a shared page,

a copy called "twin" is made. The "twin" can later be compared to the current copy of the page in order to make a "diff" - a record that contains all modifications to the page. Lazy release consistency does not require "diff" creation at each release (e.g. like Munin [8]), but allows it to be delayed. Lazy release consistency can achieve better performance than the release consistency implemented in Munin.

3 Architecture of Plurix

Plurix will be a standalone PC operating system (min. 80486) completely written in Java. The first prototype of Plurix will run within a single LAN segment. The central abstraction within our design is a global address space shared by several nodes. The global address space is organized as a distributed heap storage (DHS) containing both data and code.

Accesses (read or write) are properly monitored by the MMU. Tasks in the OS are partitioned into restartable transactions. Memory persistence and backup storage is initially provided by a Windows NT file-server connected to the Plurix cluster. The DHS in Plurix introduces a new model to maintain memory consistency: restartable transactions and optimistic synchronization [9], [10], [11] and [12].

3.1 Distributed memory consistency

Transactions emanate from the Plurix Command-Loop in each node and memory consistency can be guaranteed according to the well known ACID (Atomicity Consistency Isolation Durability) paradigm. The transactions obtain fully transparent access to the DHS and it is optional for the application programmer to partition a command into transactions of a smaller granularity by explicitly inserting "Begin-Of-Transaction" (BOT) and the "End-Of-Transaction" (EOT). Transactions (TA) are encapsulated by a central OS-Loop (e.g. user starts a command) and must be short to reduce collision probability.

At BOT all access privileges (read & write) to memory are void. If the TA accesses a

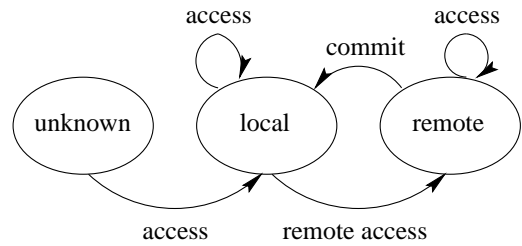


Figure 1: Location information of a page

memory location the page-fault handler of the MMU is invoked. Each page has associated status information indicating its location and access mode. In a first step the page location must be determined. A page can have three different location states according to fig. 1. The page state "unknown" means the page was never accessed before or the residence is unknown. The state "local" indicates that a page is available on the local node. The location state "remote" denotes that the page may also reside on another node and may be loaded via the network. Pages also have an access attribute "locked", "readable" or "writeable".

If a local page is requested by a write attempt the page fault handler saves the original page in a "before-image" to ensure undo capability and the page number is stored in a write-set. Read-operations are resolved in the same way, except that the page fault handler needs not create a "before-image" but only stores the page number in a read-set. If the TA is entering the commit-phase these read- and write-sets are used to detect possible collisions between participating nodes.

3.2 Collision resolution

During the commit-phase all concurrent TAs need to be ordered. Therefore we introduce TA-numbers representing the equivalent serial TA order. We denote the following order relation:

- $TA_i > TA_j$: TA_i starts processing after TA_j terminated
- $TA_i < TA_j$: TA_j starts processing after TA_i terminated

- otherwise: TA_j and TA_i overlap

During the collision detection phase we must only inspect TAs which can not be ordered by " $>$ " or " $<$ " [9]. Only TAs with overlapping lifetimes need to be considered. Collision detection is based on the three predicates below:

1. $(W_i \cap W_j = \emptyset) \wedge (R_i \cap R_j = \emptyset) \Rightarrow (TA_i < TA_j) \vee (TA_i > TA_j)$
2. $W_i \cap W_j \neq \emptyset \Rightarrow \text{collision}$
3. $R_i \cap W_j \neq \emptyset \rightarrow TA_i < TA_j$

If a collision between one or more TAs is detected all but one of these nodes must be restarted. This can easily be done, using the "before-images" saved earlier. By evaluating the predicates (1,2,3) the DHS implements an invalidate protocol allowing multiple readers and writers.

3.3 Core OS services

Our transactional DHS supports three higher level memory management services:

- distributed object relocation (DOR)
- distributed garbage collection (DGC)
- distributed peak memory allocation (DPMA)

The distributed garbage collector (DGC) and the distributed object relocater (DOR) services run as separate transactions and rely on the strong typing of Java. The DGC & DOR needs 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 "backchain" (fig. 2).

Objects in the heap can be relocated by adjusting all pointer values in the backchain.

If the backchain is empty the object is garbage and can be collected by the DGC.

