Replaying Distributed Applications with RPVM

João Lourenço   José C. Cunha
{*jml,jcc*}*@di.fct.unl.pt*
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Portugal

Abstract—**Parallel debugging is complex and difficult. Complex because the programmer has to deal with multiple program flows and process interactions, and difficult due to the very limited choice on effective and easy-to-use debugging tools for parallel programming. Simple and necessary features for parallel debugging are absent even from commercial debuggers, such as a *record-replay* feature, that allows to re-execute multiple times a parallel application assuring that during each re-execution the internal race conditions are solved in the same way they were in the first time.**

Some work has been done on record-replay techniques for parallel and distributed applications, but just a few have been applied to specific systems (such as PVM or MPI), and even less have produced working prototypes. In this paper we describe a method designed to work with the PVM system and how it was implemented to provide a working prototype.

I Introduction

Debugging is a crucial phase in the program development cycle [4], [7], [9], [11], [12], [18], [5]. When compared to sequential debugging, the increased complexity verified in parallel and distributed debugging demands new tools with new features to address new problems.

Some of the (new) significant problems that must be addressed are:

1. *Heterogeneity.* The application processes may be dispersed by several machines, with different hardware and/or operating systems. To provide the user with a unified debugging tool for parallel applications, and not with a set of individual sequential debuggers, support to handle this multitude of different architectures in a convenient way is required [8].
2. *Flexibility.* When debugging an application, the user wants to think using the same concepts and abstraction level that were used when coding the program. To support this, the debugger must "understand" the used programming abstraction level, allowing however full access to the lower abstraction levels too. For example, if a graphical language was used to develop the code, then the basic debugging activities should be done at the same graphical level, but the user should have access and be allowed to debug the application's generated source code or even its assembly code [10].
3. *Integration.* Multiple user interfaces (for the different abstraction levels) should be supported, either exclusively or simultaneously. If support is provided for multiple (simultaneous) debugging interfaces over the same target application, the debugging environment must provide support for these interfaces to have coherent views of the target application.

To tackle these problems we propose a structured debugging framework. This includes a basic infrastructure for monitoring and control of distributed applications—the DAMS system [3]—and a set of services. These services can be grouped into:

1. *Component-level services.* Deal with the application processes individually, providing services such as variable inspection and process control.
2. *Coordination-level services.* Deal with the interactions between the multiple application processes. For example, the monitoring (and logging) of the messages exchanged between the application processes is a coordination-level service.

In this paper we will describe the design and implementation of a coordination-level service of recording and replaying PVM distributed applications, and its integration into a more general debugging service.

The next section presents some related work. In Sec. III we will describe succinctly the DAMS system and its component-level distributed debugging service. In Sec. IV, and after some brief considerations about replaying of parallel and distributed programs, a replayer for PVM applications is presented and its implementation described. In Sec. V the conclusions and some ongoing work close the article.

II Related work

In order to allow the deterministic re-execution of parallel programs, several proposal of *trace*-based *replay* systems have been made [21], [1], [15], [6]. All of them based on the same principle of recording enough information during the program execution, so that there is a well-defined event ordering. This event ordering is later used to drive program execution during replay.

The intrusion due to the required program monitoring and the potentially large volume of trace information are the most critical problems in the implementation of *record-replay* systems. Concerning this later aspect, two main approaches can be identified:

1. Incremental replay systems using checkpointing techniques, so that the user may restart the program form intermediate points. The main drawback of such technique is the need to record a large amount of trace information to save the program state in the intermediary checkpoints [23], [2].
2. Other approaches require the user to replay the program from the beginning, but less information need to be included in the trace file, with the resulting smaller intrusion. In the *data-driven* replay technique, all the data exchanged

between processes is recorded. The *control-driven* replay technique reduces the trace volume by relying on data regeneration during the replay phase.
The main reference for the *control-driven* replay technique is the *Instant Replay* [14], that only require a minimal information on the event ordering, reducing drastically the volume of the trace file(s) when comparing with the *data-driven* or *checkpointing* techniques. A discussion of further optimizations to this technique, so that even smaller event logs can be obtained is reported in [20], [16]. These techniques have been applied both to shared- and distributed-memory (message passing and RPC-based) models.

Even some well known existing parallel debuggers, like p2d2 [8] and TotalView [22], lack such a *record-replay* facility.

