HEWLETT PACKARD
CONCURRENT COMPUTING DEPARTMENT
HEWLETT-PACKARD LABORATORIES May 12, 1993

Program replay in PVM

Milon Mackey

An implementation of a facility to support program replay in PVM 3.0 is nearing completion. This paper describes program replay, outlines the high level details of its implementation in PVM 3.0, and describes its user interface.
This work is still in progress and some of the details described here may change in the future.

1 Introduction

The Research on Environments and Applications for Parallel Processing (REAPP) project in HP Laboratories is currently developing parallel programming environments and tools for clusters of workstations. The first tool we are developing, Interactive Visualization Debugger (IVD), is a debugger for parallel programs that integrates debugging with performance and data visualization. One of our first goals is to have IVD work with PVM [Geist93]. The work of designing and implementing *program replay* inside PVM 3.0 grew out of this effort.

1.1 What is program replay?

Most parallel programs produce deterministic output. If they are given the same input, they will produce the same output. But if one looks inside the program at the actual computations performed during different runs of the program on the same input data, the computations can be wildly different. This non-deterministic internal behavior is caused by race conditions in the program that get resolved differently in different runs of the program even if the input data is the same during each run. An example of a typical race condition is shown in Figure 1. The messages `m1` and `m2` could arrive in either order at `P2` and the order may change from one run of the program to another. This non-

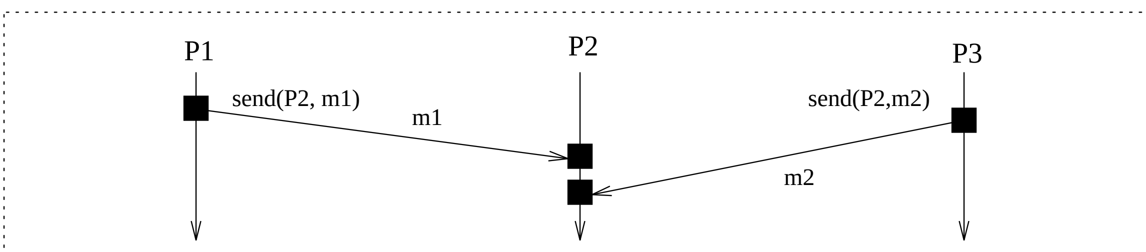

Figure 1: Example of a race condition

deterministic internal behavior is bad for debugging. It may not be possible to reproduce the exact resolution of the races that resulted in a defect when the program is re-executed using a debugger.

Program replay solves this problem through a two step process. First, the program is executed and a trace is made recording how all of the races were resolved. The program is then re-executed using the trace. The information in the trace is used to ensure that each race is resolved as it was during the initial execution when the trace was made. This way the internal behavior of a parallel program becomes deterministic and the cyclic debugging techniques used for sequential programs can be used to debug parallel programs. To isolate any defect, the only requirement is that a trace was taken when the defect occurred.

Program replay is not new. The paper [LeBlanc87] describes Instant Replay, a mechanism to do program replay for parallel programs in which all communication is through shared objects. The paper [Leu92] describes a method of doing program replay for parallel programs made up of sequential processes that communicate through message passing. The paper [Netzer92] describes a method for recording in a trace close to the minimum amount of information necessary to ensure that replay can be done.

1.2 Uses of program replay

This is a list of some of the uses of program replay.

- Program replay is one of the ways to make cyclic debugging work for parallel programs. This was already mentioned above but it is important enough to list it again.

- It can be used to minimize the "probe effect." The probe effect often occurs when trying to observe the behavior of a parallel program. The very act of observing the behavior may change the behavior.

  When using program replay, during the initial execution only enough information to do program replay need be collected, minimizing the probe effect. Then, during replay lots of other information about the execution of the program can be collected without it changing the program's behavior.

- Lamport's "happened before" [Lamport78] relation can be computed from the trace. This relation can be used as a base for more powerful debugging tools [Fowler89].

- It allows performance and data visualization to be more effective.

  Visualization of a parallel program is usually done by having the program emit trace records that are used as input to a visualization tool. These trace records typically consist of a timestamp and the data to be visualized. If these trace records are large, the act of emitting them can have a large probe effect, causing the visualization to not correspond on any real execution of the program. This effect can be minimized by using program replay. During the initial execution, in addition to the usual program replay trace information, timestamps for all the events to be visualized are also written to the trace. Visualization is then run during replay, using the timestamps recorded during the initial execution and data collected during the replay. This approach is described in [Leu92a].

