MpD:
A Multiprocessor C Debugger

Krish Ponamgi *
Columbia University
Department of Computer Science

September 16, 1991
MS Thesis

CUCS-021-91

Thesis Committee: Gail E. Kaiser and Colin G. Harrison

Abstract

MpD is a multiprocessor C debugger designed for multithreaded applications running under the Mach operating system. MpD is built on top of gdb, an existing sequential debugger. The MpD layer utilizes the modeling language *Data Path Expressions* developed by Hseush and Kaiser to provide a rich set of commands to trace sequential and parallel execution of a program. Associated with each DPE are actions that allow access to useful trace variables and I/O facilities. DPEs are useful for describing sequential and concurrent patterns of events, to be verified during execution. The patterns include conditions such as synchronization, race conditions, and wrongly classified sequential/concurrent behavior. We show in this thesis Data Path Expressions are a viable language for multiprocessor debuggers.

*Ponamgi was supported in part by IBM.

Contents

1 Introduction

Developing software for multiprocessor architectures is an area of active research. In part, this effort is fueled by the speculation that as uniprocessors push the limits of semi-conductor technology to the sub-micron range, device-physics will place a barrier on maximum gate switching speeds. This ultimately will result in single processors reaching a peak performance rate. Experts currently estimate this to be in the range of 100-200 MIPS (millions of instructions per second)[13]. Multiprocessors, however, have a much weaker restriction; theoretically, the number of processors inter-connected to solve a problem is constrained only by physical connection limits[24]. So once uniprocessors reach their maximum speeds, they can be connected to form larger more powerful machines with at least an order of magnitude better performance. Several examples of such networked multiprocessors are the Connection Machine, the BBN Butterfly, and the Encore Multi-Max.

Although parallel computation via multiprocessors has been successfully applied to a diverse number of problems such as meteorology, high energy physics, turbulence modeling, and CAD/CAM, software technology on the whole has been lagging behind hardware in fully exploiting the new architectures[9]. The software can be divided into two categories: application-oriented and system-support oriented. For applications to be developed quickly and efficiently, system support is necessary. Currently, however, parallel system support facilities are quite poor. Operating systems are still largely prototype; parallel languages and their compilers are still experimental; and debugging facilities are almost non-existent for parallel environments.

This thesis addresses the issue of multiprocessor debugging and puts forth a solution. A multiprocessor debugger, `MpD`, along with its implementation details, will be described. We believe it is a useful software tool to deal with issues of programming in a parallel environment. To begin the discussion, some of the general issues of debugging are presented.

Debugging on sequential machines is relatively well understood and there exist several excellent debuggers for this purpose (sdb, dbx, etc.). The key concept in sequential debugging is that, at any given time, only one execution flow exists. As a result, during execution, one can completely order the sequence of program events (assertions that designated points in the source code have been reached) and each segment of code has a well-defined execution time relative to any other segment. Even code segments

involving loops and recursion have a single execution stream and are unaffected by each other's execution times. Similarly branching and transfers of control flow fit predictably into the single execution stream. So to collect all relevant execution data, including ordering of events, the sequential debugger simply latches onto the single thread of execution[18]. The amassed data can be presented serially to the user, providing an accurate, perhaps voluminous, description of the program behavior.

A complete trace of a sequential debugging session would reveal an ordered list of statements executed – i.e. program events. This trace, however, could be thousands of lines long, making it difficult to read let alone understand. Even establishing an ordering on "interesting" portions of it could be tedious since on most debuggers this is accomplished by the user single-stepping through the region to determine the execution flow.

```
122   for (x= 0; x < MAXCOUNT; ++x)
123     {
124         do_action(x);
125     }
```

For example, in the program fragment involving a loop statement above, rather than iterating through each step of the program, it would be convenient to specify line 124 is executed MAXCOUNT times or an arbitrarily number of times sequentially using an abstract syntax. Suppose we denote the latter as 124*. Being able to succintly describe each type or group of program events can help the user quickly get a feel for program behavior. The user can use the abstract syntax to describe potential execution behavior which the debugger attempts to verify. Since the descriptions are easily constructed, modified, and compiled, the debugging process is accelerated. Bruegge has shown the usefulness of this approach in his thesis[5] on path expressions. He has a number of operators which model behavior such as selection, repetition, and sequencing.