In [19] a replay mechanism for the PVM system is described in detail. The approach used was somewhat different and can be summarized as:

- The application will not be changed in any way;
- The PVM system will incorporate internally all the functionalities needed to support the replay;
- Before any record or replay session, the PVM system must be shut down and restarted again, to guarantee the regeneration of the same task IDs in the replay session.

As only the PVM internals were changed, there are also some considerable drawbacks for this solution:

- Very complex implementation;
- Too much dependent on the PVM source code and PVM version;
- Requires a very specific procedure to guarantee that the PVM tasks IDs obtained in the replay phase are identical to the ones of the record phase;
- There is no knowledge that the prototype has been finished and completely operational;
- Limited access due to the (no) distribution policy.

As an alternative, the method proposed in this paper has the following characteristics:

- It does not require any modification to the PVM system;
- Compatible with new PVM versions, and easily adaptable to other systems (*e.g.* MPI);
- Requires only a minimal amount of modifications to the application source code;
- Even if linked to the application, if the record/replay features are not enabled, there is no considerable performance degradation or intrusion.

III A Structured Framework for Parallel and Distributed Debugging

The DAMS system (see Fig. 1) implements a basic infrastructure for the development of component- and coordination-level services, such as debugging and resource-management services [3]. In this figure there is a PVM target application composed by four processes and under control of DAMS, and two applications—a graphical debugging interface (GUI) and a text oriented one (TUI)—are controlling the application simultaneously. The figure illustrates a possible configuration where the GUI is only accessing the component-level debugging services, while the TUI is also accessing the replaying service.

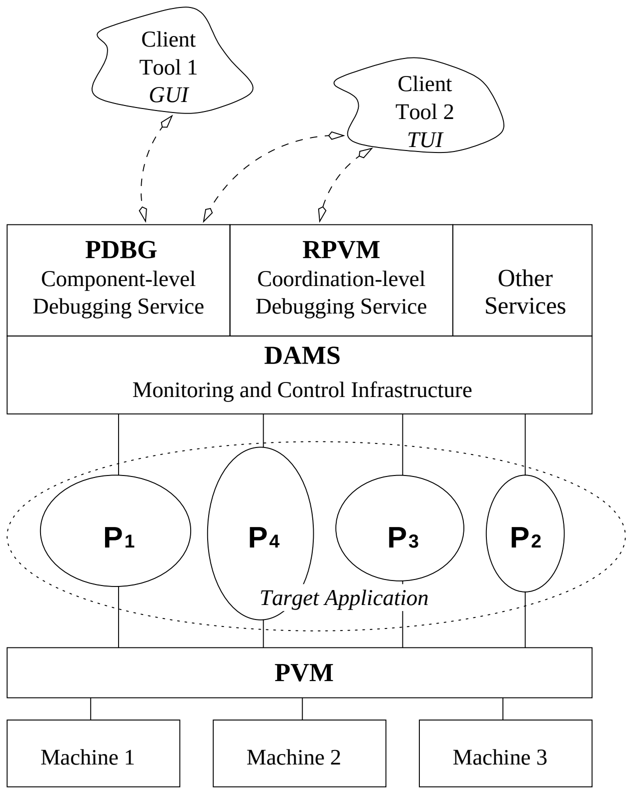


Fig. 1. The DAMS Layers

III.A The Basic Infrastructure

The DAMS system has a distributed software architecture which is composed by:

1. *A Library*, to be linked with each process that needs to access DAMS services. In the figure, both the GUI and TUI have been linked with this library.
2. *The System Manager*, to coordinate the interactions between the multiple components of DAMS.
3. *The Local Managers*, one per physical node, to coordinate the interactions with the local and target processes.

This basic architecture can then be extended with *Services* (*e.g.* a debugging or a resource management service) that may, in principle, coexist simultaneously. For each new service two components must be implemented:

1. *The Service Module*, a centralized component that implements the service's high-level system-independent part.
2. *The Service Drivers*, a distributed component (one per physical node) that implements the service's low-level system-dependent part.

The *Debugging* and *Replaying Services* are built on top of DAMS by implementing corresponding pairs of these two components.

III.B The Component-level Debugging Service

The Component-level Debugging Service in DAMS has been reported elsewhere [3], but is briefly described here. It includes primitives to deal with the target application processes individually, and can be grouped into:

1. *Debugging Session Control* primitives, to start and terminate the debugging session.
2. *Process Execution Control* primitives, to start, interrupt, resume and terminate the execution of application processes.
3. *Process Internal State Inspection and Modification* primitives, to read and/or change the process internal state, such as inspection of call stack or modification of program variables.
4. *Event Processing and Management* primitives, to provide support for multiple client tools (debugging interfaces) to have a coherent view over the target application.

