Optimistic Synchronization and Transactional Consistency

M. Wende, M. Schoettner, R. Goeckelmann, T. Bindhammer, P. Schulthess
wende@plurix.de

Abstract

Distributed Shared Memory (DSM) is an interesting alternative to build distributed object system. Explicit message passing and remote invocation is replaced by the simple and unified DSM abstraction. The recurrent performance drawbacks of DSM Systems are often caused by expensive distributed locking mechanisms. In response to this our multi purpose 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 with optimistic synchronization guarantees a sequential consistent view on the shared data to every node in the cluster.

Keywords: Distributed Shared Memory, Operating Systems, Memory Consistency, Transactions.

1 Introduction

The commercial Systems: Windows NT, Unix and MacOS offer network services based on message passing combined with remote invocation. Traditional socket interfaces force developers to superpose their own network protocols, to handle a plethora of error conditions and increase software complexity and costs of maintenance. Higher level interfaces are offered by RPC and object oriented approaches like RMI, CORBA and .NET . These higher level APIs offer a rich set of functions but fail to simplify the design of distributed software systems. The development effort is typically spread across several software layers and development tools. The available middleware centers on providing distributed database functionality and messaging, but no consistency.

Moving the distribution 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 at shared data on several computers connected via a network. Regular pointers are used for both local and remote memory accesses. OS and memory management hardware together will detect a remote memory access, fetch the desired memory block and maintain memory consistency.

In the Plurix OS we not only provide the distributed virtual memory abstraction, but each program inherits the potential to maintain consistent data structures in shared memory. Restartable transactions combined with optimistic synchronization enable the OS to restart operations and to ensure a sequentially consistent view for each application on the shared data.

The overall system structure of Plurix is patterned after the innovative Oberon system developed by Wirth & Gutknecht [18]. It uses a central event loop in each station and a simple cooperative multitasking concept. Currently a Plurix station will only participate in DSM domain with a 32 bit wide address space. The OS has been successfully demonstrated at various trade fairs and proved to be fast, compact and reliable

Our paper consists of five sections. Section two gives an overview of existing page based DSM implementations including Plurix. Section three

describes transactions and the optimistic synchronization scheme of the Plurix operating system. Section four shows that the synchronization implemented in Plurix guarantees sequential consistency. 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 defined vaguely in the literature [6]. False sharing is a characteristic performance penalty of page based DSM systems and occurs when two semantically independent variables reside on the same page and are alternatingly 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 processors writes to a page. Sequential memory consistency is enforced in much the same fashion as in tightly coupled multiprocessors. This is the primary 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 ?. This can improve the exploitation of processor locality and avoid trashing. The value of ? 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 ? 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.

2.3 Architecture of Plurix

Plurix is a standalone PC operating system (min. 80486) almost completely written in Java. Plurix runs 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 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 provided by a special node - Page Server equipped with a large harddisk. The DHS in Plurix introduces a new model to maintain memory consistency: Restartable transactions and optimistic synchronization [9], [10], [11] and [12]. But Plurix is a multi purpose operating system, not specially designed for number crunching.

3 Synchronization in Plurix

This chapter presents two kinds of optimistic synchronization, forward and backward validation and restartable transactions. This allows concurrent access to shared objects in the Distributed Heap Storage by several nodes in a Plurix cluster whereby collisions are resolved by restarting conflicting operations.

3.1 Restartable Transactions

Transactions must be resetabel in order to adhere to the well known ACID (Atomicity, Consistency, Isolation, Durability) paradigm [16]. In Plurix an aborted transaction and its input are automatically redone at some later time.

A transaction groups several read and write operations to the DHS into an atomic indivisible operation. If for any reason a transaction cannot be completed, every modification of this transaction will be restored to the state before its beginning, by restoring available shadow copies.

Transactions always begin and terminate on a consistent view of memory. When a transaction starts, the DHS is in a consistent state and at the end, the transaction leaves the DHS again in a consistent state.

During a running transaction modifications are isolated. Therefore intermediate results are invisible to other transactions.

If a transaction commits all modifications persist (durability) until a successor transaction modifies the values.

These characteristics of transactions allows the optimistic concurrency control to restart operations in case of a conflict of concurrent operations

3.2 Optimistic Synchronization

The basic assumption behind optimistic concurrency control is, that most transactions do not conflict with each other. Therefore they can tentatively proceed without locks and conflicts are checked at the end of a transaction. If two or more transactions collide, at least one needs to be aborted and is restarted later. If they do not conflict all transactions can commit successfully.

