Adaptive selection of protocols for strict coherency in distributed shared memory

Thierry Le Sergent* and David C.J. Matthews†
LFCS, University of Edinburgh.

Abstract

In this paper we present a scheme for maintaining coherency between copies of objects in a distributed system. Instead of using a fixed protocol for all objects and all time, our scheme is adaptive and is able to select the best from among a set of three protocols. The protocol is selected for each object and may change over time if the pattern of accesses to that object changes.

The scheme is in the process of being implemented as part of a distributed shared memory system for ML-like languages. We give a simulation of an example which demonstrates that between 30 and 40 per cent of messages can be saved by our scheme, compared with the dynamic distributed manager algorithm of Kai Li [LH89].

Key words: Distributed shared memory, adaptive algorithm, concurrent functional languages.

1 Introduction

In recent years there has been considerable interest in languages which add concurrency to Standard ML [MTH90], and we are currently developing a distributed implementation for two of these, Poly/ML [Mat91] and LCS [BL94]. Standard ML is a functional language with a strict type system. It has a formal definition, which among other things defines the way in which updates are performed. Most objects are in fact not updatable, and once an object is given a value, the type system will ensure that a program cannot change it. This property can be exploited by a distributed implementation, which can make copies of these immutable objects without concern for their coherency. In this paper, we use the word 'object' to describe an entity which is handled as a single unit. Although our algorithm does not make assumptions about the size of an object, in the particular applications with which we are concerned, it is likely to be a cell of only a few words.

In our implementation the immutable objects are separated out from the mutable objects and this leaves a small number of mutable objects for which coherency is required. Although in a functional language, assignment is less frequent than with a more conventional programming language, it is none the less sufficiently common, particularly in certain programs, that it needs to be implemented efficiently. In particular, by separating out the immutable from the mutable data, we are left with areas of memory containing mutable data which have a much higher frequency of assignment than if the objects had been spread evenly through the memory. Mutable objects placed adjacently in memory may well not be logically connected and so the advantages of locality, on which many algorithms for coherency depend for their efficiency, may not apply.

*E-mail: tls@dcs.ed.ac.uk
†E-mail: dcjm@dcs.ed.ac.uk

One further constraint is that we wish to make it as easy as possible for programs to be ported between different implementations, and we do not wish to require the programmer to add annotations to mutable objects to describe the expected access patterns. In addition we cannot tell whether a mutable object is in fact purely local to a particular machine, and that could depend on the way in which processes were placed on a particular network of machines.

Taken together, these constraints argue for a system of strict coherency on an object-by-object basis. Strict coherency means that whenever any machine reads from an object it receives the last value written, whichever machine has written it. Coherency in a distributed system is achieved by exchanging messages between the machines according to some protocol. Various protocols are possible, each of which is optimal for a particular sequence of reads and writes, but performs worse in other cases. Rather than choose a single protocol and use it in all cases, we have chosen to implement a scheme which adaptively discovers the best protocol to run for a particular object, and can change this protocol if the access patterns change.

In the remainder of this paper we give, first, the protocols from which the choice is made. The next section describes the algorithm. We then describe a simulation of the system on an example program. The final section covers related work and our conclusions.

2 The Protocols

Three different protocols may be used, each of which performs better with a particular access pattern. All the protocols have the property that it is possible to run them and at the same time compute the number of messages that would have been exchanged had one of the others been chosen. This property is fundamental to our algorithm.

In the protocols one machine has special responsibilities for the object and we call this the manager. In particular the manager always has a valid copy of the object. The choice of the manager critically affects the performance of the protocol and so our algorithm selects both the best protocol and the best manager. The protocol and the manager are always selected together and they remain fixed for a period, unlike for example, the dynamic distributed manager algorithm of Kai Li [LH89]. However the manager and protocol may change as result of our adaptive algorithm and so we use his scheme to allow other machines to find the manager when it changes.

The protocols are known as retention, invalidation and update and are described below. An illustration is given in figure 1.

Retention: The manager has the only copy of the object; it can read and write to it without cost. Whenever another machine wishes to read or write to it, it must send a message to the manager. The manager replies with the value of the object for a read or an acknowledgement for a write.

Invalidation: Whenever a machine wishes to read an object, it asks for a copy of it from the manager, so any further reads it performs are free. The manager keeps a set, N, containing the machines that have a copy. Whenever a machine wishes to write an object, it sends the new value to the manager. If there is more than one copy of the object the manager sends a invalidation message to all the machines that have a copy, other than the requester, and waits for an acknowledgement from them. It can then update the object and send an acknowledgement to the waiting machine. We choose that the writer also invalidates its copy so that a subsequent write from another machine does not need any invalidation.

Update: As with the invalidation protocol, any machine may have a copy of the object and the manager maintains a set, E, of the machines that have a copy. Whenever any machine wishes to update it, it asks to the manager to send the new value to all of the other machines that have a copy. To ensure coherency, these machines must lock their copy until all the copies have been updated. They send acknowledgement messages to the manager which will send them an unlock message when it has received all the acknowledgements.

