Constructive and Adaptable Distributed Shared Memory*

Jordi Bataller and José M. Bernabéu-Aubán
Dept. de Sistemes Informatics i Computacio
Universitat Politecnica de Valencia
Ap. 22012, 46071 Valencia, Spain
E-mail: {bataller,josep}@iti.upv.es

Abstract

Distributed shared memory (DSM) is a paradigm for programming distributed systems, which provides an alternative to the message passing model. DSM offers the agents of the system a shared address space through which they can communicate with each other. The main problem of a DSM implementation on top of a message passing system is performance. Performance of an implementacion is closely related to the consistency the DSM system offers: strong *consistency (all agents agree about how memory events happen) is more expensive to implement than* weak *consistency (disagreements are allowed). There have been many DSM systems proposals, each one supporting different consistency levels. Experience has shown that no one is well suited for the whole range of problems. In some cases, strong consistent primitives are not needed, while in other cases, the weak semantics provided are useless. This is also true for different implementations of the same memory model, since performance is also afected by the data access patterns of the aplications.*

In this paper, we introduce a novel DSM model called Mume. Mume is a low level layer close to the level of the message passing interface. The Mume interface provides only the minimum requirements to be considered as a shared memory system. The interface includes three types of synchronization primitives, namely, total ordering, causal ordering and mutual exclusion. This allows efficient implementations of different memory access semantics, accomodating particular data access patterns.

Keywords: *distributed shared memory, memory consistency models, synchronized memory models*

*This work has been supported in part by research grants TIC93-0304 and TIC96-0729

1. Introduction

Shared memory is a widely accepted model to build parallel applications. Its main advantage is that it provides the programmer with a well-know communication paradigm: writing and reading shared variables. When implemented in a distributed system (distributed shared memory, DSM), programs do not have to take care of the details of the underlying communication system, potentially simplifying interprocess communication. In loosely coupled distributed systems, DSM implementations are built on top of a message passing (MP) system. So, it would seem that using DSM would always offer worse performance than using the underlying MP system. However, there are many techniques which allow to reduce the overheads incurred in implementing a DSM system. As a result, DSM and MP systems are comparable [11, 25] and, in some cases, DSM is better [31]. This work focuses on shared memory implemented on loosely coupled systems.

One of the techniques used to improve DSM performance is the relaxation of the consistency model. The consistency one would expect is the same as that offered by a uniprocessor system with a single memory. Roughly speaking, all the agents agree on the ordering in which memory accesses occur. This consistency is called *strong* consistency. Strongly consistent systems are easy to understand and to use but their implementations are somehow inefficient since maintaining a global view is costly in terms of the number of messages (query, update or invalidation messages). Different strongly consistent systems have been implemented using different consistency protocols, tuned to different data access patterns, but it is an accepted fact that no single protocol is suitable for all of the possibilities.

On the other hand, it has been observed that not every algorithm needs strong consistency to solve a given problem. This fact allows the agents to disagree on the order in which they view memory access, therefore perceiving different states of the memory. This kind of consistency is called *weak* consistency. A weakly consistent system can

be implemented more efficiently because the potential disagreement on the memory views allows saving update or invalidation messages. But weakly consistent systems are more difficult to understand and use. Moreover, many different weak memory models have been proposed leading to a confusion about which one is the most suitable.

In a parallel environment, synchronization is a fundamental issue. It has been shown [2] that a distributed synchronization algorithm based on shared memory (i.e. only using the write and read primitives) can not be developed if a weakly consistent memory is used: a centralized algorithm is required. But if strongly consistent memory is used, common synchronization algorithms (based on shared memory) are also inefficient because of the high level of message transmission due do the intensive consultation of synchronizing variables.

This fact has led to introduce synchronization as the third primitive in DSM systems. We uniformly call it *sync* (there are many types, e.g. acquire, release, barriers, etc). Sync can be implemented through efficient MP algorithms. This primitive is also used in some weakly consistent systems to signal the points in which the memory has to be globally consistent to all the agents. We call these systems *synchronized* systems.

At present, things are well understood, but there is not a definitive approach on how to achieve the desirable shared memory system being capable of rivalling MP systems in terms of performance and still retaining simplicity of usage. The solution we propose is based on the (old) idea of offering simple, efficient, adaptable and constructive means which can be used to develop new tools fulfilling specific requirements and allowing ad-hoc improvements.