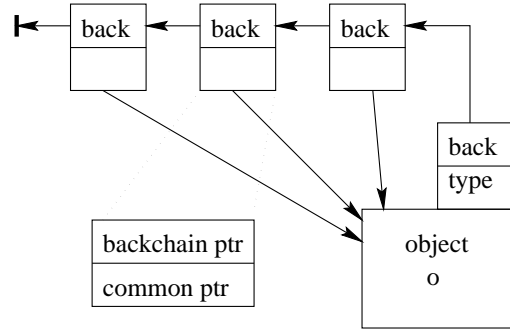


Figure 2: Backchain of references to an object

Unfortunately cyclic references cannot be detected by the backchain method. Therefore we establish an additional concept: garbage collection through heap compactification (fig. 3). This algorithm relies on the option of object relocation (DOR) and is implemented like a copy garbage collector.

Copy garbage collectors divide the heap space in two or more parts. All actual objects are located in the first part so called "from-space". The other part "to-space" is initially empty. During the garbage collection all objects are copied successively to the "to-space". Of course copying an object O includes copying of all recursive reachable objects from O. All remaining objects in "from-space" are obviously garbage.

The DHS of Plurix is divided in two parts "compact heap" and "non-compact heap". The compactification border is defined by a special pointer (fig. 3). In contrast to traditional copy collectors we mustn't copy objects but only relocate them and therefore we only reserve logical memory not physical.

Initially we define the heap compactification border. This is the highest virtual memory address used by a node. From this point of time new memory is only allocated above this border. Now we copy all objects recursively from low to high memory addresses. If the DGC transaction collides with another transaction it will always lose and learn that the object which caused the collision is not garbage. Independent of the chosen method a DGC can only work if pointer arithmetics are prohibited

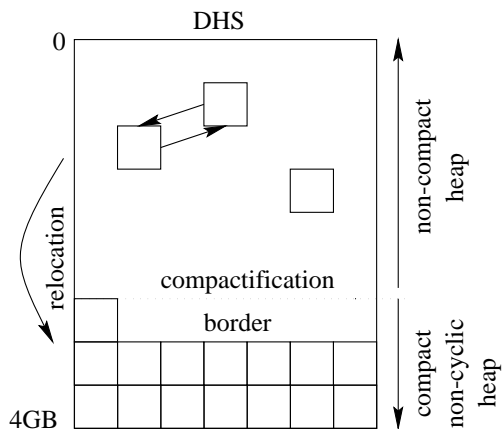


Figure 3: Detection of cycles by heap compactification

as it is the case for a strongly typed language like Java.

The distributed peak memory allocation (DPMA) subroutine is responsible for logical memory allocation from DHS. DPMA tries to avoid collisions during the search for a free memory block. The central idea of DPMA is: all nodes allocate memory in pages loosely dedicated to themselves. This is done by a page node number (PNN) on every logical page. PNN=0 means the memory is not dedicated to any node. After system startup there is only one large memory block with PNN=0 (fig. 4). If any node allocates memory on a page the whole page is reserved for this node. If a page is completely deallocated it could be made globally available. However it is more efficient not immediately pass back such a page but instead keep a contingent. During following memory allocations a node can use its own contingent. If no fitting memory block is found in the current contingent a piece of free memory is allocated from PNN=0. All pages with the same PNN are chained by a linear list (4). The first page, the root of all nodes contingent lists is typically accessed in read-only mode. Therefore no conflicts can arise on the root page. The search for free memory in a node contingent list doesn't cause conflicts either.

The DPMA allocation strategy reduces the occurrence false sharing. However, if objects are replicated by different nodes false sharing

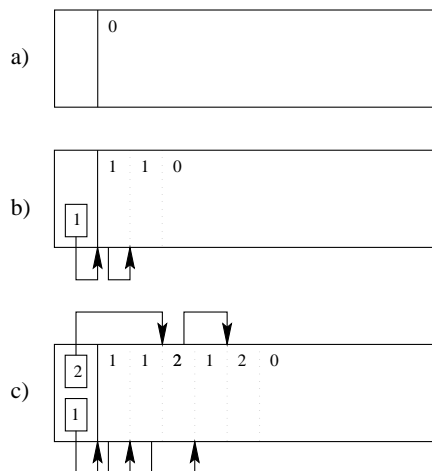


Figure 4: Distributed peak memory allocation

might still occur. The DOR can resolve a false sharing conflict between two objects in a single page by moving an involved object to another memory page. An unsolved problem is the detection of false sharing, a formal definition of false sharing and a possible detection mechanisms for it [6].