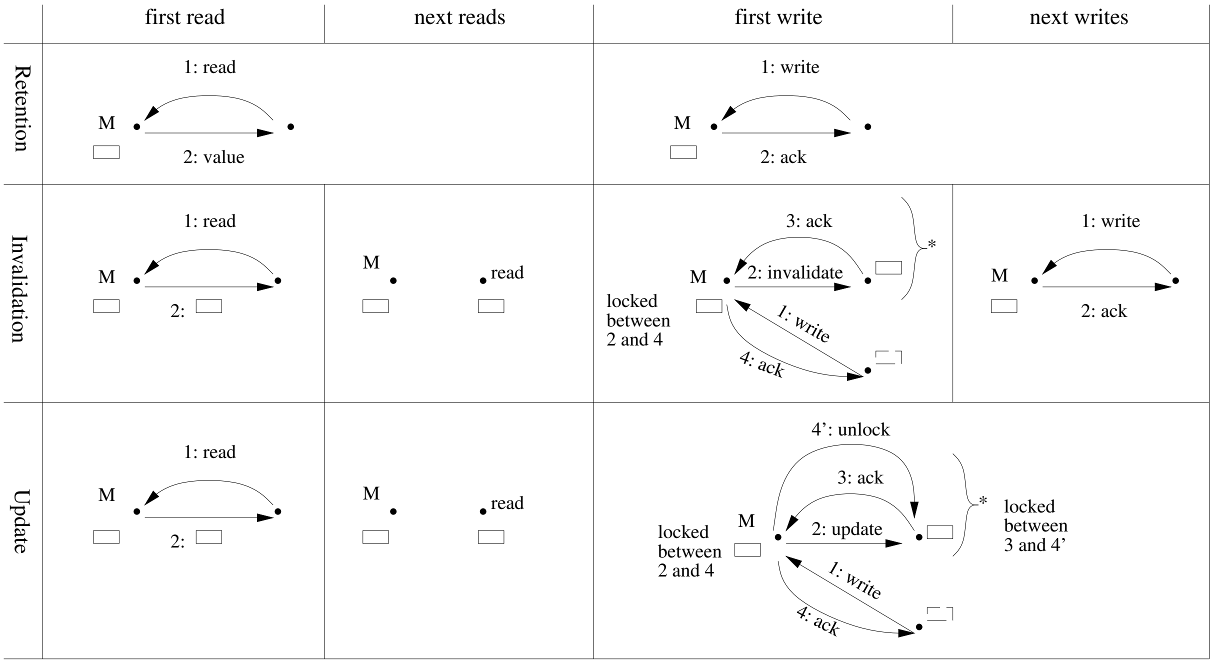

Figure 1: The protocols

3 The Algorithm

Our algorithm attempts to find the best protocol and the best manager by measuring the access pattern on an object for a period known as the 'evaluation'. At the end of the evaluation period we choose the protocol and manager which would have involved the exchange of the smallest number of messages during the evaluation. On the assumption that subsequent accesses will follow the same pattern this will be the best choice to use.

The evaluation is designed in such a way that it does not add any extra overhead in messages, except at the end, and does not require a large overhead in storage.

In this section we first present the formulae used to compute the costs of each of the protocols, and then show how these computations are used to determine the protocol and manager to be used during the lifetime of an object. We also describe some of the optimisations and the data required for the implementation.

3.1 Computations of the cost

The computation is based on a sequence of steps, each step beginning and ending with a write. More precisely, a step begins immediately after a write and ends immediately after the next write. Within a step there may be several reads. A complete evaluation involves several steps.

The cost computations are given for each of the protocols. It is important to note that the cost of each of the protocols is computed while continuing to run the existing protocol. When computing the costs of the various protocols we ignore the cost of discovering the location of the manager since this cost is incurred only for the first access by each machine.

We use the following notation:

S: set of the steps taken into account for the evaluation. *final* is the last step, and *pre(s)* is the step preceding step s.

A: set of all the machines

m: identity of the manager which always has a valid copy of the object

w^s: identity of the writer in step s

R_i^s: number of reads by machine i in step S. $R_i = \sum_{s \in S} R_i^s$

W_i^s: number of writes by machine i in step S (equal 0 or 1). $W_i = \sum_{s \in S} W_i^s$

N^s: set of the machines that have read the object at step s.

E^s: set of the machines at the step s that have read the object since the beginning of the evaluation.

3.1.1 Retention

The cost of the retention protocol is the following:

$$\begin{aligned} C_{ret}^s &= \sum_{i \in (A - \{m\})} 2R_i^s & \text{(reads)} \\ &+ \sum_{i \in (A - \{m\})} 2W_i^s & \text{(writes)} \\ C_{ret} &= \sum_{s \in S} C_{ret}^s = \sum_{i \in (A - \{m\})} 2R_i + 2W_i \end{aligned}$$

This cost is lowest if the manager m is chosen to be the machine with the greatest value for $R_i + W_i$.

In order to compute the cost of using the retention protocol each machine has two local counters, R and W. R is incremented at every read and W at every write. For this protocol it would be enough to have one counter for both forms of access, but the split is useful for the computation of the other protocols. At the end of the evaluation, the current manager collects this information, and uses it to compute the cost of using this protocol and the best machine to be the manager. This is compared with the costs computed for the other protocols to select the best.

3.1.2 Update

The cost of the update protocol is:

$$\begin{aligned} C_{upd}^s &= 2|E^s - E^{pre(s)} - \{m\}| & \text{(reads)} \\ &+ 3|E^s - \{m\}|W_m^s + \sum_{i \in (A - \{m\})} (2 + 3|E^s - \{i, m\}|)W_i^s & \text{(writes)} \end{aligned}$$

This protocol differs from retention in that the cost of repeating the same sequence of accesses does not lead to the same cost. Once an object has been read by a machine it remains there and subsequent reads have no cost. Since we are computing the costs for accesses after the evaluation period, the costs of making the copies should not included.