III.C The Coordination-level Debugging Services

Many coordination-level (debugging) services can be developed independently, and the DAMS architecture allows their incremental integration into the prototype. The *PVM Replayer* described below, is an example of such a (debugging) service.

IV REPLAYING PVM PROGRAMS

Even when, for a given input, the output of a distributed application is deterministic, it doesn't mean that the interactions between the multiple application components were the same. In fact, and due to race conditions that are resolved in different ways, these interactions can be different for each run in most of the applications. If a non-expected behavior is observed during one of these (*uncontrolled*) runs, there is the need to debug the application.

When debugging the distributed application in the most conventional way, it can be very difficult for the user to force (select) the right sequence of process interactions that lead to the bug previously observed during the initial *free* run.

To achieve this buggy application state, one of the following approaches can be used:

1. *Manual control.* This is the typically available approach, where the user has to determine the right sequence of process interactions and then, using a debugger, control the behavior of each process in order to reach the desired (buggy) application state.
2. *Pseudo-automatic control.* The user has to determine the right sequence of process interactions and writes a specification file with this set of intermediate states. A tool reads this specification file and executes the application under its control, forcing each of the intermediate states to be reached, eventually reaching the desired state. Then, control is passed to the user, to initiate the interactive debugging phase.
3. *Automatic control.* An automated tool determines a valid sequence of intermediate states that lead the application to the desired state, and then the technique described in (2) is applied. This specification file can be generated by a multitude of tools, such as static or dynamic testing tools [13], [17], or by a monitor that traces all the processes interactions during the application (as presented in this paper).

The re-execution of a parallel or distributed application requires a two-phase process:

- Phase 1: *Recording*
  In this phase, information is collected on each node, in order to generate a trace file containing the information necessary to define a partial order between all the interaction events (*e.g.* message interchanges) occurred during the application execution. In [14] is shown that it is enough to collect information about the non-deterministic message receive operations to be able to replay later all the communication events in the same (partial) order.
- Phase 2: *Replaying*
  During replay, the execution of the target application will be driven by the contents of the trace file, and the same ordering of relevant events is imposed by the replaying system.

IV.A "RPVM": a PVM Replayer

In the case of PVM, it is enough to trace the behavior of the functions presented below:

- "`pvm_recv()`" — that receives a message;
- "`pvm_nrecv()`" — that receives a message if one is available;
- "`pvm_trecv()`" — that receives a message within a specified timeout;
- "`pvm_precv()`" — that receives a uniform message (a set of identical elements) directly into a user specified buffer;
- "`pvm_spawn()`" — that launches new PVM tasks.

To trace these PVM primitives, we have designed the RPVM system that is composed by:

1. *The* "`rpvm3.h`" *header file*, that redefines the PVM primitives presented above, to have their behavior recorded in the trace file.
2. *The* "`lib??rpvm3.a`" *library file*, that includes the implementation of the redefined PVM primitives, and should be linked to each of the application processes (the "`??`" in the library name will be instantiated below).

IV.A.1 How to prepare an application to use RPVM?

In RPVM it is assumed that the application to trace has a main process that will spawn all the other application processes. If this is not the case, and the application is composed by two or more possibly distinct processes ($P_1, P_2, \ldots, P_n$) started independently, than another process P_0, whose only function is to spawn those process must be developed and should be used to start the application.

To use RPVM, there is the need to apply some minimal changes to the application:

1. Replace
   ```
   #include <pvm3.h>
   ```
   with
   ```
   #include <rpvm3.h>
   ```
   in all the application source files;
2. Add the library
   ```
   lib??rpvm3.a
   ```
   immediately before the standard PVM library
   ```
   libpvm3.a
   ```
   to all the application programs;

Recompile the application and it will be ready to use RPVM. However, the same interface and functionality will be available for two different implementations of RPVM:

- *The* RPVM *as a stand alone system*
  This version is the only one fully operational at the moment and is available by using the library file
  `libsarpvm3.a`
  in item 2 above.
- *The* RPVM *as a* DAMS *service*
  This version is currently under development, but implements RPVM as a service on top of the DAMS system, and is available by using the library file
  `libdmrpvm3.a`
  in item 2 above.

