A Distributed Execution Replay Facility for CHORUS

authors : **Martin Herdieckerhoff and Frédéric Ruget**

project :

distribution :

keywords : CHO,MES

abstract : A debugger for distributed applications running on top of the CHORUS distributed operating system is described. The debugger provides a comprehensive execution replay facility with a powerful user interface. The execution replay facility is one of the most useful features a distributed debugger can provide. The architecture of the debugger and its implementation are described, and a comparison is drawn with other published work on distributed debugging. The debugger itself has been implemented as a standard distributed application on top of the CHORUS micro-kernel.
This paper was presented at the 7th international conference on parallel and distributed computing systems, Las Vegas, Nevada, October 1994.

Copyright © Chorus systèmes, 1994

A Distributed Execution Replay Facility for CHORUS*

Frédéric Ruget
Chorus Systèmes – 6, av Gustave Eiffel
78182 Montigny le Bx – FRANCE

Abstract

A debugger for distributed applications running on top of the CHORUS distributed operating system is described. The debugger provides a comprehensive execution replay facility with a powerful user interface. The execution replay facility is one of the most useful features a distributed debugger can provide. The architecture of the debugger and its implementation are described, and a comparison is drawn with other published work on distributed debugging. The debugger itself has been implemented as a standard distributed application on top of the CHORUS micro-kernel.

1 Introduction

An important challenge in the area of system development is to provide the user with a powerful development environment. For instance, a good set of debugging tools is essential. The CHORUS distributed operating system [4] currently provides the user with two specific debuggers: the KDB tool [3] for debugging the kernel itself, and an enhanced version of GNU's GDB tool [30] for debugging multi-threaded programs running on top of the kernel [1]. However, CHORUS does not provide specific tools for debugging truly distributed applications. This paper describes the CDB tool, a debugger for distributed applications running on top of the CHORUS kernel, with a comprehensive execution replay facility.

The execution replay facility is one of the most useful features a debugger for distributed applications can provide [15]. Indeed, it allows the user to apply the traditional cyclic debugging approach for debugging distributed applications that exhibit a non deterministic behavior. Not surprisingly, the literature describes many such a replay facility: [13, 14, 29, 6, 23, 32, 11, 10, 25].

CDB itself is a standard distributed application running on top of CHORUS. However, we have had to slightly enhance CHORUS because we were lacking kernel support in two cases: (1) CHORUS did not allow for dynamically interposing debugger's code at a debuggee's system call interface, and (2) CHORUS did not provide for reporting asynchronous monitoring events to a debugger. The specification and implementation of the additional kernel support [8, 26] has been done in the context of the OUVERTURE project [2], and the additional support will be integrated with future standard releases of the CHORUS micro-kernel.

The remainder of this paper is organized as follows. Section 2 introduces CDB's user interface and the abstractions that are provided to work with the execution replay facility. Section 3 gives an overview of CDB's architecture. Section 4 is a detailed description of an important component of CDB: the global knowledge manager. Section 5 is devoted to particularly interesting implementation points and gives a few performance results. We draw a comparison with other distributed debuggers in Sect. 6. We conclude in Sect. 7.

2 User interface and debug sessions

Although CDB does not (yet) have a graphical user interface, we have tried to make it as "user-friendly" as possible. CDB provides the user with a shell similar to GNU's GDB's shell [3]. The shell provides history browsing and intelligent completion on both command name and command arguments.

*This paper was presented at the 7th international conference on parallel and distributed computing systems, Las Vegas, Nevada, October 1994.

[1] Microtec's XRay debugger [20] is also currently being ported to CHORUS. It will enable advanced host-target kernel debugging.

[2] OUVERTURE is european ESPRIT project 6603.

[3] Indeed, CDB's shell is currently implemented on top of GNU's `readline` library.

The debuggee applications that CDB handles consist of a dynamic set of CHORUS actors distributed over a fixed set of CHORUS sites. A CHORUS site is a set of tightly coupled computing resources (usually a machine). A CHORUS actor is similar to (though simpler than) a UNIX process: it includes an address space, sequential threads of execution, and communication ports. An actor is designated by a a *Unique Identifier* (UI), which is unique in space and time across the whole CHORUS network. A thread is designated by a *local identifier* (LI) that identifies the thread within the actor to which it is tied. A communication port is designated both by a LI that identifies the port within the actor to which it is attached, and by a UI [4]. In the remainder of the paper we will call *debuggee objects* (resp. *CHORUS objects*) the sites, actors, thread and ports that do (resp. that do not) belong to the debuggee application.