Similarly, the cost of a write depends on the number of copies. If we assume that at the end of the evaluation period all the machines that will read the object have done so, we can calculate the cost of subsequent writes by using the set of copies at the end of evaluation.

This leads to the computation of the cost:

$$C^s_{upd} = 3|E^{final} - \{m\}|W^s_m + \sum_{i \in (A-\{m\})} (2 + 3|E^{final} - \{i,m\}|)W^s_i$$

$$C_{upd} = \sum_{s \in S} C^s_{upd} = 3|E^{final} - \{m\}|W_m + \sum_{i \in (A-\{m\})} (2 + 3|E^{final} - \{i,m\}|)W_i$$

C_{upd} is minimum for a manager m such that, first $m \in E^{final}$, and second W_m is minimum. So the best manager is a machine that has read the object but has written it least. This may seem counter-intuitive, but is understandable since, for the manager, having a copy when another machine writes costs 2 messages (a request and a acknowledgement), but saves the 3 messages (a request to lock, an acknowledgement and an unlock message) which would result if the update was made by the manager itself.

The same counter W is used to hold W_i for both the retention and update protocols, and can be used whichever protocol is currently running. A set E giving the machines which have a copy is required by the current manager to run the update protocol. E implements directly the set E^{final} needed at the end of the evaluation. When running the retention or invalidation protocol, we must build the same set E so that the manager will also be able to compute the cost of the update protocol. It is done exactly as with the update protocol: Each time the current manager receives a read request, or makes a read access itself, it adds into E the identity of the reader.

At the end of the evaluation, the manager collects the values of W from the all the machines, as with the retention protocol, and computes the result.

3.1.3 Invalidation

The cost of the invalidation protocol is:

$$\begin{aligned} C^s_{inv} &= 2|N^s - \{m\}| && \text{(reads)} \\ &+ 2|N^s - \{m, w^s\}| && \text{(invalidations)} \\ &+ \sum_{i \in (A-\{m\})} 2W^s_i && \text{(writes)} \end{aligned}$$

This cost is more difficult to compute for several reasons:
- By contrast with the computation of the cost of the retention and update protocol, the order of the reads and writes from the different machines is significant.
- The order of the accesses is difficult to observe by the manager when running the update protocol because the machines keep their copy after a write, so they can read it without the manager being aware.

A transformation of the formulae leads to an alternative form which it is possible to implement:

$$\begin{aligned} C^s_{inv} &= \sum_{i \in (A-\{m\})} \text{if } i \in N^s \text{ then } 2 \text{ else } 0 && \text{(reads)} \\ &+ \sum_{i \in (A-\{m\})} \text{if } i \in N^s \text{ then } 2 \text{ else } 0 && \text{(invalidations)} \\ &- \text{if } w^s \in (N^s - \{m\}) \text{ then } 2 \text{ else } 0 \\ &+ \sum_{i \in (A-\{m\})} 2W^s_i && \text{(writes)} \end{aligned}$$

$$
\begin{aligned}
C^s_{inv} &= \sum_{i\in(A-\{m\})} \text{if } i\in N^s \text{ then } 4 \text{ else } 0 + \sum_{i\in(A-\{m\})} 2W^s_i - (\text{if } i\in N^s \text{ then } 2 \text{ else } 0)W^s_i \\
C_{inv} &= \sum_{s\in S} C^s_{inv} = \sum_{i\in(A-\{m\})} I_i \\
I_i &= \sum_{s\in S}(\text{if } i\in N^s \text{ then } 4 \text{ else } 0 + (\text{if } i\in N^s \text{ then } 0 \text{ else } 2)W^s_i)
\end{aligned}
$$

The computation of C_{inv} is based on a counter I per machine, that implements I_i. This is the cost for accesses made by that machine if it is not the manager. So the best manager will be the machine with the largest value in this counter, and the total cost of the invalidation protocol is the sum of all the other I_i.

I_i can be computed by each machine i from the following rules, derived from the above formulae and explained informally below:

When i reads: I := I + if $i \in N^s$ then 0 else 4

If i has not yet read, the cost of making a copy from the manager is 2. We include at this time an additional cost of 2 which will be incurred to invalidate this copy at the next write, unless i is itself the writer. This has to be included now because i will not be aware of a write by another machine if the retention protocol is currently running.

When i writes: I := I + if $i \in N^s$ then 0 else 2

In addition to the cost of the invalidation already counted, a write from a machine other than the manager involves a cost of 2 for a request to it and its acknowledgement. If i has read the object, the cost of the invalidation of its copy should not have been counted in the rule for read because it is done locally, so the correction is introduced. By beginning and ending evaluation with a write operation we ensure that these costs are correct.

Now, whichever protocol is used, machine i must be able to perform the test $i \in N^s$ which means "have I read the object since the last write on that object". The test is implemented differently depending on the protocol currently running. When running the retention protocol, a machine is not aware of the writes performed by the others. The current manager can maintain a set N of the machines that have read the object since the last write, as if it was running the invalidation protocol. It performs the test and sends the result together with the value of the object at each read and write request it receives. With the update protocol, the manager is not aware of the reads of the machines that already have a copy at the beginning of the step, but all machines are aware of every write performed. They can maintain a local boolean variable, called FLAG, set at each local read and reset at each update, and the value of the test is the value of FLAG. When running the invalidation protocol, the test is equivalent to testing if there is a local copy, except for the manager which must test FLAG since it always has a copy.

