DDB: A DISTRIBUTED DEBUGGER BASED ON REPLAY

J. Sienkiewicz
Software Engineer
Ericsson Research Canada
8400 Boulevard Decarie H3G 2H2
Ville de Mont-Royal (Quebec)

T. Radhakrishnan
Department of Computer Science
Concordia University
1455 DeMaisonneuve
Montreal H3G 1M8

Abstract

DDB is a distributed debugger that is based on the well known technique of "Instant Replay". The debugger is designed and implemented to operate under multithreaded Mach Distributed Operating System. It is complete in the sense that a graphical user interface, source code level debugging facility at the individual nodes of the distributed computation, and our experience in using the debugger are well documented. We provide, in this paper, a brief, but complete view of DDB from which the developers of new debuggers for parallel and distributed systems can benefit.

1 Introduction

1.1 Sequential vs Distributed Debugging

Distributed programs are more difficult to develop, test, and debug than sequential programs. There are several problems that a programmer must face when creating distributed systems. These problems greatly influence the debugging methods required for distributed debugging and hence different tools are required.

Non-determinism. Unlike sequential programs, due to several random factors, concurrent execution can take different paths each time the program is executed. The interaction between multiple processes taking part in the computation, and the possible race conditions present in the system cause this non-determinism. Non-determinism makes debugging difficult because the programmer can never be sure that the bug occurring in one execution will repeat itself, giving another opportunity for its examination. This problem makes it difficult particularly to detect bugs that occur very rarely.

Multiple threads of control. The level of difficulty is greatly increased when there are more than one thread of control participating in the computation. Multithreaded programs are more difficult to understand and write. It is also more difficult to follow their execution path. The interaction between threads operating on shared resources creates potential for new types of errors that would be absent in sequential programs. These errors include critical race and synchronisation problems.

Lack of precise global states. This is one of the fundamental problems of distributed debugging. Global clock synchronisation is also a classical research problem in distributed systems, but obtaining accurately synchronised global clocks alone does not guarantee the accurate global state of the system. Because of different communication delays, various speeds, and multiple Machine states, the collection of the global information can proceed at different rates. Also it is difficult to effect "control changes" on different processors at the same instant. For these reasons, only "approximate global state" can be obtained.

Complex patterns of interaction. Distributed programming introduces new types of bugs that have their origin in the interaction between multiple asynchronous processes. The interaction between processes can be data dependent or order (sequence of messages) dependent. Bugs of this type are often hard to reproduce, because they depend not only on input data, but also on the relative timing of interactions among processes.

Large state space. In most cases the data collected for the analysis of execution of a concurrent program creates a large state space. A composite state includes the machine state on a processor and a record of interaction between different processors, creating the problem of manipulating large quantities of state information to be viewed by humans while debugging.

Communication limitations. Unpredictable communication delays and limited communication bandwidth may make some of the debugging techniques (central manipulation of information) impractical.

Probe effect. Probe effect is defined as the undesired side-effects that the debugging activity itself can have on the program behaviour. This occurs when the debugging actions affect timing, and therefore the ordering, of events. The probe effect can hide otherwise present errors, or introduce new errors that do not normally occur when the program runs without a debugger. This problem is rooted in the non-determinism inherent in distributed programs. Elimination of the probe effect is not possible, but care should be taken to make it as minimal as possible.

0-7803-3529-5/96/$5.00

1.2 Instant Replay Technique

The Instant Replay method was introduced by Thomas J. Leblanc and John M. Mellor-Crummey in [2]. The mechanism of Instant Replay was demonstrated in a debugger prototype implemented on the BBN Butterfly Parallel Processor. The approach taken by the designers of the Instant Replay method treats all interactions between processes as operations on shared objects. Modifications to the objects are represented by totally ordered sequence of versions, where each version has a corresponding version number. This number is unique to each object. During the *monitoring* phase of the debugging, the partial order of the access to each object is recorded. This partial order is specified by a sequence of version numbers maintained for each object. To record the partial order, the debugging system maintains the current version of each shared object, and the number of readers of each version for each object. Each process executing as a part of the parallel program records the version number it accesses in the *monitoring* phase. In the *replay* phase this number is retrieved and used to ensure that the process sees the same value and in the same order it has seen in the original execution. The history information can be used as many times as necessary to repeat the original execution.