We propose a shared memory model called Mume. Mume has got two types of primitives: *ordinary* operations, *read* and *write*; and *synchronization* operations, *TOsync*, *CRsync* and *MEsync*. It has a set of memory areas (*caches*) accessible through ordinary operations.

Ordinary actions are similar to MP primitives *send* and *receive*, because, like them, they have a parameter indicating the cache where they have to act on. This enables an almost direct, and thus efficient, implementation. Although both kinds of primitives have similarities, they are quite different. A Mume write action does not need to be observed and it is overwritten by a subsequent write. A sent message, even out of order, has to be received. This is also discussed in [1].

Each Mume cache has a FIFO behaviour, i.e. it applies ordinary actions in the order they arrive, unless they are synchronized. Sync operations are used for relating ordinary operations, controlling the order in which they are applied to different caches. TO (total order) related operations have to be applied to every cache following the same order. Similarly, CR (causal order) related operations have to be applied to every cache accordingly to the causal relation. ME (mutual exclusion) is to gain exclusive access to a set of caches.

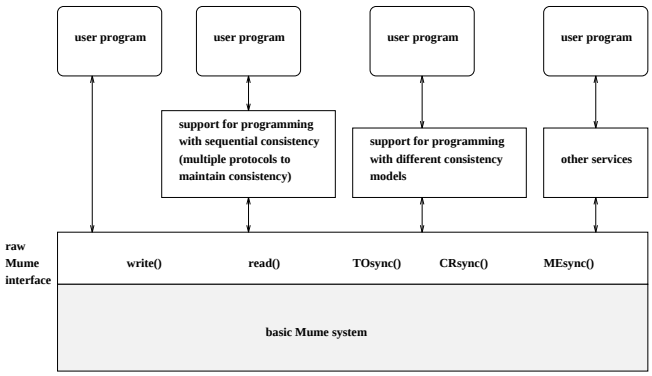

Figure 1. Mume environment.

With that set of primitives, a number of important advantages can be achieved, see figure 1. The primitives of the principal memory models proposed up to now can be emulated, thus permitting to re-use algorithms based on particular models. Within the same memory model, a single primitive can be implemented in different ways to have different behaviours, so that it can fit specific data access patterns. So, different primitives, having the desired behaviour, can be developed to be used in a single application. Finally, because a "high level" program can be transformed into a "low level" Mume program, the latter program can be fine tuned to improve performance by removing unnecessary operations.

Roughly speaking, Mume can be viewed as a shared memory system where the sharing unit is the whole memory and where the programmer, through a reduced set of primitives, is in charge of maintaining the consistency.

We want to remark that our Mume description, although being operational, does not force any particular implementation.

Mume builds on the work in [6] and [4], where the authors present a formalism with which to express and compare most memory models proposed to date. Mume has been formally defined in [3], where, in addition, it is proved that Mume can emulate other memory models.

The rest of the paper is organized as follows. Section 2 describes Mume operationally. Section 3 illustrates Mume versatility and adaptability. Section 4 shows a preliminary design of a Mume system and discusses which techniques can be used to build it. Finally, in section 5 we summarize this paper and discuss further work.

2. Description of Mume

As we have previously indicated, Mume memory primitives are classified into ordinary (or data) operations, *read* and *write*; and synchronizations, *sync*. Mume offers a shared memory which may be accessed through these operations. The memory is arranged as a set of areas (*caches*) that can be independently accessed by each agent. Therefore,

- A cache is a copy or a version of the whole memory.
- At a given moment, caches can have different states.
- There is no cache designated as the primary copy. (This type of meanings are left to the programmer.)

Implementations are not forced to associate a cache with each agent to use it as a local copy of the memory. Anyway, this seems to be a good choice. By default, caches apply operations as soon as they are received, i.e. FIFO. This means that Mume data actions are totally independent of each other even if they are from the same agent or process.

We denote data actions as $write(i, x, v)[A]$ and $read(i, x, v)[a]$. i is the identifier of the agent issuing the operation, x is the variable and v the value being written or read. A is the set of caches where the value is to be written and a is the cache from which the value is to be fetched.

The basic orderings we consider are process order (PO), the order in which an agent issues actions; writes-to ($\mapsto$), which relates a write with the read which gets the value; and causal relation (CR), the transitive closure of PO and $\mapsto$. This is, a and b are CR related if b follows a in PO, or $a \mapsto b$, or there exists c such that a and c are CR related and c and b are CR related. See [1] for details about causal relation in shared memory.