CDB assigns symbolic (i.e. ASCII) names to all CHORUS objects. The names are specified explicitly by the user, or automatically by the debuggee application via a special system call. CDB organizes the CHORUS objects in a hierarchy similar to a file system. Actors are viewed as directories of threads and ports. Sites are viewed as directories of actors. Thus the user may designate any debuggee object with a symbolic "pathname", and CDB provides completion on these pathnames. There is also a notion of current working object and most commands usually refer to the current working object (e.g. the `lo` command invoked without argument lists the objects within the current working object). The commands `pwo` and `co` respectively print and change the current working object.

Along the lines of emacs [31], CDB is internally based on a micro-LISP interpretor. This should give the user more flexibility for entering complex commands or defining its own "macros". However, CDB normally hides the LISP syntax at the user interface level, and CDB's commands look like ordinary shell commands.

CDB provides many of the common facilities of traditional debuggers. It can set breakpoints and query the state of a debuggee actor. It can do symbolic disassembling [5], display back-traces of stack frames, read and write an actor's address space.

[4] The UI is used as the port's IPC address.

[5] However CDB does not yet provide source level debugging. This will be implemented in a future version, probably in the form of a cooperation between CDB and GDB

In addition, CDB provides a comprehensive execution replay facility. This facility applies to applications with the following characteristics: they consist of a dynamic set of (debuggee) actors distributed over a fixed set of mono-processor sites (multi-processors are discussed in Sect. 6.1). Threads within the debuggee actors are scheduled preemptively. They may communicate with themselves and with the application's environment via the CHORUS IPC (using reliable and/or non-reliable primitives). Threads residing on the same site may also communicate via shared memory. However, CDB puts the restriction that debuggee threads do not share memory with the application's environment or with debuggee threads residing on other sites (virtual distributed shared memory is discussed in Sect. 6.1).

The code of the debuggee actors needs not be recompiled or re-linked for the re-execution facility to work [6].

The execution replay facility is based on the notion of *debug session*. A debug session includes a fixed set of sites and simulates a virtual CHORUS network with special properties. When an application executes within a *record* debug session, it's behavior is automatically recorded in a *session log*. When it executes within a *replay* debug session, its behavior is constrained to match a previously recorded session log. The replay session also simulates the interactions with the application's environment that may possibly have occurred at record time. It is possible to run several replay sessions of the same session log simultaneously. These replay sessions won't interfere, although the several application's "clones" all use the same UIs and LIs.

Debug sessions are also part of the file-system-like hierarchy of CHORUS objects: they are viewed as directories of sites. A special `default` debug session is provided that includes all CHORUS objects known to CDB (monitored or not). It enables the user to designate objects to be inserted in other regular debug sessions. The `default` session itself does no recording and no replaying.

To create a record session, the user must specify a set of sites and a symbolic (i.e. ASCII) name which is used to identify the session in the file-system-like hierarchy of CHORUS objects. Once the session is created, the user may load actors in the session's site, and their behavior will be recorded in the session's

[6] With a slight exception related to the implementation of the instruction counter (see 5.2).

```
> # Print current working object
> pwo
    /default    # the default session
> # List sites in the default session
> lo
    becarre  bemol  diese
> # Create record session "mysession"
> # with sites "becarre" and "bemol"
> session mysession becarre bemol
> # List sites within created session
> lo /mysession
    /mysession/becarre   /mysession/bemol
> # Load some actors in the session
> load server site /mysession/becarre
> load client site /mysession/bemol
> # Wait for termination of the session
> # then use the session's log to replay
> # execution in session "mysessionreplay"
> replay mysession mysessionreplay
```

Figure 1: Creation of a session

log. To create a replay session, the user must specify a previous session's log and a symbolic name for the replay session. This is illustrated in Fig. 1.

The actors *inside* the debug session represent the debuggee application. The actors *outside* the debug session represent the application's environment. The debug session may dynamically extend when new actors are "forked" by the application's threads.

At record time a debug session produces one sequential event log per (mono-processor) site [7]. Tools are provided that analyze the log's contents. The simplest one produces an ASCII human readable listing of a log's events. The PARTAMOS tool [8] [28] uses the logs to draw space-time diagrams showing the causal relationships between events.