3.2 The evaluation cycle

An object is allocated and a default protocol is used for a short period. It then enters a period of evaluation while the access pattern is observed. At the end of the evaluation a protocol and manager are chosen for the object and used for a period of 'non-evaluation'. Some time later a new evaluation period is started, and a new protocol and manager found. This cycle repeats for the lifetime of the object.

The default protocol and the length of the periods of evaluation and non-evaluation all have an effect on the performance of the algorithm and have to be chosen carefully. We describe below the choices in detail.

3.2.1 Allocation and initialisation

There is no way of predicting which protocol will be the most appropriate for a newly created object. The best compromise appears to be Kai Li's dynamic protocol, an invalidation based protocol where the manager is always the last writer. The cost of this protocol is difficult to compute, so it is not used as an alternative to the three protocols we use, but it does allow us to compute the cost of these protocols while it is running. This can be done in a similar way as for the invalidation protocol. The only difference is that the information managed by the current manager must be passed to the new one each time the manager changes.

The evaluation is not started at the time of the allocation for two reasons:

- The evaluation period is used to predict the way the object will be accessed in the future. It is often the case that an object undergoes an initialisation phase which does not reflect the subsequent access pattern.

- Many objects may actually be local to a particular machine and it is wasteful to attempt to run an evaluation for them. We have no way in our system of discovering from the compiler or the programmer whether an object is shared by several machines or if it is local.

The solution we use is to wait until a certain number of messages have been exchanged before starting an evaluation. For local objects, the evaluation will never start.

3.2.2 Ending evaluation

The period of evaluation should be long enough so that we can distinguish clearly between the three protocols. To do that, the period must include a minimum number of write accesses. However, fixing a bound on the total number of writes in the system is difficult: in the case where every machine writes to the object, the bound would have to be larger than the number of machines in the system; but if only one machine writes to the object, that value would lead to a very long evaluation period.

The solution chosen is to have a small bound W_{bound} on the counter W of each machine. When any machine overflows its counter, it asks the manager to end the evaluation. On receipt of this message the current manager locks the object, asks all the machines for the counters they hold, and computes the cost of the three protocols, choosing the best protocol and the best manager.

If the manager must change, the current manager sends the new manager a message with the information it requires, i.e. the value of the object if it does not yet have a copy, the set of the machines that have a copy, the "name" of the protocol chosen, and some information relating to the computation of the costs as explained in the next section. The current manager sets the identity of the probable manager to the new one, and can then unlock the object. Any requests it receives from other machines will then be forwarded to the new manager. When the new manager receives that message, it locks the object and initialises its header to be the manager.

As well as a possible change of manager, there may be a change of protocol. If the new protocol is retention and there are several copies at that time, these are invalidated, otherwise the set N, if the new protocol is invalidation, or the set E if it is update, is initialised to the set of copies. The counters used for the non-evaluation period are initialised, and the object is unlocked.

3.2.3 Restarting evaluation

It would be possible to run the evaluation continuously, starting a new evaluation cycle as the previous one finishes. However, evaluation and in particular the exchange of messages at the end of evaluation, does impose an overhead so we would prefer not to run the evaluation too frequently. In addition, we expect that the evaluation period will provide us with a good prediction of the access pattern over a longer period, so that it should not be necessary to repeat the evaluation.

The evaluation period should be a good prediction for the subsequent non-evaluation, but if that turns out not to be the case, it is better to start a new evaluation to choose a new protocol and manager as soon as possible. We achieve this by monitoring the performance of the chosen protocol. The actual cost of running the protocol is compared with the cost predicted from the evaluation, and an evaluation is started if the actual cost becomes significantly higher. To do that, the manager keeps the value of the computed cost of the protocol, Ceval, together with the number of accesses, Aeval, performed during the evaluation period.

Aeval $= \sum_{i \in A} R_i + W_i$

For the retention protocol, we use Aeval $= \sum_{i \in A} W_i$ for reasons given below.

During the non-evaluation period, the manager counts the number of messages, Cactual, it receives and sends. This does not include the messages used to locate the manager if it has changed, so this value is comparable with the cost predicted.

The manager has also to count the total number of accesses (writes for retention), Aactual, performed by all the machines. The machines that have a copy of the object count locally the numbers of read accesses they perform on their copy. The manager gets these values at each write with the acknowledgement or the request it receives from all these machines whichever protocol is used, invalidation or update.

The computation of the actual cost is carried out over a period equal to the duration of evaluation, i.e. for Aeval accesses. As soon as Aactual is greater than Aeval, the manager checks if the actual cost Cactual is significantly greater than the predicted cost Ceval. If this is the case, an evaluation is started. If not, the counters Aactual and Cactual are reset and a new non-evaluation cycle is started.

This algorithm starts an evaluation when the current protocol becomes more expensive than expected. However it does not detect the case when the sequence of accesses changes so that another protocol would cheaper. To take that into account, an evaluation is started after the counters have been reset a certain number of times, even if the actual cost is lower than predicted.

The retention protocol is appropriate when a significant proportion of the accesses are writes. If the access pattern changes so that there are many more reads, the protocol should be changed. For the retention protocol the cost of an access by another machine is the same whether it is for read or for write. Simply counting the cost over a certain number of accesses provides no useful information. Instead, when running this protocol, we compute the cost for a certain number of writes. In this way we are able to detect when the number of reads from other machines has increased compared with our prediction, and to change to another protocol as necessary.

