Integration of Performance Evaluation Facilities into Distributed Shared Memory based Programming Environments

Lionel Brunie, Laurent Lefèvre, Olivier Reymann

Laboratoire de l'Informatique du Parallélisme (LIP)
Ecole Normale Supérieure de Lyon
46, allée d'Italie
69364 Lyon cedex 07
Tél. : (+33) 72 72 84 81
Fax : (+33) 72 72 80 80
E-mail : Prénom.Nom@lip.ens-lyon.fr

RÉSUMÉ. *Cet article propose un nouveau modèle de traçage pour des systèmes à Mémoire Distribuée Partagée basé sur deux nouveaux concepts : les meta-objets et les processus de gestion d'événements. Ce modèle constitue la base d'un système de traçage, appelé DOSMOS-Trace, qui a été developpé au dessus du système à MDP DOSMOS. Cet environnement de traçage est analysé en termes de fonctionnalités et d'interface utilisateur. Des expériences montrent l'efficacité et la robustesse du modèle sous-jacent ainsi que la pertinence d'un tel outil de traçage pour le programmeur.*

ABSTRACT. *This paper proposes an original monitoring model for Distributed Shared Memory systems 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 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.*

MOTS CLÉS : *Mémoire Distribuée Partagée (MDP), traçage, évaluation de performance, visualisation de programme*
KEYWORDS : *Distributed Shared Memory (DSM), monitoring, performance evaluation, program visualization.*

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

1 Introduction

With the development of standard and portable communication libraries (*e.g.*, PVM, MPI), networks of workstations have recently received an increasing attention. Indeed they now actually offer efficient and very cost-effective parallel platforms. However, for most basic programmers, programming using communication libraries remains largely esoteric. In that framework, by releasing the programmer from the management of inter-process communications and by allowing to access shared data in a transparent way, Distributed Shared Memory (DSM) systems provide a much more intuitive and easy-to-use programming environment. However, such systems often appear, from the programmer point of view, as a "black box" since no information about the actual communications is available. Hence, in the lack of monitoring tools, optimizing 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 2.2, 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 model we propose is discussed in section 5. Section 6 presents the DOSMOS-Trace monitoring environment which implements the concepts introduced by this model. An analysis of the intrusion generated by the monitoring is proposed in section 7. Section 8 draws the general lines of a performance evaluation and automatic optimization tool. Section 9 discusses the pertinence and the effectiveness of our model. 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 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.* a 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 6) 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; 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.

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).

In that framework, this paper proposes both a user-oriented model[2], designed to minimize the intrusion and increase the scalability, and an actual monitoring environment, implementing the principles developed in this model.

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 sub-groups. 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.

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

[2]Though this model has been initially designed for variable-oriented systems, it applies with very slight modifications to page-oriented systems (section 9).

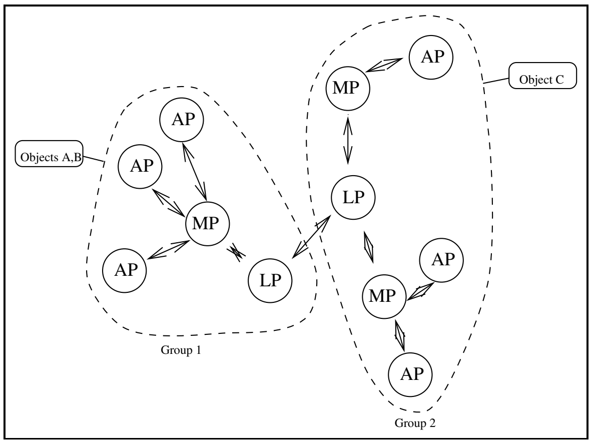

Figure 1: DOSMOS system: an example of software configuration (MP: memory process; AP: application process; LP: link process)

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. This approach presents important advantages. First, it minimizes the work requested from MPs, and consequently, the intrusion due to monitoring[3]. 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.

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

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

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.

6 Interface and Experiments

Designed to trace applications developed on top of DOSMOS, the DOSMOS-Trace system actually implements all 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[4].