IV.A.2 How to run an application using RPVM?

An application linked to the RPVM system can be ran in three modes:

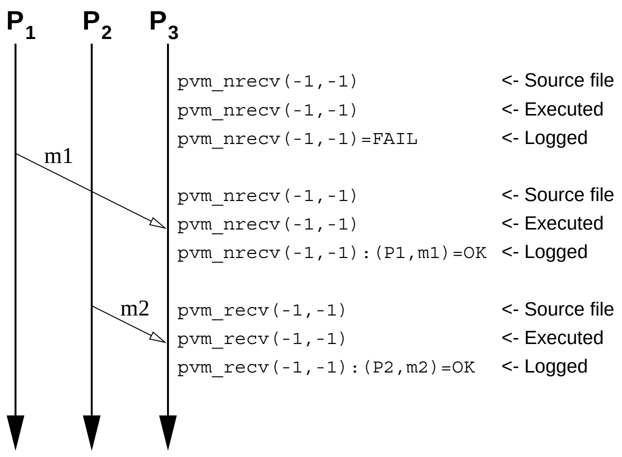
1. *Record mode.* If the application is launched with the command line argument(s)
   `-wlog <log_file_name>`
   the relevant PVM primitives (these primitives were listed above, in the beginning of Sec. IV.A) will be traced and their behavior logged in the specified trace file;
2. *Replay mode.* If the application is launched with the command line argument(s)
   `-rlog <log_file_name>`
   the behavior of the relevant PVM primitives will be driven by the contents of the specified trace file;
3. *Normal mode.* If none of the above RPVM specific command line arguments are given, the application will run as a conventional PVM application, without tracing or control from the RPVM system. In this case, the overhead of having the application linked with the RPVM library is minimal and can be ignored, as the access to the right set of functions for each running mode only requires an extra single access to a table in memory.

In both cases (1) and (2) above, the "`-wlog...`" and "`-rlog...`" options must be the first ones in the command line, immediately after the program name and before any other command line arguments the program may expect. These options will be automatically removed from the command line, so the programmer should assume they are never present. Currently it is assumed that the trace files reside in the "`/tmp`" directory, but this will be changed in a near future by using a configuration file for RPVM.

IV.A.3 How does RPVM work?

The file that replaces the standard PVM include file contains wrappers to all the relevant PVM functions (listed in Sec. IV.A) and to the C "`main()`" function. It changes the user code to invoke alternative implementations in the "`lib??rpvm.a`" library that behave as described below:

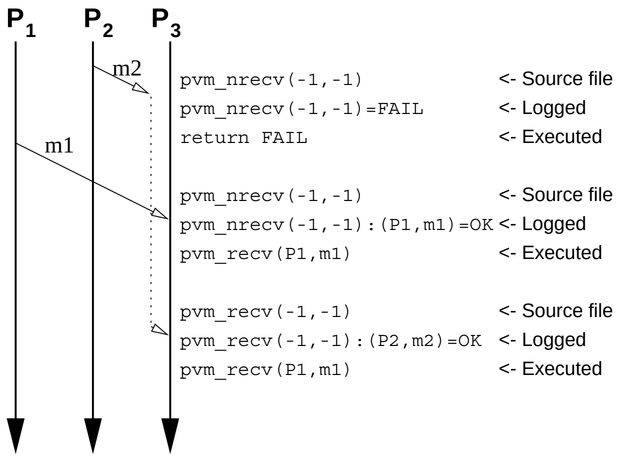
- `main()`
  Detect the running mode (*Normal*, *Recording* or *Replaying*) and determine which function should be called for each of the wrapped functions. If the application in being executed in *Normal* mode, the original PVM function will be called and the replacement functions described below will be ignored, otherwise they will be called and will behave according to the execution mode.
- `pvm_?recv()`
  If in *Record* mode, call the original PVM function and then record the arguments and the results of the call, including its exit status: *success* or *failure* (the "`?`" in the function name stands for all the variants of the PVM receive function).
  In Fig. 2 is shown the contents of the source code file of process P_3, what's really executed and what's logged in the trace file, given a race condition in the reception of the messages sent by processes P_1 and P_2.