If, during the evaluation period, the only machine to have accessed an object is its current manager, the cost computed at the end of the evaluation will be zero. In this case the current protocol is continued and Aeval and Ceval are initialised so that a new evaluation is started only when there are some messages exchanged. The same initialisation is done at the allocation, as explained in 3.2.1.

3.3 Optimisations

Several optimisations are introduced in the algorithm described so far. They are briefly described below:

"False writes": For the invalidation and the update protocols the cost of a write is much higher than the cost of a read. Invalidation or update is actually useful only if the value written is different from the previous value held in the object. Testing whether the value has changed may in some cases save a large number of message exchanges. We call a write access that does not change the value of the object a "false write".

A false write could be implemented in the same way as a read access but without making a copy when the invalidation or update protocol is used. However, by implementing it in that way it is no longer possible to place a bound on the counters as described in 3.4 while keeping a minimum number of write accesses during every evaluation period.

Our solution is to perform a real read access when there is a false write, allowing us to use the existing rules for computing the cost of a read access. Compared with the other option, this may or may not be more expensive if there is no local copy. In practice it is likely to be rare for a machine to perform a write, either false or real, on an object without first reading it, so when running the invalidation or update protocols it is likely there will already be a local copy.

End of evaluation: At the end of the evaluation, the manager needs to get the information to compute the costs and to do this it must contact all the machines that have read or written to the object. The set E contains the machines that have read the object but not those that have written to it without reading. To avoid requesting all the machines in the system, it would be possible to use another set to hold the machines that have written the object but that requires extra space. Instead, during an evaluation period, the manager includes the writers in the set E. This may result in unnecessary messages being sent when running the update protocol, but this case is probably sufficiently rare for it to be acceptable.

When running the update protocol the set E contains all the machines that have read the object since the start of the *previous* evaluation. It may be that some of these machines no longer need their copies. To remove unnecessary copies, if a machine has not read the object during the *current* evaluation it discards its copy when it receives the request at the end of the evaluation period. The manager can remove it from the set E, and use the revised set E to compute the cost of the update protocol.

Location of the manager: The adapted version of Kai Li's dynamic manager algorithm we use can result in a chain of probable managers. The chain collapses to a chain of length two as soon as it traversed by a read request as shown in figure 2, by setting the probable manager to a machine that will shortly know the identity of the manager. This algorithm has a small disadvantage: The machine for which the probable manager was the real manager now has a chain of length two to find it. The length of the chain can also increase for other machines if they also read. For example, in the third diagram, machine C has a chain of length three instead of two at the beginning.

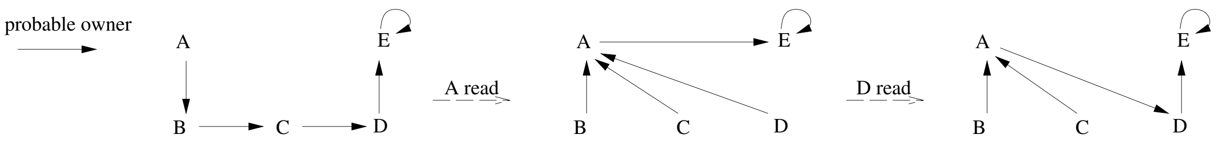

Figure 2: Location of the manager

To avoid this, whenever the manager of an object receives a read request that has been forwarded from another machine, it returns a message saying it that it is the real manager.

This additional message allows the chain of probable managers to be shortened. It may not be a good solution in Kai Li's algorithm where the manager changes at each write, but it is certainly worthwhile here because the manager changes very infrequently.

Evaluation period started with the retention protocol: The retention protocol is very inefficient if there are a large number of reads. If, while running the retention protocol, the access pattern changes so that no further writes are performed, we would like to ensure that the protocol changes to either invalidation or update. However, an evaluation period must always contain a minimum number of writes and so the algorithm will never change from retention. The solution is to switch the protocol to one which creates copies as soon as it is clear that the retention protocol is no longer a good alternative, but to continue the evaluation. The retention protocol has to be rejected as soon as one machine exceeds a bound R_{bound} on its number of read access. When that occurs, the machine sends a message to the manager which simply changes the protocol to Kai Li's dynamic algorithm, as at the allocation time.

Fast access: The code required to implement this algorithm is quite extensive. However, many objects will be purely local and a read or write can be implemented by a straightforward load or store. To avoid the overhead of the cost of the full algorithm in these cases we introduce a test which indicates whether a simple load or store can be used, or whether the more general case is required.

3.4 Size of information and values for the parameters

The information needed to handle an object is kept in a header local to each machine.

By choosing a bound on the number of machines in the system and on the maximum number of writes done by each machine during the evaluation period, it is possible to compute a maximum value for all the counters used during the evaluation and the non-evaluation period, so we can determine the number of bits required to hold these counters.

Since we only finish the evaluation when one of the counters of the number of write accesses overflows, it would appear that it is not possible to place a bound on the read counters. However, the read counters are only used in the computation of the retention protocol. This protocol should be rejected in favour of update or invalidation if there are a large number of reads. It is sufficient to choose an upper limit on the read counters which ensures that the retention protocol will not be chosen in that case.

Since we aim to minimise the size of the header we use the same space to hold the counters needed during the evaluation period and those for non-evaluation. A limit of 32 machines and a value for W_{bound} less than or equal to 6 allows us to pack all the information into 4 words of 32 bits.

For the simulation described below we use values of 4 for W_{bound} and 50 for the maximum number of periods of non evaluation between two evaluations. We choose to start the first evaluation when the number of messages exchanged exceeds twice the number of machines.