Bruegge's work applied path expressions to model sequential execution of a program. So, for example, there was no operator to model line 124 in the above example executing concurrently. In general, concurrent execution implies multiple threads of control flow. Events occurring in different control threads are not constrained to a particular execution ordering; they can, in fact, occur asynchronously with respect to each other. The actual execution sequence of such events is unpredictable and may change with each execution of the program with the same inputs. So a parallel debugger has to cope with a variable execution stream and present this as a coherent

picture the user can understand. The command language of the parallel debugger also has to have the expressive power to model common "bugs" of parallel programming.

Concurrency-related bugs often involve synchronization problems among multiple threads sharing information[17]. These typically manifest themselves in programs as race conditions, deadlocks, livelocks and starvation. A debugger for a parallel environment does not necessarily have to run in parallel itself to recognize these errors. Instead, it needs to be able to gathher events from the different processors and analyze their behavior to catch such errors. To model inter-dependencies properly, the debugger needs to examine the usage of semaphores, shared memory, message passing, locks, and conditional wait mechanisms. A multiprocessor debugger needs to peer into the run-time details as well as know what statements are executed.

The multiprocessor debugger, `MpD`, we describe deals with these issues. It implements the event modeling language Data Path Exppresions, developed by Hseush and Kaiser[15], on top of an existing sequential debugger, gdb. The work was done on an experimental multiprocessor the IBM 8CE.

2 Related Work

A number of approaches have been suggested to deal with these issues. One method taken by Peter Bates and Jack Wileden[3] relied on event-based behavioral abstraction of a parallel program. Their idea was to describe expected program events in a high-level language that would be compared against events generated during actual program execution. Their approach was similar to Bruegge's except for their "shuffle automata" which they implemented for recognizing patterns in the event stream.

Shuffle automata are a finite state machine-like formalism for describing event sequences for recognition purposes. The input alphabet to the shuffle automaton is the set of program events to be recognized in a described ordering. The shuffle automaton is composed of a set of states and a transition control mechanism that controls the state movement from an initial state to a (possibly) final state based on the input events. Transitions between states are made if the input symbols match exactly the transition symbols. The fundamental difference between shuffle automata and finite state automata (FSA) is that sets of input events, rather than individual symbols, form the input alphabet. By examining sets as opposed

to individual symbols. Bates uses a simple mechanism to model concurrency. Shuffle automata, in addition, are capable of describing the usual sequential behaviors: sequence, iteration, and selection.

The shuffle automata of sequential, repetition, selection, and concurrent patterns are given in Figure 2.1.1. They are the basic blocks from which more complicated, nested automata can be created.