Instant Replay presents a general solution to the problem of non-determinism in concurrent debugging. This method can be applied to any concurrent system since it does not depend on any particular form of interprocess communication, and it does not require synchronized clocks or globally consistent logical time. One of the important advantages of this method, is the low overhead that the debugging system introduces. The overhead is small because only the order of relevant events is saved but not the data associated with the events. The time overhead is less than 1 percent, and the space overhead is also reasonable, making the technique possible to record events of even a large scale production system.

The disadvantages of this method include the fact that the replay must include all "related processes", not just the subset of processes suspected to contain the bug. In a time window, the processes interacting with each other are said to be related if they depend on each other's input to proceed with the computation. If one of such processes is to be replayed, all the related processes have to be replayed, because all input values have to be recalculated. This can be rather wasteful in the case of large systems. Another disadvantage of this method surfaces when the granularity of the communication is very small. Then the space used for recording the version numbers associated with shared objects could become prohibitively large.

Deterministic replay mechanism used in the DDB debugger is an adaptation of the method used in Instant Replay [2]. In the *record* phase of the DDB, just like Instant Replay, we collect control information about the execution. No data values associated with the user program are recorded. The debugger stores only access patterns to the shared variables within each task, and order of send and receive operations for each port used for interprocess communication (IPC).

1.3 Mach Distributed computational model

The Mach operating system, developed at Carnegie Mellon University (CMU), incorporates many innovations from operating system research. The key goal of Mach is to be a distributed system capable of functioning on heterogeneous hardware platforms. Mach programs are built from several basic abstractions:

- **Task :** collection of resources including virtual memory and communication ports. Tasks are passive, they do not run on a processor.
- **Thread:** active execution environment. Each task may support one or more threads; all threads have equal access to the task's resources. Each thread has private execution state that consists of a set of registers, such as general purpose registers, stack pointer, program counter, and a frame pointer.
- **Port :** communication channel - a logical queue of messages protected by the kernel. Ports are location transparent, and they can change owners. Access to ports is granted according to access rights (for receive, send, ownership). Ports are attached to tasks and any thread of that task is entitled to use that port to send or receive.
- **Message:** typed collection of data structures used for communication between threads. Messages may be of any size and may contain inline data, pointers to data, and capabilities for ports.

2 Other Debuggers that Existed Before

Table 1 presents information about functional characteristics of **twenty nine** distributed and parallel debuggers developed in the recent years. A complete reference to them can be found in Sienkiewicz's thesis [4]. The notations used in the table are as follows:

1. Is replay facility provided by the debugger ?
   (a) *complete* - entire program state is available
   (b) *commun* - communication state can be deduced
   (c) *none* - no replay facility available
2. What types of breakpoints are supported by the debugger ?
   (a) *global* - state breakpoint can be set
   (b) *local* - state breakpoint can be set

Debugger	*Replay*	*Type of Breakpoints*	*Breakpoint Effect*	*Event Type*
dbf Agora	complete	local, single	process	shmem
dbf Amoeba	complete	local, stmt, mult	either	ipc
CDB	complete	local	program	none
Instant Replay	complete	local, stmt	program	shmem
Recap	complete	local, stmt	process	ipc, shmom
RP3	complete	single	program	stmt
Bugnet	complete	none	none	ipc
Panorama	complete	none	n/a	none
ParaGraph	complete	none	n/a	stmt
PPUTT	complete	none	n/a	none
IGOR	complete	n/s	n/s	none
dbf ML	commun	stmt	program	none
VISIT	commun	stmt	program	stmt
IDD	none	global, mult	program	ipc
bdb	none	local	program	none
CXdb	none	local	program	none
ldb	none	local	program	none
LPdbx	none	local	process	none
MDB	none	local	n/s	none
MpD	none	local	n/s	n/s
UDB	none	local	program	none
IPD	none	stmt	process	none
NodePrism	none	stmt	program	none
Parasight	none	stmt	process	stmt
Ariadne	none	none	n/a	shmem
EBBA	none	none	n/a	stmt
HeNCE	none	none	n/a	stmt
MULTVISION	none	none	n/a	stmt
Voyeur	none	n/s	n/s	stmt