4 Simulation of the algorithm

Before considering implementing our algorithm on a real system, we have simulated it using a trace driven simulator in a Poly/ML program. This allows us to develop, improve, and test the algorithm before implementing it as part of a run-time system. The trace driven simulation is a reliable way of comparing and measuring the efficiency of different algorithms in terms of the number of messages exchanged. The example chosen to show the results of using our algorithm is an implementation of the life game. The advantage of this example is that it is easy to configure different situations within the same program, and to understand the individual behaviour of each mutable object used. It is illustrated in figure 3.

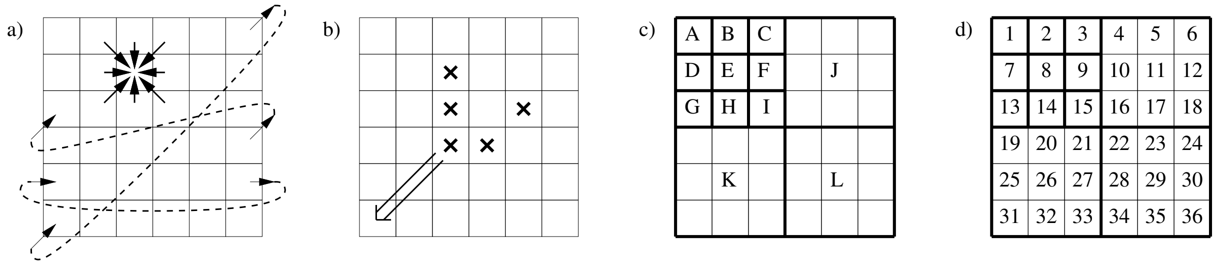

Figure 3: The life game

As shown in a) the life game is a set of cells organised in a torus of 4 dimensions, i.e. each cell is connected to the 8 adjacent cells. Each cell has a state, marked or not marked, stored in a variable. The whole system works as a sequence of steps: Every cell reads the state of the adjacent cells and computes the state for the next step. Then the state variables are updated. The classical function[1] and a suitable initialisation of the states allows the well known "glider", shown in b) to cross the torus of the cells.

This life game can be programmed as a set of co-operating processes, each managing a subset of cells, with the suitable synchronisation between each step of the game. In order to show as many situations as possible in a single example we have split the torus of the 36 cells numbered in picture d) irregularly between 12 machines, called A to L, as pictured in c). So the reference that holds the state of the cell 11 for example is accessed only by machine J, while the state of cell 8 will be read by nine different machines.

4.1 Basic algorithm

We first run our algorithm without the optimisation of testing at each write if the value has really changed. The reason for this is twofold. First, it is probably the case that many algorithms, particularly those involving floating point numbers, have very few assignments which do not change the value of the object and so this is likely to be the most accurate simulation of those cases. Secondly, it provides a comparison to show whether messages are saved when the optimisation of testing for writes which do not change the value of the object is introduced.

Three diagrams are shown in figure 4. The curves in the upper graph represent the accumulated number of messages exchanged during the simulation for the five protocols we simulated: our adaptive protocol (shown as ap) and the four basic protocols it uses including Kai Li's dynamic protocol, marked kld. The lower graph shows the number of messages per step of the life game for the ap and kld algorithms. The right hand diagram shows for the each cell the protocol chosen by our algorithm after the learning period.

At the beginning of the simulations (steps 0 to 4), the ap and the kld algorithms behave the same because the kld is the default protocol chosen. Then there is an additional cost for the ap

[1]The new state is marked if either three adjacent cells are marked or two adjacent cells plus the current state are marked.