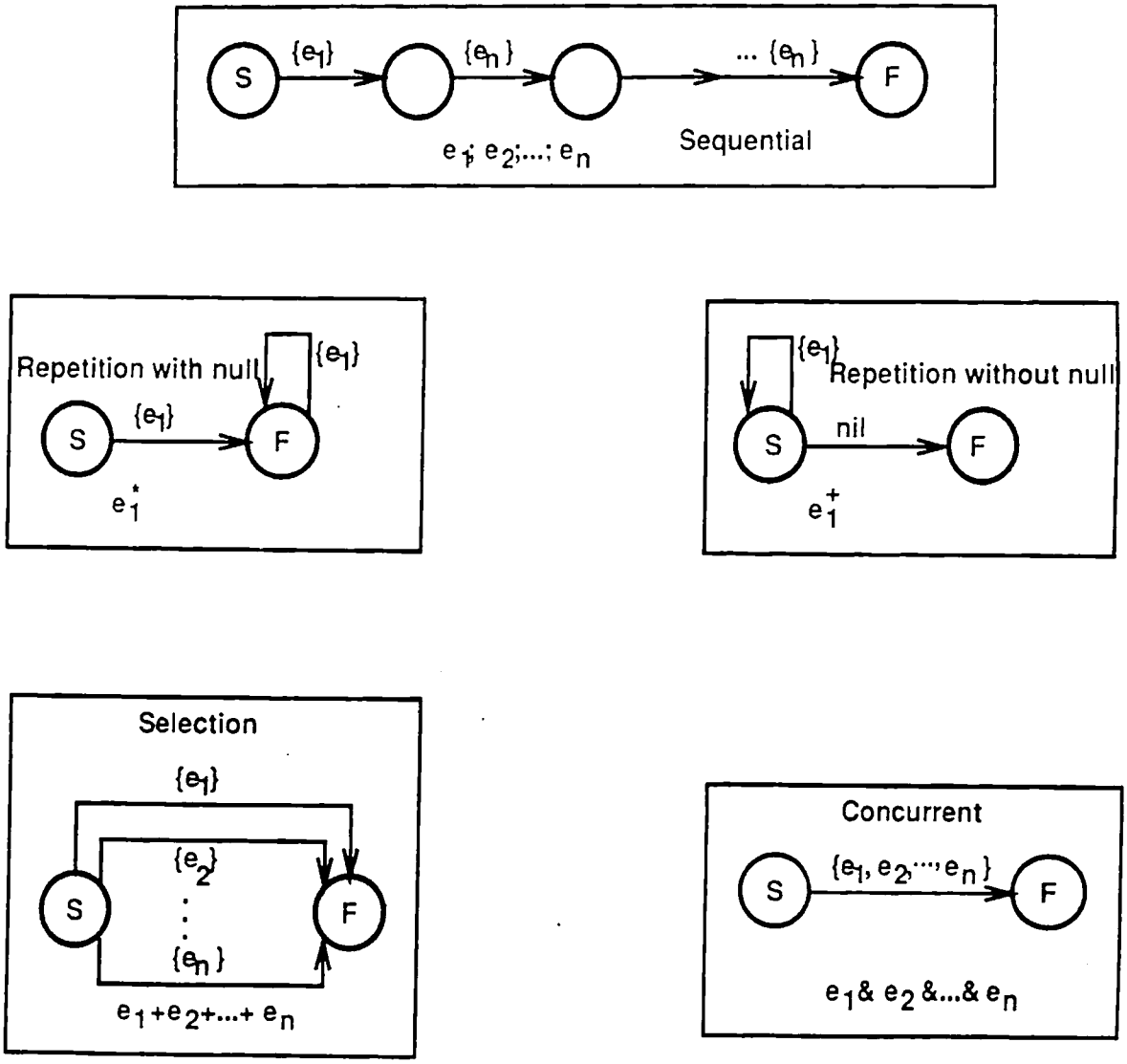

Figure 2.1.1: Shuffle Automata

The weakness in this approach is that events that are described as concurrent may actually be interleaved due to implicit synchronization. Furthermore, there is no way to test for the difference between interleaving and concurrent execution. By interleaving, we mean that events e_1 and e_2 occur in the context of different threads T_1 and T_2, respectively, but due to implicit or explicit synchronization, perhaps via semaphores, they may actually occur in a sequential manner. A typical example of this situation is I/O serialization on a multiprocessor. Independent threads attempt to do an I/O operation, but since there is single I/O channel each must wait in turn - a problem we encountered on the SCE.

The end result of this is that users can describe interleaved expressions as concurrent and the shuffle automata cannot catch the error. Clearly this is not a satisfactory solution. In our implementation of MpD, we deal with this situation—in fact, this exact scenario was the initial motivation for our research. Our solution utilizes an additional piece of information in describing event relationships. Each event has associated with it a predecessor event. An event's predecessor is simply the last event occurring prior to this current event. For instance, in the expression $e_1;e_2;e_3$ the event predecessor to e_2 is e_1 and for e_3 it is e_2. The event e_1, however, is an *original event* since there are no events specified prior to it. The more complicated cases of this predecessor relationship, especially those involving concurrency, will be discussed in section 4.3. Basically, in our solution, by determining an event's relationship with its predecessor, we can decide whether the current event fits the execution pattern described by the user.

Another parallel debugger project addressing some of these same issues is the Amoeba debugger developed by I.J.P. Elshoff[11]. It operates under a distributed operating system similar to our own (Mach). In the Amoeba environment, a program is a collection of clusters that interact with their environment through messages. A cluster is composed of an address space and a collection of threads called tasks. The address space is a collection of contiguous blocks of memory, called segments, which have assigned to them specific access rights with respect to threads.