4 The Java Compiler

Plurix is implemented from scratch with a new Java compiler producing native Intel code. Therefore inefficiencies problems caused by the interpretation of Java Byte Code are avoided. The development of a new compiler became necessary due to the following requirements:

- lean runtime support
- addition of low level Java extensions
- support for a distributed garbage collection (see also DGC in section 3)
- integration of the compiler into the OS

We investigated modifying an existing Java compiler, but were reluctant to inherit the excessive runtime requirements of the imported Java libraries. Importing a plethora of library classes into the compiler would introduce additional overhead when bootstrapping the compiler to the Plurix machine. We also avoided

to use a parser generator, because the symbol attributes generated by the compiler will be an integral part of the OS - run-time structures. As described in section 3 unconventional run-time structures are needed to support the DHS.

4.1 Bootstrapping

The bootstrapping process of the Plurix OS occurs in several steps:

1. developing and testing the compiler in a Windows NT environment
2. loading a serial line driver from disc
3. loading a small kernel together with the compiler across the serial line
4. compiling and loading additional source texts across the serial line
5. starting additional stations on the LAN

We have established a testing environment under Windows NT (fig. 5). The Java Compiler (JC) is executed on the Microsoft JVM (JVIEW) and deposits its output not in a file but into virtual memory.

Plurix offers a persistent DHS which is represented by the VMB (Virtual Memory Block) (fig. 5). The VMB is a large virtual memory block allocated during process startup of the testing environment on the same Windows NT machine. The JC can access the VMB using native methods encapsulated in "memory.dll" (fig. 5). All method calls to "memory.dll" are passed to the "test" process by inter process communication. The run-time structures and the code located in VMB can easily be examined with a common debugger (e.g. SoftICE) and we can step through a compiled Plurix program setting a breakpoint at the entry point of program execution. After verification of the code generation we copy the used part of the VMB to the Plurix machine (5).

On the native machine we run a small boot-loader which switches to protected mode and waits on a serial link for the kernel image. The VMB is copied to the same memory location

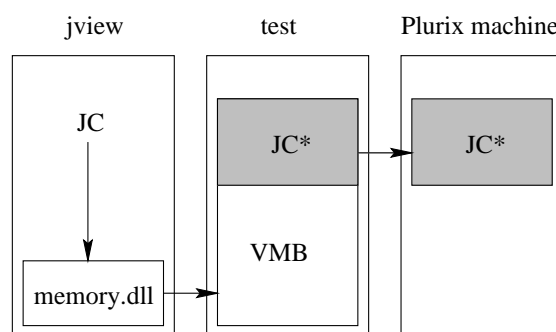


Figure 5: Test environment for bootstrapping

on the target machine and its execution starts at a given entry point.

The main runtime structures are class descriptors and code segments. We would have liked to adopt the class descriptors of JVIEW to the VMB. However, the memory layout of JVIEW's class descriptors is undocumented and full of indirections, and after examining the public available sources of the JVM kaffee we were dissatisfied with the class descriptor structures of kaffee. Many fields in a kaffee's class descriptor are useless for the purpose of Plurix.

It was considered to model the run-time structures in Java to make visualisation and verification easy. However, abundant use of pointers and indirections in Java would lead to excessive heap fragmentation and run-time penalties. Therefore the compiler includes a non standard variant of the "new" method to allocate the required run-time structures more efficiently. For the bootstrapping process we create these compact class descriptors manually in the VMB. After JC is compiled to JC* this will become unnecessary.

4.2 Java extensions

Distributed garbage collection is only feasible if the system can find all global references to an object and pointer arithmetics are prohibited. Therefore the compiler allocates additional space for each pointer to allow the linking of all the pointers to one object - creating a separate backchain from each object block in the heap (fig. 2 and [11]). The backchain is

updated during a pointer assignment.

In order to make code segments relocatable as well, we do not use absolute jumps in the code but always branch (and return) through a jump table.

The Java compiler accepts inline assembler code to access the real hardware and offers a facility to access logical memory via an array operator for memory words or bytes.

Another area of concern are the static variables. According to the DHS concept they are shared by all nodes. However, we deviate from this concept to allow for individual initialization of variables by each user. Only variables carrying the attribute "global" can be shared on a system wide scale. Other static variables are initialized when they are first imported by a user.