P1
P2
P3
m1
m2
pvm_nrecv(-1,-1) <- Source file
pvm_nrecv(-1,-1) <- Executed
pvm_nrecv(-1,-1)=FAIL <- Logged
pvm_nrecv(-1,-1) <- Source file
pvm_nrecv(-1,-1) <- Executed
pvm_nrecv(-1,-1):(P1,m1)=OK <- Logged
pvm_recv(-1,-1) <- Source file
pvm_recv(-1,-1) <- Executed
pvm_recv(-1,-1):(P2,m2)=OK <- Logged

Fig. 2. Recording the application behavior

When in *Replay* mode (see Fig. 3) look at the status of the logged operation. If it failed when logging, return immediately with the same status simulating a call failure (like the first call to receive a message in Fig. 3), otherwise call the adequate PVM function with the arguments adapted according to the analysis of the log file (note that the second non-blocking receive has been changed to a blocking receive and the "*ANY* (-1)" flags were changed to the right process and tag identifiers) and compare the obtained results with the logged ones.



P1
P2
P3
m2
m1
pvm_nrecv(-1,-1) <- Source file
pvm_nrecv(-1,-1)=FAIL <- Logged
return FAIL <- Executed
pvm_nrecv(-1,-1) <- Source file
pvm_nrecv(-1,-1):(P1,m1)=OK <- Logged
pvm_recv(P1,m1) <- Executed
pvm_recv(-1,-1) <- Source file
pvm_recv(-1,-1):(P2,m2)=OK <- Logged
pvm_recv(P1,m1) <- Executed

Fig. 3. Replaying the application behavior

- `pvm_spawn()`
  The behavior is similar to the described above for the

"`pvm_?recv()`" functions, except that a new trace file is created/opened for each new process spawned.

If in *Record* mode, a new trace file is created for each spawned process, with the corresponding PVM *task ID* as the log file name. The main difficulty in handling the dynamic spawning of processes in PVM is the fact that the generation of task identifiers changes from one execution to another in a system dependent way.

If in *Replay* mode, an heuristic is used to map the new *task IDs* of the spawned processes to the ones used when recording the session. For easy of experimentation, a simple heuristic has been used in the first prototype:

> If the mapping between old and new task IDs of the sender process is already known, the correct task ID will be used. Otherwise just wait for a message from any task with the same message tag as the one received when recording.

It is known that in some cases this heuristic doesn't work adequately, but this situation will be detected and the execution replay aborted. When this situation arises, the application cannot be reexecuted with the used heuristic and there is the need to change the RPVM configuration file to use a different heuristic/method.

A new method, using *symbolic process identifiers*, that always assures the right mapping between the recorded and the current process identifiers is currently under development:

– If in Record mode, the behavior is the same as above, except that less information needs to be logged, namely only the PVM task IDs should be saved. The sender task ID is saved on message reception and the child task IDs are saved on process spawning.
– At the end of the execution in Record mode there is enough information to build the dynamic process tree, that must used to reproduce the execution during the replaying phase. Based on this information, unique symbolic process identifiers are generated for each process and are made globally accessible to the processes when in Replay mode. On each process spawn the corresponding entry in the dynamic process tree is updated with the actual PVM task ID. If this ID is not yet available on a message receive, the receive operation suspends until the corresponding process is created.

When compared to the method described in Sec. IV.A.3, this method has the advantage of providing a complete mapping between symbolic and real process identifiers, however it requires the access to a global database that must be dynamically updated.

V Conclusions and Ongoing Work

As a continuation of our research on parallel and distributed systems, this paper has described the design and implementation of a *Replay service* for PVM distributed applications, and reported on its application on parallel debugging. We have shown how this simple but effective replay mechanism can be supported as a stand alone tool on top of PVM, without requiring modifications to the PVM system source code.

Ongoing work includes the full implementation of the symbolic process identifiers mechanism and its performance evaluation, and the experimentation with alternative designs for the replay mechanism using the DAMS software architecture.

Acknowledgements

Thanks to João Vieira, Bruno Moscão e Daniel Pereira for their work in the DAMS system.

The work reported in this paper was partially supported by the Portuguese CIÊNCIA Programme in the PRAXIS XXI SETNA-ParComp (Contract 2/2.1/TIT/1557/95).

References

[1] Richard H. Carver and Kuo-Chung Tai. Replay and testing for concurrent programs. *IEEE Software*, 8(2):66–74, March 1991.

[2] Jong-Deok Choi and Janice M. Stone. Balancing runtime and replay costs in a trace-and-replay system. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 26(12):26–35, December 1991.

