# Formal and experimental validation of a low overhead execution replay mechanism*

Alain Fagot and Jacques Chassin de Kergommeaux

IMAG, APACHE project
46 avenue Félix Viallet,
F-38031 Grenoble Cedex 1, France.
{Alain.Fagot,Jacques.Chassin-de-Kergommeaux}@imag.fr

**Abstract.** This paper presents a mechanism for record-replay of parallel programs written in a remote procedure call (RPC) based parallel programming model. This mechanism, which will serve as a basis for implementing a user-level debugger, exploits some properties of the programming model to limit drastically the number of records that need to be done. A formal proof of the equivalence between recorded and replayed executions is given. Systematic measurements of the time overhead of the recording indicate that it is sufficiently low for the recording mode to be considered as normal execution mode. Similar techniques can be applied to other programming models.

*Keywords:* Instant Replay, parallel debugging, deterministic reexecutions, Remote Procedure Call.

## 1 Introduction

This paper presents a mechanism allowing programmers to cope with the inherent non-determinism of parallel executions, when debugging programs written in a remote procedure call (RPC) based programming model, designed for parallel multiprocessors. Many parallel programs present a non-deterministic behavior, even if they produce deterministic computation results. Non-deterministic execution behaviors originate mainly in execution environments of programs. Such an environment depends on a large number of factors that cannot be controlled by the programmer, such as the initial contents of cache memories, the behavior of the operating system, etc. Programs adapting to the execution environment for efficiency reasons, using dynamic load balancing techniques, for example, are very prone to exhibit non-deterministic execution behaviors. Non-deterministic execution behavior of erroneous parallel programs may result in transient errors which appear very unfrequently or vanish when debugging tools are used, because of changes introduced by these tools in the causal relationship between parallel processes.

* **This work was partially supported by the French Ministry of Research under the inter-PRC project Trace.**

The most classical technique used to catch transient errors appearing during executions of parallel programs is to *record* an initial execution and to *force* subsequent executions to be deterministic with respect to the initial execution, using the recorded information. Debugging an erroneous program then amounts to record an erroneous execution and to apply cyclic debugging techniques during subsequent replayed executions. In order for this technique to be effective, the perturbation resulting from the recording operation ought to be kept sufficiently low so that errors appearing in un-recorded executions do not vanish in recorded ones and vice-versa. If this overhead is low enough, recording can be left active during each execution of a parallel program, so that an error occurring unfrequently can be captured and subsequently reproduced.

Efficient record-replay techniques are mostly based upon the "Instant Replay" mechanism of LeBlanc and Mellor-Crummey [5]. The efficiency of the instant replay comes from the observation that it is sufficient to record the order of accesses to shared objects to be able to reproduce "indistinguishable" executions. The instant replay mechanism was adapted to message passing programming models [6], where each process records on a tape the identifiers of received messages. The replay system forces re-executing processes to treat incoming messages in the same order as during the initial recording. This mechanism was used as a basis for the implementation of parallel debuggers [8, 6, 3].

This paper describes an optimized record-replay mechanism for ATHAPASCAN[2], the programming model of the APACHE research project. APACHE aims at designing and implementing a parallel programming environment for parallel computers, providing both static and dynamic load balancing facilities [9]. The mechanism described in the sequel of this paper, exploits the characteristics of remote procedure calls to reduce drastically the volume of traces that need to be recorded in order to be able to replay programs deterministically with respect to the original recorded computation. This mechanism can be applied to any RPC-based programming model. A formal proof of the equivalence of executions controlled by the described mechanism is also given. In addition the time overhead of recording is measured systematically, for the most classical parallel numerical algorithms, showing that it always remains very low.

## 2 The Athapascan programming model

In ATHAPASCAN, the execution of parallel programs is performed by a set of identical *virtual processors* operating asynchronously [2]. Expression of parallelism is achieved by blocking and non-blocking remote procedure calls (requests), thereby hiding the underlying communication protocols under the parameters and results transmission mechanisms. Thus the ATHAPASCAN model is well suited for expressing control parallelism. Each virtual processor includes several *Entry Points*, which are the targets of remote procedure calls (see figure 1). No other communications are available in ATHAPASCAN.