Table 1: Functional Characteristics

(c) *stmt* - breakpoint can be set at a source code level

(d) *single* - event occurrences can be used to specify breakpoints

(e) *mult* - breakpoints on occurrence of combination of events

3. What is the effect of breakpoint on the program ?

(a) *program* - the entire program is stopped

(b) *process* - one process is halted

(c) *either* - either one process or the entire program is halted

4. What event Types supported by the system ?

(a) *ipc* - explicit interprocess communication

(b) *shmem* - shared memory references

(c) *stmt* - each statement execution

The table entries are sorted according to the replay facility which reflects our belief that execution replay is a crucial feature that every concurrent debugger should possess. The replay facility allows for repeating the same execution sequence many times, eliminating the problem of non-determinism inherent in concurrent programs. This makes it possible to use other well known debugging techniques, like cyclic debugging, top-down analysis, breakpoint insertion, or single stepping.

Nine of the debuggers presented in this table provide "complete" replay facility. The "complete" replay means that in the replayed execution all the information about the program state is available, as opposed to the "commun" (communication) replay, provided by two of the debuggers, where only the history of the communication events is available for examination. The second type of replay is provided by the VISIT debugger, where the trace data about the high level events is gathered during the program run, and stored for later use. This limited trace information is used for offline visualization of program execution. Although not all the program state is available, this type of visualization can be useful for constructing a high level view of the execution, which can be used for choosing the granularity of analysis and recording.

The method used in providing the replay facility differs from one debugger to another. One of the popular methods is the Instant Replay, which is also used in the Panorama debugger. In the replay facility provided by the Bugnet debugger, the IPC messages exchanged between processes are saved during the monitoring phase in a log. In the replay phase the *send message* operations are not executed, and the *receive message* operations are replaced by reading the contents of the message from the log. In this way, during replay the user can run one or more processes, depending on his debugging needs. The disadvantage of this method is the overhead in terms of time taken in saving the messages, and space used for storing the data included in the messages.

The discouraging observation that must be made when looking at this table, is the fact that only eleven out of the twenty nine debuggers provide replay facility. The remaining eighteen debuggers ignore the problem of non-determinism. Although it does not mean that they are not good at all, these debuggers are missing a very relevant feature.

Other type of information presented in this table refers to breakpoint and their effects on the program execution. The most common type of breakpoint is the one denoted as "local", which is based on the state of the execution of a single process. Some of the debuggers provide only statement breakpoint, which allows only for breakpoint specification based on the location in the code. The global breakpoint, where the distributed program state is considered for breakpoint condition evaluation, is provided by only one of the debuggers included in this table (IDD). The effect that the breakpoint has on the program also varies from one debugger to another. The breakpoint can stop one process or the entire program. In some cases the user is able to choose the breakpoint effect according to his needs.

The last column in the table is concerned with types of events recognized by the debuggers. Some of the debuggers recognize interprocess communication events (marked "ipc"), or shared memory events (marked "shmem"). There are also debuggers that allow the user to define the events according to his needs. In such cases, execution of any statement can be defined as an event. Many of the debuggers in this survey do not support event based views of the debugged programs. This is the case with the debuggers that have been built by extending sequential debuggers.

