Supporting Multiple Memory Consistency Models on a Shared Virtual Memory Parallel Programming Environment

Alba Cristina Magalhães de Melo Balaniuk
UnB - Universidade de Brasília - Brazil
albamm@cic.unb.br

Abstract *- In order to make shared memory programming possible in parallel or distributed architectures where no physical shared memory exists, we must create a shared memory abstraction that parallel processes can access. The behavior of this abstraction is dictated by the memory consistency model. It has been observed that simulating exactly the real shared memory behavior in such architectures creates a huge network traffic. Several relaxed memory models have been proposed to reduce this traffic. We claim that no single memory model will be able to provide both performance and ease of programming for all classes of parallel applications. Hence, we must allow programmers to choose the memory model that best fits to his parallel application needs.*

This article describe the design and implementation of a shared virtual memory system that supports multiple user-defined memory consistency models. The idea here is to allow programmers to choose the memory model that best fits to its parallel application sharing patterns. The programmer can also define a memory model by his own. This article is divided in three parts. In the first part, we introduce the problem of memory consistency model definition and define formally what is a memory consistency model. In the second part, we describe the design of a general model manager that supports multiple memory models. In the last part, we present the implementation of a prototype of the system in a parallel machine and examine some preliminary results.

1 Introduction

Programming loosely coupled parallel architectures has been traditionally achieved by message passing. However, the use of the shared memory paradigm in such architectures has received a lot of attention last years. This is basically for two reasons. First, the shared memory programming model is often considered easier than message passing, since communication is implicit. Second, this can be the first attempt in order to make portable parallel programs that are independent of the parallel architectures families.

In order to make shared memory programming possible in parallel architectures where no physical shared memory exists, we must create a shared memory abstraction that parallel processes can access. This abstraction is called Distributed Shared Memory (DSM) and can be implemented basically by two different approaches: shared objects and shared virtual memory (SVM). In the first approach, we treat the shared memory abstraction as an object or a set of objects. Operations in this shared memory are methods that access the memory object. The second approach was first proposed by [Li86]. In this case, we use the virtual memory mechanism to map the shared memory

abstraction into the virtual space of each parallel process. Operations in the shared memory are issued by memory addressing mechanisms.

Even though these two approaches are quite different, they both suffer from the same problem: memory consistency. The first DSM systems tried to give parallel programmers the same guaranties they had when programming uniprocessors. It has been observed that providing such a memory model creates a huge coherence overhead, slowing down the parallel application and bringing frequently the system into a trashing state [NL91] [Mos93]. To alleviate this problem researchers have proposed to relax some consistency conditions, thus creating new memory behaviors that are different from the traditional one. Many new memory consistency models have been proposed in the literature but, by now, none of them has achieved the goal of providing both performance and ease of programming.

We claim that no single memory model will be able to achieve this goal for all categories of parallel applications. The best memory behavior for a given application depends on its own shared memory access patterns. Indeed, each parallel application must be given the opportunity of choosing a memory model that best fits to its performance and ease of programming wills.

In this article, we describe the design and implementation of a shared virtual memory system where multiple user-defined memory models can be used. In order to do this work, we had to go through three different stages:

1) *Elaborate a general and formal memory consistency model definition.* The objective of this part of the work is to identify the characteristics that are inherent to all memory consistency models and the characteristics that are model-specific. A simple and general definition of memory consistency model is proposed.
2) *Design of a system architecture that supports multiple user-defined memory consistency models.* We propose a system that uses the general memory consistency model definition produced in the preceding stage to guide the design of a general model manager that will control parallel executions.
3) *Implementation of a prototype of the system.* In order to validate our ideas, we implemented the proposed system in a 52-node Intel Paragon parallel machine. With this prototype, we were able to examine some parallel executions and have some preliminary results.

The rest of this paper is organized as follows. In section 2, we introduce the problem of memory consistency models definition and propose our own general memory model definition. Some well-known memory models are then defined using this definition. Section 3 describes the design of a model manager that supports multiple user-defined memory models. Section 4 describes the implementation of the prototype of a complete shared virtual memory system and presents some preliminary results. Finally, related work and conclusions are presented in sections 5 and 6.

