Execution Analysis of DSM Applications: A Distributed and Scalable Approach

Lionel Brunie, Laurent Lefèvre and Olivier Reymann
Laboratoire de l'Informatique du Parallélisme
Ecole Normale Supérieure de Lyon
69364 LYON Cedex 07 , France
(lbrunie, llefevre, oreymann)@lip.ens-lyon.fr

Abstract

In the last five years, Distributed Shared Memory (DSM) systems have received increasing attention. Indeed, by releasing the programmer from the management of inter-process communications, they offer a very intuitive and easy-to-use programming paradigm. In compensation, such systems often appear, from the programmer point of view, as a "black box" since no information about the actual communications is available. Consequently, in the absence of visualization and monitoring tools, optimizing, debugging or evaluating the performance of DSM applications is very difficult. In that framework, this paper proposes an original monitoring model based on two new concepts: *meta-objects* and *event manager processes*. This model constitutes the basis of an actual monitoring system, called DOSMOS-Trace, that has been designed and implemented to monitor applications developed on top of the DOSMOS DSM system[1]. This monitoring environment is analysed in terms of functionalities, protocols and user interface. Experiments show the efficiency and the robustness of the underlying model as well as the pertinence, for the programmer, of such a monitoring tool.

Keywords: distributed shared memory, monitoring, performance evaluation, program visualization.

1 Introduction

In the last five years, Distributed Shared Memory (DSM) systems have received increasing attention. Indeed, by releasing the programmer from the management of inter-process communications, they offer a very intuitive and easy-to-use programming paradigm. In compensation, such systems often appear, from the programmer point of view, as a "black box" since no information about the actual communications is available. Consequently, in the absence of visualization and monitoring tools, optimizing, debugging or evaluating the performance of DSM applications is very difficult.

Until now, most of monitoring tools have been designed for message passing applications. However, in spite of evident points of convergence, monitoring DSM applications differs from monitoring message-passing applications because pertinent information to be traced is clearly different. In that framework, this paper proposes an original monitoring model based on two new concepts: *meta-objects* (*i.e.* specific distributed data-structures designed for the storage of monitoring data) and *event manager processes* (*i.e.* specialized distributed processes in charge of the on-the-fly collection and management of execution traces).

This paper is organized as follows. In section 2, we recall some basic points about DSM systems and analyse the functionalities that should be provided by DSM-oriented monitoring tools. Previous works on parallel monitoring are studied in section 3. Then, the DOSMOS system is briefly presented in section 4. The basics of the monitoring model we propose are described in section 5 while protocols are analysed in section 6. Section 7 presents the DOSMOS-Trace monitoring environment which implements the concepts introduced in this paper. An analysis of the intrusion generated by the monitoring is proposed in section 8. Section 9 discusses the pertinence and the effectiveness of the model presented in this paper. Finally, section 10 concludes this paper and analyses the main perspectives of this work.

2 Monitoring DSM applications

2.1 What is a DSM System?

Basically, Distributed Shared Memory systems (DSM) allow, above a distributed memory architecture, the manipulation of shared data in a transparent way. In other words, in such systems, a programmer can make the processes of his application share data without explicitly programming the inter-process communications which are actually handled by the system.

Two basic approaches have been studied:

virtual sharing of memory pages: the environments of this type [Li88, LP92, FP89, CBZ91, HS92] merge various memory pages distributed in the system into a single address space.

virtual sharing of variables or objects: such systems [RAK89, TKB92, CG89, BL94, BL96], more programming oriented, allow the user to define *shared variables* (or *shared objects* for object-oriented systems) which

[1] DOSMOS is the acronym of Distributed Objects Shared MemOry System. The DOSMOS system has been developed on top of PVM in our laboratory.

will be accessible in a transparent way from any node in the network.

The purpose of this paper is not to discuss the respective advantages and drawbacks of these two kinds of systems. Indeed, from the programmer point of view (and thus for the monitoring point of view), all these systems implement the same basic functionalities, *i.e.* transparent manipulation of shared data.

2.2 Monitoring DSM applications

The goal of any monitoring tool is to collect information about the execution of the application (called execution traces) and to display it in a pertinent way in order to allow the programmer to understand the behaviour of his application. In the framework of DSM applications, execution traces can be grouped in two classes:

Information about the DSM system administration: creation, destruction, migration of processes and, to speak more generally, of system "entities" (*e.g.* group of processes for the DOSMOS system (cf. section 4)).

Information about the shared data: duplication of shared data (evolution of the number of copies distributed in the system), migration, number and types of accesses (*e.g.* read-only, exclusive write, concurrent write, ...), a list, for each shared datum, of the processes that frequently modify it, *etc.*

Analysing such traces will allow (section 7) the programmer to detect and correct most critical situations, *e.g.*:

bottlenecks: a bottleneck occurs when a shared datum is too frequently accessed in an exclusive way. A possible solution consists in splitting the shared datum into several sub-variables in order to distribute the accesses over several objects;

ping-pong effects: this occurs when a variable is, for a long while, concurrently accessed by two, or more, processes. Possible solution: splitting of the variable;

no-sharing: when a variable is declared as shared but only one process actually accesses it. Solution: declaration of the variable as a local variable in order to circumvent the DSM system layer.

specific features: for instance, in the DOSMOS system, a bad group structure.

The purpose of this paper is twofold: first, proposing and studying a software architecture able to efficiently implement such monitoring functionalities (section 5); second, illustrating the relevance of this model by analysing the facilities provided by the DOSMOS-Trace environment which has been designed and implemented according to these principles (section 7).