  In addition, during replay a visualization can be synchronized with the program. This allows a high level view of a program's behavior to be given by the visualization tools and a low level view to be given by a debugger. If visualization is run from a trace after the program has terminated, the low level view of the program is not available.

- It can be used to support causal distributed breakpoints [Fowler90]. See Section 3.

- The initial execution can be done using optimized code and the replay done with code that can be symbolically debugged. This is useful for people who do not have a debugger that can debug optimized code.

1.3 How does program replay work?

During the execution of a PVM program, the tasks and PVM daemons communicate through message passing, and so, non-determinism arises because of the races between arriving messages. It is the resolution of these races that must be recorded in the trace files. This can be done quite simply.

During the initial execution of the PVM program, each PVM task and daemon has its own private trace file. In its trace file it records the order in which it received its messages. The only requirement for this to be possible is that each message sent during the execution of the PVM program have a unique identifier. To guarantee this, each message sender maintains an "event counter." The value of the event counter is initially zero. Every time a message it sent, the sender increments its event counter and stores the new value of the event counter into the message's header. This gives every message a unique identifier made up of the task id of the sender and the value of the event counter stored in the message's header. All of this is illustrated in "tracing" portion of Figure 2. The value of the event counter is shown by "`event_c`." The task id of the message sender is shown by "`src`." When the receiver receives a message, it records this fact by appending the pair $\langle \texttt{src}, \texttt{event\_c} \rangle$ identifying the message to the end of its trace file.

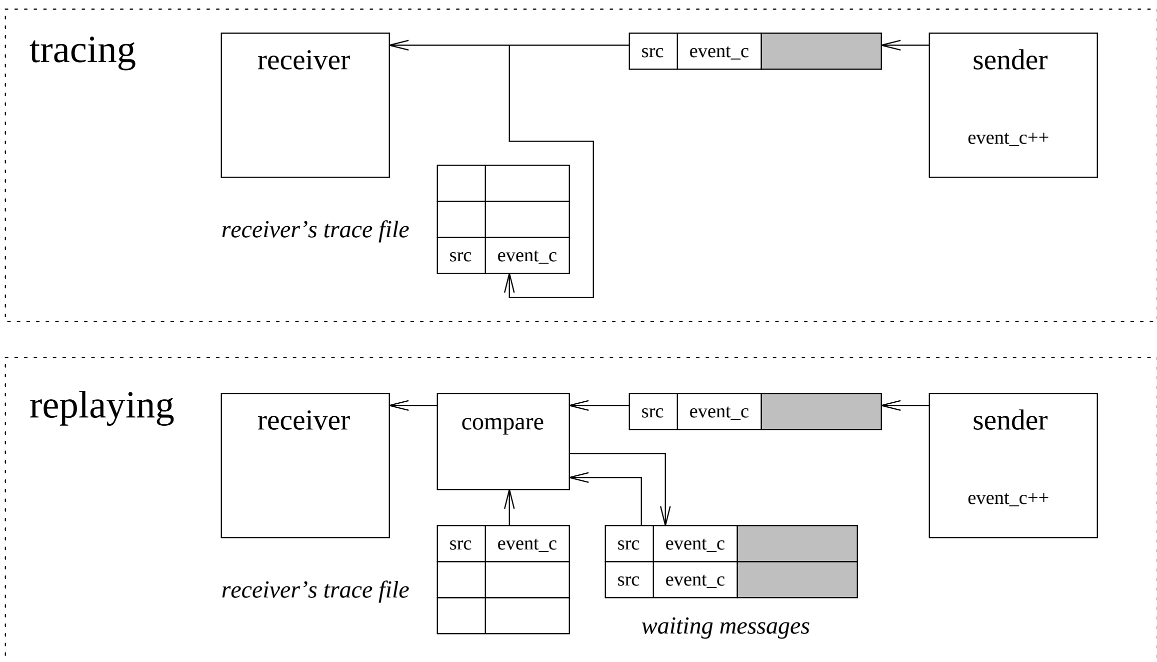

Figure 2: How program replay works.