[2] ATHAPASCAN is the language of the Apaches.

A remote procedure call results in the execution of a light-weight process (thread) within the virtual processor holding the target entry point. This thread may in turn create new threads by issuing remote procedure calls. Upon completion, each thread returns a result to its caller thread. Several light-weight processes execute concurrently within each virtual processor to hide the latency of communications in parallel systems. ATHAPASCAN offers two types of remote procedure calls:

- blocking (*Call*): control is returned to the caller after receiving the result of the called procedure,
- non blocking (*Spawn*): control is returned to the caller after the creation of the remote thread. Two operators are provided to test (*TestSpawn*) or wait (*WaitSpawn*) for the completion of non-blocking remote procedure calls.

## 3 Minimal trace recording

The non-deterministic behavior of an ATHAPASCAN program execution is due to the variable order in which requests are handled by entry points and to the results of non-deterministic primitives which are related to the current state of the system. The two causes can be tracked down separately.

### 3.1 Basic mechanism

The principle of the control driven replay is to record the order of accesses to shared resources. Classical implementations of Instant Replay record the order of system-level primitives for passing messages [6] or accessing shared variables [5]. In the ATHAPASCAN model, shared resources are entry points, accessed by requests. Our record-replay mechanism uses an intermediate level of abstraction where several system-level events can be abstracted in one, thereby reducing the number of records while being independent of the underlying communication system.

Each call to an entry point results in a typical sequence of such "abstract" events. Figure 1 represents a complete sequence of events generated by a call to an entry point, from the request emission (event **a**) to the result receipt (event **d**) passing through the request receipt (event **b**) and the result emission (event **c**). A replay can be driven by forcing the execution order of request receipts (event **b**). Since an entry point controls racing requests, it can be responsible for recording the order in which it serves incoming requests.

The order of request emissions (event **a**) is not recorded since an emitting thread will reproduce this order if all its non-deterministic operations produce the same values. However, this order may not be significant since the ATHAPASCAN model does not impose this order to be followed by the request receipts.

The order of request receipts (event **b**) is the order in which incoming requests are processed by an entry point. This fundamental order represents the order in which access is granted to shared resources and is recorded by each entry point.

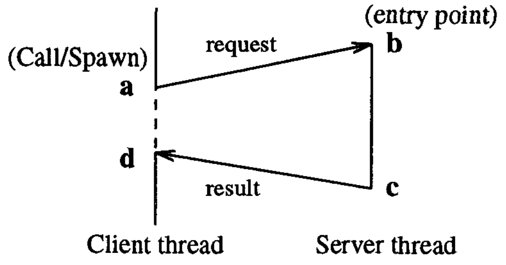

**Fig. 1.** Sequence of events for a call to an entry point.

The order of result emissions (event **c**) is not significant since a single result is emitted by a thread and this occurs at the end of its execution.

The order of result receipts (event **d**) is not visible from the point of view of the client thread. A client thread is informed of the presence of the result only through primitives *TestSpawn* and *WaitSpawn* in the case of a non blocking call or implicitly in the case of a blocking call.

Each entry point is responsible for recording its request receipt history (event **b**). This history contains the order of request unique identifiers. A request identifier is constructed independently by the client thread emitting the request in a deterministic way: in particular, it is independent of the other threads sharing the same virtual processor. For the ATHAPASCAN model, it has the following form: ⟨VirtualProcessorID, EntryPointID, ThreadID, RequestID⟩.

### 3.2 Non-deterministic primitives

Non-deterministic primitives may be considered as predefined non-deterministic entry points of the ATHAPASCAN kernel. For this type of entry points, results cannot be computed during replayed executions as during a recorded execution. Therefore, the instant replay mechanism must record the results computed for each non-deterministic request along with the request identifier in order to provide the same result to the same request during the replay. This technique mixes data driven replay with the general control driven strategy. It is used to record-replay the ATHAPASCAN *TestSpawn* primitive whose result is dependent on communication delays.