During execution replay, the user can still set breakpoints and query the actors' data. S/he can also control the pace of the re-execution: Execution is replayed at normal speed, or context switch by context switch, or step by step. In step by step mode, the user (or CDB) chooses an appropriate thread [9] and makes it execute a single machine instruction. In context switch by context switch mode, the user (or CDB) chooses an appropriate thread and makes it run for a whole "schedule" (i.e. up to the point where the thread was preempted or did block).

In a future version of CDB the debug session will also be the unit of checkpointing, that is, the user will have the possibility to take and develop consistent snapshots of a whole debug session. CDB already provides a way to determine a consistent snapshot of a distributed debug session by using a marking algorithm similar to the "colored" algorithm described in [18] which is a generalization of Chandy and Lamport's well known algorithm [2]. However the difficulty lies not so much in guaranteeing consistency as in saving and restoring the application's local states (at each site). The problem is that the local states may include complex kernel state [17, 21], such as the fact that a given thread is blocked in an RPC transaction. We are currently investigating ways to solve this problem by combining the restoration of a local state without kernel state and the re-execution of the events leading to the recorded kernel state.

3 Architecture of CDB

CDB consists of one *remote module* per site under control of the debugger. Each remote module monitors and possibly modifies the behavior of the debuggee actors that execute on its local site. The remote modules cooperate to maintain global knowledge about the debuggee application. For example, they compute the (dynamic) set of unique identifiers (UIs) assigned to debuggee objects. They also cooperate with *storage modules* to write the sessions' logs to stable storage. The storage modules may or may not reside on the same sites as the control modules. Finally, a *user interface module* provides a shell for the user and organizes the work of the remote modules.

The most interesting part of CDB is the remote modules. Each remote module consists of a number of managers:

- The *global knowledge manager* (KM). The KMs cooperate to replicate debug information concerning the debuggee application at each debuggee site. They implement a kind of replicated database. For instance, they replicate the set of all UIs assigned to debuggee objects. This enables CDB to make the difference between inside objects (i.e. the debuggee application it-

[7] On multiprocessors, there would be one log per processor.

[8] Still to be ported on CHORUS.

[9] There may be several appropriate threads, because threads execute concurrently and the events in the session's log are only partially ordered.

self) and outside object (i.e. the application's environment).

In a future version of CDB, the KMs will also replicate the set of sessions' logs [10]. This will make CDB's log recording tolerant to site crashes.

- The *logical time manager* (TM) is part of the global knowledge manager. It implements a matrix clock [33] that represents the knowledge the remote modules have about each other. This aspect of CDB will be optimized in future versions, for the sake of scalability.

- The *scheduling manager* (SM) is responsible for precisely recording and reproducing the scheduling of the debuggee threads. It is based on an instruction counter.

- The *debuggee manager* (DM) interposes its code at the system call interface of the debuggee actors [12]. The DM provides the debuggee actors with the illusion of an independent virtual CHORUS network. It makes it possible to run several execution replays of the same debug session simultaneously, by translating identical virtual UIs and LIs (as seen by the application's clones) into real different UIs and LIs (as seen by the kernel).

- The *log manager* (LM) is responsible for writing the sessions' logs to stable storage. It cooperates with a storage module.

This architecture is illustrated by Fig. 2.

4 The global knowledge manager

During the execution of a debuggee application, the CHORUS kernel assigns Unique Identifiers (UIs) to various debuggee objects (actors, ports and groups). Since UIs are unique in time and space the kernel will assign different UIs to the debuggee objects at replay time. However, the replayed application "believes" that the UIs are still the same. A translation is thus needed between UIs as seen by the application and UIs as seen by the system.

[10] More precisely, only the portion of the sessions' logs that has not yet been written to stable storage will be replicated.

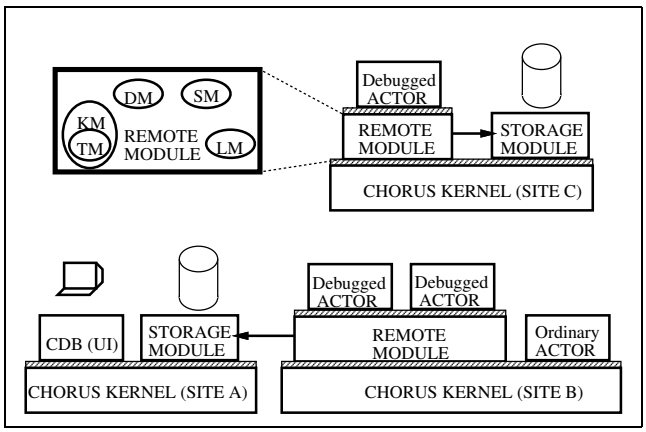

Figure 2: Architecture of CDB

An important role of the global knowledge managers (KMs) is to perform this translation in a efficient way. For that purpose, the KMs maintain a replicated set of all UIs assigned to debuggee objects. This set also enables CDB to know if an outgoing message is destined for a debuggee port (then it is tagged with debug information) or for the application's environment (then it is not tagged).