The Amoeba debugger is composed of three layers. The bottom level monitors the behavior of the program being debugged and generates events for the debugger when appropriate. An event is a 4-tuple identifying event type, the identity of the task generating the event, the capability of the cluster containing the task, and the arguments (possibly none) of the event. The arguments depend on the event type: for events related to message passing, the argument is the header and body of the message; for semaphore

events it is the name of the semaphore; for segment management events, it is the capability of the segment in question and so on[11].

Sitting above the monitor is a layer of tasks, one for each cluster in the target. Each task is called a correspondent; it receives streams of events from the corresponding cluster in the target program and functions in part as a pattern matcher.

At the top-most level is the user interface. The user can examine and control individual clusters and tasks, as well as deal with the target program as a whole (e.g., make checkpoints or initiate I/O).

The portion of the Amoeba debugger that is of main interest to us is the filter/recognizer. The filter and recognizer work together to recognize patterns in the event stream generated by the monitor. A pattern consists of zero or more events fashioned together by logical operations (i.e., conjunction, disjunction, negation, etc.) from primitive patterns. The filter removes "noise" or irrelevant events as specified by the user, and the recognizer attempts to analyze the incoming data stream. The recognizer has the basic FSA operators and a permutation operator. For example, "[a b c]" means the primitive events a, b, and c can appear in any order, and is equivalent to "(abc + acb + bac + bca + cab + cba)."

Using permutations to deal with concurrent events is again insufficient. Like the Bates-Wileden method, there is no mechanism to differentiate interleaving from true concurrency. Synchronization done implicitly or explicitly can cause parallel threads to execute in lock-step fashion. The Amoeba debugger cannot detect this situation. It only allows users to verify that all events in the concurrent threads have been executed identical to the shuffle automata method.

Moreover, although the Amoeba debugger reads the arguments to an event, it does not make full use of their information. In our system when we read the arguments of certain system calls, we check to see if these calls are used improperly. This technique, for example, helps us detect deadlocks and identify in the source code the origins of the locking cycle.

The Durra debugger developed by Dennis Doubleday took a slightly different approach to parallel debugging[10]. Debugging is seen as an application monitoring activity in addition to providing basic breakpointing facilities. The Durra approach stems from the stress placed on message passing in the programming environment. The debugger's monitoring facility provides useful services such as monitoring ports to identify the origin of messages. We incorporated this idea into MpD. The Durra debugger had

many of its services intimately linked with its environment. The language for application development in the Durra environment provided the necessary facilities to do monitoring type debugging. The Durra language had built into it debugging facilities to directly provide a debugging monitor messages about its executing state. This was achieved by passing messages to a debugging message port.

Since in our system we could not alter the C language, hence the C-compiler, we chose to provide the monitoring activities by creating an interface to some of the standard system and library calls. In our system the usage of these routines via the interface was analyzed for correctness. MpD also provides means to specify interesting relationships among breakpoint events, an aspect the Durra debugger ignored.

The Parasight debugger[22], in some respects, is also a language specific/constrained debugger. Its approach is to insert "parasites" into the application source code to directly monitor execution state. The parasites execute as independent threads on spare processors to check assertions. The assertions can range from indicating points in the execution have been reached to time stamps of actual events. The modification of code gives more detailed information about the program, but carries the penalty of extra overhead. The basic idea of this approach is to give accurate descriptions of the program at each parasitic incursion. The relationships between these snapshots is not considered. This is a fundamental weakness we addressed.

Generally most multiprocessor based debuggers take the Parasight or Durra approach. Their scope of use is for low-level mechanisms such as halting threads in a consistent state, restarting a specific thread, and capturing a single snapshot of a multithreaded program. These facilities are essentially the equivalent to those found in a sequential debugger. Our interest is at a higher level, since we are interested in establishing *relationships* among these low-level events. Furthermore, we are interested in "automated debugging" where the user has only a vague notion of where the program is incorrect but the debugger, by examining the programs system usage and parameters to functions, detects the exact error. The information from these conditions indicates the true state of a program's concurrent environment.