### 3.3 Interest of the proposed mechanism

The simplifications brought to the classical model result from the communication simplicity of the ATHAPASCAN-0 model. Processes obey a Client-Server protocol which is a sub-class of the model of communicating processes. Each request corresponds to a result and a result is emitted only if a request was received. The ATHAPASCAN request and result transmissions can be implemented in several different ways without consequences on the design of the record-replay mechanism. This independence leads to a reduction of the number of trace-points for some implementations, up to a factor of six relative to the classical solution of implementing the record-replay at the system level. This reduction factor is obtained at low cost since no additional information is appended to the messages.

## 4 Execution equivalence demonstration

In this section, we prove formally that using the record mechanism defined in section 3 results in deterministic replayed executions of ATHAPASCAN programs with respect to an original recording. An execution model is given for ATHAPASCAN, in the framework of which it is possible to define the equivalence between two executions of the same program. The notion of equivalence is such that equivalent executions of a program exhibit the same behavior from the programmer point of view.

The demonstration is similar to the demonstration of equivalence given by Mellor-Crummey in [8]. A parallel program execution is composed of a set of (light-weight) processes, each of which executes a function to compute the response (result) to a request. A mapping relation $M$ defines the bijection between emitted requests and computing processes. The demonstration shows that it is sufficient to enforce the same mapping relation $M$ in both executions to make them equivalent. The modified ATHAPASCAN run-time kernel implements the record-replay mechanism by recording the mapping relation of the initial execution and forcing the replayed execution to follow this mapping relation.

### 4.1 Execution model

**Assumptions.** In the following, we will assume that:

1. *Recorded* and *replayed* executions of a parallel ATHAPASCAN program are done using a fixed and ordered set of virtual processors $VP$. Each execution is started with the same initial parameters. Each virtual processor offers a fixed and ordered set of entry points $EP_{vp}$. All replayed executions can use at least the same amount of resources as the initially recorded one: processors, memory and disk space, etc.
2. The global variables of any *Entry Point* can only be accessed through a call to this *Entry Point*. Such an *Entry Point* declaration defines the limit of concurrent accesses allowed for these global variables. Therefore the state of each *Entry Point* during the computation of an ATHAPASCAN program is only determined by the order according to which the requests are received and processed by the *Entry Point*.
3. ATHAPASCAN programs do not use system non-deterministic primitives. The equivalence result obtained for such programs can be simply extended for programs using non-deterministic primitives, provided the results of these primitives are recorded in the recording phase and read in subsequent replayed phases.

4. The use of inputs/outputs in ATHAPASCAN programs is restricted. The programmer must ensure that the inputs of replayed executions are the same as the inputs of the recorded one. Access to shared output devices need to be encapsulated in ATHAPASCAN *Entry Points.*
5. Transmission times of requests and results are finite.

**Definition 1.** A *history of request receipts* $hr_{vp,ep}$ is associated with each entry point of each virtual processor. It defines the order according to which incoming requests are handled. The denotation of the history of request receipts of entry point $ep$ of virtual processor $vp$ is the following:

$$hr_{vp,ep} = c_0^{vp,ep}, c_1^{vp,ep}, c_2^{vp,ep}, \ldots$$

The execution model refers to the histories of request receipts $hr_{vp,ep}$ which are defined as the sequence of processes executed on the entry point. Each computation $c_{th}^{vp,ep}$ is performed by the process identified with the unique triple $\langle vp, ep, th \rangle$ expressing the thread $th$ running the entry point $ep$ on the virtual processor $vp$.

**Definition 2.** A *history of request emissions* $he_p$ is associated with each process $p = \langle vp, ep, th \rangle$. It defines the sequence of requests emitted by this process during execution. The denotation for the history of request emissions of process $p$ is the following:

$$he_p = e_0^p, e_1^p, e_2^p, \ldots$$

Each request emission $e_i^p = \langle vp, ep \rangle$ is directed towards entry point $ep$ on virtual processor $vp$. For each process $p$, the history of request emissions $he_p$ reflects the interactions of this process with the rest of the program.