3 Previous Works

Basically, monitoring an application requires dealing with three main problems: first, the collection of execution traces; second, the management and storage of the traces; finally, the analysis and visualization of the traces. Until now, most monitoring tools have been designed to trace message-passing applications. In that framework, the trace collection is usually performed by instrumenting the application code in order to monitor the most important events occurring during the execution. The instrumentation can be placed at various levels:

- operating system level (*e.g.* Tapestry [MR90]): tracing of events like communications, creation of processes, memory accesses, system calls;
- run-time environment level: (*e.g.* IPS [MY87]): entries into and exits from parallel sessions, use of barriers, procedure calls;
- application level (*e.g.* IVE [FLK+91] or PVVT [Str90]): working at this level allows focusing the monitoring on the most interesting parts of the source code. The systems of this type are all based on the same principle: inserting additional code into the user program in order, as previously, to trace the most important events.

Once the traces are collected, it is necessary to manage them, to organize them in the memory. On the other hand, post-mortem analysis requires the storage of traces on disk. Though a few monitoring tools (*e.g.* SIEVE [SG92] or The Belvedere system [HC87]) implement database techniques, most monitoring systems use "ordinary" files. Recently, in order to increase the scalability and the efficiency of monitoring tools, some works (*e.g.* PIMSY [TV94a, TV94b]) have proposed the use of distributed trace files. The DOSMOS-Trace system lies within this approach.

Last point: the visualization of traces. Basically, all systems allow the visualization of inter-processes communications [KS93]. Furthermore, each system proposes its own specific features: hierarchical visualization (*i.e.* grouping of processes), performance analysis, traffic analysis, processors activity,...

DSM-oriented monitoring tools are not over-abundant. Most tools have been designed for virtual shared memory systems (*i.e.* page-based systems (section 2.1)). In that framework, attention has mainly been focused on *system* events: accesses to pages[Ede93], false sharing, cache misses. However, for the basic programmer, such information is very difficult to use (*e.g.*, what measure is to be taken if a page is over-used ?). In other words, such monitoring systems provide pertinent data for DSM system designers, but not for end-users.

Nevertheless, some works have proposed focusing on higher-level features. Thus, Brorsson and Strenstrom [BS92], propose an analysis tool based on the study of four parameters: the spatial granularity, the degree of sharing, the access mode and the temporal granularity. These parameters allow the comparison of various coherence strategies in order to choose the most pertinent one (for the target application...). SHMAP [DBKF90] provides a visualization of memory access patterns, cache strategies and processor assignment. In a different context, designed to trace applications developed on top of shared-memory multiprocessors, Robinson, David and Enbody [RCE92] propose the observation of causal dependencies between events. In the same framework, [LMCF90] implements a specialized library in which every access to a shared data is assigned a logical sequence number used to infer a partial ordering of the execution. MTOOL [GH91]) tries to isolate memory bottlenecks by comparing the actual execution with an ideal execution performed on a perfect shared memory machine.

However, most of these approaches suffer from important drawbacks. First, most of them provide information hardly

usable by the end-user. Second, tools designed to monitor shared-memory multiprocessors use specific features provided by these machines and so are not portable. Thus, like most message-passing monitoring tools, MPTrace [EKKL90] instruments the application code. However, this is not realistic for software DSM systems. Indeed, no internal information is available at the application level, *e.g.* it is impossible, from the application, to know where a shared variable is located, which process manages it, etc. So, working at the application level would require a complete modification of the interface between the DSM system and the applications, which would deeply affect the actual behaviour of the applications. Finally, most existing systems (and specifically DSM-oriented systems) are intrusive (*e.g.* systems instrumenting the DSM management routines) and not scalable (*e.g.* systems using a centralized trace file).

4 The DOSMOS System

The DOSMOS-Trace monitoring environment (section 5) has been designed to monitor applications programmed on top of the DOSMOS DSM system developed in our laboratory. The purpose of this paper is not to study this DSM system (see *e.g.* [BL94, BL96]). However, to better understand the functionalities provided by the DOSMOS-Trace environment, it may be necessary to say a few words about the DOSMOS system.

DOSMOS is a variable-oriented DSM system (section 2.1) developed on top of PVM. The user can declare either basic type variables (*e.g.* integers, floats,...) or arrays that will be split into several "system objects". Various splittings are provided: by row, by column and by block. Basically, a DOSMOS application is composed of two kinds of processes:

Application Processes (AP) which execute the user's code;

Memory Processes (MP) which manage the shared variables and handle the access requests issued by Application Processes.

To avoid expensive synchronizations and useless communications which break down the efficiency of the applications, DOSMOS allows grouping together the processes which actually share a common set of variables. These groups can be hierarchically structured into groups and subgroups. Furthermore, to maintain the coherence of the shared data, the DOSMOS system implements a weak consistency protocol called *release consistency*. This protocol is based on two primitives: *acquire* which allows obtaining an exclusive access to a variable and *release* which unlocks this variable (for a complete discussion on consistency protocols, see [RM93]).

Though Application Processes and variables can be structured in groups, any shared variable is accessible from any Application Process. Link Processes (LP) are specialized MP devoted to inter-group operations on shared variables. Thus, thanks to LPs, an Application Process can access variables managed by other groups than its own. However such inter-groups accesses are, of course, more expensive than intra-group accesses. Figure 1 shows an example of software configuration including 6 APs, 3 MPs and 2 LPs.