The multiprocessor C debugger outlined here, MpD, attempts to address these issues. The MpD system is built on top of an existing sequential C debugger, gdb, and it too takes an event-based approach but it does not consider event patterns. The debugged program is viewed as a generator

of interesting events. An event in our system is a 3-tuple: the event type (usually line number reached), the thread it belongs to, and its (optional) arguments.

Our debugger technology consists of two main components, a debugger command language called Data Path Expressions (DPEs) and a mechanism called Predecessor Automata (PAs) for recognizing the runtime behavior described by DPEs. The programmer specifies a set of DPEs representing patterns of concurrent events (desirable or undesirable) that might happen during program execution, and the debugger generates the corresponding PAs to recognize these patterns. During the execution, the actual event stream is compared against each PA.

DPEs were inspired by the Generalized Path Expressions developed by Bruegge and Hibbard for debugging sequential programs written in Ada or Pascal[5]. DPEs are essentially regular expressions over program events, but with a concurrency operator, indicating causal independence, added to the regular expression notation. The DPEs also have a concurrent repetition operator which was not implemented and cannot be recognized by PAs.

The PAs are analogous to FSAs, which recognize regular expressions, but in PAs each transition is labeled with both a program event and its relationship with its immediate predecessor. Thus while FSAs can recognize only totally ordered streams of events, PAs can recognize partially ordered streams and, most importantly, distinguish between causally dependent versus causally independent events (those that are interleaved in their arrival at the recognizer but not while in progress).

The remaining sections of this thesis details our solution. We begin with the background of our system: the Mach operating system, gdb, and the 8CE multiprocessor. Then an overview of the solution is sketched. A detailed look of the implementation is then described. Then follows the system evaluation and conclusions.

3 Background

The Mach operating system was the environment in which MpD was developed. Mach was developed at Carnegie Mellon University for multiprocessing and distributed systems, and is compatible with the 4.3BSD Unix[1]

[1]Unix is a trademark of AT&T Bell Laboratories.

operating system. The Mach environment provides four fundamental computing abstractions[2]:

- A TASK is the execution environment and is the basic unit of resource allocation. It includes a paged virtual address space and protected access to system resources such as processors and ports.
- A THREAD is the basic unit of execution. It consists of an execution stack, a processor state including program counter and hardware registers, and a limited amount of static storage. A thread shares memory and resources with all other threads executing in the same task. A thread can only execute in one stack.
- A PORT is a channel of communication between tasks or threads, a logical queue of messages protected by the kernel. Send and receive are primitive operations of a port.
- A MESSAGE is a typed collection of data objects used in communication between tasks or threads on ports.

Mach splits the traditional notion of a process into the task and thread abstractions. All threads within a task share the address space and communication rights of that task[2], that is access to the ports available to the task. A familiar Unix process would consist of a task with a single thread of execution. So a task, in effect, provides the environment in which the threads execute the program.

To facilitate easier use of the task/thread abstractions for C programming, Mach provides a cthreads interface. A cthread is a C function forked similar to a Unix *fork()* call. The following program shows several cthreads being created from the function `rtn()` using the routine `cthread_fork()`. A Mach programmer can use the cthreads library to create, execute, and manipulate threads rather than deal with low-level system calls. The constructs serve a similar purpose as to those found in Mesa or Modula2 - namely, forking and joining threads, protection of critical regions with mutex variables, and synchronization by means of condition variables[8].

```
/* Program calculates 2^5 + 3^5 + 4^5 + 5^5 on four separate threads */

#include <cthreads.h>                  * Cthreads include file */
mutex_t         pr_lock;
int             final_value, val1, val2, val3, val4, values[3];

static void
rtn(ret_val)                    /* function being cthread_fork'ed */
   int  *ret_val;
{
  int ctr;

  mutex_lock(pr_lock);
  printf("Thread: %d.\n\n", ret_val[2]-1);
  mutex_unlock(pr_lock);
  for(ctr= 0; ctr < ret_val[1]; ctr++)
    {
        ret_val[0] *= ret_val[2];
        cthread_yield();          /* hint to scheduler we can yield */
    }
}

main()
{
   cthread_t t1, t2, t3, t4;

   setbuf(stdout, NULL);
   cthread_init();
   pr_lock = mutex_alloc();
                    /* create 4 cthreads by forking function */

   values[0] = 1; values[1] = 5; values[2] = 2;
   t1 = cthread_fork( rtn, values);
   val1 =  values[0]; values[2] = 3;
   t2 = cthread_fork( rtn, values);
   val2 =  values[0]; values[2] = 4;
   t3 = cthread_fork( rtn, values);
   val3 =  values[0]; values[2] = 5;
   t4 = cthread_fork( rtn, values);
   val4 = 5;

   cthread_join(t1);          /* join the cthreads after execution */
   cthread_join(t2);
   cthread_join(t3);
   cthread_join(t4);
   final_value = val1 + val2 + val3 + val4;
   printf("2^5 + 3^5 + 4^5 + 5^5 = %d.\n", final_value);
}
```