Collision detection can be done as a forward or as a backward validation.

In a *backward validation scheme* the addresses of all modified objects (write sets) of a transaction are compared against all overlapping transactions that already have committed. If a conflict is detected, the currently validating transaction is aborted; if there is no conflict the transaction can commit.

If *forward validation* is used the write set of a validating transaction is compared against all active transactions. If a conflict is determined one or more transactions must be restarted.

The forward validation scheme has the advantage that there are more than one candidate transactions to abort in case of a collision. Therefore a fairness protocol can help to decide which transaction to abort.

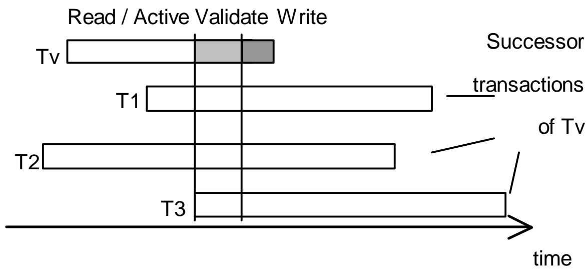
3.3 Collision detection

During validation phase at the end of transaction T_v collisions between concurrent transactions are detected by comparing the read references and the write references of participating objects Three rules must be satisfied to exclude concurrent read or write operations [15].

(1) Read-Write: T_v must not read an object written by an overlapping Transaction T_j.

(2) Write-Read: T_j must not read an object written by an overlapping T_v.

(3) Write-Write: T_j must not write an object written by an overlapping T_v and vice versa.



T_v is a transaction in the validation phase and T_j are running transactions overlapping with T_v in the case of forward validation.

4 Transactional Consistency

In the following chapter we discuss a set of rules which define a special type of consistency which we choose to name "transactional consistency"

4.1 Definition of Sequential Consistency

A transactional distributed system is a pair *(T, M)*, where *T* is a set of *N* transactions and *M* is the shared memory. Each transaction executes read and write operations on data objects of *M* in the order defined by the operations of *T*.

Write operations are denoted by $w_i(x)v$, where *v* is the value written on *x* by transaction T_i. A read operation is denoted by $r_i(x)u$, where *u* is the value read by T_i at object *x*.

A "*reads from*" relation is given, if transaction T_j reads a value written by T_i; $r_j(x)$ *reads x form* $w_i(x)$.

An execution of transaction T_i is a pair of $(S_i, <_{Ti})$ where S_i is a set of read and write operations on *M* and $<_{Ti}$ is the total order on S_i defined by the program running in transaction T_i. This is also called internal operation order relation.

A complete execution history on a DHS *M* is defined by a pair $H^®$ with $H^® = (H, <_H)$ where $H = \bigcup_{i=1..n} S_i$ and $<_H$ is a well formed irreflexive transitive relation defined by the following rules:

(1) If $op_1 <_{T_i} op_2$ then $op_1 <_H op_2$, i Î N.

(2) If $r_j(x)v$ *reads from* $w_i(x)v$ in *H*, then $w_i(x)v <_H r_j(x)v$.

A history $H^®$ is *legal* if for every read operation $r_j(x)v$ exists a write operation $w_i(x)v$ in *H* such that $r_j(x)v$ *reads from* $w_i(x)v$ and there exists no $w_k(x)u$ with $w_i(x)v <_H w_k(x)u <_H r_j(x)v$ and u ¹ v.
Further on, a history $H^® = (H, <_H)$ is a sequential history if $<_H$ is a total order.

A transaction based DHS is comprehensively described by $H^® = (H, <_H)$, the reads from relation and the *legality* condition.

Mizuno, Raynal and Zhou [14] define the terms "*equivalent*" and "*sequentializable*" to define a formal definition for sequential consistency.

Definition Equivalent: Two histories $H^® = (H, <_H)$ and $H^{®'} = (H', <_{H'})$ are *equivalent* if:

(1) They are over the same set of transactions T_i and the same set of operations ($H = H'$).
(2) They have the same internal operation order relation.
(3) They have the same set of "reads from" relations.

Definition Sequentializable: A history $H^{®} = (H,<_H)$ is *sequentializable* if $H^{®}$ is equivalent to some *legal sequential* history.