3 Design Requirements of DDB

3.1 Non-determinism in OS/Kernel

The programming model used in Mach is built on the notions of *task* and *thread*. *Task* is a repository of resources. *Thread* is a light weight process that executes program instructions. A distributed program can consist of many tasks, working towards a common computational goal and communicating through message passing or shared memory. A task can contain one or more threads. Every thread belongs to a single task and may access all of that task's resources. The sources of non-deterministic program behaviour in such a programming model can be viewed at two levels:

- **Interprocess communication level.** The communication patterns can differ from one execution to an-

other. The order of arrival of messages at a port can be different. In the case of unreliable communication channels, some messages can get lost in one execution but reach the destination in another execution, which adds more complexity.

- **Thread interaction level.** In a multithreaded environment the additional source of non-determinism is the interaction between threads sharing the same resources, which include accessing the same memory locations or using the same interprocess communication primitives. When two threads access the same program variable, the access patterns might vary from one execution to another, creating different program behaviour.

We require that our DDB should take care of the above two sources of non-determinism.

3.2 Division of responsibility between the user and debugger

Debugging is a user centered activity wherein human knowledge and intuitions, assisted by the software system, play the key role in detecting the "bug". It is highly interactive activity in which the user performs the following tasks repeatedly, hopefully homing towards the place of the bug in the source code.

1. Halt the distributed program execution when one of the user specified breakpoint events occur [Breakpoint Specification, Program Monitoring].
2. Visually inspect the various state variables and conditions, using the dynamically collected history file [Visualization, Browsing, Navigation].
3. "Step" the program execution one step at a time, giving the user the opportunity to follow the sequence of changes that take place so that a mental model can be built [Stepping semantics in a distributed computation].
4. Rollback the program to the previously stored checkpoint and restart the execution, now using perhaps a "fine comb" in building the dynamic history file. For example, in DDB we have not included the Checkpoint and Rollback module, although they were included in our earlier attempt (CDB) in building a distributed debugger. As a result, the DDB user, when he wants to rollback the execution, the distributed program is started from the beginning. This aspect will limit the utility of the DDB to "not too large" programs.

In the interactive process of debugging, the above mentioned tasks are shared between the human user and the debugger system. As a design requirement for the simple minded DDB, we placed more responsibility on the user and **minimal responsibility** for the DDB. Thus, no new stepping semantics for distributed programs is included in DDB. The user has to manage with stepping facilities provided in the modified local or node-level debugger (GDB).

3.3 User interface requirements

Even for a simple minded debugger like DDB, a proper user interface is essential if it is intended to be useful in practice. We required that DDB must have a user interface based on the GUI principles. It should support at least two levels of abstracted presentation: (i) graphical display of the time-process diagram showing various threads, threads grouped into tasks, and messages exchanged between the various ports of the tasks; (ii) source code level presentation of the computation at the various nodes using multiple split-windows.

4 Architecture of DDB

4.1 Reusing the GDB from CDB project

The idea of introducing a local debugger into the structure of a distributed debugger has been demonstrated in the past in the design and implementation of the CDB debugger [5] [3]. The local debugger used in CDB as well as in DDB is an extension of the GDB debugger from Free Software Foundation.

The single threaded version of the Unix GDB debugger has been modified by Caswell and Black [1] to support multiple threads of Mach. The resulting enhanced GDB debugger (version 3.4) allows user to select any thread and analyse its execution during a debugging session. Designers have adopted the concept of the *current thread* to handle the selection of threads to be examined. Any thread can be selected as the *current thread* and the selection can be modified during execution. All the GDB commands that are used for examination and modification of a Unix process can be applied to the *current thread*. Some modifications to these commands include change of the *single step* command to step only the current thread while preventing all the other threads from running.

This modified version of GDB was adapted for the special needs of distributed debugging by C. Yep. [5]. The interface of the debugger was changed so that it uses IPC messages for communication, instead of standard input and standard output. With this adaptation the debugger can receive commands from a process executing on a remote host. Within the DDB debugger, the GDB plays the role of local debugger. The local debugger is used in the *replay* phase of debugging for extracting the source level information, which can be used to refine the high level view of the execution provided by the time-process diagrams.