The way of relating ordinary actions is by means of synchronizations. Mume has three kinds of sync actions, namely, total ordering, causal ordering and mutual exclusion. They are respectively denoted as $TOsync(i, s_{TO}, t)$, $CRsync(i, s_{CR}, t)$ and $MEsync(i, s_{ME}, t)$. i is the agent identifier. s_{TO}, s_{CR} and s_{ME} are synchronization variables. t is the type of action, acq (*acquire*) or rel (*release*), and they are to mark which data actions are affected.

Acquire sync marks the beginning of a synchronization and release sync marks the end. So, we say that ordinary actions issued between an acquire and a release and the sync actions themselves are *synchronized actions*. Issuing a second acquire before releasing the previous one is not allowed. In other words, it is not allowed nested synchronizations even if they are different types of synchronizations or the sync acts on different synchronization variables.

We say that synchronized actions from different agents are *related* (*TO-related, CR-related, ME-related*) when they are synchronized by syncs acting on the same synchronization variable.

Program:

agent a : $TOsync(a, s_1, acq)$ $a_1[c_1, c_2]$ $a_2[c_1, c_2]$ $TOsync(a, s_1, rel)$

agent b : $TOsync(b, s_1, acq)$ $b_1[c_1, c_2]$ $TOsync(b, s_1, rel)$ $b_2[c_1, c_2]$

Correct execution:

cache c_1 : $a_1\ b_1\ a_2\ b_2$

cache c_2 : $b_2\ a_1\ b_1\ a_2$

Because a_1, a_2 and b_1 are TO-related on s_1 they are applied in the same order to both caches. PO between a_1 and a_2 is also respected. Because b_2 is not TO-related it can be applied in any order to any cache.

Figure 2. TOsync example.

The meaning of each synchronization type is natural and obvious. TO-related actions are applied in the same order to every cache they act upon. In other words, if action a and b should be applied to caches 1 and 2, it can not happen that cache 1 applies first a and then b and cache 2 applies b before a. TO-related actions respects PO. See figure 2.

Program:

agent a : $CRsync(a, s_1, acq)$ $w(a, v_1, 1)[c_1, c_2]$ $CRsync(a, s_1, rel)$

agent b : $CRsync(b, s_1, acq)$ $r(b, v_1, 1)[c_1]$ $b_1[c_2]$ $CRsync(b, s_1, rel)$

Correct execution:

cache c_1 : $w(a, v_1, 1)\ r(b, v_1, 1)$

cache c_2 : $w(a, v_1, 1)\ b_1$

$w(a, v_1, 1)$ and b_1 are CR-related (CR-synchronized on s_1). b_1 causally follows $w(a, v_1, 1)$ ($w(...)$ writes to $r(...)$ and b_1 PO follows $r(...)$). Therefore, cache c_2 must apply $w(...)$ before that b_1.

Figure 3. CR example.

Similarly, two CR-related actions should be applied to every cache according to the causal relationship between

them (see [1]). If they are not causally related any order is fine. See figure 3.

Program:

agent a : $MEsync(a, s_1, acq)$ $a_1[c_1, c_2]$ $a_2[c_1, c_2]$ $MEsync(a, s_1, rel)$

agent b : $MEsync(b, s_1, acq)$ $b_1[c_1, c_2]$ $b_2[c_1, c_2]$ $MEsync(b, s_1, rel)$

agent c : $MEsync(c, s_2, acq)$ $c_1[c_1, c_2]$ $MEsync(c, s_2, rel)$

Correct execution:

cache c_1 : $a_1\ a_2\ b_1\ c_1\ b_2$

cache c_2 : $a_1\ c_1\ a_2\ b_1\ b_2$

Because actions from agent a and from agent b are ME-related, caches can't interleave them and have to apply in the same order. c_1 is not ME-related to actions from a or b.

Figure 4. ME example.

MEsync is the strongest synchronization since it is used to achieve mutual exclusion among ME-related accesses. A MEsync acquire can not proceed until the agent holding the exclusion right (for the affected synchronization variable) issues a MEsync release. Obviously, not two agents are allowed to simultaneously hold the right. Roughly speaking, when an agent gains the exclusion on a given synchronization variable it can access any cache knowing that no other agent will access any cache at the same time if this second agent synchronizes on the same variable. See figure 4.