Definition SC: A distributed shared memory system *(T, M)* is *sequential consistent* if for all possible execution histories $H^{®}$ are *sequentializable.*

Theorem Sequentializable: A history $H^{®}$ is *sequentializable* if and only if $H^{®}$ has a topological sort $S^{®}$ such that $S^{®}$ is *legal*.

The proof for the theorem can be found in [14].

4.2 Definition of Transactional Consistency

The Definition SC requires that each possible execution history $H^{®}$ is *sequentializable*. In a transactionally consistent system with optimistic synchronisation an execution history is selected and tentatively executed. If it turns out not to be sequentializable, then one or more transactions are aborted and restarted later resulting in a modified execution history, which is again tested for *sequentializability*.

Definition Transactional Consistency: A distributed shared memory system *(T, M)* is transactionally consistent if it implements a transaction mechanism which allows only those transactions to commit which contribute to a sequentializable history.

Which means that a transaction only commits if and only if the execution history $H^{®}$ is a topological sort such that $H^{®}$ is *legal*.

The separate execution history S_i of a transaction T_i has a priory a topological sort given by the program running in transaction T_i. This sort could only be destroyed by other transactions reading and writing objects concurrently.

Regarding the rules of chapter 3.3 every kind of concurrent read/write or write/write on an object will abort one or several transactions. Therefore the topological sort of all transactions is maintained if a transaction commits.

Suppose there is a execution history $H^{®'}$ that is not *legal*. Then there is a relation $r_j(x)v$ *reads from* $w_i(x)v$ and a $w_k(x)u$ such that $w_i(x)v <_{H'} w_k(x)u <_{H'} r_j(x)v$ and *k ? j and u ? v*.

The isolation property of transactions prevent intermediate results to become visible to other transactions. Therefore transaction T_i and transaction T_j must be identical or $w_i(x)v$ is the result of a committed previous transaction T_i.

Suppose $T_i = T_j$: Then $w_k(x)v$ of T_k is overlapping with T_i. Due to the rules of chapter 3.3 concurrent read/write and write/write operations will abort transactions.
Suppose T_i is a previously committed transaction. Then the write operation of T_i: $w_i(x)v$ is the last committed write operation on x and therefore T_j and T_k are working concurrently on x. Due to the rules of chapter 3.3 this will abort T_j or T_k.

Consequently the execution history $H^{®}$ must be legal and $H^{®}$ is a topological sort. For that reason the execution history of a committed transaction T_i is sequentially consistent. Because of the test and restart of operations in case of an abort, we suggest to call this type of consistency transactional consistency.

5 Conclusion and Perspective

We have shown that our multi purpose operating system Plurix guarantees sequential consistency of the shared heap, using optimistic synchronization combined with restartable transactions. This allows to develop applications using shared objects in the DHS without special synchronization mechanisms.

The consistency of the objects, shared between the applications is assured by the memory management of the Plurix operating system and the network protocols.

Future work will be done on false sharing, a fairness protocol for the abort of transactions in case of

collisions and an implementation of a weaker consistency model for ”lazy objects”, e.g. live video frames.

6 Literature

[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 Scienes, 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. Ben0net. 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 Proc. of the ACM Transactions on Database Systems, 1981.

[11] S. Traub. The Design of a Distributed Oberon System. 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] D. Mosberger: “Memory Consistency Models”; Tech. Report TR93/11, Dept. of Computer Science, Univ. of Arizona, 1993.

[14] M. Mizuno, M. Raynal, J. Zhou : “Sequential Consistency in Distributed Systems: Theory and Implementation“; Publication Interne no. 871 Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes, France, 1994,

[15] G. Coulouris, J. Dollimore, T. Kindberg: "Distributed Systems. Concept and Design"; Third Edition, Addison Wesley, Harlow England, 2001.

[16] P. Dadam: "Verteilte Datenbanken und Client/Server-Systeme Grundlagen, Konzepte, Realsierungsformen"; Springer-Verlag, Heidelberg, 1996.

[17] L. Lamport: “How to make a multi-processor computer that correctly executes multiprocess programs”; IEEE Transactions on Computers, C-28(9):690–691, 1979.

[18] N. Wirt, J. Gutknecht: „Project Oberon“; ACM Press, Addison-Wesley New York 1992.