[3] J. C. Cunha, J. Lourenço, J. Vieira, B. Moscão, and D. Pereira. A framework to support parallel and distributed debugging. In *Proceedings of the International Conference on High-Performance Computing and Networking (HPCN'98)*, Amsterdam, The Netherlands, April 1998.

[4] T. Delaitre, G. Justo, F. Spies, and S. Winter. A graphical toolset for simulation modeling of parallel systems. *Parallel Computing Journal*, 22:1823–1836, 1997.

[5] A. Fagot and J. Chassin de Kergommeaux. Optimized record-replay for rpc-based parallel programming. In *Working conference on programming environments for massively parallel distributed systems*, pages 347–352, Ascona, Switzerland, April 1994. IFIP, WG10.3, Birkhaeuser, Basel.

[6] Eddy Fromentin, Noël Plouzeau, and Michel Raynal. Replaying distributed executions. In Mireille Ducassé, editor, *AADEBUG, 2nd International Workshop on Automated and Algorithmic Debugging*, pages 1–18, Saint Malo, France, 22–24 May 1995. IRISA-CNRS.

[7] L. Hluchý, M. Dobrucký, and J. Astaloš. Hybrid approach to task allocation in distributed systems. In *Lecture Notes in Computer Science*, volume 1277. Springer, 1997.

[8] Robert Hood. The p2d2 project: Building a portable distributed debugger. In *Proceedings of the 2^{nd} Symposium on Parallel and Distributed Tools (SPDT'96)*, Philadelphia PA, USA, 1996. ACM.

[9] Kacsuk. Macrostep-by-macrostep debugging of message passing parallel programs. In *IASTED PDCN'98*, Las Vegas, USA, 1998.

[10] P. Kacsuk, J. C. Cunha, G. Dózsa, J. Lourenço, T. Fadgyas, and T. Antão. A graphical development and debugging environment for parallel programs. *Parallel Computing*, 22(1997):1747–1770, 1997.

[11] P. Kacsuk and T. Dózsa, G.and Fadgyas. Designing parallel programs by the graphical language GRAPNEL. *Microprocessing and Microprogramming*, 41:625–643, 1996.

[12] B. Krawczyk, H.and Wiszniewski. *Software Enginneering for Parallel and Distributed Systems*, chapter Interactive Testing Tool for Parallel Programs, pages 89–109. Chapman & Hall, 1996.

[13] H. Krawczyk and B. Wiszniewski. Interactive Testing Tool for Parallel Programs. In P. Crolll Chapman & Hal: I. Jelly, I. Gorton, editor, *Software Engineer for Parallel and Distributed Systems*, pages 98–109, London, UK, 1996.

[14] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Cumputers*, C–36(4):471–482, April 1978.

[15] Eric Leu and Andre Schiper. Execution replay: A mechanism for integrating a visualization tool with a symbolic debugger. In *Proceedings of CONPAR '92*, pages 55–66, Lyon, France, September 1992.

[16] L. J. Levrouw and K. M. R. Audenaert. Reducing the space requirements of instant replay. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 205–207, San Diego, California, May 1993. [Extended abstract].

[17] J. Lourenço, J. C. Cunha, H. Krawczyk, P. Kuzora, M. Neyman, and B. Wiszniewsk. An integrated testing and debugging environment for parallel and distributed programs. In *Proceedings of the 23^{rd} Euromicro Conference (EUROMICRO'97)*, pages 291–298, Budapeste, Hungary, September 1997. IEEE Computer Society Press.

[18] E. Luque, R. Suppi, T. Margalef, J. Sorribes, P. Hernandez, E. Cesar, M. Serrano, J. Falguera, and C. Ortet. Simulation of parallel systems in SEPP & HPCTI. In *Proceedings of the 3rd SEIHPC Workshop*, Madrid, Spain, January 1998.

[19] Milon Mackey. Program replay in PVM. Technical report, Hewlett Packard, Concurrent Computing Department, Hewlett Packard Laboratories, May 1993.

[20] R. H. B. Netzer and B. P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. *The Journal of Supercomputing*, 8(4):371–388, 1995.

[21] M. A. Ronsse and L. J. Levrouw. On the implementation of a replay mechanism. *Lecture Notes in Computer Science*, 1123:70–??, 1996.

[22] Dolphin ToolWorks. *TotalView*. Dolphin Interconnect Solutions, Inc., Framingham, Massachusetts, USA.

[23] Larry D. Wittie. Debugging distributed C programs by real time replay. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):57–67, January 1989.