3. Using Mume

In this section we want to show the versatility of Mume, illustrating how to use it to emulate most memory models, how to develop primitives with different implementations within the same memory model or simply how to directly use it to build final applications.

3.1. Emulation of memory models

The first thing to be noted is that each model can be emulated in different ways. The way we have chosen here is by associating each agent with a "local" cache. Therefore, writes are to act on all of the caches, while reads should fetch values from their respective local cache. The correctness of the descriptions that follows has been formally proved in [3], where many more emulations can be found.

Sequential consistency

This model was introduced by Lamport in [24]. A sequentially consistent system must behave (produce the same results) as a uniprocessor system with a single (non distributed) memory does.

If one cache per process is used, all of the actions must be TO-related so as to ensure that all of them are applied in the same order to all of the caches.

The manner to do this is simple: the first and last action of each process should be $TOsync(i, o, acq)$ and $TOsync(i, o, rel)$, respectively. That relates every action between the pair acquire and release, and, because the only synchronizing variable used is o, every action (from any process) is TO-related.

PRAM consistency

PRAM was introduced in [27]. This is a weakly consistent model. It has only to ensure that each agent sees writes from other agents in PO. Distinct agents may disagree in the order in which writes from different agents are seen.

The way to emulate it is by TO-relating actions from the same process, that is, $TOsync(i, o_i, acq)$ and $TOsync(i, o_i, rel)$ should be the first and the last action issued by agent i, respectively. Note that here, a different synchronizing variable, o_i, is used for each process, so as not to globally TO-relate all actions.

Causal consistency

The Causal model was introduced in [21]. This is another weakly consistent model. Each agent should perceive writes from other agents in accordance with causal order. In other words, two writes can be seen in a different order only if they are not causally related.

That means that we have to CR-relate every action of every process so that they are causally applied. Thus, the first and last action of every process should be, respectively, $CRsync(i, r, acq)$ and $CRsync(i, r, rel)$. The causal synchronizing variable used, r, is the same for every agent.

Cache consistency

This weak model was introduced in [18]. Actions on the same variable should be seen in the same total order. But actions from the same process can be perceived out of PO if they act on different variables.

The way to emulate this model, having one cache per agent, is by TO-relating actions acting on the same variable. Thus, one different synchronizing variable, o_v, is to be used per ordinary variable v. So, an ordinary action acting on v is to be surrounded by $TOsync(i, o_v, acq)$ and $TOsync(i, o_v, rel)$.

Synchronized models

These models include the sync primitive, which is used to synchronize processes as well as copies of the memory. They are Weak consistency [14], Release consistency [17], Entry consistency [7] and Scope consistency [19].

The ME synchronization of Mume has a behaviour similar to the sync primitive of those models (perhaps, it is more close to the Scope model), therefore, a Mume system can be used in an similar way.

The key differences, as we know, are that Mume memory is a set of independently accessible caches and that Mume has other two types of synchronizations to relate data accesses. This enables Mume to emulate more memory models and allows the programmer to fine tune the programs.

3.2. Derived memory primitives

The results of several studies [5, 15, 33, 10] show that the performance of an implementation can be improved if different types of protocols are used to maintain (the same model of) consistency. This is because no single protocol can efficiently support every data access pattern. Even within a single program, different variables can be accessed in a different manner. Thus, the programmer, knowing how the variables are accessed, may choose the most suitable protocol to manage them.

The native primitives of Mume explained in the previous section are low level, which allows to use them in a constructive manner in order to develop new primitives with customized behaviour.

- Libraries of new access operations (derived primitives) can be developed. These libraries can offer support for different memory models. Within the same model, the same primitive can be supported with different implementations.
- A programmer can develop an application using a given memory model emulated by Mume. The Mume equivalent basic program (i.e. in terms of native Mume primitives) can be obtained an then he/she can inspect and fine tune it.
- For a single application, the programmer can, obviously, build his/her own access operations fulfilling specific needs.

Now, we are going to discuss how Mume native primitives can be used to address typical problems and issues.

3.2.1 False Sharing and Multiple Writers Protocols

The false sharing problem appears when "unrelated" or "logically independent" variables are held together in the same shared unit (e.g. page, block, etc.) managed by the consistency protocol. When accessing one of these variables, messages due to the consistency maintenance targeted to other copies of the unit may be unnecessary if remote agents do not actually need to access them.