4.3 Integration of compiler and OS

When JC* is available on the Plurix machine the symbolic class descriptor (SCD) and the runtime class descriptor (RCD) will be amalgamated resulting in one Plurix class descriptor (PCD). Furthermore the scopes of the compiler are extended by "userscope" and "globalscope" thus creating the starting point for a naming service. Module management is greatly simplified by the persistent heap storage: after a class is compiled it is already loaded and ready to be used. There is no need for loaders and complex debuggers because the PCD contains already the necessary information. Optionally, after the compilation a user can publish the class globally and than everyone can use it without installation.

5 Deployment areas of Plurix

Plurix is the first DSM System intended for Computer Supported Cooperative Work scenarios. Available OS support groupwork rather poorly. As a result a multitude of complex distributed applications or application sharing packages have been developed [13]. They all struggle to compensate the lack of OS

support for CSCW. We expect that the development of groupware applications can be significantly simplified by a fully transparent underlying DSM heap. Most likely that collaborative document processing or distributed time planners are applications which will perform well on a transactional DSM. Transactions and optimistic synchronization for CSCW applications have been used by [14] and [15]. Plurix does not aim to compete in high performance parallel computing. Rather we assume an optimistic synchronization strategy and expect that collisions are rare. This assumption typically holds in CSCW and office scenarios. In CSCW meetings we expect only a limited number of competing participants. If these participants are working in different sections of a shared document there may be no collisions at all.

6 Future work and outlook

Existing DSM systems are typically user-level implementations or modifications of existing operating systems. The overhead inherited (e.g. interprocess communication in microkernel systems like Windows NT) results in poor system performance. Weaker consistency models can enhance DSM performance but conversely they make distributed programming more complex. This is contradictory to the aim of DSM: simplify distributed programming of message passing systems. Therefore we develop a totally new operating system made-to-measure for supporting a page based DSM. Consistency is maintained by restartable transactions respecting the ACID-principle and optimistic synchronization. The application programmer has the perspective of a fully transparent DSM heap (DHS). No additional programming complexity to explicitly observe special consistency models is imposed. We are aware of the fact that the duration of a TA is a critical parameter for overall system performance. This is achieved by a lean and fast OS and associated short transactions and small working sets. Using a typesafe language like

Java makes OS services like garbage collection and object relocation possible at all. The object relocater provides a heuristics for reducing false sharing problems. An early prototype of Plurix runs on MS-DOS based machines. This version was written with a modified Oberon compiler. Currently we are finishing the development of the Java Compiler. We are planning to have an early Java prototype consisting of two Plurix machines at the end of the year.

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] A comprehensive bibliography of distributed shared memory. Technical Report TR96-17, Department of Computing Science, University of Alberta, Department of Computing Science, 1996.
- [3] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *1988 Int'l Conf. Parallel Processing*, 1988.
- [4] B. D. Fleisch and G. J. Popek. Mirage: A Coherent Distributed Shared Memory Design. In *Proc. 14th ACM Symp. Operating Systems Principles*, 1989.
- [5] P. Keleher et al. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *USENIX Winter 1994*, 1994.
- [6] M. L. Scott W. J. 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.
- [7] K. Rajamani C. Amza, A. Cox and W. Zwaenepoel. Tradeoffs Between False Sharing and Aggregation in Software Distributed Shared Memory. In *Principles and Practice of Parallel Programming*, 1997.
- [8] W. Zwaenepoel J. B. Carter, J. K. Bennet. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symposium Operating System Principles*, 1991.
- [9] J. T. Robinson H. T. Kung. On Optimistic Methods for Concurrency Control. In *ACM Transactions on Database Systems*, 1981.
- [10] H. Kung and J. Robinson. On Optimistic Methods for Concurrency Control. In *In Proc. of the ACM Transactions on Database Systems*, 1981.
- [11] S. Traub. The Design of a Distributed Oberon System. In *In Proceedings of the Joint Modular Languages*, 1994.
- [12] S. Traub. *Speicherverwaltung und Kollisionsbehandlung in transaktionsbasierten verteilten Betriebssystemen*. PhD dissertation, University of Ulm, Germany, Distributed Systems Department, 1996.
- [13] A. Lupper P. Dudzik M. Schoettner, A. Kassler and P. Schulthess. Application Sharing - Architecture and Performance Aspects. In *Proceedings of the 2nd Mobile Communications Summit*, 1997.
- [14] D. Marwood S. Greenberg. Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface. In *Proceedings of the conference on Computer supported cooperative work*, 1994.
- [15] S. J. Gibbs C. A. Ellis. Concurrency Control in Groupware Systems. In *Proceedings of the ACM SIGMOD '89 Conference on the Management of Data*, 1989.