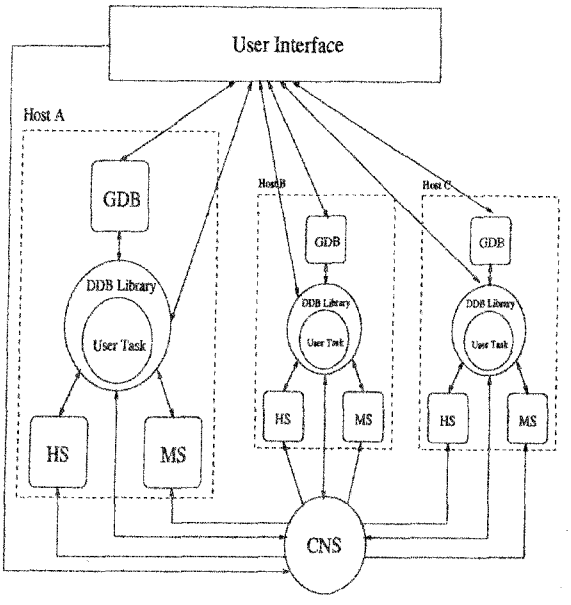

Figure 1: DDB system architecture

4.2 How Mach influenced DDB ?

The design and implementation of our DDB was influenced by the underlying components of the Mach environment.

The computational model of Mach operating system had a great influence on the architectural design of DDB as well as on the internal design of different modules of the debugger. The distributed nature of of Mach programs determined the design of DDB as a distributed system where debugging activities for each user task are supported by several independent debugger tasks. The internal design of DDB modules was often influenced by multithreaded capabilities of Mach programs. For example, the dynamic creation of threads within the user task results in creation of new threads within the History Server task, which is necessary for fast collection of the history data.

The design of the deterministic replay mechanism for DDB was based on our examination of different sources of non-determinism in Mach programs. Different non-deterministic scenarios for Mach system calls as well as for calls to the netmessage server were analysed. Based on the analysis, the appropriate information was collected during the *record* phase and stored in the history files for future use. In the *replay* phase of debugging, the previously stored information was used by the mechanism enforcing the deterministic replay.

C Threads is a C language interface library provided by Mach designers to facilitate creating multithreaded applications. This interface allows the programmer to create and use multiple threads without a need to handle low level operating system details. One of the benefits of using this library package is the availability of the synchronization primitives for controlling access to shared data by multiple threads. The C Threads library has been extensively used during implementation of the DDB debugger since all servers of the debugger are multi-threaded. We also make the assumption that the user programs handle thread related operations through the C Thread package. This assumption is particularly important in the case of synchronization variables, since DDB monitors the access to shared data through access to the C Threads by synchronization primitives.

4.3 The key modules of DDB

The architectural design of the DDB debugger is depicted in figure 1. Almost all modules of the debugger operate as separate multi-threaded tasks and communicate with each other via IPC messages. The exception is the DDB Library which is compiled with user program.

- **User Interface** - provides the user with easy access to all services offered by the debugger. It displays the time process diagrams and textual interpretation of contents of history files.
- **GDB** - the Modified GDB debugger is used for source level debugging during the *replay* phase. GDB is not invoked in the *record* phase.
- **DDB Library** - provides control over execution of Mach system calls and communicates with other parts of the debugger.
- **HS** - History Server maintains the control information related to execution of the user task. It stores the data received in *record* phase. In the *replay* phase it provides the data when it is requested.
- **MS** - Message Server is invoked only in *replay* phase. It intercepts messages sent to the user task ports and resends them in the order of the original execution.
- **CNS** - Central Name Server distributes identification numbers to the tasks participating in the computation in the *record* phase, and starts the server processes (HS, MS) for each user task.

The distributed architecture of the debugger follows the structure of the user program. Except for the User Interface and the Central Name Server, all the debugger modules are invoked for each user task in the application program. The History Server, Message Server, and GDB are always invoked on the same host the user task resides to make communication as efficient as possible. In case if there is more than one task executing on a host, the debugger servers will be duplicated, since each server can provide services to only one task. There is no communication among servers belonging to different tasks nor among different servers of the same tasks.