In order to mitigate this drawback, protocols allowing different agents to simultaneously access the same unit have been developed. They are called *multiple writers protocols* (MW), for instance TreadMarks [23], and Munin [9] in a recent release [10].

Since native Mume writes and reads are "targeted", the (low level) Mume programmer can select which set of caches are to be updated. For example, if an array is being independently processed by two agents, each one may use only its "local" cache. When the processing ends, each agent can update its cache or the cache of other agents by performing a "block update", a read or a write of part of the cache using a single message. Obviously, this simple technique can save unnecessary messages in the processing phase.

Also, MW protocols can be developed in Mume to be used in a "higher" level of programming. Anyway, as [22] shows, the consistency model has a higher impact on the overall performance of a system than the use of a (complex) MW protocol. The use of a weak model, like Release consistency, masks most of the false sharing problem. If such a model is used, the improvement obtained by employing a MW protocol is relatively small. In some cases, a single writer protocol performs better than a MW protocol, due to the overhead of this second protocol.

A particular implementation of the MEsync primitive of Mume can make it behave as a release system does, this is, to delay messages to Mume caches until the release of the lock. According to [22], this will improve part of the false sharing problem. Because as discussed before, specific improvements can be made to the low level Mume code in crucial phases, this would permit to improve the performance of programs more than the use of a general MW protocol: it is not incurred in the MW protocol overhead but crucial performance problems can be solved.

3.2.2 Access Patterns

[10] classifies variables according to how they are accessed and suggests that each type should be maintained using a particular protocol to enhance performance. We will show how they can be managed in Mume, what is obvious in most cases. In order to follow [10] suggestions and to make things easy, a new Mume layer or library, with a new primitive for each type of variable, could be developed upon the basic Mume primitives. This layer can include a manager who makes intelligent decisions in real time basing on the execution access patterns. [32] is an example of a self-

adjusting coherence support.

Mostly Written and Mostly Read Variables Mostly read variables can be efficiently managed by using "broadcast" writes, that is, writes acting on every cache. Since reads are frequently issued, updated local copies reduce access latency.

On the other hand, if a variable is often written, it is advisable to centralize it into a single cache. In this case, writes should be addressed to the cache where the variable is hosted.

If a real time manager is being used, the host cache for each variable can be dynamically changed changed according to the agent issuing the highest number of writes.

Migratory Variables These variables (see [10]) are alternatively accessed by different agents, each one issuing multiple accesses. This behaviour is typical of data accessed inside a critical section. Therefore we should protect the accesses with MEsync. An efficient implementation of this type of primitives will only deliver updates at release time thus saving messages.

Write-shared These variables (see [10]) are concurrently written by multiple agents without synchronization since each agent accesses a disjoint set of them. They may produce false sharing if the implementation tries to keep all of them consistent all of the time. We have already discussed this problem.

Synchronization As we have discussed, the MEsync primitive must be used for synchronizing purposes instead of "normal" data variables. MEsync should be implemented with a special and efficient protocol that, compared to a conventional protocol for maintaining the consistency of data variables, dramatically reduce the number of messages required to synchronize agents.

4 Implementing Mume

In this section, for the sake of exemplifying that a Mume system is implementable, we want to show a preliminary design in which we are currently working. We also want to briefly comment which techniques and protocols may be used.

4.1 Architecture

The basic sketch of our general Mume architecture is shown in figure 5. Processes use the system through a homogeneous interface which offers the primitives introduced