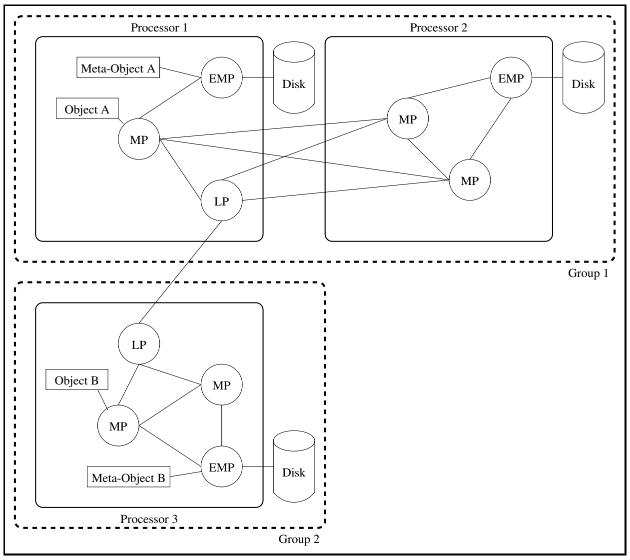

Figure 2: DOSMOS-Trace: example of monitoring environment

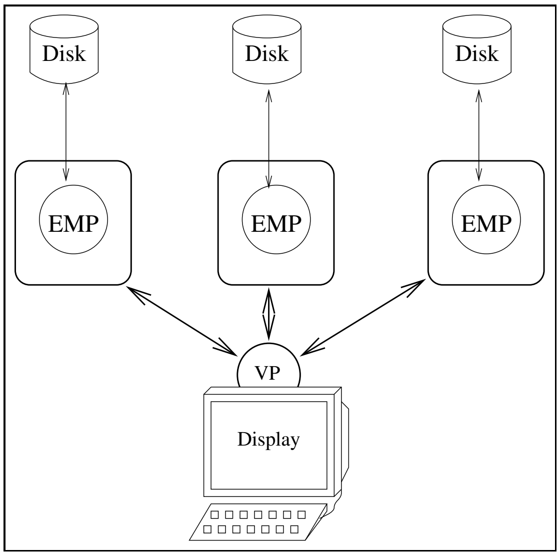

Figure 3: DOSMOS-Trace : the Visualization Process (VP) communicates with all the Event Manager Processes distributed in the network.

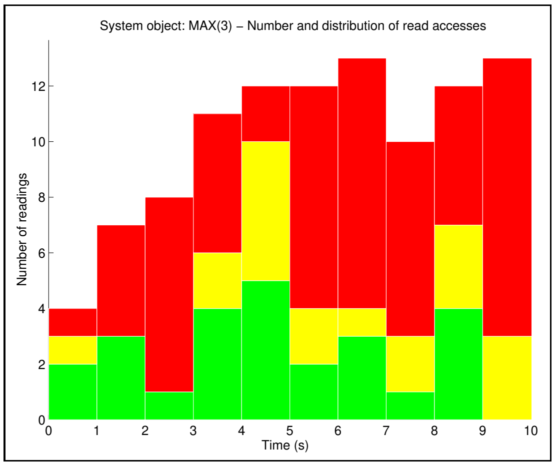

Figure 4: Number and origin of the read accesses performed on an object vs execution time (in black: inter-group accesses)

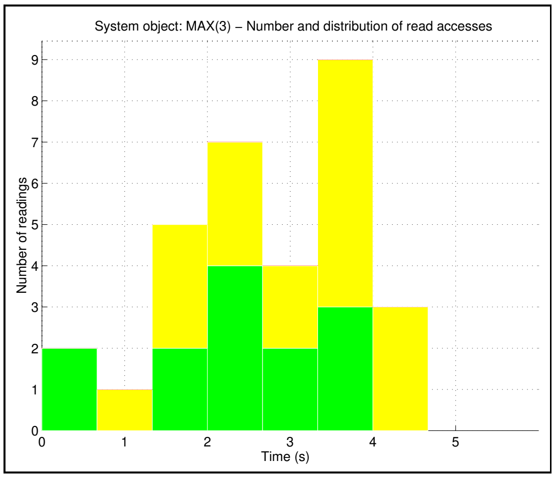

Figure 5: Number and origin of the read accesses performed on an object vs execution time (note this execution does not include inter-group accesses)

6.1 Accesses to shared Variables

Figures 4 and 5 display the histogram of the read accesses performed on a variable during the execution of an application. Various colors[5] 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 4, the predominance of inter-group accesses shows clearly that the group structure is not pertinent.On the contrary, in figure 5, one can verify that no inter-group accesses are performed.

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

6.2 Histories

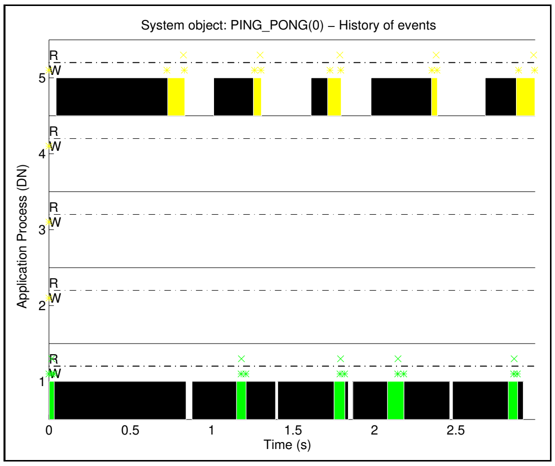

Figure 6: Object activity vs execution time

This functionality provides an analysis of the “history” of any shared variable (figure 6) or any application process (figure 7). 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.