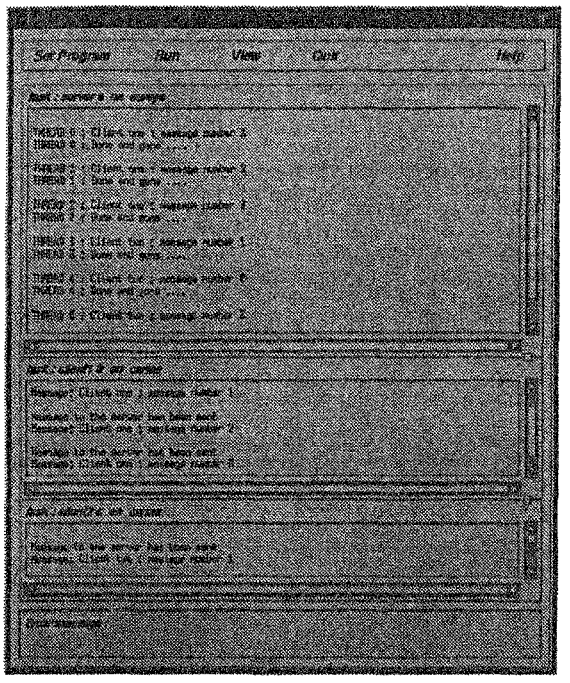

Figure 2: Main window of DDB

4.4 GUI

The graphical user interface for DDB was developed with the widgets of the OSF/Motif toolkit. The user interface provides easy access to all features of the debugger. The main window of the interface is presented in figure 2. The menu bar visible at the top of the window contains the following items :

- **Set Program** - allows the user to interactively set the names of the tasks participating in the computation and the names of the host machines that the tasks are to be run on. This eliminates the need for configuration files.

- **Run** - allows the user to start the program in either *record* or *replay* phase. The user starts each task of the program separately so that the appropriate order and delays can be applied.

- **View** - provides high level view (time-process diagram or textual interpretation of the history files) of the program execution based on the control information gathered during *record* phase. The user can choose to view the time-process diagram where each task is represented by events placed in the horizontal line according to the elapsed time. The textual description of system calls executed in the program can be accessed from the "View History Files" submenu. It gives more detailed information than the time-process diagram.

- **Quit** - terminates the DDB debugger.

- **Help** - gives access to help facility.

The area below the menu bar is used to display output of the user tasks during the *record* phase. In the *replay* phase it allows for communication with the base debugger controlling each task. During program execution, each task has a dedicated scrollable window. The windows are re-sizable to allow more space for tasks of interest.

The bottom part of the window is used for displaying error messages. This layout of the main window complies with the guidelines of the *Motif Style Guide*.

5 A Sample Debugging Session with DDB

This section gives step by step walk through of a debugging session.

Set up the program
To set up the program, select **Set Program/Set New Program** option from the menu bar. A window appears where the program specifications can be entered. Figure 3 shows the window with the complete list of tasks in the program and hosts on which the tasks are to be executed. In our case the program will be executed on two hosts: europa and carme.

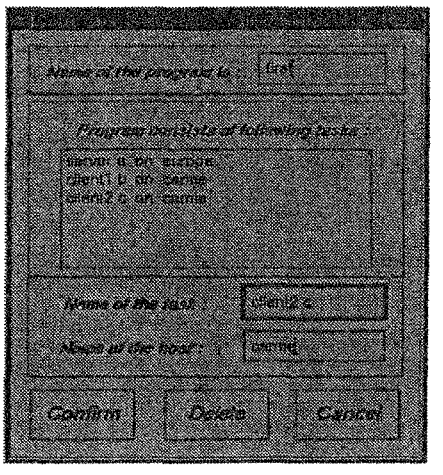

Figure 3: Set Program window

Start the program
To start execution of the user program we have to select

the **Run/Record** option from the menu bar. The Run window will appear on the screen. The program is started incrementally by initiating execution of each task belonging to the program. There will be a scrollable window created in the main window of the debugger for the newly started task. The output from started task will appear in the scrollable window.

