LEMMA: A Distributed Shared Memory with Global and Local Garbage Collection*

David C. J. Matthews, Thierry Le Sergent,
University of Edinburgh, LAAS-CNRS

Abstract

Standard ML is an established programming language with a well-understood semantics. Several projects have enriched the language with primitives or constructions for concurrency, among them Poly/ML and LCS. We describe first the previous run-time systems for Poly/ML and LCS and then the development of a new memory management system, LEMMA, which allows parallel programs to be run on networks of workstations. LEMMA provides a distributed shared virtual memory with global garbage collection, and in addition allows each machine to run independent local collections. LEMMA is written as a separate layer from the run-time systems of Poly/ML and LCS and is independent of the object representations and process management schemes used in the languages. We give a description of LEMMA and the results of the execution of some preliminary test programs.

Key Words: Distributed shared virtual memory, garbage collection, Standard ML.

1 Introduction

Standard ML is an established programming language with a well-understood semantics [MTH90]. Several projects have enriched the language with primitives or constructions for concurrency, primarily to provide a better way to describe parallel applications such as interactive multi-windows systems. LCS [BL94], designed at LAAS-CNRS, and Poly/ML [Mat89], being developed at the University of Edinburgh, are such languages. Our investigations concern parallel implementations of these languages in order to speed up the execution of parallel programs. The parallel machines we are interested in are networks of workstations, because they are widely available.

Poly/ML [Mat91] has been implemented on various targets including distributed architectures. The sequential and distributed implementations use the same compiler; only the lowest levels of the implementations, the run-time systems, are different. In fact, the run-time system of the distributed implementation has been designed as an extension of the sequential one. Unfortunately, this distributed implementation suffers from

*This is a preliminary version of a paper accepted for the International Workshop in Memory Management, Kinross, Scotland, September 1995 some inefficiencies, detailed below, and the data structures and algorithms it uses have become so complicated that it seems difficult to improve that implementation.

By contrast, a "real" distributed run-time system has been designed for the LCS system [Le 93]. Its implementation is only a prototype, done to help the design process. A clean implementation of the algorithms is highly desirable.

Although they are both based on Standard ML and have explicit constructions for introducing parallelism, Poly/ML and LCS differ in the syntax and semantics of their concurrency primitives. Even the technique used to implement them is different: the Poly/ML compiler produces binary code, while LCS is a byte-code interpreter. From the point of view of the run-time system, however, the essential characteristics are the same, namely:

- the systems are composed of a small number of cooperating virtual machines, typically one per physical processor, each handling possibly many application threads;
- these machines may share data structures consisting of large numbers of small objects connected as a graph;
- most of the cells built by the execution of typical programs are immutable, although mutable cells are permitted;
- the feature of a cell, mutable or immutable, can always be determined at the time of its creation;
- objects are not explicitly deallocated, thus there is a requirement for an automatic mechanism to reclaim unused space.

With the experience of the previous work done independently on Poly/ML and LCS, it seemed appropriate to combine our efforts. We set out to design and implement a single distributed software platform which would support the efficient execution of programs in both Poly/ML and LCS. We have called this LEMMA.

2 Analysis of the existing run-time systems

Although the parallelism is introduced in a different way in LCS and Poly/ML, one of the characteristics they have in common is the presence of channels which allow arbitrary data structures to be sent between processes. For processes on a single machine a communication involves nothing more than the transfer of a single word, since even if that word is a pointer to a large data structure the two processes operating in a single memory can simply share the data.

The situation is much more complicated if the processes are running on separate machines without a shared memory. In such a system communication requires the data structures to be physically copied from one machine to another. One possible implementation, used for example in the Facile implementation [GMP89], is to copy the whole data structure whenever a communication is made. This has advantages in a widely distributed system where communications can break down because it means that a machine always has a complete copy of the data it operates on, and failures are limited to the points of communication.