2 Memory Consistency Models

Ideally, a SVM system must provide to its users all the consistency guarantees of a real shared memory. Unfortunately, it has been observed that systems that offer such guarantees suffer from significant efficiency problems. To reduce the cost of maintaining consistency, researchers have proposed to weaken consistency conditions. Following this approach, many relaxed consistency models have been proposed in the literature.

Memory consistency models, strong or relaxed ones, have not been originally formally defined. In most cases, the semantics of memory behavior has to be induced from the protocol that implements the memory model. The lack of a unique framework where memory models can be formally defined makes it difficult to compare and understand the memory model semantics. This fact was also observed by other researchers and some work was indeed done in the sense of formal memory model definitions. [Adv93] has proposed a quite complete methodology to specify memory models. Nearly all known memory models were considered. The aim of her work, however, was to define relaxed models in terms of strong consistency. The central hypothesis of this study is that all parallel programmers would prefer to reason as if they were programming a time-sharing uniprocessor. A set of formal definitions was also proposed in [RS95]. The objective of this study was to understand memory models and compare them. The following memory models were defined using the proposed notation: atomic consistency, sequential consistency, causal consistency and PRAM consistency. [KNA93] also proposed a formal framework to relate memory models where sequential consistency, TSO, processor consistency and release consistency were defined.

In our case, we need a general, simple and formal framework where memory models can be defined. As we consider that the programmer can define his own memory models, we must consider even memory models that do not exist yet. For such a degree of generality, we must define the minimal properties that a memory model should have. Every memory model that fits these minimal properties could be defined in our system.

2.1 System Model

To describe memory models in a formal way, we propose a history-based system model that is related to the models described in [Adve93] and [KNA93]. In our model, a *parallel program* is executed by a *system*. A *system* is a finite set of *processors*. Each *processor* executes a *program* that is composed by a set of *operations* on *the shared global memory M*. The *shared global memory M* is an abstract entity composed by all addresses that can be accessed by a program. Each *processor* p_i has its own local *memory* m_i. Each *local memory* m_i caches all memory addresses of M. There are two basic operations on M: *read (r)* and *write (w)*. Each memory operation is first issued and then performed, i. e., memory operations are non-atomic. At the end of the execution, all memory operations must be performed. For the sake of simplicity, we assume that each processor executes only one process.

Each process running on a processor p_i is described by a local execution history H_{pi}, that is an ordered sequence of memory operations issued by p_i. An execution history H is the union of all H_{pi}. An example of an execution history is shown in figure 1.

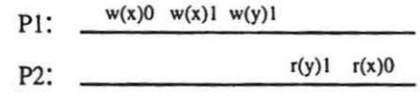

Figure 1 - An Execution History

In this figure, the notation *w(x)v* represents the instant where the write of the value *v* on memory position *x* was issued and *r(x)v* represents the instant where the read of value *v* on memory position *x* was performed by p_i.

In our definitions we often use the notion of *linear sequences*. If *Q* is a history, a *linear sequence* of *Q* contains all operations in *Q* exactly once. A linear sequence is

legal if all read operations *r(x)v* return the value written by the most recent write operation. For the complete description of the system model please refer to [Bal96].

In execution histories, we have some operation orderings that are allowed and some orderings that are forbidden . The decision of which orderings are valid is made by the *memory consistency model.* This observation leads to our memory consistency model definition: *a memory consistency model (MCM) defines an order relation* ($-\!\!>_{,}^{R}$) *on the set of shared memory accesses.* One execution history is possible on a memory consistency model if it is possible on the relation order defined by the model.

In the rest of this section, we describe four well-known memory consistency models (atomic consistency, sequential consistency, causal consistency and release consistency) using this generic memory consistency model definition. For the definitions of PRAM, processor consistency and slow memory please refer to [Bal96].

2.2 Atomic Consistency

Atomic consistency is the strongest and the oldest memory consistency model. Our definition of atomic consistency is derived from the definition presented in [RS95]:

> A history H is *atomically consistent* if there is a legal linear sequence $-\!\!>_{,}^{AT}$ of the set of operations on H such that:
>
> i) $\forall\ o_1, o_2$ where $o_1 -\!\!>_{,}^{po} o_2$ then $o_1 -\!\!>_{,}^{AT} o_2$ and
>
> ii) $\forall\ o_1, o_2$ where $t(perform(o_1)) < t(perform(o_2))$ then $o_1 -\!\!>_{,}^{AT} o_2$

In this definition, the function $t(x)$ returns the real time when the operation x is performed. Above, we defined a new order relation ($-\!\!>_{,}^{AT}$) where all processors must perceive the same execution order of all shared memory accesses (legal linear sequence of H). In this order, all operations performed by a process must respect the program order $-\!\!>_{,}^{po}$ [1] (i). Also, non-overlapping memory accesses must respect real-time order (ii).