When the program is re-executed during replay, the information in each trace file is used to force the messages to arrive in the same order as they did during the initial execution. This is shown in the "replaying" portion of Figure 2. The sender behaves as before, tagging each message that it sends with the value of its event counter. But the receiver behaves differently. This time it reads its trace file. When a message arrives, it compares the unique identifier in the message with the unique identifier at the head of the trace file. If they match, it advances to the next identifier in the trace file and passes the message on to the receiver. If they do not match, it adds the message to the pool of waiting messages. The message stays in the pool until its identifier appears at the head of the trace file. At that time, the trace file is advanced and the message is removed from the pool and passed on to the receiver. The pool of waiting messages can be thought of as the set of messages that arrived earlier during replay than they did during the initial execution.

Notice that nearly all the work of program replay, both during tracing and during replay, is done by the receiver. Notice also that the individual trace files do not need to be merged to do replay.

At first thought it may seem that it would not be necessary to trace the PVM daemons, but this is wrong. The daemons maintain among themselves shared state that is readable and modifiable by the PVM tasks. The daemons must be traced and replayed to ensure that during replay the tasks see the same view of this shared state.

One of the underlying assumptions of the whole discussion up to now has been that if a task receives the same messages in the same order, it will send the same messages in the same order. If this assumption is true, program replay works. If this assumption is not true, it may not. For example, the assumption is false for any task that reads the system clock and then uses the clock value to determine whether a message should be sent. But in this case, program replay can be made to work again by tracing the values returned by the system clock during the initial execution and replaying the same values to the task during replay. Hence, program replay can always be made to work by extending it to trace any other sources of non-determinism.

1.4 How is it implemented?

Program replay was implemented in PVM 3.0 by small modifications to the PVM library, PVM console, and the PVM daemon. The PVM daemon required the most modifications. Approximately 2000 lines of new code were added to implement program replay and routines that manipulate the trace files.

2 User interface

The user interface to program replay in PVM is made up of three parts: (1) a new option in the PVM hostfile that controls where the trace files are kept, (2) new PVM console commands, and (3) new PVM library routines. The new commands and library routines are used to control program replay.

2.1 PVM hostfile

A minor change was made to the PVM hostfile to support program replay. During tracing, trace files are produced by both the PVM tasks and daemons. Where are these trace files kept?

The location of the trace files can be controlled on a host by host basis by using a new option, `tf=`, in the PVM hostfile. The complete syntax is "`tf=`<*path*>", where <*path*> is a path name that leads to the directory containing the trace files. The path name can be specified in two ways. If the path name begins with "`/`", it is treated as a rooted path name. If it does not begin with "`/`", it is treated as a relative path name from `$HOME`, the users home directory. If the `tf=` option is not specified for a host and no `tf=` option is in effect from a `*` line, the default location of the trace files is `$HOME/pvm3/traces`. If `HOME` is not defined in the environment, `/tmp` is used in its place. Examples of using the `tf=` option are shown in Figure 3.

It is strongly recommended that a hostfile be used to configure the PVM virtual machine when tracing and replay are used.

If a hostfile is not used, before starting a replay the hosts must be added to the PVM virtual machine in the *same order* that they were added when the trace was made. This is because during a replay each PVM daemon must have the same task id that it had when the trace was made. The task id's of the PVM daemons are determined by the order in which their hosts are added to the PVM machine, and so, if the order is changed they will not have the same task id's and the replay will fail.

```
# The trace files on the host "betti" are kept in $HOME/pvm3/traces.
betti

# The trace files on "pina" are kept in $HOME/traces.
pina tf=traces

# The trace files on "marina" are kept in /tmp/traces.
marina tf=/tmp/traces

* tf=/tmp/junk/monday

# The trace files on "giovanna" are kept in /tmp/junk/monday.
giovanna
```

Figure 3: A PVM hostfile showing the use of the `tf=` option.

2.2 PVM console

This section describes the interface to program replay provided by the PVM console.

2.2.1 Trace names

Many of the new PVM console commands related to program replay take a "trace name" parameter. A trace name parameter is always shown in the descriptions given below by a string containing the substring "*trname*." A trace name has the syntax {*dir*/}+*name*, where {}+ means 0 or more occurrences of the enclosed text. *dir* is the name of a directory and *name* identifies the trace.