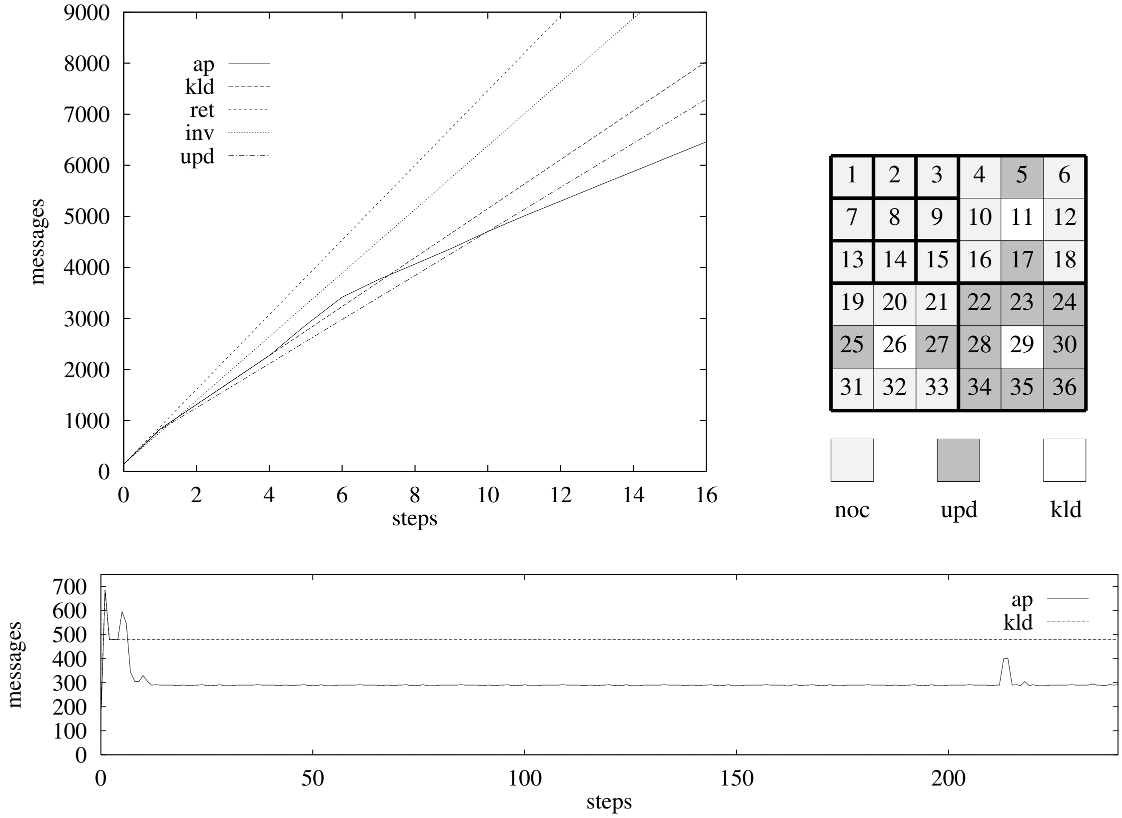

Figure 4: Simulation without test on writes

due to the end of the evaluation of the variables (steps 5 to 10). After this the ap algorithm performs better.

There are some detailed comments that can be made on this experiment:

- All the state variables shared by three or more machines are managed with the retention protocol. The state variables shared by two machines are managed with the update protocol. Cells 11, 26 and 29 are each local to one machine; the evaluation has not been started, so they are still managed with the kld algorithm.

- Although the retention protocol has been chosen for majority of the cells choosing it for all the cells would produce a very poor performance.

- Invalidation performs badly because the machines that have made a copy and then invalidate it because of a write, will always read it again before the next write, requiring a new copy.

- The update protocol performs better in our example than kld, but it should not be viewed as a better protocol in general. In particular cases, a simple protocol such as upd, or ret or inv in other examples, performs better than kld, and leads us to believe that our algorithm that is able to switch dynamically between these simple protocols will perform better than others.

- The end of evaluation is clearly shown on the lower graph by a higher cost per step from step 5 to step 10. Because different variables exchange different numbers of messages, they start their evaluations at different times, resulting in the evaluation cost being spread over a range of steps. A new evaluation is started after a time equal to fifty times the time of the previous evaluation, leading to another peak of the number of messages per step at step 210. Clearly the cost introduced by this is negligible.

After the learning period, ap saves 40% of the messages compared with kld.

4.2 With tests on the writes

In this second experiment the full algorithm described in this paper is run, including the optimisation to detect writes which do not change the value. The result is shown in figure 5 in the same way as for the previous example. In comparison with the previous simulations, two important things can seen:

- kld has one tenth of the previous cost, ap has one ninth of the previous cost.

- The protocols chosen by the ap algorithm to manage the variables are quite different.

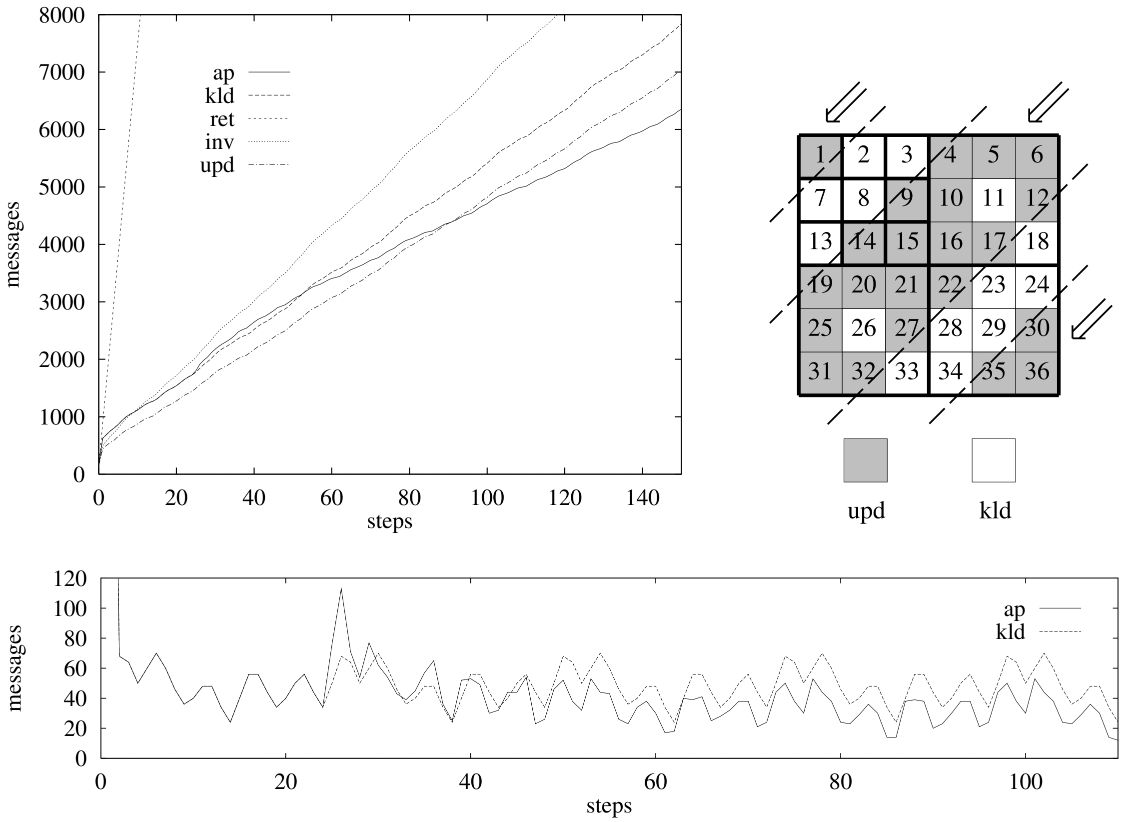

Figure 5: Simulation with a test on writes

The curves representing the accumulated cost for the different protocols are similar to the previous ones: The ap curve follows exactly the kld one till step 25, then is above it till step 46, and then is always under it.