We do not have examples of shared virtual memory systems that implement atomic consistency. Preserving real time order is very difficult (if not impossible) in a system where no global physical clocks exists.

---

[1] Program order: $o_1 -\!\!>_{,}^{po} o_2$ if and only if: a) both operations are issued by the same processor and o_1 immediately precedes o_2 or b) $\exists\ o_3$ such that $o_1 -\!\!>_{,}^{po} o_3$ and $o_3 -\!\!>_{,}^{po} o_2$.

2.3 Sequential Consistency

Sequential Consistency was proposed by [Lam79] as a correctness criterion for shared memory multiprocessors. Many shared virtual memory systems proposed in the literature implement sequential consistency. Ivy [Li86], Mirage [FP89] and KOAN [LP92] are examples of sequentially consistent shared virtual memory systems.

Our definition of sequential consistency is derived from the definition presented in [A+92]:

> A history H is *sequentially consistent* if there is a legal linear sequence $\xrightarrow{SC}$ of the set of operations on H such that:
>
> i) $\forall\, o_1, o_2$ where $o_1 \xrightarrow{po} o_2$ then $o_1 \xrightarrow{SC} o_2$.

For sequential consistency, we define a new order $\xrightarrow{SC}$. The only difference between $\xrightarrow{AT}$ and $\xrightarrow{SC}$ is that real-time order is no longer necessary in sequential consistency. These two models are called strong memory models because they pose restrictions on the order of all shared memory accesses issued by parallel processes.

2.4 Causal Consistency

Causal consistency is a relaxed memory model that is based upon the potential causal relation defined by [Lam78]. Causal consistency was implemented by the researchers that proposed it [AHJ90] and was also implemented in the system Clouds [JA93].

For the formal definition of causal consistency, we must use a new history - H_{pi+w} - that is called the *History of the write operations seen by p_i*. H_{pi+w} is a subset of the operations of H that contains all memory accesses issued by p_i and all write accesses issued by every other processes. Our definition is derived from the one proposed by [JA93]:

> A history H is *causally consistent* if there is a legal linear sequence $\xrightarrow{CA}$ of the set of operations on H_{pi+w} for all processor p_i where:
>
> i) $\forall\, o_1, o_2$ where $o_1 \xrightarrow{po} o_2$ then $o_1 \xrightarrow{CA} o_2$ and
>
> ii) $\forall\, o_1, o_2$ where $o_1 \xrightarrow{rb} o_2$ then $o_1 \xrightarrow{CA} o_2$ and
>
> iii) $\forall\, o_1, o_2, o_3$ where $o_1 \xrightarrow{CA} o_2$ and $o_2 \xrightarrow{CA} o_3$ then $o_1 \xrightarrow{CA} o_3$.

By this definition, we can see that it is no more necessary for all processors to agree on the order of all shared memory operations. Instead of using H, we use a subset of it, H_{pi+w}. We have one H_{pi+w} for each processor and the order $\rightarrow,^{CA}$ is defined for these histories. In this order, the order *read-by*[2] must be respected. Also, the transitivity of the new order $\rightarrow,^{CA}$ must be preserved.

2.5 Release Consistency

Release consistency is one of the most popular relaxed memory models and we find in the literature many systems that implement it: Munin [Car93] and ThreadMarks[K+92] are examples of release consistent distributed shared memory systems. DASH [L+93] is a release consistent parallel architecture.

Our definition of release consistency is derived from the one presented by [KNA93]. In this definition, release accesses are seen as special write accesses. Similarly, acquire accesses are treated as special read accesses.