**Definition 3.** The *mapping relation* $M$ is a set of triples in the form $\langle p_1, i, p_2 \rangle$ indicating that the request emission $e_i^{p_1} = \langle vp, ep \rangle$ is computed by process $p_2 = \langle vp, ep, th \rangle$.

The relation $M$ realizes a bijection between the set of emitted requests and the set of computations. This bijection guarantees that each emitted request is computed and each process computation corresponds to a request.

Following these definitions, the execution $X$ is characterized by the triple $\langle H, E, M \rangle$, where $H$ is the set of histories of request receipts, $E$ is the set of histories of request emissions and $M$ is the mapping relation.

In the following we assume that for all the computations of the same programs, the same mappings are enforced.

## 4.2 Execution equivalence

**Definition 4.** Two executions $X$ and $X'$ are said to be equivalent if for each process $p = \langle vp, ep, th \rangle$ both executions assign the same history of request emissions to process $p$. The equivalence of two executions $X$ and $X'$ is denoted as $X \approx X'$.

This definition of execution equivalence is suitable for debugging a program since the behavior of each individual process is identical in all equivalent executions. These identical behaviors enable a programmer to refine his understanding of the execution of a program through repeated executions. A cyclic debugging technique can then be applied.

***Lemma 5.*** *Sequential* ATHAPASCAN *processes having no external interaction are deterministic.*

This lemma expresses the basic hypothesis of all instant replay mechanisms.

*Consequences:*

1. For any execution of a parallel ATHAPASCAN program, a process started with the same initial conditions will emit the same first request or the same result, if it does not emit any request.
2. For any execution of a parallel ATHAPASCAN program, a process started with the same initial conditions and whose previously emitted requests were identical and returned the same results, will emit the same following request or result, if it does not emit any more request. Here the difference is that the process interacts with its environment. However its interactions remain the same through all its computations.

For the purpose of the equivalence demonstration, let us define a vector clock [7] for ATHAPASCAN.

**Definition 6.** A *vector clock* for ATHAPASCAN is defined as a vector whose dimension is the number of Entry Points used by an ATHAPASCAN program execution and updated for each Entry Point in the following way:

1. The $i^{th}$ component of the vector clock of an Entry Point is incremented each time an incoming request $c_{th}^{vp,i}$ is handled on the Entry Point, that is a new process is created and started:
$$VC_i[i] := VC_i[i] + 1 = th$$
2. The vector clock $VC_i$ of $EP_i$ is piggy-backed to each message sent by a process of $EP_i$, be it a request or a result message.
3. The vector clock $VC_i$ of $EP_i$ is updated on receipt of each message by the Entry Point: if the message is a request, it is when starting a new process, just before incrementing the $i^{th}$ component of the vector clock (see above); otherwise, if the message is a response, the incrementation takes place when the message is passed to the requesting process. Updating performs the following operation:
$$VC_i := sup(VC_i, VC_{mes})$$
*sup* being a component-wise maximum operation.

We now use this vector clock to define a partial order on the messages emitted during a computation. Let $n$ be the number of Entry Points during both computations. Let $m_i$ and $m_j$ be two messages emitted from Entry Points $EP_i$ and $EP_j$ during the computation of an ATHAPASCAN program and $VC_i$ and $VC_j$ the values of the vector clocks piggy-backed to $m_i$ and $m_j$.

**Definition 7.** The partial order between messages induced by the ATHAPASCAN vector clock will be denoted $\prec_{VC}$:

$$m_i \prec_{VC} m_j \Leftrightarrow VC_i \prec VC_j$$

with

$$VC_i \prec VC_j \Leftrightarrow VC_i[k] \leq VC_j[k], \forall k \in [1, n]$$

$\prec_{VC}$ is a partial order since non causally linked messages cannot be ordered. Several requests emitted by the same process $p$ may hold the same vector clock. However they can be ordered by using the indice $o$ of the request emission $e_o^p$ to extend the order $\prec_{VC}$.