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 6), over-accessed variables, bottlenecks, not actually shared variables, etc.

7 Estimation of the Intrusion

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

[4]Figures are displayed using Matlab.

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

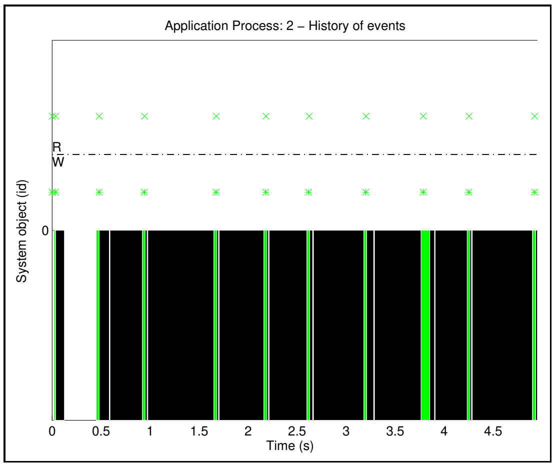

Figure 7: Process activity vs execution time

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;
- 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

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

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 $R \times 21.90$. 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.

8 From monitoring to performance evaluation and automatic optimization

Monitoring is the first step toward performance evaluation and automatic optimization. Performance evaluation refers to the automatic detection of problematic situations, *e.g.* bottlenecks, unshared variables, ping-pong effects, over-loaded Memory Processes, over-loaded Application Processes, bad group structure, bad process mapping,. . . Automatic optimization refers to the capability of inferring possible improvements from the analysis of these problematic situations. Automatic optimization can be performed either in an on-line or in a post-mortem fashion.

8.1 Performance Evaluation

- **Group Structure**
  A good group structure is achieved when (1) inter-group accesses are few in number; (2) groups are "well-dimensioned" *i.e.* groups gather processes that actually share the variables. The first requirement can be checked by analysing the access distribution. The verification of the second one is more difficult

to implement since it requires a deep analysis integrating a study of the time distribution of the accesses.

- **Load Balance and Process Mapping**

  Estimating the quality of the software configuration can be done by analysing either the distribution of the access requests or the CPU time used by each MP. In the same way, the load balancing over the APs can be evaluated by studying the amount of CPU time used by each AP. Oppositely, the pertinence of the software configuration requires a much more complex analysis. However, some basic checkings can be executed. For instance, the load of a processor running at least one MP should not much exceed the load of a processor without MP. In fact, this would mean that the MPs are not numerous enough.

- **Bottlenecks, Unshared Variables, Ping-Pong Effects**
  These verifications can be easily done by an analysis of variable histories (see section 6.2).

8.2 Automatic Optimization

- **Group Structure**
  Two basic operations can be proposed when groups are badly structured. (1) Group redefinition: if the inter-group accesses are too numerous, one can expand the groups. On the contrary, if groups are over-dimensioned, they can be shrunken. (2) An AP can "subscribe" to the variable which is accessed in a inter-group manner. This AP will now be authorized to own a copy of the variable in order to perform local accesses and to be notified of value modifications. If the AP did not access the variable for a long while, its subscription will be canceled.

- **Load Balance and Process Mapping**
  If tasks allocated to the MPs are not well-balanced, various strategies can be used: changing the ownership of the variables, modifying the array splitting in order to adapt it as well as possible to the access distribution.

  If a processor running a MP is over-loaded, either the number of MPs in the system can be increased, or MPs can be mapped on dedicated processors. Such modifications of the software configuration should improve the efficiency of the application. Nevertheless, there is no guarantee of this mainly because they affect the network traffic.

- **Bottlenecks, Unshared Variables, Ping-Pong Effects**
  In front of bottlenecks, the only measure an automatic tool can propose is a new array splitting. Concerning unshared variables and ping-pong effects, optimization strategies can only be taken by the user since the DSM system does not have any semantic information about the application.

9 Discussion

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

weak intrusion, (section 7) due mainly to the introduction of devoted 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 to specifying the protocol between EMPs and memory processes;

user-orientation: as illustrated in section 6, 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,. . .

shared data type independence: 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;

hierarchical paradigm: DOSMOS-Trace relies on the hierarchical architecture on which the DOSMOS system is based (notion of group). This feature is especially well suited for networks organized in sub-networks of workstations.

10 Conclusion and Future Works

DSM systems appear as very promising programming environments, especially in the context of networks of workstations. However, in the lack of monitoring tools, optimizing and evaluating DSM applications is *de facto* impossible. In that framework, this paper has described a novel model for the monitoring and the visualization 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 run-time, of typical problematical situations (*e.g.* bottlenecks, ping-pong effects, bad group structure). The general lines are presented in section 8.

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.

[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.