For our applications, where machines are more closely coupled copying the data in this way has two major disadvantages. If the structure includes mutable objects, such as references or arrays, these will not be correctly shared if communicated between processes on different machines. Thus the semantics of communication is different depending on whether the processes are on the same or different machines. This is a situation we would like to avoid. The other disadvantage is that making a copy of the whole data structure each time it is sent places an arbitrarily large delay on each transmission. Even if the process receiving the data uses only part of it, it nevertheless must be prepared to accept and store the whole structure. Although copying the whole data structure has problems, there are nevertheless advantages if some of the data structure can be copied between machines, provided the semantics of communication is not affected.

The existing implementations of Poly/ML and LCS both solve these problems by providing the illusion of a single shared memory. They differ in how they implement it.

2.1 The distributed Poly/ML run-time system

The Poly/ML run-time system makes use of two sorts of addresses, local and global. Each machine has its own local address space, in which it allocates objects. A local address is only valid within a particular machine, and when an address is to be communicated to another machine, a local address must be converted into a global address. A global address encodes within it the particular machine on which the object it refers to resides, the manager. A machine holding the global address for an object can therefore fetch it by sending a message to the manager. The system uses tables to keep track of local objects that may be referenced by another machine and builds structures by copying cells in a special area when transferring objects between machines.

Distinguishing local and global addresses has both advantages and disadvantages. On the one hand having multiple address spaces results in a large amount of copying and address translation, because the processes may share a lot of data. The management of the tables can become complicated especially if optimisations are applied.

On the other hand, the advantage of having the distinction between local and global spaces is that each machine is able to run a local garbage collector on its local space, independently from the others. A global garbage collector is, however, still needed because the local collections cannot reclaim the space occupied by local objects which were referenced by other machines but are no longer referenced.

An important difference between the management of immutable and mutable cells is that while the former can be copied without problem onto several machines, copies of the latter need a coherency protocol. The Poly/ML run-time system uses information given by the compiler to distinguish the mutable objects. They are allocated in the same way as immutable objects, but are treated in a different way when another machine accesses them. They are never copied or moved between different machines. Instead, all remote accesses are made by an exchange of messages, similar to a Remote Procedure Call.

2.2 The LCS run-time system

Compared to the Poly/ML system, the obvious advantage of LCS's run-time is its simplicity. It is based on the notion of distributed shared virtual memory. It means that an address refers always the same object, whichever machine uses it.

A convenient implementation is described by Kai Li [LH89]. The *single writer/multiple readers* paradigm is implemented by an invalidation protocol where the granularity of the coherency is a page of memory. Each page is managed by a particular machine, its manager. The role of the manager is statically distributed among the set of machines. If a machine wishes to read a page of which it does not have a copy it traps. The handler procedure of the trap requests a copy of the page from its manager. If the trap is for a write, the handler has also to invalidate all the other copies of the page.

A disadvantage of the Kai Li algorithm is that parallelism may be reduced because the invalidations involve complete pages. Two machines that wish to write different objects in the same page are forced to synchronise, a problem known as "false sharing".

The garbage collector for LCS [LB92] is based upon the [AEL88] algorithm. This two-space copying technique has many advantages, such as compacting live objects in memory which improves the management of the virtual memory. It is an incremental algorithm (the application is not suspended for the entire execution of the garbage collection) that also relies on the use of traps on access to non-coherent pages.

The trap handler of the LCS distributed run-time system supports both the [LH89] and [AEL88] algorithms. A trap on access to a page could occur either because the processor does not have a copy of the page (when the Kai Li algorithm is executed), or because the page needs to be scanned (when the handler procedure consists of scanning and updating the page), or possibly for both reasons.

Copying garbage collectors have a major problem when used on a distributed shared virtual memory scheme. In order to ensure that sharing of objects is preserved in the copied graph, whenever an object is copied the old object is overwritten with a forwarding pointer to the new copy. This results in every object in the system being written to at least once. While this does not cause problems in a single memory, when used with distributed shared virtual memory it results in a high number of page invalidations. It is possible to avoid this problem by not using normal writes to the distributed memory to store the forwarding pointers. For example, Ferreira [FS94] in the BMX system writes forwarding pointers only into local copies of objects, but requires separate tables to identify all the remote references to an object.