In a future version, the KMs will also replicate the session logs until they are put to stable storage. Thus if a site crashes, the remaining remote modules will be able to retrieve a consistent view of the session logs, and execution replay will be enabled up to crash.[11]

These two roles of the KMs are in fact pure instances of two theoretical distributed database problems known as the replicated dictionary and the replicated log problems [7, 33]. Indeed, the KMs are based on algorithms similar to those described in [33].

For example, in order to maintain the replicated set of session logs, each debuggee message is tagged with the view the remote module on the sending site has of the session logs. This enables the diffusion of the logs to all sites. To avoid diffusing information already known to other sites, the remote modules use a matrix clock that is a conservative approximation of the knowledge the remote modules have of each other.

However this strategy is not applicable when a debuggee message is sent outside the debuggee application, because then it is not possible to tag the message with debug information. To handle that case, we perform an operation similar to a "commit": the remote

[11] More precisely up to the minimal global state of the application [1] associated with the crash.

module on the sending site first broadcasts its view of the session logs and waits for acknowledgments from all other modules, before it actually allows the message to be sent.

The algorithm for the replicated set of UIs is similar. These distributed algorithms tolerate arbitrary delayed or lost messages, or even site crash failures. At the same time they are not very intrusive: they do not request additional control messages to be sent. Their potential weakness resides in the size of the message tags. In the current implementation, each tag includes a matrix clock which size is the square of the number of sites in the debug session. This is not really a problem since a debug session will hardly ever include more than three or four sites. However, the algorithms can be optimized to become more scalable as is suggested in [33] or done in [5]. The protocols described in [33, 5] trade tag size for additional control/acknowledgement messages. We plan to implement one of them in a future version of CDB(as described in [27]).

5 Some Implementation points

In this section, we detail the implementation of CDB's remote modules, and we show how we have benefited from our new CHORUS kernel support for monitoring and debugging [8].

5.1 Garbage collection

The remote modules are written in C++ and do a lot of dynamic object allocation. In order to free the memory that is no longer needed, we use a garbage collection (GC) mechanism based on reference counts. A debugger thread must increment the reference count of an object before it can access the object, thus "locking" the object in memory. It must decrement the reference count after it has accessed the object (in fact, this is done automatically by a clean C++ interface based on constructors/destructors).

A special case must be handled separately, when a debugger thread is deleted while it is currently locking objects. The reader may wonder why in the first place, a debugger thread would be deleted. The reason is that there are two kinds of threads that execute the debugger's code: true debugger threads, and debuggee threads that have been intercepted by the debuggee manager. While the former threads are never deleted, the latter may a priori be deleted.

To solve this problem, a possible solution is to mask thread deletion while a debuggee thread is running debugger code. In that case however, if the thread invokes a potentially blocking call, it must first release all the locks it holds and unmask thread deletion. This practice leads to a rather complicated programming model. In CDB we have chosen another way: thanks to the CHORUS kernel monitoring upcall mechanism, we can connect a deletion handler to all debuggee threads. When a debuggee thread is deleted, the handler automatically releases the locks held by the thread. The advantage of this method is that it is completely transparent to the CDB's programer.

5.2 Software instruction counter

In CHORUS threads are scheduled preemptively. To be able to record and replay the scheduling, CDB uses an instruction counter: at record time, CDB's scheduling manager (SM) counts the number of (atomic machine level) instructions executed by a debuggee thread until it is preempted. This information is used at replay time to reproduce the scheduling. Since appropriate hardware support was not available on our target machines (486DX PCs), we have implemented a software instruction counter along the lines of [19]: the debuggee code is instrumented so that a *branch counter* is incremented before each branch at the machine instruction level. The value of the branch counter is stored at a fixed address in memory private to the processor.[12] The pair (branch_counter, program_counter) gives information equivalent to a true instruction counter. It is important to realize that the instrumentation implementing the software instruction counter must be applied to the whole debuggee code, *including* libraries.