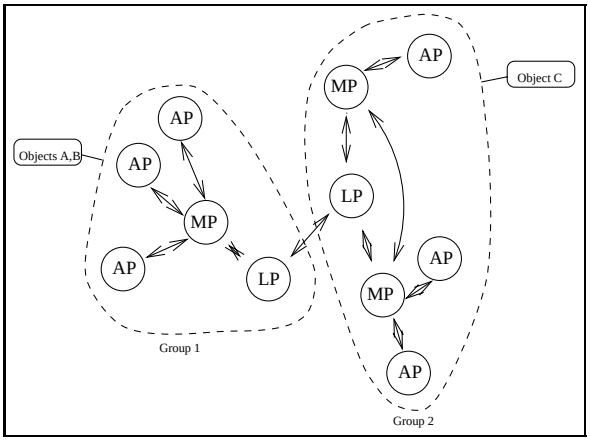

Figure 1: DOSMOS system: an example of software configuration with two groups and three objects A, B and C

5 A Model for DSM Application Monitoring

5.1 Trace Detection and Collection: Event Manager Process

As previously discussed, instrumenting the user's code is not realistic. Consequently two approaches are possible: first, modifying the DSM code, *i.e.*, in the DOSMOS context, modifying the code of the memory processes (MP). This approach presents important drawbacks: as in any DSM system, the whole efficiency of DOSMOS relies on the memory processes. Thus, loading MPs with the monitoring would deeply affect the behaviour of the system. So, in order to minimize the monitoring intrusion, we propose to introduce a new kind of system processes called *Event Manager Processes* (EMP). An EMP is linked to one or more memory processes. Once an MP detects an event, it sends a message to the EMP it depends on. EMPs are in charge of the whole management of the execution traces. Protocols defining the coordination procedures between memory processes and EMPs are described in section 6. This approach presents important advantages. First, it minimizes the work requested from MPs, and consequently, the intrusion due to monitoring[2]. Furthermore, in the case of post-mortem utilization, traces have to be stored on disk. However, as traces are managed by EMPs, which are distributed in the whole network, the storing on disk is not performed by a single process but by all the EMPs, which is clearly more scalable and efficient. The scalability of the system can even be increased if several distributed trace files, located on several disks, are used. In fact, the best (but the most expensive...) solution is to attach a local disk to each processor on which an EMP runs. Indeed, by increasing the I/O bandwidth, such an architecture allows reducing the bottleneck constituted by the transfer of traces to disk. Figure 2 shows an example of the monitoring environment. This configuration uses three processors, two logical groups and two shared variables A and B.

Finally, experimentally it appears that tracing realistic applications generates a huge amount of traces which affects the intrusion. That is why, to reduce the volume of traces, the DOSMOS-Trace system allows the user to specify which information he is interested in.

[2] This intrusion will be even reduced if EMPs are located on dedicated processors (in order not to "steal" CPU time from memory processes).

5.2 Trace Management: the Meta-Object Concept

In contrast to post-mortem analysis, on-line monitoring tools require keeping execution traces in memory. To manage these traces, we propose to introduce new data structures, called *meta-objects*. A meta-object is a tuple (record) with as many fields as different monitoring informations.

However, in the DOSMOS system, for efficiency purposes, a variable can be duplicated, *i.e.* several read-only copies of a variable can be distributed (within the group of processors sharing the variable). Therefore, it is necessary to distinguish between two types of meta-objects:

a *primary* meta-object is attached to each shared variable. It contains information about the variable such as the number and the type (read, write, acquire) of the accesses performed on the variable. It also maintains the list of the processes that recently accessed the variable, the origin (local, intra-group, inter-group) and the characteristics of the accesses they requested (*i.e.* type (read/write/acquire)). This information is very useful to analyse the behaviour of the application and to propose optimizations. The primary meta-object of a variable is managed by the EMP monitoring the MP owner of the variable.

***secondary* meta-objects** are attached to each copy of a shared variable. Because a copy can only be accessed in a read-only fashion, a secondary meta-object does not have to store as much information as a primary meta-object does. In practice, secondary meta-objects record the identification of the MP that owns the copy, the identification of the EMP that manages the primary meta-object and the number of read operations performed on this copy. A secondary meta-object is managed by the EMP attached to the MP owner of the copy.

Secondary meta-objects allow the user to know the actual distribution of the read accesses among the processes. This information is important because it deals with the group structure of the application and with the efficiency of the implemented consistency protocol. Indeed, to be efficient, DSM applications should perform as many local accesses as possible (because remote accesses are more expensive).

Remark: write accesses require bringing invalidation protocols into play. These protocols are triggered by the Memory Process owner of the variable. This Memory Process is connected with the EMP which manages the primary meta-object. Consequently, all the write accesses are traced in this primary meta-object.

5.3 Analysis and Visualization of Execution Traces

Whether they work in an on-line or post-mortem fashion, analysis tools must interact with EMPs which are the only processes able to access monitoring data. This argues for implementing a client-server architecture in which EMPs act as servers and tools as clients.

The DOSMOS-Trace system implements the following approach (figure 3): a Visualization Process (VP) is started at the beginning of the execution (on-line monitoring) or after the execution (post-mortem analysis). The user submits queries to this process which passes them to all the EMPs concerned. These latter return the requested information to the VP which is in charge of the fusion of these data. Finally, the VP displays the results.

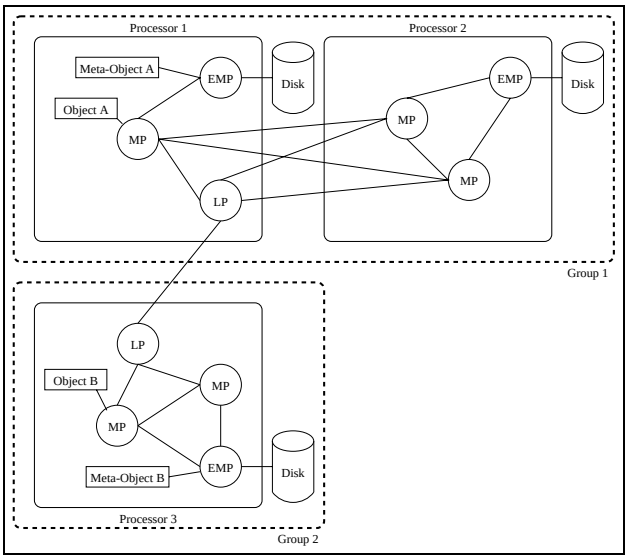

Figure 2: DOSMOS-Trace: example of monitoring environment

6 Implementation and System Architecture

6.1 Meta-Objects

As described in section 5.2, meta-objects are designed to store and manage the traced information in memory. However, as several copies of the same variable can be distributed in the network, several kinds of meta-objects must be distinguished.