> A history H is *release consistent* if there is a linear sequence $\rightarrow,^{RC}$ of the set of operations on H_{pi+w} for all processor p_i where:
>
> i) $\forall\, o_1, o_2, o_3$ where $o_1 \rightarrow,^{so} o_2 \rightarrow,^{po+} o_3$ in H and type(o_1) = *release* and type(o_2) = *acquire* and type(o_3) $\in$ {r,w} then $o_1 \rightarrow,^{RC} o_2$ and
>
> ii) $\forall\, o_1, o_2$ where $o_1 \rightarrow,^{po+} o_2$ on H_{pi+w} and type(o_1) $\in$ {r,w} and type(o_2) = *release* then $o_1 \rightarrow,^{RC} o_2$.

In short, the definition of $\rightarrow,^{RC}$ imposes that synchronization order[3] $\rightarrow,^{so}$ must be preserved. In addition, all basic memory operations that follow the acquire must be issued after the acquired is performed (i) and all basic memory operations must be performed before the release is issued (ii).

3 Diva: a SVM System that Supports Multiple Memory Consistency Models

[2] Read-by order: if the operation r(x)v, issued by processor p_i, reads the value written by the operation w(x)v issued by processor p_j and $i \neq j$ then $w(x)v \rightarrow,^{rb} r(x)v$

In the preceding section, we showed that generically a memory consistency model can be defined in terms of ordering relations. To prove that this generic model definition can be used, we defined some well-known memory models using the notation we proposed. In this section, we describe the design of a shared virtual memory system, called DIVA[4], that guarantees that all ordering restrictions imposed by a particular memory model will be observed in the execution of a parallel application. For this, we use the memory model definition to guide parallel execution.

We must note, however, that there are several possible implementations of the same memory consistency model. At this design level, we only provide the basic mechanisms to make possible the execution of parallel applications under memory model implementations. Providing the best memory model implementation for a particular memory model definition is beyond the scope of our work.

3.1 The Model Manager

The execution of a parallel program on the generic memory consistency model is treated by the Model Manager Module that is illustrated in figure 2:

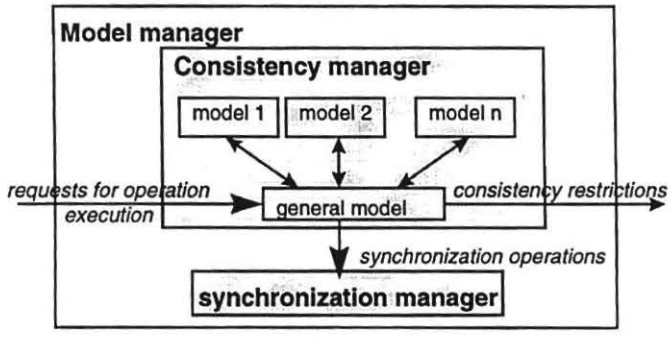

Figure 2 - The model manager

The model manager is activated by the reception of requests for the execution of an operation on the shared virtual memory. The module that implements the general model forwards the request to the memory consistency model that is currently active for the parallel application. The current memory model decides if there are consistency or

[3] Synchronization order: $o_1 \dashrightarrow^{so} o_2$ if o_1 and o_2 are synchronization operations and o_1 is performed before o_2 is issued.

synchronization operations that must be executed before the requested operation can proceed. After the execution of all such consistency and synchronization operations, the model manager sends a message to the module that called him, allowing the operation to be now issued.

3.2 Definition of New Memory Models

To include new memory models in the DIVA system, the programmer must follow the model description scheme illustrated in figure 3.

```
model_New(operation_type)
{
   case (operation_type)
      OP_TYPE1: consistency_restrictions_type1();
      OP_TYPE2: consistency_restrictions_type2();
      OP_TYPEn: consistency_restrictions_typen();
}
```

Figure 3 - The model description scheme

In fact, the programmer associates the type of the operation (read, write or any other one) with the ordering constraints established by the model. After that, the programmer writes the model description onto a configuration file. This configuration file will be read into the code of DIVA. The detailed explanation of this incorporation process is found in [Bal96].