We have implemented the instruction counter on top of our kernel monitoring upcall mechanism [8]: a *run*, a *preempt* and a *sleep* handlers are connected to all debuggee threads. At record time, the run handler sets the value of the branch counter to zero, and the preempt and sleep handlers store the value of the branch and program counters in the session's log. At replay

[12] In [19], Mellor-Crummey and LeBlanc investigate the possibility of optimizing the implementation of the instruction counter by making the compiler dedicate a processor register for storing the counter. We have not implemented this kind optimization, because we want our instruction counter technique to apply to any piece of code, including pieces written in assembly language that already use all processor registers.

time, the run handler sets the value of the branch counter to minus the logged branch count. Then the thread runs until the branch counter reaches zero. At this point, an exception is raised that gives control to the SM. Then the SM sets a breakpoint at the address corresponding to the logged program counter. When the thread finally reaches the breakpoint, it has re-executed exactly the same number of instructions as during the record phase.

5.3 Identifying a debuggee thread

CDB's debuggee manager transparently interposes its own code at the debuggee system call interface [12]. When a system call is intercepted, the associated thread is "migrated" in CDB's remote module and begins executing the code of the DM [26]. However, the identity of the intercepted thread is not explicitly given to the DM.

Instead of invoking a system call to get the identity of the intercepted thread, we use the following (more efficient) way. A run handler is connected to each debuggee thread (see Sect. 5.2). When a debuggee thread is run by the CHORUS scheduler, the run handler is invoked with a parameter that is a pointer to the object that represents the debuggee thread within the remote module. The run handler stores this pointer in memory private to the processor. If the debuggee thread is later intercepted by the DM, the DM may retrieve the pointer by directly reading at the appropriate memory location.

5.4 Log manager

At each remote module, the log manager (LM) receives requests from its local clients (DM, SM, ...) to write event records onto stable storage. The LM cannot process the write requests synchronously, because it may be invoked while hardware interrupts are masked, or in the context of a monitoring upcall, at a time when it is allowed to re-enter the kernel only via a restricted set of system calls. Also, synchronous writing may not even be desirable because it might impact performance. Instead, event records are first put in a circular memory buffer [13] and later written to stable storage by an asynchronous thread (as in [25]).

[13] Thanks to CHORUS virtual memory, we could implement a truly circular buffer by mapping twice the same memory region consecutively !

The interface to the LM is the following. First the client of the LM requests the allocation of a chunk of the circular buffer of a given size. The LM always allocates chunks consecutively. Then the client may write the event record into the allocated chunk. Finally the client reports write completion to the LM, and eventually, the LM will wake up the asynchronous thread that flushes the circular buffer onto stable storage. The LM is capable of handling "simultaneous" requests (i.e. more than one write request before the first completion report [14]).

5.5 Performance

We have run some performance measurements on a network of PCs/AT with 486DX33 processors. These include micro- and macro-benchmarks to measure the overhead introduced by the new kernel support for monitoring and debugging, and an evaluation of the re-execution facility itself. The overhead associated with the new kernel support was found to be negligible (less than the standard deviation for the benchmark results).

The performance of the re-execution facility was found to depend only on the number of monitoring events recorded/reproduced. Average event size is 14 bytes (on the session's log). Average time to record an event is $300\mu s$ (including the actual writing onto stable storage). Average time to reproduce an event is $3ms$. Let us give examples of what this yields in practice. For an application that issues only deterministic system calls, the only recorded events correspond to thread preemptions/runs, i.e. around 30 events per second on our target machines. This yields an overhead of 1% for the recording phase and of 10% for the replay phase. For the average distributed application producing around 100 events per second, the overhead is 3% for the recording phase and 30% for the replay phase.

6 Related work

The literature describes quite a few debug tools that provide execution replay. In this section we compare

[14] This case may happen when a processor interruption generates an event or when a thread is preempted (generating a preempt event), while an event is currently being written to the circular buffer.

these tools and relate them to CDB. The tools differ in several aspects.

6.1 Targeted systems

Instant replay's re-execution facility [14] applies to a tightly coupled system where debuggee processes communicate via shared memory only [15]. On the other hand, Bugnet [13], Amoeba's debugger [6], EREBUS [10] or the CAC's debugger [25] apply to systems with distributed memory where debuggee processes communicate only via messages passing They assume a reliable network with FIFO channels [6, 10, 25].