Halstead in the Multilisp system [Hal84] pointed out the disadvantage of statically partitioning the address space between the processors for the allocations. As soon as any one machine has exhausted its partition the whole memory must be garbage collected even if other machines have space available. For LCS a distributed algorithm is used for globally allocating pages managed by all the machines, but its cost is not negligible:

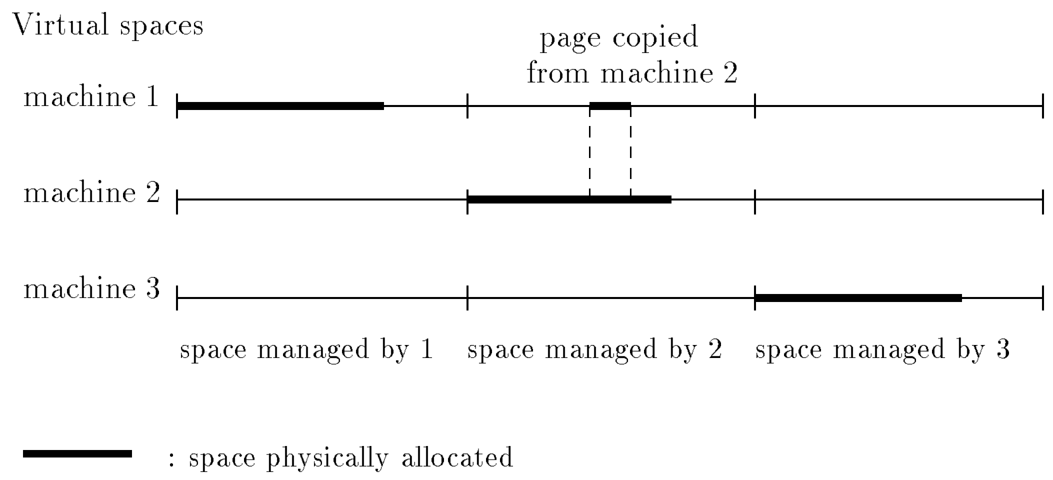

Figure 1: Global view of the memory space

to decide when to start garbage collection, messages need to be transferred along a ring involving all the machines. Furthermore, another distributed algorithm is used to allow the machines to cooperate to perform a global collection; the advantage is that machines which do not have any processes to run will garbage-collect for the others. The inconvenience is that the data are spread over the distributed memory; it is therefore not possible for the machines to perform local collections.

3 A new distributed software platform

The starting point of our work was the LCS distributed run-time system. Page invalidations, both as a result of the need of coherency for the mutable objects, and as part of the garbage-collection process, have a significant cost. We therefore made a number of changes to improve the efficiency of the system.

3.1 The memory space

The solution we adopted uses the fact that a typical workstation has a large virtual address space, but only uses a small portion of it for real memory. We can therefore statically partition the virtual address space between the machines and allow each machine to allocate independently within its portion. Provided the number of machines is not too large it is possible for each machine to decide locally when to stop allocating pages and start a garbage collection. With the advent of 64bit addressing this is unlikely to be a problem in the future.

As in Kai Li algorithm, accessing a page managed by another machine causes a trap. The handler makes a local copy of the page at the same virtual address. Figure 1 shows with bold lines the pages physically allocated by the machines.

3.2 Immutable objects

One of the significant properties of typical ML programs is that most objects are immutable. Coherency is only required between objects that can be updated, the mutable objects. It is therefore sensible to allocate mutable and immutable objects in different pages. Pages containing immutable objects are never invalidated. The machine that manages a page will always have a copy of it and can respond to requests for the page from other machines. It is simple to discover the manager of an immutable object from the address, since the address space is statically partitioned.

3.3 Mutable objects

By contrast with immutable data, mutable data requires a coherency protocol. If we pack the mutable objects together into pages, and continue to use the page invalidation scheme, the "false sharing" problem will result in serious contention for pages containing mutable objects. To avoid this we either have to allocate only one mutable object per page, resulting in much increased memory requirements or abandon the use of page protection as a mechanism to control access. We chose the latter solution, adding an explicit test whenever a mutable object is read or updated. A header is added to each mutable object to handle the information needed for the coherency protocol.