Thus, in the DOSMOS-Trace system, a *primary meta-object* is associated with the main copy of a variable[3]. This meta-object contains general information such as:

- Variable identification (name, system identication)
- Group: this field contains the identification of the group the shared variable belongs to. It is used to analyse and visualize the group structure.
- Number of copies of the variable distributed in the system
- Memory process owner of the variable
- Number of read operations performed on this main copy
- Total number of read accesses performed on all the copies (see below)
- Number of write accesses
- Number of acquire and release operations
- List of last acquire operations
- List of last write operations
- List of delayed acquire operations

[3] In the DOSMOS system, a shared variable is managed by one Memory Process (Distributed Static Owner protocol ([LH89])). This Memory Process controls the duplication of the shared variable, handles the copies invalidations and manages the write accesses.

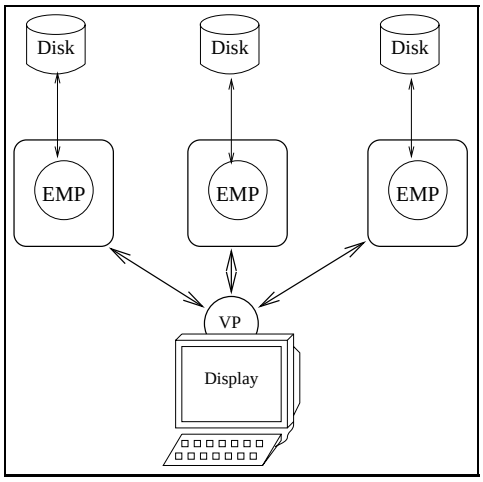

Figure 3: DOSMOS-Trace: the Visualization Process (VP) communicates with all the Event Manager Processes distributed in the network.

These last three lists store triplets containing the identifier of the Application Process from which the operation was issued, the identifier of the Memory Process that received this query and the group the Application Process belongs to.

Variable copies are monitored using *secondary meta-objects*. A secondary meta-object is attached to each copy of a variable. It contains the following information:

- Variable identification
- Memory process owner of this copy[4]
- EMP which manages the primary meta-object
- Number of read operations performed on the copy

Secondary meta-objects permit a very accurate view on the execution. More precisely, secondary meta-objects allow knowing the actual distribution of the read accesses among the processes. This information is important because it concerns the group structure of the application. Indeed, the efficiency of DSM applications is largely determined by the ratio of the number of local accesses to the number of remote accesses[5]. So, analyzing the read access distribution is extremely important to understanding the behavior of DSM applications well.

Moreover, using secondary meta-objects allows the EMP managing the primary meta-object to be discharged from the management of the traces generated by the copies. As a consequence, it increases the scalability of the monitoring system (both from a CPU point of view and an I/O point of view (if, of course, EMPs use several disks)).

6.2 System Architecture

Figure 2 shows an example of process configuration during a monitored execution. This architecture follows a few rules:

- One Event Manager Process at most can be run on one processor;
- Each EMP must be connected to at least one Memory Process;
- A Memory Process sends its trace information only to its dedicated Event Manager Process;
- A Memory Process must deal with at least (and eventually more than) one Application Process;
- An Application Process communicates with only one Memory Process.

Event Manager Processes can switch between two modes. During execution, EMPs receive messages from the memory processes concerning the various operations performed on the shared objects. They store these traces in memory (meta-objects) and/or on disk (trace files). At the end of the execution, EMPs remain alive in order to answer to the queries issued by the user.

6.3 Protocols

This section describes the protocols implemented for passing traces information from MPs to EMPs.

During a variable access, two kinds of memory processes must be distinguished:

Primary Memory Process (PMP): the Memory Process that owns the requested variable;

Secondary Memory Process (SMP): any Memory Process that received an access request from one of its Application Processes but does not own the requested variable. It possibly has one copy of that variable.

The management of shared memory is based on four standard operations: write access, read access, acquire and release.

6.3.1 Write Operation Protocol

The protocol used for a write access is the simplest one. Two cases are possible:

Local write access: (figure 4.a) The PMP directly receives the AP write request (1) ; it modifies the variable and informs its EMP (2) in order to store the operation on disk and update the meta-object.

Remote write access: (figure 4.b) The SMP receives the AP write request (1), informs its EMP (2) to store the operation on disk and forwards the request to the PMP of the variable (3) which performs the write access. Then the PMP sends a message to its EMP (4) to update the variable's primary meta-object.

6.3.2 Read Operation Protocol

Local read access: (figure 5.a) The MP receiving the AP read request (1). It owns either the variable itself or a valid copy of this variable. It then returns the requested value to the AP (2) and informs its EMP (3) to store the operation on disk and update the meta-object.

Remote read access: (figure 5.b) In this case, the MP receiving the AP read request (1) does not own a valid copy of the variable. This request is then forwarded to the variable PMP (2) which returns the value of the variable (3) and sends a message to its EMP (4) to

[4] This information is mandatory because an EMP can monitor several Memory Processes (see section 6.2).

[5] Remote accesses are much more expensive.