A trace itself is made up of a collection of trace files, one trace file for each daemon and task in the PVM program that was traced. The name of a trace file has two components. The first component is *name* followed by a suffix, which is the PVM task id (in hexadecimal) of the daemon or task that wrote the file. The suffix is separated from *name* by "`@`", and so, "`@`" cannot appear in *dir* or *name*.

A trace name is "really" a relative path name from the trace files directory defined by the `tf=` option in the PVM hostfile. For example, suppose that a PVM console user types the trace name "`run1/foo`" and that the hostfile is as shown in Figure 3, then the trace files on the host `pina` are kept in the directory `$HOME/traces/run1` and have names `foo@*`.

2.2.2 PVM console commands

Below is a list of the new and existing PVM console commands that were modified to add program replay. The `reset` command was modified. All of the other commands are new.

- `reset` — This command is used to reset the PVM daemons. In standard PVM 3.0, the `reset` command deletes the string "`pvmgs`" from the name database in the master PVM daemon and kills all of the PVM tasks except for the PVM console executing the `reset` command.

  Executing the modified `reset` command removes the "`pvmgs`" string from the name database in the master PVM daemon, kills all of the tasks that are not PVM consoles, and stops any tracing or replay that is in effect.

- `trace` *trname* `[`*topts*`]` `[[`*spawnopts*`]` *file* `[`*argv*`]]` — This command is used to trace the initial execution of a PVM program.

The `trace` command resets the PVM daemons (see the `reset` command above), prepares the PVM daemons to write trace records to the trace files specified by *trname*, and spawns the PVM program *file*, if specified. All PVM tasks spawned by this command or after it are traced.

The optional *topts* parameter determines what information is written to the trace files. It has the syntax {`to=`*opt*}`+` , where *opt* is one of `TSENDS`,[1] ..., etc.

The option `TSENDS` causes message sends to be traced. This must be specified if the "happened before" relation is going to be computed from the trace. The default is to trace only message receptions, which stores enough information in the trace to replay the execution.

The optional parameters *spawnopts*, *file*, and *argv* have the same meaning and syntax as described for the `spawn` command and define the program to be executed. Collectively, these options will be call the "*pgmopts*." If the *pgmopts* are specified, the `trace` command automatically spawns the program and records values of the *pgmopts* so that when the `replay` command is used to replay *trname*, the same program is automatically spawned by default.

If the *pgmopts* are omitted, no program is spawned. This allows a user to start tracing from a PVM console and then start the PVM program that is going be traced from the command line of a shell.

Tracing is terminated by the `reset` command.

- `replay` *trname* `[[`*spawnopts*`]` *file* `[`*argv*`]]` — This command is used to replay the execution of a PVM program.

  The `replay` command resets the PVM daemons, prepares the PVM daemons to guide their execution by reading trace records from the trace files specified by *trname*, and spawns the PVM program specified by the *pgmopts*, if specified. All the tasks belonging to the PVM program will have their execution guided by the trace *trname*.

  If during replay an EOF is encountered on any trace file, the corresponding task or daemon automatically enables tracing mode and appends new trace records to the trace file. The tracing options will be the same as for the original trace file. This allows an interrupted trace to be extended.

  If the *pgmopts* are not specified, the `replay` command spawns the program specified by the *pgmopts* recorded by the `trace` command when the trace was made.

  Replaying is terminated by the `reset` command.

- `replaytrace` *trname* *new-trname* `[`*topts*`]` `[[`*spawnopts*`]` *file* `[`*argv*`]]` — This command is used to replay and trace the execution of a PVM program. This command is used to make a trace containing more information from an existing trace.

  It is similar to the `replay` command. The trace, *trname*, is used to control the replay. The trace, *new-trname*, is written and *topts* determines what information goes into *new-trname*.

  Replay and tracing is terminated by the `reset` command.

- `rmtrace` *trname* — This command removes the trace files belonging to *trname*.

- `traces` — This command returns a list of all the traces that exist.

- `traceinfo` *trname* — This command returns detailed information about the trace *trname*. See the discussion of `pvm_traceinfo()` in Section 2.3.

- `debug` — This command toggles the state of using debuggers between "enabled" and "disabled."

  If disabled, a newly spawned task is run under control of a debugger only if the `PvmTaskDebug` flag was passed to `pvm_spawn()`. If enabled, all newly spawned tasks are run under control of a debugger.