There are a number of possible protocols that can be used to ensure the coherence of mutable objects. We describe in [LM94] an algorithm which selects dynamically between three alternatives according to the access pattern on the object. The current method uses a distributed dynamic manager algorithm [LH89].

3.4 Global garbage collection

Our garbage collection algorithm is based on the well known two-space copying garbage collection. In a uniprocessor implementation, all reachable objects are copied into a new area of memory, the *to* space and subsequently scanned. Scanning a cell involves copying all the objects that the cell refers to into the *to* space for future scanning. As each address is scanned it is updated to the new location of the object. When all reachable objects have been copied, the collection is finished and the old space (*from* space) can be discarded. The role of the two spaces is reversed for the next garbage collection.

In a multiprocessor implementation, many of the problems can be avoided by using a "serialized" global collection, i.e. at most one processor at a time performs the GC. This scheme is used by [AEL88] and [DL93] with a single parallel process performing the collection. The disadvantage is that a single collector can only support a limited number of machines if each machine is allocating heavily. [DL93] measured that on average, their collector could support up to 20 machines, but should be restricted to four to ensure that the memory would never overflow.

Our garbage collector is based on the LCS distributed collector, using the same technique of protecting pages to permit incremental collection. After a global synchronisation, all the machines perform an incremental garbage collection in parallel. As with the LCS collector a page is scanned either as a result of memory allocation or because of a trap when an unscanned page is accessed. The new system differs from that of LCS in two important aspects. First, it does not use the distributed shared virtual memory to ensure consistency of the forwarding pointers, and second it introduces asynchronous local garbage collections.

The space managed by each machine is divided in two to constitute the *from* and *to* spaces. The collective *from* and *to* spaces are the concatenation of these local spaces. The task of each machine is to ensure that all the cells it can reach from its own roots have been copied into the collective *to* space. When that is done, the machine has finished its collection, but that does not mean that the entire global collection is finished. Only when all the machines have finished is the global collection complete. A simple asynchronous protocol is executed to let the machines know when they can discard the *from* space they manage.

A single machine is responsible for the forwarding pointer of each cell in order to ensure that the garbage collector copies an object exactly once. For an immutable cell this machine is the manager of the *from* space page containing that cell, i.e. the machine that created the object. For a mutable cell it must be a machine that has a valid copy of the cell; we chose to use the last machine which wrote to the cell.

3.4.1 Basic protocol

The complete protocol for the garbage collector is quite complex, but it can be described in terms of a basic protocol and some optimisations. The basic protocol is the following:

- When a machine A wishes to copy a cell managed by a machine B, it sends it a REQUEST message.

- If the cell has already been copied, B sends back the forwarding pointer, so machine A can update the cell it was scanning. Otherwise, machine B copies locally the cell, and sends back the new address.

This protocol has the property that an object is always copied by the machine that initially created it, or in the case of mutable objects, last wrote to it, even if that machine no longer has a reference to it. It is often the case that a machine will create an object and pass a pointer to it to another machine, subsequently discarding its own pointer. In that case it is preferable that the object should become managed by a machine that actually has a reference. This would avoid the need to request a copy of the page containing the copy of the object if that machine subsequently uses the object, and will reduce the number of messages which need to be exchanged during the next garbage collection. We have therefore added two optimisations which ensure that in most cases the object will be migrated.

3.4.2 Use of local copies

It is frequently the case that a machine will have a copy of a cell which it does not actually manage, if the application program has used the page. These pages will, of course, be in the *from* space. The reachable cells they contain can be copied locally, and these pages can be used to store the forwarding pointers. A protocol to ensure uniqueness of the copies is nevertheless necessary, but it is only executed the first time the cell is reached by that machine:

- When a machine A reaches a cell managed by a machine B, and A has a copy of the page containing that cell, it copies the cell locally, and sends to the manager a COPIED message containing the address of the new copy, to be stored as the forwarding pointer.