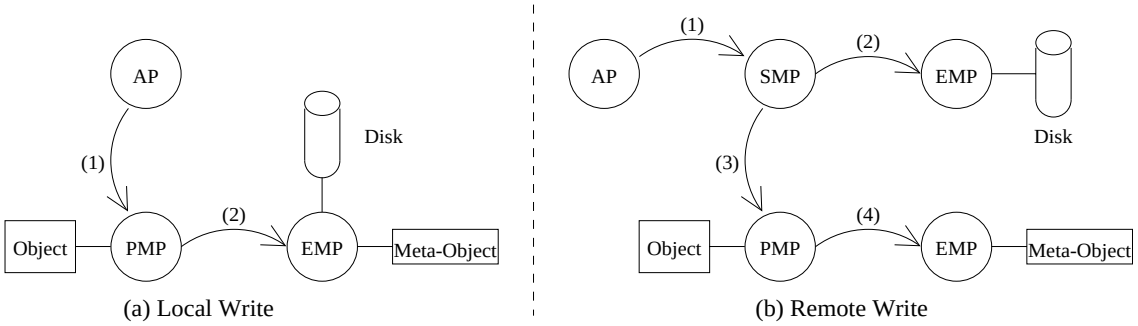

Figure 4: Protocol implemented to collect the trace information about a write operation.

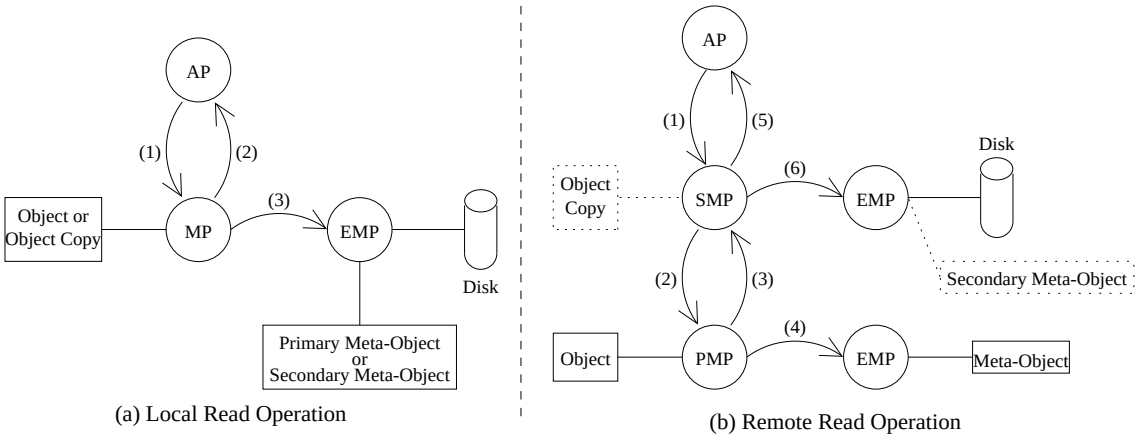

Figure 5: Protocol implemented to collect the trace information about a read operation.

update the meta-object attached to it. The secondary MP forwards the value to the calling AP (5), creates a variable copy and notifies its EMP to store the operation on disk and to generate a secondary meta-object.

6.3.3 Acquire Operation Protocol

To obtain an exclusive write access right on a variable (figure 6), an AP must generate an Acquire request message and sends it to its MP (1). Two cases must be distinguished:

This MP is the PMP of the variable: (figure 6.a) If the variable is free (*i.e.* not acquired by another AP), it gives the exclusive write access right to the AP (2) and sends information to its EMP (3) in order to store the operation on disk and update the meta-object. If the variable is already acquired by another AP, the MP informs the EMP that a new AP is waiting for the variable (3). When it is released, the PMP gives the exclusive write access right to the AP (2) and reports it to its EMP (3).

This MP is not the PMP of the variable: (figure 6.b) The SMP forwards the request to the PMP of the variable (2). This latter verifies if the variable is free. In this case, it returns the exclusive write access write to the SMP (3) which forwards it to the AP (5). The EMP attached to the PMP updates the primary meta-object (4) while the EMP attached to the SMP stores the operation on disk (6). If the variable was already acquired, the PMP informs its EMP that a new AP is waiting for the variable (4). When it is released, the same action sequence is performed as in the case where the variable was immediately available.

6.3.4 Release Operation Protocol

The management of a Release operation requires a lot of communications. Indeed, we must guarantee the consistency of all the object copies but also update the primary meta-object by sending to it all the data contained in the secondary meta-objects. This generates additional communications between EMPs.

Figure 7 shows a diagram of the protocol used by a Release operation in the most general case, *i.e.* when the release request is sent by an AP to a SMP (1). This latter forwards the request to the variable's PMP (2) which performs either an invalidation or an update of all the copies distributed in the system (3). Each SMP that has a copy informs its EMP (4) that it must send the data stored into the secondary meta-object to the EMP attached to the PMP in order to update the primary meta-object (5). When the PMP receives all acknowledgement messages issued by the SMPs (6), it updates the primary meta-object (7) and requests the SMP linked to the calling AP to inform this AP (8 and 9) and the EMP attached to this SMP (10) that the release operation is finished.