***Theorem 8.*** *Let $X = \langle H, E, M \rangle$ be an* ATHAPASCAN *program execution. Let $X'$ be an execution of the same program under the assumptions defined above (see beginning of section 4.1).*

*For $X'$ to be equivalent to $X$, it is sufficient to map all requests of $X'$ using $M$.*

*Proof:* The proof is done by contradiction. Let us assume there exists at least one message of $X'$ without corresponding identical message in $X$. If there exist several messages of this kind, there exists a set of smallest messages, under the $\prec_{VC}$ relation. Let $r_i'^j$ be one of the messages of this set, the $j^{th}$ message emitted by process $i$. Two possible cases may arise:

1. Either $j = 1$, which means that $r_i'^j$ is the first request emitted by process $i$, or the answer emitted by process $i$ if it does not emit any request. Again two possible cases:
   (a) Either $i = 1$, $r_1'^1$ is the first request emitted during the ATHAPASCAN program execution. The program was started with the same input parameters in $X$ and $X'$ and therefore the sequential computations before the first request emissions should be identical in $X$ and $X'$ (consequence of lemma 5). Therefore requests $r_1^1$ and $r_1'^1$ are identical.
   (b) Or $r_i'^1$ is the first request emitted during the computation of process $i$ or the answer of process $i$, if it does not emit any request. Since process $i$:
      – was created as a consequence of the same request in both executions $X$ and $X'$, this request being lower than $r_i'^1$ in the order $\prec_{VC}$,
      – was started under the same initial conditions because the mapping $M$ defined by execution $X$ is used during execution $X'$.
      – did not receive any other external input before emitting $r_i'^1$,

it will perform the same sequential computation between its initialization and the emission of $r_i'^1$ (consequence of lemma 5). Therefore $r_i'^1$, emitted during $X'$ is identical to $r_i^1$ emitted during $X$.

2. Or $j > 1$. But prior to the emission of $r_i'^j$, process $i$:
   - was started under the same initial conditions because the mapping $M$ defined by execution $X$ is used during execution $X'$.
   - received the same inputs, in the same order as in execution $X$, since otherwise there would be some message $m_i'$ of $X'$ with $m_i' \prec_{VC} r_i'^j$ without corresponding identical message $m_i$ in $X$, which contradicts the assumption above.

   Because of lemma 5, it is not possible for $r_i'^j$ to be different from the $j^{th}$ request of process $i$ in $X$ and the assumption of the proof is contradictory.

The assumption done is proven false under any circumstances which demonstrates by contradiction that every request from $X'$ has a corresponding request in $X$. A similar reasoning proves that it is impossible for a request of $X$ not to have its counterpart in $X'$. □

## 5 Time-overhead measurement of trace recording

A prototype ATHAPASCAN kernel was instrumented with the mechanism described in this paper. The kernel is built on top of PVM and several thread libraries available for different hardware architectures. The determinacy of reexecutions was tested with a highly non-deterministic process-farm implementation of the N-queens problem, by observing the order of solutions in the result list.

### 5.1 Time-overhead measurement method

Performance measurements were done using the ANDES (Algorithms aNd DEScription) modeling language and synthetic programs generator (see figure 2) [4] which was adapted to model and generate ATHAPASCAN programs. A synthetic parallel program is a real program whose resource consumption, processor, memory and communication, can be easily controlled. The main advantages of synthetic programs are the possibility to generate them automatically and the ease of changing parameters regulating the consumption of resources. Synthetic programs can be used to measure the overhead of trace recording since this overhead is the same for real or synthetic programs. Another advantage of this method was the availability of a set of existing ANDES models. This is the approach of the ALPES project (ALgorithms, Parallelism, and Evaluation of Systems), combining synthetic program generation tools with software monitoring of parallel programs [11].