CDB and Recap [23] handle both shared memory and message passing (CDB even assumes a non reliable network). Recap handles shared memory by detecting accesses to shared variables at compile time and by instrumenting the debuggee code to save/restore the values read. CDB handles shared memory by assuming mono-processor sites and precisely monitoring the threads' scheduling. This technique would generalize to multi-processor sites by having the processors share memory through MMU management only, and by precisely monitoring the page fault traffic [24]. By doing so, one effectively serializes all accesses to the same shared page, which might lead to serious performance degradation. In some cases however, an efficient implementation is possible, by piggy-backing on the overhead already paid by the underlying cache coherence protocol [22].

6.2 Targeted programs

Some replay tools assume programs written in a specific language : ADA programs in [29], Estelle programs for EREBUS [9], Guide programs for THESEE [11] or concurrent ML programs in [32]. The benefit of working at the language level is that it is possible to record execution on one kind of machine and replay it on another. Also, the replay tools that work at the language level can usually do with a coarser grain of monitoring [9] which results in execution replay being easier to implement. Distributed checkpointing may also benefit from a coarser grain of monitoring because intermediate program states associated with complex kernel state may be avoided [17].

[15] However the technique on which Instant replay is based generalizes to systems with distributed memory and message passing [16].

On the other hand, Amoeba's debugger or CDB take a lower level approach, and handle applications written in any language, as long as they run on a specific system (e.g. CHORUS or Amoeba).

6.3 Debuggee instrumentation

Amoeba's debugger or Recap rely on linking the debuggee program with special system call libraries. The drawback is that part of the debugger's code and data thus resides in the same address space as the debuggee and may be corrupted [6]. CDB avoids this drawback by not instrumenting the debuggee program [16] and using an interposition technique.

6.4 Interactions with the environment

The concurrent ML debugger or EREBUS handle debuggee programs that do a computation on their own without any reference to an environment. On the other hand, THESEE handles debuggee programs that access shared persistent GUIDE objects. The objects are part of the debuggee's environment, and their state must be reseted before each execution replay. For that purpose, THESEE automatically creates special copies of the objects that were accessed at record time, and replays the debuggee in a special *island mode* so as to prevent any interferences with the external world through these shared objects. An important restriction is that thus it is not possible to replay an application that communicates with the external world through shared object [11].

On the other hand, Bugnet, the CAC's debugger or CDB manage the interactions between the debuggee application and its environment by simulating the environment at replay time. For example, CDB records the contents of the messages received from the environment so that they can be recreated at replay time. This technique is called *data driven*. Usually, execution replay of interactions with the environment is data driven, while execution replay of events internal to the debuggee are *control driven*, so as to reduce the amount of logged information. This is the approach taken in [14, 6, 16, 11, 10, 25] [17]. However, some older

[16] With the exception of the software instruction counter.

[17] The EREBUS debugger even goes to using compression algorithms to make the logs smaller, as it needs to keep them in main memory and does not have the ability to store them onto disk.

tools are solely based on data driven execution replay [13, 23].

7 Conclusion

A debugger for distributed applications running on top of the CHORUS operating system has been presented. A first version of the debugger has been fully implemented. It provides a comprehensive execution replay facility with a powerful user interface. A future enhanced version will provide a distributed checkpoint facility. The performance of the debugger is acceptable, with an average 3% overhead for the record phase and a 30% overhead for the replay phase. CHORUS itself has proven to be an adequate platform for implementing the distributed execution replay mechanism, and we were able to implement the debugger as a standard CHORUS distributed application on top of a slightly modified kernel.

References

[1] M. Adam, M. Hurfin, N. Plouzeau, and M. Raynal. Distributed debugging techniques. Technical note, IRISA, 1991.

[2] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb 1985.

[3] Chorus Team. CHORUS kernel v3 r4.0 – debugger user's manual for COMPAQ deskpro386. Technical Report CS/TR-91-72, Chorus Systèmes, 1991.

[4] Chorus Team. Overview of the Chorus distributed operating system. In *USENIX Workshop on Micro Kernels and Other Kernel Architecthres*, Seattle (USA), 1992.

[5] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast ouput commit. *IEEE Transactions on Computers, Special Issue on Fault-Tolerant Computing*, 41(5):526–531, May 1992.

[6] I. J. P. Elshoff. A distributed debugger for amoeba. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 1–10. ACM, 1988.