[1] Currently, only `TSENDS` is defined.

- `status` — This command displays the current status of tracing and replay.

  If no tracing or replay are in effect, it displays nothing. If tracing is in effect, it displays the name of the trace being made. If replay is in effect, it displays the name of the trace being replayed. It also indicates whether tasks are automatically run with a debugger (see the `debug` command above).

Figure 4 shows how these PVM console commands can be used to trace and replay an application, `wmod`. This fictitious application takes two parameters, the staring year and the ending year. It models the global climate between the two specified years.

```
> trace junk/r1 wmod 57 93            # Trace the application.
>
> ps -a                               # Verify that wmod is finished.
> reset                               # Stop the trace.
> replay junk/r1                      # Replay the application.
>
> reset                               # Stop the replay.
> debug                               # Run all tasks with debugger.
> replay junk/r1                      # Replay the application again.
>
> reset                               # Stop the replay.
> replaytrace junk/r1 junk/r1.a to=TSEND
>                                     # Replay the application and
>                                     # this time make a trace that
>                                     # also records when messages
>                                     # were sent.
>
> reset                               # Stop the replay.
> replay junk/r1.a                    # Replay from the new trace.
>
> reset                               # Stop the replay.
> spawn wmod 57 93                    # Runs the application with no
>                                     # tracing or program replay.
```

Figure 4: Example of using the new PVM console commands.

2.2.3 Relation of PVM console to program replay

The messages sent and received by a PVM console are not traced. This allows a PVM console to be used during tracing or replay to examine what is going on but to have no effect on the trace or replay.

The PVM console commands, shown in Figure 5, that could in fact modify the state of the PVM machine or PVM program cannot be used from the PVM console during tracing and replay.

2.3 PVM library

This section describes the interface to program replay provided by routines in the PVM library. All of the routines listed below except for `pvm_reset()` are new.

- `void pvm_console(void)` — informs the PVM daemon that the task calling this is a PVM console. It must be called before any other routine in the PVM library.

add	delete	kill	replay	replaytrace	sig	spawn	trace

Figure 5: PVM console commands that cannot be used during tracing or replay.

- `pvm_pgmreplay(...)` — used to control program replay. A detailed discussion is given below.
- `pvm_traceinfo(char *`*trname*`, struct traceinfo *`*ti*`)` — returns a structure *ti* containing descriptive information about the trace *trname*. This structure is not yet defined, but it will contain something like this: (1) whether the trace was made from another trace and if so, which one; (2) a list of what information is contained in the trace; (3) the time the trace was started and stopped; (4) a list of the hosts that were involved in the trace; (5) a list of all the PVM tasks that were traced; etc.
- `pvm_spawnflags(int `*flags*`, int `*op*`)` — causes the PVM daemons to modify the spawn flags used when they spawn a new task. The PVM spawn flags listed in *flags* are set or un-set when a task is spawned depending on the value of the corresponding bit in *op*. If the corresponding bit is 1, it is set. If the corresponding bit is 0, it is un-set. The values of the `PvmTaskHost` and `PvmTaskArch` flags cannot be modified. This routine makes it easy to have all the tasks in a PVM program be run under control of a debugger. For example, `pvm_spawnflags(PvmTaskDebug, PvmTaskDebug)` causes all tasks to be spawned under control of a debugger and `pvm_spawnflags(PvmTaskDebug, 0)` causes all tasks to spawned without a debugger.
- `pvm_traces(char ***`*traces*`)` — returns a list of the names of all the existing traces.
- `pvm_status(struct pvmstatus *`*st*`)` — reports whether tracing or replay are in effect, and also reports the current setting of the spawn flags set by `pvm_spawnflags()`.
- `pvm_reset()` — Resets the PVM daemons. Kills all tasks that are not PVM consoles. Stops any tracing and replay that are in effect.
- `pvm_rmtrace(char *`*trname*`)` — Removes the trace named *trname*.

2.3.1 pvm_pgmreplay()

This is a detailed discussion of `pvm_pgmreplay()`.