3.3 The Synchronization Manager

The decision of supporting multiple memory consistency models led to transformations in the design of our shared virtual memory system. The most significant one concerned the design of the synchronization module. There is a class of memory models - hybrid memory models - that verify consistency constraints when synchronization operations are issued. Hence, in order to be general, our system must allow the execution of consistency operations at synchronization time. For this reason, we conceived two synchronization operations (diva_lock and diva_unlock) that have

[4] DIVA - DIstributed Virtual memory Approach

dynamic semantics. In other words, the semantics of these operations depends upon the current memory consistency model. These two operations are illustrated in figure 4.

```
diva_lock (lock)
{
 acquire (lock);
 c_lock(memory_model);
}
```

```
diva_unlock (lock)
{
c_unlock(memory_model)
release (lock);
}
```

Figure 4. Synchronization operations

The primitives diva_lock and diva_unlock have an invariant part that does not depend upon the current memory consistency model. This invariant part deals with traditional lock management that is done by the acquire and release operations. The other part, represented by c_lock and c_unlock is specified by each memory consistency model in the memory model definition scheme explained in the preceding subsection.

4 Implementation and Results

In order to validate our design and have some performance measures, we implemented a prototype of the DIVA system on a 52-node Intel Paragon Parallel Machine [Int93]. The operating system used was the Mach micro-kernel [Ope92]. DIVA was implemented as memory manager and uses the Mach Server Interface. Combined with the general memory consistency manager, the prototype of DIVA has all functionalities a normal shared virtual memory server. The architecture of the prototype can be seen in figure 5.

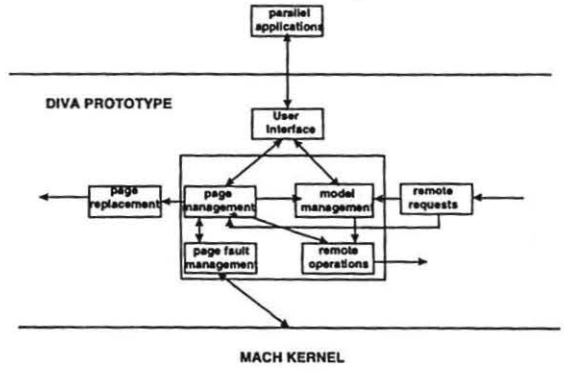

Figure 5 - The prototype architecture

We have one full copy of the DIVA server in each node. Each copy is multi-threaded, composed by four threads. The most important thread is called memory manager and has the following functions: page table management, page fault management and memory consistency model management. These three modules can request remote operations that are issued by the remote operations controller. If local memory is full or nearly full, the page replacement thread is called to place some local pages in a remote node. For this, it uses a page replacement algorithm that moves the page to a neighbor node that has free space in its memory. The detailed description of this algorithm can be found in [BaM94].

The DIVA prototype is activated by the Mach kernel, in the occurrence of a page fault, or directly by parallel applications, when they issue diva primitives. We have eight diva primitives, shown in the following table:

Syntax	Description
`model_set = diva_set_memory_model (model_wanted)`	Definition of the memory consistency model to be used in the execution of a parallel application
`result = diva_map (region, address, size)`	Creation of a shared memory region in the address space of the parallel process
`result = diva_unmap (region, address, size)`	Destruction of a shared memory region in the address space of the parallel process
`result = diva_prefetch (address)`	Prefetching of the page that contains the address <address>
`lock_number = diva_get_lock ()`	The user obtains a lock identifier
`result = diva_remove_lock (lock_number)`	The user remove the lock identifier <lock_number> from the system
`diva_lock (lock_number)`	Lock acquisition operation
`diva_unlock (lock_number)`	Lock release operation

Table 1 - Primitives of the prototype

To evaluate the performance of a single application under more than one memory consistency model, we implemented two memory models in DIVA: sequential consistency and release consistency.

For the implementation of sequential consistency, we used firstly the implementation suggested by [Li86]. This implementation will be called *SC+inv*. As *SC+inv* suffered from trashing caused by false sharing, we implemented sequential consistency again in a different way. The implementation now chosen was the one proposed by [FP89] and will be called *SC+timer*. Both implementations of sequential consistency were considered in our evaluation. To implement release consistency, we chose the implementation proposed by [Car93]. We executed a 16x16 matrix

multiplication using these three memory model implementations and the results are shown in figure 6.