7 Interface and Experiments

Designed to trace applications developed on top of DOSMOS, the DOSMOS-Trace system actually implements all

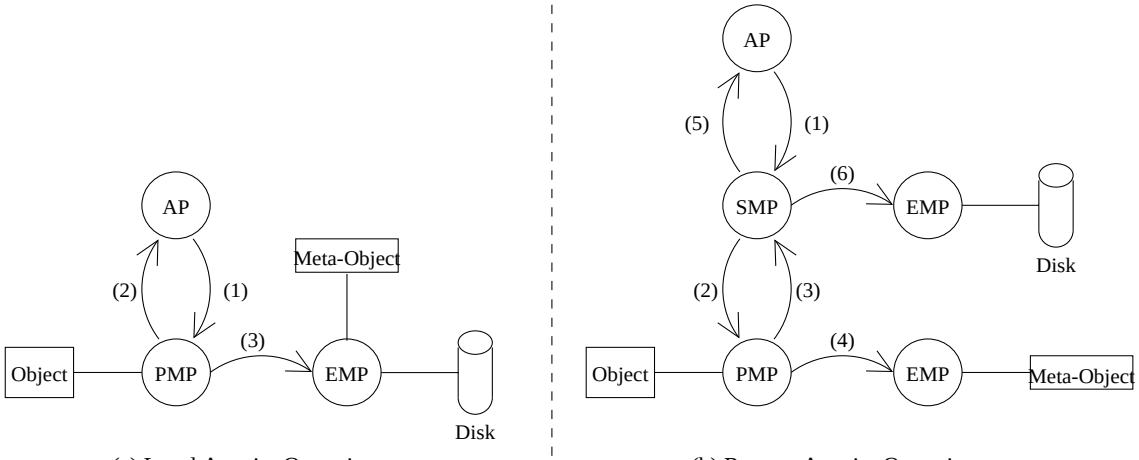

Figure 6: Protocol implemented to collect the trace information about an Acquire operation.

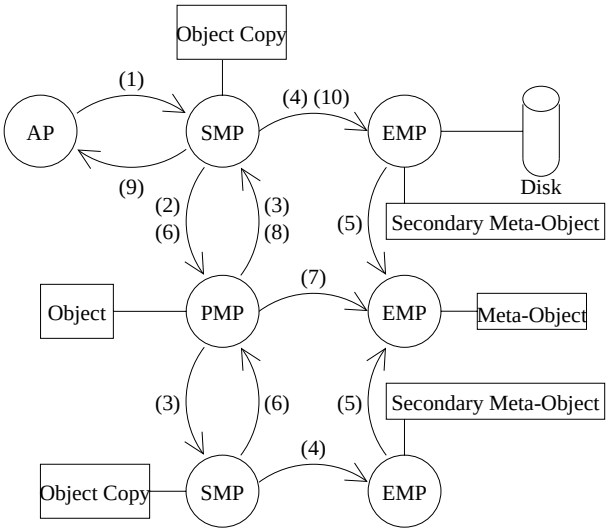

Figure 7: Protocol implemented to collect the trace information about a Release operation.

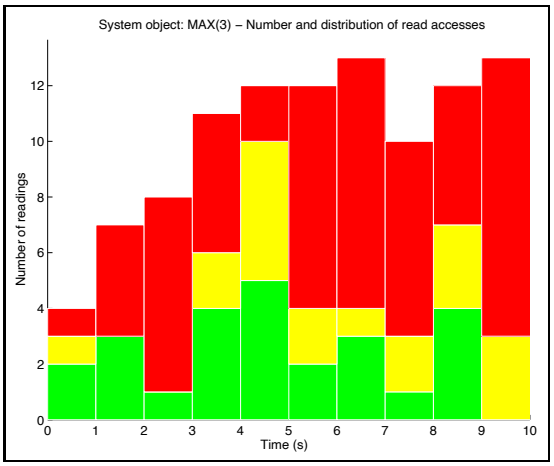

Figure 8: Number and origin of the read accesses performed on an object vs execution time (in black: inter-group accesses)

the concepts developed in the above sections: EMPs, meta-objects, distributed traces files, visualization process. The aim of this section is to illustrate its functionalities by presenting two examples of information provided by this system[6].

7.1 Accesses to shared Variables

Figures 8 and 9 display the histogram of the read accesses performed on a variable during the execution of an application. Various colors[7] are used in order to differentiate the origin of the accesses: local accesses are represented in green, intra-group accesses in yellow and inter-group accesses in red.

Such diagrams allow the user to detect a bad group structure. Thus, in figure 8, the predominance of inter-group accesses shows clearly that the group structure is not pertinent.On the contrary, in figure 9, one can verify that no inter-group accesses are performed.

In the same way, it is possible to visualize write and acquire accesses.

7.2 Histories

This functionality provides an analysis of the "history" of any shared variable (figure 10) or any application process (figure 11). In other words, it allows the visualization of all the accesses performed on a variable or, reciprocally, all the accesses performed by an application process.

On these figures, a "*" represents a write operation (under the dotted line), an "x" symbolizes a read access (above the dotted line) and a green "+" represents an optimized read access (*i.e.* a read access performed on a local copy of the variable). Black boxes are used to represent the amount of time that a process was waiting before it can perform either an acquire or a release.

[6] Figures are displayed using Matlab.

[7] Grey level correspondence: green=dark grey, yellow=light grey, red=black

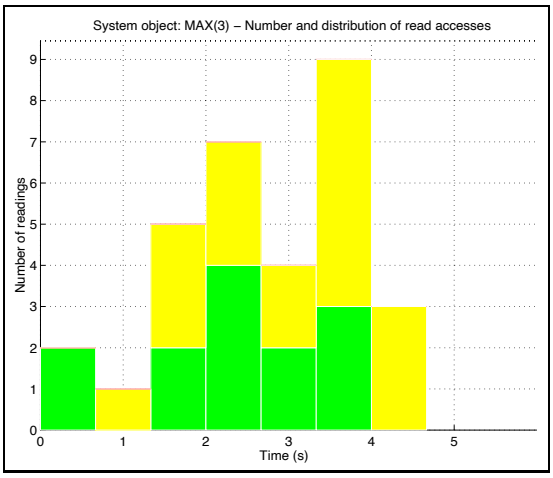

Figure 9: Number and origin of the read accesses performed on an object vs execution time (note this execution does not include inter-group accesses)