Synopsis

```
int info =
pvm_pgmreplay(pgmreplay_t *p,
              char *task, char **argv, int flag, char *where, int ntask, int *tids)

typedef struct flag_t_ {
  int tid;         /* tid of task whose spawn flags get modified */
  int flags;       /* pvm spawn flags to be modified */
  int op;          /* the value of the corresponding flags */
} flag_t;

typedef struct stop_t_ {
  int tid;         /* tid of task to stop */
  event_t event_c; /* event it should stop after */
} stop_t;
```

```
typedef struct error_t_ {
  char *host;      /* host that had an error */
  int cc;          /* the error code */
} error_t;

typedef struct pgmreplay_t_ {
  char *replay;    /* R name of trace to be replayed */
  int checkpoint;  /* R check point to start replay from */
  flag_t *flags;   /* R spawn flags to change during replay */
  stop_t *stop;    /* R tasks to stop during replay */

  char *trace;     /* T name of trace to make */
  topt_t *topts;   /* T what should be traced */

  error_t *errors; /* B what went wrong */
} pgmreplay_t;
```

Parameters

task, *argv*, *flag*, *where*, *ntask*, *tids* — are exactly the same as the parameters to `pvm_spawn()`.

p — controls program replay.

The fields labeled with "`R`" in `pgmreplay_t` control replay. The fields labeled with "`T`" control tracing. The fields labeled with "`B`" are used by both tracing and replay.

replay — the name of the trace used to guide the replay. If set to 0, no replay is done.

checkpoint — in the future we hope to add a checkpoint and restart facility. This parameter allows the program to be replayed beginning at a specified checkpoint. Specifying 0 for this parameter means to start the replay from the beginning of the program. This is very useful for replaying long running programs.

flags — is 0 or a pointer to an array of `flag_t` with the end of the array indicated by a `flag_t` with `tid` set to 0. This parameter allows the `pvm_spawn` flags to be changed for specified tasks during replay. In each `flag_t`, `tid` is the task id of the task whose spawn flags are to be modified. `flags` and `op` have the same meaning as the parameters passed to `pvm_spawnflags()`. This parameter is useful since it allows a subset of the tasks in a PVM program to be run under control of the debugger during replay without having to modify the program.

stop — is 0 or a pointer to an array of `stop_t` with the end of the array indicated by a `stop_t` with `tid` set to 0. This parameter allows a specified set of tasks to be stopped automatically when they reach the specified events during replay. These tasks are automatically spawned under control of a debugger. This parameter is used to support things like causal distributed breakpoints [Fowler90].

trace — the name of the trace to be made. If set to 0 no trace is made.

topts — flags that determine how much information is recorded in the trace. Currently only one flag is defined, `PvmTraceSend`. If it is specified, the message sends are also recorded in the trace. The default is to record only the message receptions.

errors — used to report errors. Set to 0 if no errors occurred. If any errors occurred, while trying to initialize tracing or replay, this parameter returns error information. It is an array of `error_t` with the end of the array indicated by an `error_t` with `host` set to 0. `host` is the name of the host that had the error and `cc` is the error number.

info — the return value. If it is negative an error occurred. The return value itself is an error number and more information may be available in `p->errors`. If it is zero and no spawn was requested, it indicates success. If it is zero or positive and a spawn was requested, it indicates success or partial success of the spawn as described for `pvm_spawn()`.

Discussion

This routine is used to control tracing and replay and to optionally spawn a PVM program to be traced or replayed. The tracing and/or replay is in effect until stopped by calling `pvm_reset()`.

3 Causal distributed breakpoints

To control the execution of a parallel program, it must be possible to stop its execution. One way to do this is a breakpoint. A breakpoint is a user defined event that causes the program to stop when it occurs. Examples are, (1) stop when this statement is executed in this process, (2) stop when this process tries to write to this memory location, (3) stop when this type of message arrives, (4) stop when process x has executed statement z and process a has also received a message from process b, etc. No mater how a breakpoint is defined, once it occurs the program must be stopped. This is usually done by stopping the process that detected the breakpoint and then broadcasting a signal to the other processes to cause them to stop. One of the problems of debugging a distributed parallel program is that the state in which the processes stop is not unique, since the broadcast of the stop signal cannot be done instantaneously. This situation is shown in Figure 6. A breakpoint A has occurred and stopped the program. Some of the possible stopped states for the breakpoint are shown by the dotted lines. Please note that all of the dotted lines should cross Process 1's timeline at A. They are drawn separately to make them easier to distinguish.