[7] M. J. Fischer and A. Michael. Sacrifying serializability to attain high availability of data in an unreliable network. In *Proc. ACM SIGACT-SIGMOD Symp. on principles of database systems*, pages 70–75, Los Angeles, March 1982.

[8] M. Herdieckerhoff and F. Ruget. Matching operating systems to application needs – a case study. In **local** *Proc. of SIGOPS'94*, 1994.

[9] M. Hurfin, N. Plouzeau, and M. Raynal. Implementation of a distributed debugger for estelle programs. In *ERCIM workshop*, 1991.

[10] M. Hurfin, N. Plouzeau, and M. Raynal. Erebus: A debugger for asynchronous distributed computing systems. In *Proc. of 3rd IEEE Work. on Future Trends in Distribued Computing Systems*, Taipei, Taiwan, April 1992.

[11] H. J. Jamrozik, C. Roisin, and M. Santana. A graphical debugger for object-oriented distributed programs. In *Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 117–128, July, 1991.

[12] M. B. Jones. Transparently interposing user code at the system interface. In *Proc. of the 2nd Work. on Workstation Operating Systems*, April 1992.

[13] S. H. Jones, R. H. Barkan, and L. D. Wittie. Bugnet: a real time distribued debugging system. In *Proc. of the 6th Symp. on Reliability in Distributed Software and Database Systems*, pages 56–65, March 1987.

[14] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Computers*, C-36(4):471–481, April 1987.

[15] E. Leu and A. Schiper. Techniques de déverminage pour programmes parallèles. *Technique et Science Informatiques*, 10(1):5–21, 1991.

[16] E. Leu, A. Schiper, and A. Zramdini. Execution replay on distributed memory architectures. In *Proc. of 2nd IEEE Symp. on Parallel and Distributed Processing*, Dallas, December 1990.

[17] W. Lux, W. E. Kuhnhauser, and H. Hartig. The birlix migration mechanism. In *Workshop on Dynamic Object Placement and Load Balancing in Parallel and Distributed Systems Programs*, pages 83–90, June 1992.

[18] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 1993.

[19] J. M. Mellor-Crummey and T. J. LeBlanc. A software instruction counter. In John L. Hennessy, editor, *Proc. of Third Int. Conf. on Architectural Support for Programming Languages and Operating Systems, Boston, MA*, pages 78–86. ACM/IEEE, 1989.

[20] NewBits: A quarterly newsletter from microtec research, inc., 10(3), 1993.

[21] D. S. Milojicic, W. Zint, A. Dangel, and P. Giese. Task migration on the top of the mach microkernel. In *Mach III Symposium*, pages 273–289. Usenix association, 1993.

[22] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 235–244, Santa Clara, CA, April 1991.

[23] D. Pan and M. Linton. Supporting reverse execution for parallel programs. *SIGPLAN NOTICES*, 24, January 1989.

[24] J. Riotto. Software checkpointing. Part of UNISYS architecture task force monograph series.

[25] J. F. Roos, L. Courtrai, and J. F. Mehaut. Execution replay of parallel programs. In *Proc. of the Euromicro Work. on Parallel and Distributed Processing*, pages 429–434, 1993.

[26] F. Ruget. Actor-wide trap tables for CHORUS. Technical note, Chorus Systems, 1994.

[27] F. Ruget. Cheaper matrix clocks. In *Proc. of the 8th Int. Workshop on Distributed Algorithms (WDAG-8)*, Terschelling, the Netherlands, September 1994.

[28] M. Schiefert. PARTAMOS: Parallel real-time application monitoring system. Product sheet, Alcatel Austria–ELIN Research Center, Ruthnergasse 1–7, A1210 Vienna, 1991.

[29] D. Snowden and A. Wellings. Debugging distributed real-time applications in ada. Technical report, University of York, UK, April 1987.

[30] R. Stallman. The gnu debugger. Technical report, Free Software Foundation, Inc, 675 Mass. Avenue, Cambridge, MA, 02139, USA, 1986.

[31] R. Stallman. *GNU Emacs Manual.* Free Software Foundation, 1987.

[32] A. P. Tolmach and A. W. Appel. Debuggable concurrency extensions for standard ML. In *Proc. ACM/ONR workshop on parallel and distributed debugging*, pages 120–131, 1991.

[33] G. T. J. Wuu and A. J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proc. 3rd ACM Symp. on PODC*, pages 232–242, 1984.