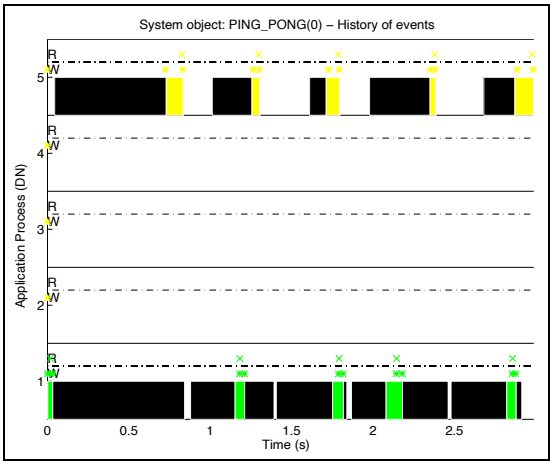

Figure 10: Object activity vs execution time

Such diagrams are extremely useful for the user in analysing problematical situations. Indeed they allow the very easily isolating ping-pong effects (*e.g.* figure 10), over-accessed variables, bottlenecks, not actually shared variables, etc.

8 Estimation of the Intrusion

This section presents the methodology we have followed to estimate the overhead time introduced by the monitoring.

The experiments were made on a network of SUN Sparc workstations. They are based on a sample application which consists of a sequence of exclusive write and read accesses applied to a variable without any computation. In terms of overhead time, this application is especially unfavorable. Indeed:

- This application does not perform any computation. The ratio overhead/execution time is consequently the worst possible;

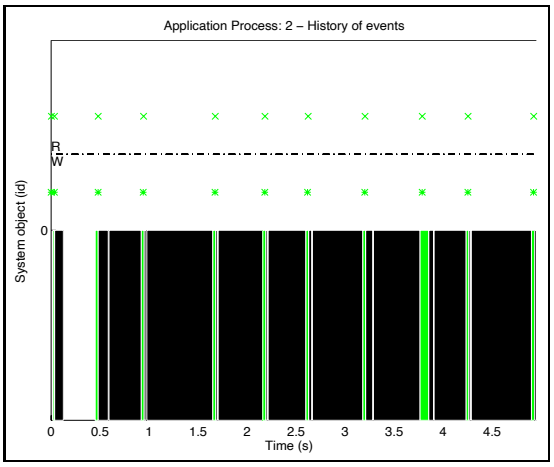

Figure 11: Process activity vs execution time

- All requests are made on the same variable. So one cannot take advantage of the variable's owner distribution;
- Only acquire and release operations are performed. However these operations are, as we have seen, the most exepensive ones.

In other words, results obtained using this sample application can be considered as an upper bound.

Four system configurations were used:

1. 4 workstations each one holding one Application Process and one Memory Process. This is the reference configuration.
2. 4 workstations each one holding one Application Process, one Memory Process and one Event Manager Process.
3. 4 workstations each one holding one Application Process and one Memory Process. Furthermore, one of them also runs one Event Manager Process which collects the events from all the MPs.
4. 4 workstations each one holding one Application Process, one Memory Process plus another workstation holding only an Event Manager Process which collects the events from all the MPs.

Table 1 shows the execution time obtained on these various system configurations. Obviously, the three monitoring configurations do not provide the same results. It is clear that an architecture containing dedicated processors only executing an EMP is the best solution. In this case, an overhead of 29% was obtained.

Consider now a "real" application, *i.e.* an application which not only makes accesses to shared data but also performs some computations. Let R be the ratio between the computation time and the access data time. The sample application represents the case where R equals zero. Its execution time, without monitoring, is equal to 21.90 seconds. For a given value of R, the execution time without monitoring can therefore be estimated as $21.90 \times (1 + R)$. In case of monitoring, it can be estimated as the time for the monitored execution without computation plus $R \times 21.90$.

Configuration	1	2		3		4	
Execution time	21.90	40.00	(+83%)	48.20	(+120%)	28.30	(+29%)

Table 1: Execution time (in seconds) for several configurations

Ratio \ Configuration	1	2		3		4	
R=1	43.80	61.90	(41%)	70.10	(60%)	50.20	(15%)
R=2	65.70	83.80	(28%)	92.00	(40%)	72.10	(10%)
R=3	87.60	105.70	(21%)	113.90	(30%)	94.00	(7%)

Table 2: Calculated execution time (in seconds) for different (computation/shared data access) ratios

Table 2 shows the estimated execution times (and the corresponding overhead (in percentage)) for 3 values of R. Thus, it appears that as soon as the computation time is higher than the access data time (which seems reasonable), the intrusion falls below 15%.

One question remains open: how many MPs must be managed by one EMP in order to keep the intrusion below a predefined limit? We are currently making tests in order to collect more experimental data.

9 Discussion

In comparison with previous approaches, the model presented in this paper presents several important advantages:

weak intrusion, due mainly to the introduction of dedicated distributed processes (EMPs).

scalability, due to the distributed architecture on which relies the model. Thus, EMPs are distributed as well as trace files;

flexibility: meta-objects are very flexible data structures. Adding a functionality to the monitoring environment only requires adding fields to the meta-object structure and specifying the protocol between EMPs and memory processes;

user-orientation: as illustrated in section 7, by working at the variable level, the DOSMOS-Trace system allows the user to clearly understand the behaviour of his application, especially to detect the most important problems: bottlenecks, ping-pong effects, bad group structure, activity imbalance,...

independence towards the shared data type: though designed for variable-oriented DSM systems, this model allows to deal with page-oriented systems. Thus, tracing the accesses to shared pages can be very simply handled by associating one meta-object to each page.

10 Conclusion and Future Works

This paper has described a novel model for the monitoring of DSM applications. This model relies on two original concepts: Event Manager Processes and meta-objects. In comparison with previous systems, this approach, based on a distributed architecture, has shown it was weakly intrusive and scalable. Implementing these concepts, the DOSMOS-Trace monitoring system has proved their efficiency and robustness.