- When machine B receives that message, there are two cases to consider, depending on whether or not the cell has already been copied. This can be found by checking to see whether the old object on B contains a forwarding pointer. If the object has not previously been copied the copy made by A can be used and the forwarding pointer stored in the old copy and an acknowledgement is sent. However, if the object has already been copied B must send back to A the pointer to the "real" new copy together with the acknowledgement. The copy previously made by A is discarded.

- On receipt of the acknowledgement, machine A updates the cell it was scanning, and stores the forwarding pointer in the copy of the page it has. In this way a forwarding pointer in a local copy of an object will always be a correct copy of the forwarding pointer on the manager. If A reaches that cell again while collecting, it can use this forwarding pointer directly instead of executing the protocol.

3.4.3 Explicit migration of cells

In addition to the implicit migration described above, it is also possible for the manager of an object to force the migration of a cell to a machine which has a reference to the cell, but does not have a copy, if the manager is certain that it itself does not have a reference.

- When the manager B receives a REQUEST message for a cell that has not yet been copied, it normally copies the cell locally and sends back the forwarding pointer. However, if it has completed its collection and the cell has not already been copied, it means that B cannot reach that cell, so it is better to migrate it by explicitly sending a copy to A.

- When machine A receives the cell, it copies it locally, and sends back to the manager a COPIED message containing the address of the new copy, to be stored as the forwarding pointer, in case a similar request is received from another machine. The protocol is then the same as the one described above.

3.5 Local garbage collection

Even though all the objects are in a shared space, they are not necessarily actually shared by the machines. At the moment of allocation, only the process that has allocated the cell has access to it, so only the machine that runs that process can access it. A cell can become truly shared between machines only if a pointer to it is passed to another machine, either explicitly, or by assigning a shared mutable object.

The consequence is that it is possible to perform local collections, i.e. collection of space occupied by objects that have never been accessed by any other machine. Each machine can perform local collections independently since a local collection does not require any exchange of messages. Local collections will be executed much more often than the global garbage collection to deal with short lived objects. In this way local collections can be seen as playing the role of a minor collection in a generational garbage collection system.

A local garbage collection must not move any object referred to by a page that has been sent to another machine because a remote machine may use the original address to request the object. The scheme described in [DL93] ensures that no other machine has a pointer to a local object by copying into the global area the graph pointed to by an object when it becomes global.

In our scheme, pages of objects can be in one of three states. In addition to the purely local and global states we allow a third, intermediate, state: locked. A global page is one which has been copied to another machine. It cannot be modified by a local garbage collection so the objects in the page and the values in them are frozen until the next global collection. A locked page is one which contains at least one object whose address is in a global page. The objects in a locked page cannot be moved during a local garbage collection, but the addresses within them can be changed. They are roots for the local garbage collection. The remaining pages are purely local and objects in them can be moved and the space reclaimed.

A request for a copy of a page can only be made for a global or locked page, because only those pages contain objects whose addresses are known from other machines. When a request is received for a locked page the sending machine must scan the page and lock every local page that it points to. The locked page now becomes global. The cost of this scan is proportional to the size of the page, not the size of the graph, so the overhead is constant. As a rough measure, we found that on average each page in Poly/ML referred to six other pages.

Once a page is locked or global it cannot become local again. The global collection involves each machine in copying objects from global, locked or local pages into pages which are, initially, local. Because of the optimisations described above objects which are no longer shared between machines will become local. Pages containing shared objects must, however, be locked as part of the garbage collection protocol.

Apart from the fact that the roots for the garbage collection include the locked objects, the local garbage collection algorithm follows that of the global collection. No messages need to be exchanged with any other machine because all the objects in the *from* space are local. The local garbage collector, like the global collector, is incremental, allowing execution of the application program to be overlapped with the garbage collector.

The locked and global pages may be distributed randomly through the memory space. This makes it extremely difficult to ensure that the *from* and *to* spaces for the local collection are contiguous areas of memory. Instead of attempting to maintain contiguous areas we intersperse the *from* and *to* pages throughout the memory. Each machine keeps a table with the status of the pages of which it is the manager, including whether a page is in the *to* space or the *from* space. In effect the spaces are sets of pages.