From a model of algorithm written in ANDES, it is possible to generate a wide range of different synthetic programs with different structures. For example with the *Prolog-like search tree* algorithm, it is possible to change the branching factor of the nodes or the depth of the tree. In the experiments, only one structure was

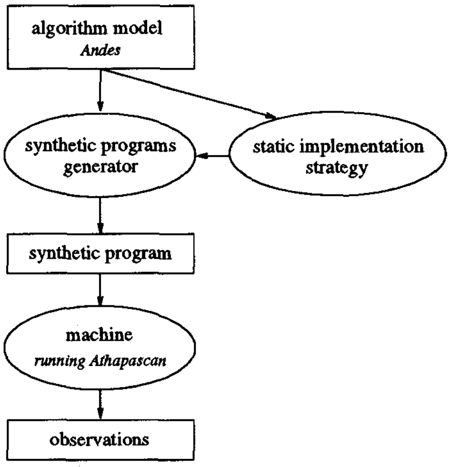

**Fig. 2.** Evaluation chain.

retained for each model of algorithm. The computation/communication ratio of synthetic programs was adjusted by tuning the values of the parameters of the program models. Experiments were performed by generating, for each model of algorithm, a synthetic program for each of the selected ratios. Execution threads defined in the models were mapped on the virtual processors executing the synthetic programs using one of the greedy algorithms of the mapping toolbox of the APACHE project [1]. From these mappings and the ANDES models of algorithms, synthetic ATHAPASCAN programs were generated. Then the execution times of all synthetic programs were measured, for all the selected ratios.

We restricted ourselves to program models having a deterministic behavior. The behavior of a non-deterministic program can indeed be so different for each of its executions that comparisons become impossible, as it was experienced with the N-queens program where some executions recording traces executed faster than unmonitored executions. For programs whose behavior is non-deterministic, it is not possible to apply a statistical measurement method, based on the hypothesis that observed executions have similar behaviors. The selected models include the following structures of algorithms *Divide and Parallelize* (balanced tree), *Prolog-like Search Tree* (unbalanced tree), *Regular Iteration* (same number of forks in each step), *Master-Slaves* (variable number of forks in each step) and *Strassen's Matrix Product* [10] (recursive numerical algorithm).

To measure the time overhead of the recording, the execution times of synthetic programs were measured "with" and "without" the recording mode set. For each benchmark, the desired precision was to make sure that, with a probability of 95%, the real mean execution time was enclosed within an interval of 3% of the mean execution time centered around the estimated mean execution time. The whole experiment represented 2400 different executions of ATHAPASCAN programs. Each overhead was computed as the ratio of the difference of mean execution time "with" recording mode set and mean execution time "without" recording mode set, divided by the mean execution time "without" recording mode set. The division between two results known with a precision of 95% each reduced the precision of the overheads to a certitude of 90%. Synthetic programs were executed by a prototype of the ATHAPASCAN kernel running on a 32 nodes IBM SP1 entirely dedicated to the measurements.

## 5.2 Time-overhead measurement results

Measurement results are summarized in figure 3 which displays the measured recording overheads with their confidence ranges. The main outcome from these measures is that recording overheads are lower than 5%, even for the improbable cases where communication costs represent 10 times computation costs. No algorithm seems pathological with respect to the time overhead of recording.