Based on this model, further developments are mainly focused on the definition and implementation of an on-line automatic optimization tool (data migration, load balancing), *i.e.* on the automatic detection and correction, at runtime, of typical problematical situations (*e.g.* bottlenecks, ping-pong effects, bad group structure).

Acknowledgment

We would like to thank Mr Robert Halstead for his proofreading of this paper. His pertinent advices were a great help for us for improving the quality of this paper.

References

[BL94] Lionel Brunie and Laurent Lefèvre. DOSMOS : A distributed shared memory based on PVM. In *First european PVM users group meeting*, Universita di Roma, October 1994.

[BL96] Lionel Brunie and Laurent Lefèvre. New propositions to improve the efficiency and scalability of DSM systems. June 1996. to be published in the proceedings of the IEEE ICA3PP'96 conference (Singapore).

[BS92] Mats Brorsson and Per Strenstrom. Visualizing sharing behaviour in relation shared memory management. In *International Conference on Parallel and distributed systems*, Hinschu Taiwan ROC, December 1992.

[CBZ91] John B. Carter, John K. Bennet, and Willy Zwaenepoel. Implementation and performance of MUNIN. *ACM - Operating Systems Review*, 25(5):152–164, 1991.

[CG89] Nicholas Carriero and David Gerlenter. LINDA in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[DBKF90] J. Dongarra, O. Brewer, J. A. Kohl, and S. Fineberg. A tool to aid in the design, implementation and understanding of matrix algorithms for parallel processors. *Parallel Distributed Computing*, 9(2):185–202, June 1990.

[Ede93] Daniel R. Edelson. Fault interpretation: Fine-grain monitoring of page accesses. In *Winter USENIX*, pages 395–403, San Diego, CA, January 1993.

[EKKL90] Susan J. Eggers, David R. Keppel, Eric J. Koldinger, and Henry M. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. *Performance evaluation review*, 18(1):37–47, May 1990.

[FLK+91] M. Friedell, M. LaPolla, S. Kochhar, S. Sistare, and J. Juda. Visualizing the behavior of massively parallel programs. In *Supercomputing*, pages 472–480, Albuquerque, November 1991.

[FP89] Brett D. Fleisch and Gerald J. Popek. MIRAGE: a coherent distributed shared memory design. In ACM PRESS, editor, *Proceedings of the twelfth ACM Symposium on Operating Systems Principles*, volume 23, pages 211–223, The Wigwam Litchfield Park, Arizona, December 1989.

[GH91] Aaron Goldberg and John Hennessy. MTOOL: a method for isolating memory bottlenecks in shared memory multiprocessor programs. In *International Conference on Parallel Processing*, volume 2, pages 251–257, 1991.

[HC87] A. A. Houch and J. E. Cuny. Belvedere: Prototype of a pattern-oriented debugger for highly parallel computation. In *International conference on parallel processing*, pages 735–738, August 1987.

[HS92] Abdelsalam Heddaya and Himanshu Sinha. An overview of Mermera: a system and formalism for non-coherent distributed parallel memory. Technical report, Computer Science Department, Boston University Boston, MA 02215, September 1992.

[KS93] E. Kraemer and J. T. Stasko. The visualization of parallel systems: an overview. *Journal of Parallel and Distributed Computing*, 18:105–117, 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.

[Li88] Kai Li. IVY: A shared virtual memory system for parallel computing. In *International Conference on Parallel Processing*, volume II, pages 94–101, August 1988.

[LMCF90] T. J. LeBlanc, J. M. Mellor-Crummey, and R. J. Fowler. Analysing parallel program execution using multiple views. *Parallel Distributed Computing*, 9(2):203–217, June 1990.

[LP92] Zakaria Lahjomri and Thierry Priol. KOAN: a shared virtual memory for the iPSC/2 hypercube. In Springer-Verlag, editor, *Parallel Processing : CONPAR 92-VAPV*, pages 441–452, September 1992.

[MR90] A. D. Malony and D. A. Reed. Visualizing parallel computer system performances. *Parallel computer systems*, 1990.

[MY87] B. P. Miller and C. Q. Yan. IPS: an interactive and automatic performance measurement tool for parallel and distributed programs. In *Seventh international conference on distributed computing systems*, University of Wisconsin, September 1987.

[RAK89] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Coherence of distributed shared memory: unifying synchronization and data transfer. In *International conference on parallel processing*, volume II, pages 160–169, 1989.

[RCE92] David F. Robinson, Betty H. C. Cheng, and Richard J. Enbody. A transparent monitoring tool for shared-memory multiprocessors. *IEEE*, pages 227–232, 1992.

[RM93] Michel Raynal and Masaaki Mizuno. How to find his way in the jungle of consistency criteria for distributed object memories (or how to escape from minos' labyrinth). Technical Report 1962, INRIA, IRISA, Rennes, July 1993.

[SG92] S. R. Sarukka and D. Gannon. Peformance visualization of parallel programs using SIEVE. In *International conference on supercomputing*, pages 157–166, Washington D. C., July 1992.

[Str90] Per Strenström. A survey of cache coherence schemes for multiprocessors. *IEEE computer*, 23(6):12–24, June 1990.

[TKB92] Andrew S. Tanenbaum, M. Frans Kaashoek, and Henri E. Bal. Parallel programming using shared objects and broadcasting. *IEEE computer*, 25(8):10–19, August 1992.

[TV94a] Bernard Tourancheau and Xavier-François Vigouroux. Parallel trace file management on top of PVM. In *PVM UG*, Oak Ridge, TN, 1994.

[TV94b] Bernard Tourancheau and Xavier-François Vigouroux. PIMSy – a parallel trace file analyzer. In IEEE computer society press, editor, *Scalable High-Performance Computing conference*, Knoxville, Tennessee, May 1994.