3.6 Optimisation: Object copying

Our distributed memory and garbage collector maintain a *uniqueness* property for the memory, that is: every copy of an object is at the same address on every machine and, immediately after an object is copied by the global collection, there is exactly one copy in the *to* space. A considerable improvement in performance can be obtained by slightly weakening this constraint. This is possible because the definition of equality for immutable data in ML does not permit the user to distinguish between two copies of the same data structure at different addresses and two pointers to the same structure. It is therefore possible to duplicate immutable data within the shared memory without changing the semantics of the program.

When a pointer to a local data structure is sent from one machine to another, either explicitly or through assignment to a shared reference, the data structure is scanned and packed into a buffer. The buffer is sent as part of the message and copied into new local memory on the receiving machine. This is, in effect, an *eager* transmission of data, by contrast with the *lazy* transmission used previously. To preserve the semantics only immutable objects are transferred in this way: mutable data is handled simply by passing the address, as before. A fixed size buffer is used and if there is insufficient room for all the objects then the addresses are passed and the corresponding pages locked. In addition, objects in pages which are already global or locked are not copied into the buffer, since it is likely that those pages have been or will be shared.

The major advantage of copying data in this way is that it very much reduces the number of pages which have to be locked and thus improves the effectiveness of the local garbage collector. A case which frequently occurs is that a function on one machine creates a small data structure and passes it to another machine. The structure will be in pages which have been recently allocated and much of the other data in those pages will be short-lived objects. If the pages are locked then all those objects, and any local objects they refer to, will not be capable of being recovered by the local garbage collector. In addition, when the receiving machine reads the data structure it receives a copy of the full page, with all the other objects. Instead, by copying the data between machines only the data which may actually be used by the receiving machine is sent. The page is not locked and can be garbage collected by the local collector. This problem is to a large extent caused by the fact that we are forced by the virtual memory system to use a granularity of page which is many times larger than the size of a typical object.

This appears to be a return to the communication by copying data described in section 2 above and to some extent it is. However there are two significant differences. The first is that the maximum size of the data copied in this way is fixed, so there is an upper bound on the delay in communication. The other difference is that mutable objects are not copied so the semantics are preserved.

The possibility of making multiple copies remains a problem. If a pointer to the same data structure is sent twice then the receiving machine will have multiple copies in its local space. This could result in the space filling up. It is also possible that the receiving machine only requires part of the structure, so this eager transmission of data may result in more data being sent than is necessary.

3.7 Interface

One of the aims in building LEMMA was to be able to support both Poly/ML and LCS, and possibly other languages of the ML-type. To this end we were careful to specify an interface which would separate out those issues, such as storage allocation, which were common to both languages, from inter-process communication and scheduling, which differ. In addition, to preserve backwards compatibility it was necessary to cater for the different ways Poly/ML and LCS encoded addresses and objects.

All the memory management and low-level communications between the machines are handled by LEMMA. That leaves a much simpler language-specific run-time system which deals with process scheduling and the details of object representation. The interface between LEMMA and this residual run-time system is quite simple. It must provide LEMMA with information about the objects it creates. Basically the distributed platform needs to know at least:

- the size of a cell given its address;
- if a cell contains constants or addresses and where;
- the roots of all the cells accessible by the machine (registers, stacks).

In return, LEMMA provides functions to:

- allocate space for immutable objects, and for mutable objects;
- handle the traps when accessing immutable objects;
- read and write mutable objects;
- transfer small messages between machines on behalf of the process schedulers.

Other functions are provided by LEMMA, mainly to improve efficiency of the whole system. For example, requests between machines (for mutables or for pages of memory) can be synchronous, or asynchronous. In the latter case, LEMMA does not block until the reply is received, but instead returns to the caller with an appropriate result code. The process scheduler can then schedule another ML process until the answer is received.

Servers	Elapsed Time (secs)	GCs	Messages
1	153	15, 1	23476
2	94	7, 7, 1	27846
3	69	4, 4, 6, 1	32873

Figure 2: Ray-Tracing Example

This can mean that machines are blocked waiting for answers much less often compared with traditional DSVM systems.

A technical report describing the interface in detail is available [ML95].