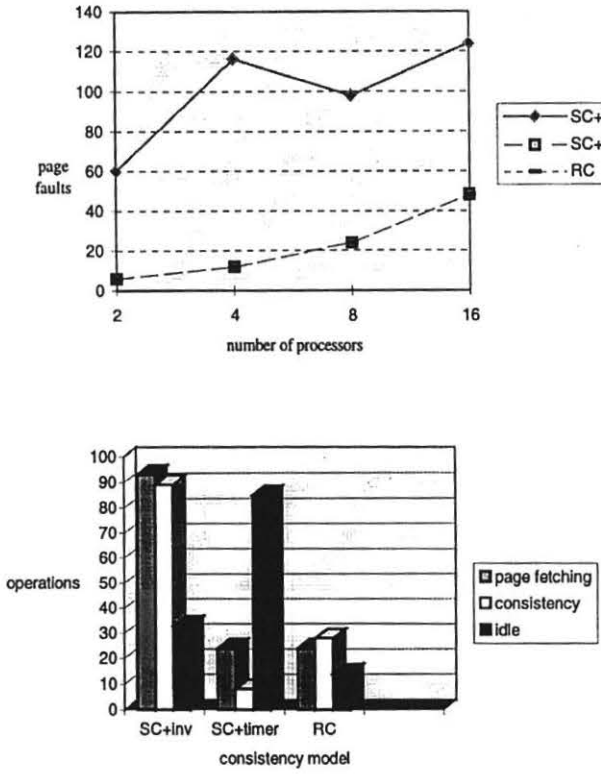

Figure 6 - Evaluation of the matrix multiplication

In the first graphic, we can see that *SC+inv* presents a very high number of page faults, even though matrix multiplication has no write data dependencies. As explained before, this was due to false sharing. Analyzing only the first graphic, one can conclude, erroneously, that *RC* and *SC+timer* have nearly the same performance. The data in the second graphic prove that this is not true. Although both models have nearly the same number of page faults, *SC+timer* achieves this low number by slowing down the computation, because it imposes that a page must be in one node for at least *n* units of time. This can be easily seen by the high amount of idle time that matrix multiplication presents under this model. The careful analysis of these graphics lead us

to say that matrix multiplication behaves better under release consistency than under sequential consistency, in a shared virtual memory system that manipulates pages.

5 Related Work

In the literature, some work has been done to provide users of a distributed shared memory system with multiple memory models.

[HS93] proposed a formalism and a system - Mermera - that permits multiple memory models in a single execution. Shared memory is accessed by read and write primitives. Reads are always local. There is one write primitive for each memory model offered by the system: CO_write (sequential consistency), PRAM_write (PRAM consistency), SLOW_write (slow memory) and LOCAL_write (local consistency). This approach differs from DIVA in three important ways. First, DIVA treats memory accesses and Mermera deals with primitives. Second, DIVA allows the user to define its own memory consistency models. In Mermera, the user must choose among a set of pre-defined models. Third, in a single execution DIVA allows only one memory model to be active while Mermera allows many models to coexist.

Midway is a DSM system first proposed by [BZ91] and then modified in [BZ93]. It was introduced to evaluate a new memory model: entry consistency. As this model was too relaxed, the authors decided to support also some memory models that offer more consistency guarantees. The models supported are entry consistency, processor consistency and release consistency. Similarly to Mermera, Midway provides a pre-defined set of models that can be chosen by the programmer. Multiple memory models can coexist in a single execution. As DIVA, Midway treats memory access operations but it needs a particular compiler to deal with these operations.

We consider that our system is more flexible than these two systems. As a user-defined memory model is added to the set of models accepted by the system, the programmer can also choose among many memory models in DIVA. Besides, the programmer can define himself the memory model needed for a particular application execution. This characteristic is not present in Midway or Mermera. On the other side, DIVA poses the restriction of only one memory consistency model per parallel execution. This restriction that is not present in the other two systems. However, this restriction can be relaxed in DIVA in a relatively simple way [BAL96].

6 Conclusion

In this article, we presented the formal definition of memory consistency models and the design and implementation of a shared virtual memory system that uses this formal definition to provide a multiple memory consistency model support. As our main goal with this work was to show that the choice of a memory consistency model depends on how the parallel application accesses the shared memory, we implemented two memory models (sequential consistency and release consistency) and executed the same parallel application (matrix multiplication) under these models.

Our experience with the matrix multiplication led us to two main conclusions. First, the choice of a memory model affects the performance of a parallel application that uses distributed shared memory to communicate. Second, the choice of an efficient implementation of the memory model is essential and a bad choice can lead to bad results that are not produced by the model itself.