Observe the output
During execution of the program, the output of each task will appear in a task-dedicated scrollable window. The windows are re-sizable to allow for increasing the size of the most interesting window. The output from a program can be seen in figure 2.

View time-process diagram
Since a picture is worth a thousand words it might be good to take a look at the time process diagram of an execution. This can be done by selecting **View/View Graph** from the menu bar. A window appears on the screen showing the time process diagram as shown in figure ??.

Look at the history files
The time-process diagram does not show the details of system calls to preserve the clarity of the presentation. These details can be obtained by selecting **View/History Files** from the menu bar. A window that appears on the screen contains two push buttons: **Load** and **Close**. The history files can be brought to the window by clicking the mouse on the **Load** button.

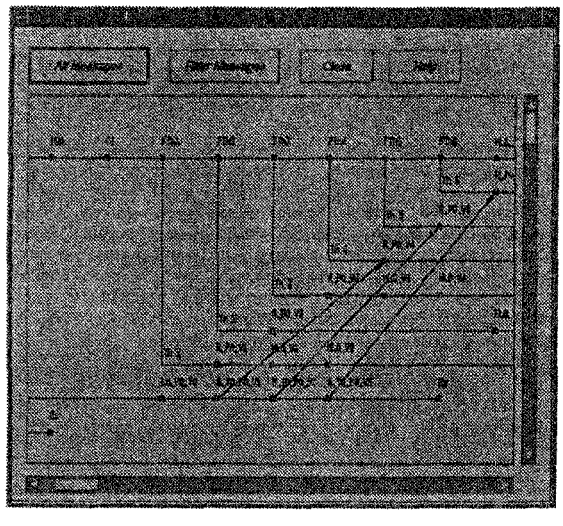

Figure 4: Time-process diagram

Play it again ...
To replay the program we have to select **Run/Replay** option from the menu bar and start all tasks one by one as we have done in *record* phase. For each task in the computation there is a scrollable window created on the main pane of the debugger window. Each window displays a **GDB** prompt. To start execution of a task, the GDB command **run** has to be entered.

6 Summary

At Concordia University we have been working on a team project to develop a complex and fully integrated distributed debugger called CDB for C programs operating under Mach. It involved top-down design and several master's level theses. Unlike CDB, the DDB debugger described in this paper is a simple minded but self-contained and a usable system. It can be incrementally enhanced as supported by its architecture. In fact we have used DDB to debug several programs including the debugger itself, in some cases. The systematic methodology followed in the design and development of DDB can be useful for the developers of other debuggers. We have identified the the minimal requirements of a debugger, the sources of non-determinism as per the distributed computational model, and the minimal user interface requirements. They are all incorporated in DDB and was developed as the work of a single master's thesis. However, the DDB development has benefited from the reusable modules developed as a part of the CDB project.

References

[1] Deborah Caswell, David Black, "Implementing a Mach Debugger for Multithreaded Applications", *Conference Proceedings of Winter 1990 USENIX Technical Conference and Exhibition*, Washington, DC, January 1990.

[2] Thomas J. LeBlanc, John M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay", *IEEE Transactions on Computers*, Vol. C-36, No.4, pp. 471-481, April 1987.

[3] Alain Sarraf, "Checkpoint and Rollback Recovery in a Non-FIFO Multi-Channel Distributed Environment", M.Comp.Sc. Thesis, Concordia University, Montreal, June 1993.

[4] Joanna Sienkiewicz, "A Distributed Debugger Based on Deterministic Replay", M.Comp.Sc. Thesis, Concordia University, Montreal, December 1995.

[5] C.Yep, "A Debugging Support Based on Breakpoints for Distributed Programs Running Under Mach", M.Comp.Sc. Thesis, Concordia University, Montreal, November 1992.