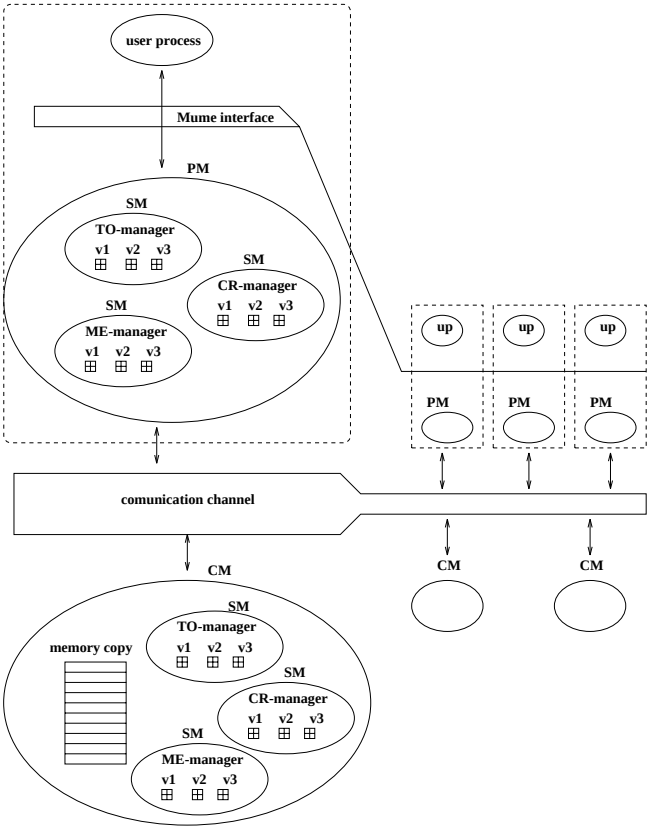

Figure 5. Basic System Architecture.

in previous sections. Each process is associated with a "process manager" (PM) which directly serves its calls. Similarly, each cache is managed by a "cache manager" (CM). PMs and CMs communicate through a peer-to-peer channel which is only assumed to offer send and receive primitives without loosing messages, although it may deliver them out of order.

Given that nesting syncs is not allowed, only one type of relation is possible for each ordinary action, e.g. not-synchronized, TO, CR or ME. Therefore, each PMs and CMs have "synchronization managers" (SM) specialized in each type of synchronization. Since synchronization modes are independent a SM only talks with SMs of its same type.

When a process issues an acquire, the local PM knows which type of synchronization is desired and which sync variable is to be used, given the last sync acquire used. This situation ends when a release is performed. Then, actions issued between an acquire-release pair are addressed to the corresponding local SM which, using the implemented protocol, communicates with the other SMs to perform the requests accordingly to the selected quality of service. When an agent issues a non-synchronized action, the PM simply forwards the request to the target CM.

CMs are not just passive memory containers. Some pro-

tocols may need coordination among CMs or making intelligent decisions. For instance, existing Causal Memory protocols, [1], require to block update messages until causally previous ones are applied. This is why CMs have SMs as well.

The motivation for having separated PMs and CMs is only for sake of generality, i.e., a cache does not need to be associated to each user process nor to each machine. But in a real implementation, it seems to be more convenient to host a CM in each machine, thus allowing fast local accesses.

4.2 Protocols

Implementing the architecture presented is not difficult. The architecture is modular and the protocols for each type of synchronization run independently. Therefore, it should be specified which protocol is to be used for each type of SM, e.g. TO, CR and ME. Obviously, we need to implement the best existing protocol of each type, specially in terms of the required number of messages, since within a certain range of message sizes, there is no difference in sending one small or one big.

For mutual exclusion, we plan to use a "path compression" algorithm ([26, 13]): a token is interchanged among agents, only the agent holding it is allowed to enter its critical section. In order to locate the token and manage requests to enter the critical section, each agent maintains a "guess" variable pointing to the agent supposed to hold the token. Globally, the whole set of guesses forms a tree, the root of which indicates the agent holding the token.

Like Release consistency, the semantics of MEsync allows to delay data modifications until an agent releases the token. In that moment, the modifications are propagated to caches before the following process is allowed to proceed.

For causally ordered operations, we are developing the algorithm proposed in [1]. It uses vector clocks, [29], to determine causal precedence. Such a vector has one component per agent, storing the number of its last known write. We plan to improve the algorithm by using the technique for causal broadcasts explained in [28], which dramatically reduces the size of such vectors. In this case, a vector only contains a reference of each write observed (read) since the last local write.

There are many broadcasting protocols suitable to achieve totally ordered cache accesses. For example, [12, 20, 30] as well as those in the "Isis Toolkit" [8].

Because the work on this subject is currently in progress and we are still evaluating each one of the above techniques, we can not discuss them, their merits and possible improvements more deeply.

Some minor improvements are been considered as well. For instance, packing messages [16] seems to be a technique which significantly improves performance in message passing systems.

5 Conclussions

In the present work, we have introduced a novel shared memory model called Mume. Mume offers ordinary access primitives (read and write) which operate on memory deposits as well as three types of synchronizations (total ordering, causal ordering and mutual exclusion) so as to relate ordinary accesses.

We have argued why we think Mume is a suitable and efficient model for the shared memory paradigm in a loosely coupled system: it can be implemented using existing efficient protocols upon a message passing system, it allows to develop new high-level primitives to support other memory models or to adapt to different data access patterns. In these cases, because low level Mume code can be obtained, the programmer can fine-tune critical parts of it.