3.8 Implementation

The parallel machine we are using is a network of workstations running the system UNIX. UNIX provides us with the facilities we need: functions to protect and unprotect pages of the memory, handlers for the traps, allocation of memory at any address we want in the virtual space.

Because of the complexity of the whole project, we implemented and tested the distributed platform gradually. We have now reached a state where the Poly/ML language is entirely supported. LCS is in the process of being ported.

As an example to test the system and to measure the speed-ups possible, we used a simple functional ray-tracing program in Poly/ML which used a task-farm to distribute the work. The scene was created on a client machine which sent out the tasks to server processes running on other machines, and then collected the results.

Figure 2 shows the timings that were obtained, the numbers of garbage-collections performed on each machine and the number of messages exchanged. The message numbers include messages used internally by LEMMA as well as those transmitted on behalf of the application. The figures for the garbage-collections give first the number of collections performed by the client followed by the figures for each of the servers. By comparison, running the whole problem on one machine in a single space took 187 seconds and required 42 garbage collections. There was therefore an effective speed-up of 22% simply by splitting the problem between the client and one server.

3.9 Further work and concluding remarks

The purpose of LEMMA is to support ML-like languages on local networks of workstations. This has implications on the way we implement consistency: to maintain the semantics we must use strong coherence. It also means that to maintain some degree of independence of the specific language we are able to use only one significant piece of information from the application about the way it intends to use any object: namely whether or not the object is mutable. In both these ways LEMMA is distinguished from other work on garbage collection and distributed shared memory, most notably that of Shapiro and Ferreira [FS94] who are interested primarily in object-based languages on loosely-coupled networks.

The system is working and gives useful speed-ups on a number of test programs. Nevertheless, there is considerable work to be done in a number of areas. For example, there is the question of what to do when the current memory allocated to LEMMA on a particular machine is exhausted. The machine can start a local or global garbage collection, it can discard immutable pages read from another machine or it can increase the space which is available. The choice is by no means obvious.

Another possible area of research is to look at the way LEMMA interacts with other UNIX processes on the same machine. A very useful application area would be to allow LEMMA servers to run on workstations so as to absorb spare cycles when the machine is not heavily used. The process should be able to adapt the memory available depending on the requirements of other processes. This is related to another area of research, that of process migration.

References

[AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In *ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 11–20, June 1988.

[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

[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.

[DL93] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proc. of the 20th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 113–123, Charleston SC (USA), January 1993.

[FS94] Paulo Ferreira and Marc Shapiro. Garbage Collection and DSM Consistency. *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey, California, USA, November 1994.

[GMP89] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, pages 121–160, 1989.

[Hal84] Robert H. Halstead Jr. Implementation of multilisp : Lisp on a multiprocessor. In *1984 ACM Symposium on LISP and Functional Programming*, pages 9–17, August 1984.

[LB92] Thierry Le Sergent and Bernard Berthomieu. Incremental Multi-threaded Garbage Collection on Virtually Shared Memory Architectures In *Memory Management - IWMM'92*, LNCS 637, pages 179–199, September 1992.

[Le 93] Thierry Le Sergent. *Méthodes d'exécution, et machines virtuelles parallèles pour l'implantation distribuée du langage de programmation parallèle LCS.* Thèse de doctorat de l'Université Paul Sabatier, Toulouse, Février 1993.

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

[LM94] Thierry Le Sergent and David C. J. Matthews. Adaptive selection of protocols for strict coherency in distributed shared memory. Report ECS-LFCS-94-306, LFCS, September 1994.

[Mat89] David C. J. Matthews. Papers on Poly/ML. Technical Report 161, Computer Laboratory, University of Cambridge, 1989

[Mat91] David C. J Matthews. A distributed concurrent implementation of Standard ML. In *Proceedings of EurOpen Autumn 1991 Conference, Budapest, Hungary*, September 1991. Also in LFCS Report Series ECS-LFCS-91-174.

[ML95] David C. J. Matthews and Thierry Le Sergent. LEMMA Interface Definition. Report ECS-LFCS-95-316, LFCS, January 1995.

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