It needs 24 steps of the life game for the glider to make a complete revolution around the torus, and the the state variables modified at each step are only local around the glider, so that any individual state variable is not modified very often. This explains why:

- The performances are much better with this run than with the previous one.

- Since the game is organised as a torus in 4 dimensions, during the time the glider is going from the left lower side of the diagonal to its right upper side, there are also modifications to the cells

1, 30, 35 and 36. All the cells modified by the passage of the glider and shared by at least two machines have been evaluated and the result is the choice of the update protocol. All the others are still running the initial protocol, i.e. kld. This is clearly shown in figure 5.

- The curve is locally much more irregular than before because the cost of a step depends on the position of the glider on the torus, but the curves for kld and ap show a pattern with a period of 24 steps.

- The first ends of evaluation appear only at step 25, because since the duration of the evaluation is based on the number of "real" writes, the evaluations of the variables last longer.

As a result, after the learning period, ap saves 33% of the messages compared with kld.

5 Related work and conclusions

Distributed shared memory provides a very interesting way to use distributed memory machines. The survey paper [NL91] introduces all the notions of granularity, coherency semantics and coherence protocols inherent to distributed shared memory systems.

Several systems have been built on the algorithms proposed by Li and Appel [LH89]: IVY [Li86], CHORUS [ARG89], KOAN [LP91], etc. A lot of work have been done in other systems to reduce the cost of the write-invalidate protocol used.

One way is to weaken the principle of strict coherency. Several models have been built, release consistency as in [CBZ91], lazy release consistency [KCZ92], entry consistency [BZS93], etc. The number of messages exchanged is reduced by delaying the propagation of the writes and merging them. They require the exclusive use of specific synchronisation primitives provided by the distributed shared memory system.

We did not consider this approach because in our system, distributed implementation of concurrent functional languages, it would be very difficult for the compiler to introduce all the necessary synchronisation points, and it is not desirable to ask the programmer to perform this task.

Another way used of reducing the cost of distributed shared memory is to use multiple coherency protocols to better match the way the objects are accessed. Munin [CBZ91] is an example. The shared variables are annotated by the programmer so that a suitable protocol can be used for each shared variable. Again, we do not want to introduce this information in the functional languages we have in mind, and it is not possible for the compiler to compute which protocol is the most suitable for each variable.

Our algorithm is also based on the idea of using several protocols to better match the way the shared variables are used. As far as we know it is the first algorithm which choose automatically the best protocol among a set of coherence protocols to handle shared objects.

The algorithm uses three basic protocols, retention, invalidation and update, each of them with a fixed manager that has always a valid copy of the object. The simplicity of these protocols allows the computation of the cost of all of them together with the identity of the best machine that would be the manager while any of these protocols is used. As a result, the algorithm changes automatically the protocol and manager used for each shared object. The fact that the manager may change is handled by a chain of probable managers as in Kai Li's dynamic algorithm [LH89].

Preliminary results obtained from simulations showed a saving of 30 to 40% of messages compared with Kai Li's dynamic algorithm.

References

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proceeding of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, April 1991.

[AMMR92] R. Ananthanarayanan, Sathis Menon, Ajay Mohindra, and U. Ramachandran. Experiences in integrating distributed shared memory with virtual memory management. *Operating Systems Review*, 26(3):4–26, July 1992.

[ARG89] Vadim Abrossimov, Marc Rozier, and Michel Gien. Virtual memory management in chorus. In *Workshop on Progress in Distributed Operating Systems and Distributed Systems Management, Berlin, LNCS, Springer Verlag*, April 1989. Chorus systèmes, CS/TR-89-30.

[BL94] Bernard Berthomieu and Thierry Le Sergent. Programming with behaviors in an ML framework: the syntax and semantics of LCS. In *Programming Languages and Systems - ESOP'94, LNCS 788*, pages 89–104, April 1994.

[BZS93] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway distribued shared memory system. In *COMPCON*, 1993.

[CBZ91] John B. Carter, John K. Bennet, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[KCZ92] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Li86] Kai Li. *Shared Virtual Memory on Loosely-coupled Multiprocessors*. PhD thesis, Yale University, October 1986.

[LKBT92] Willem G. Levelt, M. Frans Kaashoek, Henri E. Bal, and Andrew S. Tanenbaum. A comparison of two paradigms for distributed shared memory. *Software Practice and Experience*, 22(11):985–1010, November 1992.

[LP91] Zakaria Lahjomri and Thierry Priol. KOAN: a shared virtual memory for the iPSC/2 hypercube. Technical Report 597, IRISA, July 1991.

[Mat91] David C J Matthews. A distributed concurrent implementation of Standard ML. In *Proceedings of EurOpen Autumn 1991 Conference, Budapest, Hungary*, September 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[NL91] Bill Nitzberg and Virginia Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8):52–60, August 1991.