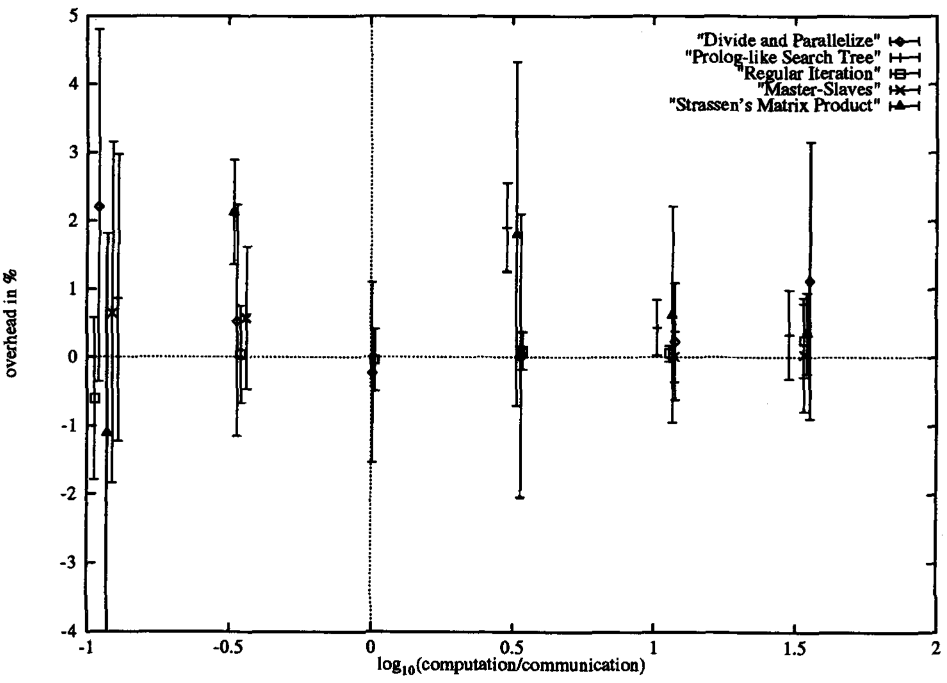

**Fig. 3.** Preliminary results.

## 6 Conclusion

Cyclic debugging of inherently non deterministic parallel programs can be done using the Instant Replay technique. This paper describes an adaptation of the

Instant Replay mechanism to a hierarchical, RPC-based programming model, where parallel programs are executed by a potentially high number of lightweight processes, grouped in virtual processors. This adaptation was optimized by exploiting the characteristics of the programming model, resulting in an important reduction of the number of records necessary to replay programs deterministically. The techniques described in the paper can be used for any RPC-based parallel programming model such as applications structured according to a Client-Server architecture. Similar techniques could be adapted to some object-oriented parallel programming models.

A prototype implementation of the RPC-based ATHAPASCAN programming model including record-replay techniques was done and tested. Systematic measurements indicated that the costs of the recording –time overhead and volume of recorded traces– remain very limited and that the recording mode can be used as the normal ATHAPASCAN execution mode, enabling to capture unfrequent errors as soon as they occur and debug them using a cyclic method. The implementation of the instant replay mechanism will serve as a basis for the development of an ATHAPASCAN debugger which is currently being designed.

## References

1. P. Bouvry, J. Chassin, and D. Trystram. Efficient solutions for mapping parallel programs. In *Proceedings of EuroPar'95*. Springer-Verlag, August 1995.
2. M. Christaller. ATHAPASCAN-0A control parallelism approach on top of PVM. In *Proc PVM User's group meeting*. University of Tennessee, Oak Ridge, 1994.
3. H. Jamrozik. *Aide à la Mise au Point des Applications Parallèles et Réparties à base d'Objets Persistants*. PhD thesis, Université Joseph Fourier, Grenoble, 1993.
4. J. P. Kitajima and B. Plateau. Modelling parallel program behaviour in *ALPES*. *Information and Software Technology*, 36(7):457–464, July 1994.
5. T.J. LeBlanc and J.M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–481, 1987.
6. E. Leu and A. Schiper. Execution replay: a mechanism for integrating a visualization tool with a symbolic debugger. In *CONPAR 92 - VAPP V*, volume 634 of *LNCS*, September 1992.
7. F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms*, Bonas, France, September 1988. North Holland.
8. J.M. Mellor-Crummey. Debugging and Analysis of Large-Scale Parallel Programs. Technical Report 312, University of Rochester, September 1989.
9. B. Plateau. Présentation d'APACHE. Rapport APACHE 1, IMAG, Grenoble, December 1994. Available at `ftp.imag.fr:imag/APACHE/RAPPORTS`.
10. V. Strassen. Gaussian Elimination is not Optimal. *Numerische Mathematik*, Band 13(Heft 4):354–356, 1969.
11. C. Tron et al. Performance Evaluation of Parallel Systems: the ALPES environment. In *Proceedings of ParCo93*. Elsevier Science Publishers, 1993.