The comparison with other systems showed that, as far as we know, DIVA is the only full SVM system that allows programmers to define generically memory models and incorporate them into the system. This flexibility allows DIVA to be also used as a testbed for evaluating and creating new memory models.

7 References

[Adv93] S. V. Adve, *Designing Multiple Memory Consistency Models for Shared-Memory Multiprocessors*, PhD thesis, University of Wisconsin-Madison, 1993.

[AHJ90] M. Ahamad, P. Hutto, R. John, *Implementing and Programming Causal Distributed Shared Memory*, Technical Report GIT-CC-90/49, Georgia Institute of Technology, 1990.

[A+92] M. Ahamad et al., *The Power of Processor Consistency*, Technical report GIT-CC-92/34, Georgia Institute of Technology, 1992.

[Bal96] A. Balaniuk, *Conception d'un Système Supportant des Modèles de Cohérence Multiples pour les Machines Parallèles à Mémoire Virtuelle Partagée*, PhD Thesis, Institut National Polytechnique de Grenoble, France, 1996.

[BM94] A. Balaniuk and T. Muntean, *Programming with Shared Data in Parallel Loosely Coupled Machines: the Shared Virtual Memory Approach*, In IEEE/USP International Workshop on High Performance Computing, pages 129-142, 1994.

[BaM94] A. Balaniuk and T. Muntean, *Adaptive Page Replacement in the DIVA Shared Virtual Memory Parallel Server*, In Journées des Jeunes Chercheurs en Architectures de Machines et Systèmes, Tunisia, 1994.

[BZ91] B. Bershad and M. Zekauskas, *Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors*. Technical Report CMU-CS-91170, CMU, 1991.

[BZ93] B. Bershad and M. Zekauskas, *The Midway Distributed Shared Memory System*. COMPCON'93, pages 34-42, 1993.

[Car93] J. Carter, *Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency*, PhD Thesis, Rice University, 1993.

[FP89] B. Fleisch and G. Popek, *Mirage: a Coherent Distributed Shared Memory Design*, In 14th ACM Symposium on Operating Systems Principles, pages 211-221, 1989.

[HS93] A. Heddaya and H. Sinha, *An Implementation of Mermera: a Shared Memory System that Mixes Coherence with Non-Coherence*, Technical Report BU-CS-93-006, Boston University, 1993.

[Int93] Intel Corporation. *Paragon System Administrator's Guide*, 1993.

[JA93] R. John and M. Ahamad, *Causal memory: Implementation, Programming Support and Experiences*, Technical Report GIT-CC-93/10, Georgia Institute of Technology, 1993.

[K+92] P. Keleher et al. *Lazy Release Consistency for Software Distributed Shared Memory*, In 19th Symposium on Computer Architecture, pages 13-21, 1992.

[KNA93] P. Kohli, G. Neiger, M. Ahamad, *A Characterization of Scalable Shared Memories*. Technical Report GIT-CC-93/04, Georgia Institute of Technology, 1993.

[Lam78] L. Lamport, *Time, Clocks and Ordering of Events in a Distributed System*, Communications of the ACM, pages 558-565, 1978.

[Lam79] L. Lamport, *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs*, IEEE Transactions on Computers, pages 690-691, 1979.

[Li86] K. Li, *Shared Virtual Memory on Loosely Coupled Architectures*, PhD thesis, Yale University, 1996.

[LP92] Z. Lahjomri and T. Priol, *KOAN: a Shared Virtual Memory for the iPSC/2 Hypercube*, Lecture Notes on Computer Science 634, pages 442-452, 1992.

[L+93] D. Lenosky et al. *The DASH Prototype: Logic Overhead and Performance*, IEEE Transactions on Parallel and Distributed Systems, January, 1993.

[Mos93] D. Mosberger, *Memory Consistency Models*, Operating Systems Review, pages 18-26, 1993.

[NL91] B. Nitzberg and V. Lo. *Distributed Shared Memory: A Survey of Issues and Algorithms*, IEEE Computer, pages 52-60, 1991.

[Ope92] Open Software Foundation and CMU. *Mach 3 Server Writer's Guide*. 1992.

[RS95] M. Raynal and A. Schiper, *A Suite of Formal Definitions for Consistency Criteria in Distributed Shared Memories*, Technical Report PI-968, IRISA, France, 1995.