We have also presented a preliminary and general design of a Mume system and we have briefly discussed which protocols and techniques can be used in the development.

References

[1] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9, 1995.

[2] H. Attiya and R. Friedman. Limitations of Fast Consistency Conditions for Distributed Shared Memories. *Information Processing Letters*, 57(5):243–248, Mar. 1996.

[3] J. Bataller and J. Bernabeu-Auban. An adaptable DSM model: a formal description. Technical Report II-DSIC-17/97, Dep. de Sistemes Informatics i Computacio (DSIC), Universitat Politecnica de Valencia (Spain), 1997.

[4] J. Bataller and J. Bernabeu-Auban. Synchronized DSM models. In *Europar'97. Third Intl. Conf. on Parallel Processing*, Lecture notes on computer science, pages 468–475. Springer-Verlag, August 1997.

[5] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proc. of the 17th Annual Int. Symposium on Computer Architecture*, pages 125–135, May 1990.

[6] J. Bernabéu-Aubán and V. Cholvi-Juan. Formalizing memory coherency models. *Journal of Computing and Information*, 1(1):653–672, May 1994.

[7] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 528–537, Feb. 1993.

[8] K. P. Birman and R. van Renesse, editors. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[9] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. *Operating System Review*, 25(5):152–164, October 1991.

[10] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, Aug. 1995.

[11] S. Chandra, J. R. Larus, and A. Rogers. Where is time spent in message-passing and shared-memory programs. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 61–73, October 1994.

[12] J. Chang and N. Maxemchuck. Reliable broadcast protocols. *ACM Trans. on Computer Systems.*, 2(3):251–273, August 1984.

[13] Y. Chang, M. Singhal, and M. Liu. An improved $o(\log(n))$ mutual exclussion algorithm for distributed systems. In *Proc. of the International Conf. on Parallel Processing*, pages III295–302, 1990.

[14] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 434–442, June 1986.

[15] S. Eggers and R. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 373–382, 1988.

[16] R. Friedman and R. Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. Technical report, Dept. of Computer Science. Cornell University. Ithaca NY, july 1995.

[17] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26. IEEE, April 1990.

[18] J. Goodman. Cache consistency and sequential consistency. Technical Report 1006, University of Wisconsin-Madison, February 1991.

[19] L. Iftode, J. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. Technical report, Dept. of Computer Science, Princeton Univ., 1996.

[20] X. Jia. A total ordering multicast protocol using propagation trees. *IEEE Trans. on Parallel and Distributed Systems*, 6(6):617–627, 1995.

[21] R. John and M. Ahamad. Causal memory: Implementation, programming support and experiences. Technical Report GIT-CC-93/10, College of Computing, Georgia Institute of Technology, 1993.

[22] P. Keleher. The Relative Importance of Concurrent Writers and Weak Consistency Models. In *Proc. of the 16th Int'l Conf. on Distributed Computing Systems (ICDCS)*, pages 91–98, May 1996.

[23] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Winter USENIX*, 1994.

[24] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[25] T. J. LeBlanc and E. P. Markatos. Shared memory vs. message passing in shared-memory multiprocessors. In *Fourth Symposium on Parallel and Distributed Processing*, December 1992.

[26] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239. ACM, August 1986.

[27] R. Lipton and J. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Department of Computer Science, September 1988.

[28] R. Prakash, M. Raynal, and M. Singhal. An efficient causal ordering algorithm for mobile computing environments. In *Proc. of the 16th Int'l Conf. on Distributed Computing Systems (ICDCS)*, pages 744–751, May 1996.

[29] R. Schwarz and F. Mattern. Detecting causal relationship in distributed computations: In search of the holy grail. Technical Report SFB124-15/92, Department of Computer Science, University of Kaiserslautern, 1992.

[30] I. Stoica, R. Yerraballi, and R. Mukkamala. A simple ordered and reliable group multicast protocol. Technical Report TR-94-26, Old Dominion University, September 1994.

[31] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *Computer*, 23(5):54–64, May 1990.

[32] H.-H. Wang and R.-C. Chang. A distributed shared memory system with self-adjusting coherence scheme. *Parallel Computing*, 20(7):1007–1025, July 1994.

[33] W.-D. Weber and A. Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 243–256, April 1989.