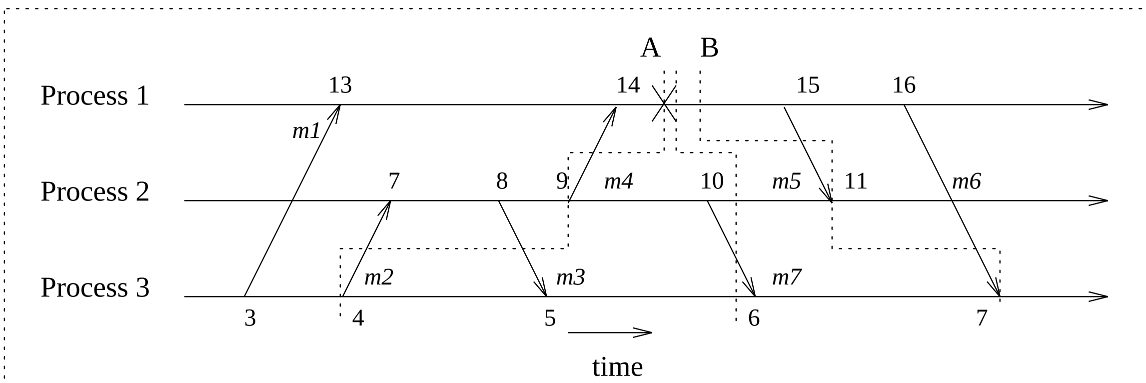

Figure 6: Some consistent system states for breakpoint A.

One possible question is, "Which consistent system state is best for debugging the program?" Two of the system states shown in Figure 6 are uniquely defined. The one shown by the dotted line passing through A is called the *causal distributed break point* [Fowler90] associated with A. In it, every process in the program is stopped at the "earliest" possible time, meaning, each process has performed all events that happened before A but nothing more. The one shown by the dotted line passing through B is the "dual" to the one passing through A. In it, every process has done everything that it can without executing any events that depend on A. It can be found by simply stopping Process 1 when A occurs and letting all of the other processes run until they block on a message receive. To stop at the causal distributed breakpoint, program replay is needed. The program is stopped and the trace examined to compute the event in each process where it must be stopped. The program is then replayed from the trace, with each process stopping immediately after the appropriate event.

4 Acknowledgments

Thanks to Ming Hao, Alan Karp, and Vineet Singh for lots of useful comments.

References

[Fowler89] Robert J. Fowler, Thomas J. LeBlanc, and John M. Mellor-Crummey. An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors. *Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* (Madison, Wisconsin, May 1988). Published as *SIGPLAN Notices*, **24**(1):163–73, January 1989.

[Fowler90] Jerry Fowler and Willy Zwaenepoel. Causal distributed breakpoints. *Proceedings of 10th International Conference on Distributed Computing Systems* (Paris, France, May 1990), pages 134–41. Institute of Electrical and Electronics Engineers Computer Society Press, May 1990.

[Geist93] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng J iang, Robert Mancheck, and Vaidy Sunderam. *PVM 3.0 user's guide and reference manual.* Technical report ORNL/TM-12187. Oak Ridge National Laboratory, Oak Ridge, TN 37831, February 1993.

[Lamport78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, **21**(7):558–65, 1978.

[LeBlanc87] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, **C–36**(4):471–82, April 1987.

[Leu92] Eric Leu and AndréSchiper. Execution replay: a mechanism for integrating a visualization tool with a symbolic debugger. *Proceedings of 2nd Joint International Conference on Vector and Parallel Processing (CONPAR 92 — VAPPV)* (Lyon, France, September 1992), volume 634 of Lecture Notes in Computer Science, pages 55–66, L. Bougé, M. Cosnard, Y. Robert, and D. Trystram, editors. Springer Verlag, Berlin, 1992.

[Leu92a] Eric Leu and André Schiper. ParaRex: a programming environment integrating execution replay and visualization. *CNRS NSF* (Saint Hilaire du Touvet, France, September 1992).

[Netzer92] Robert H. B. Netzer and Barton P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. *Proceedings of Supercomputing '92* (Minneapolis, Minnesota, November 1992), pages 502–11. Institute of Electrical Engineers Computer Society Press, November 1992.