Program 1: Basic Cthreads Example

Task communication in Mach is implemented by the port and message abstractions. Each task has assigned to it a receiving port, and messages sent to it can be viewed by authorized threads executing within that task. Threads can send messages to a port whenever the task they belong to has sending rights registered for that port. This communication facility is the

basic building block by application and system level message communication takes place; in fact, it is also the underlying mechanism for the Mach exception handling facility.

Exceptions are synchronous interruptions to the normal flow of program control caused by the program. Exceptions include illegal accesses (bus errors, segmentation, and protection violations), arithmetic errors (overflow, underflow, and divide by zero), and hardware instructions intended to support emulation, debugging, or error detection [4]. Hardware exceptions cause traps into the operating system; the system handles certain exceptions transparently (e.g., recoverable page faults), but the remaining exceptions are exported to the user by the operating system's exception handling facility[4].

Low-level debugging in Mach is implemented via exception handling, using an extended version of the Unix *ptrace()* system call. The Unix *ptrace* is primarily used for implementing breakpoint debugging. A debugger, such as gdb, forks a child process (task + threads) to be traced and controls its execution with *ptrace*, setting and clearing breakpoints by modifying object code (explained fully in section 4.1), and reading and writing data in its core image. Tracing amounts to synchronizing with the debugged application and controlling its execution, which includes examining and setting its virtual address space. Essentially the operating system allows the parent process complete access to its child's address space.

```
if ((pid = fork()) == 0)
{
    /* child -- being traced */
    ptrace(0, 0, 0, 0);
    exec("name of traced process here");
}
/* debugger -- controls child */
for(;;)
{
     wait((int *) 0);
     read(input for tracing instructions);
     ptrace(cmd, pid,...);
     if (quitting trace)
         break;
}
```

Figure 3.1: Debugger Pseudo-Code

The pseudo-code in figure 3.1 shows the typical structure of a debugger program. The debugger forks a child, which invokes the *ptrace* system call requesting special treatment from the kernel. The kernel sets a trace bit (STRC) in the child process table, indicating the child is being traced. The child then *execs* the program being traced, basically tells the operating system to begin execution overlaying the child's address space at the *exec* point. The kernel executes the *exec* call in the usual manner, except at the end when it sends the child a "trap" signal due to the trace bit being turned on. After returning from *exec*, the kernel checks for signals. The trap signal it sent itself and any other signals—generated by either the user or program exceptions—are processed by signal handlers. The trap signal merely becomes a special case of the general signal handling facility.

The child process, after returning from *exec*, sees its trace bit is on and proceeds to awaken its parent via the *wait* system call. It then enters a special trace state similar to the sleep state and does a context switch. The parent (debugger), once awakened by its child (the traced application), reads user input commands and translates them into the appropriate *ptrace* calls to control the child. The syntax of the *ptrace* system call is *ptrace(cmd, pid, addr, data)*. The parameter cmd specifies various commands such as reading data, writing data, resuming execution and so on; pid is the process ID of the traced process; addr is the virtual address (including register) to be read or written in the child process; and data is an integer value to be written.

When executing *ptrace*, the kernel verifies that the debugger has a child whose ID is pid and that the child is in the traced state. It sets up a global trace data structure to transfer data between the two processes. The trace data structure is locked while cmd, addr, and data are copied to prevent other tracing processes, such as another copy of the debugger, from overwriting it. The kernel then revives the child, puts it into the ready-to-run queue, and sleeps until the child responds. When the child resumes execution (in kernel mode), it does the appropriate trace command, writes its reply into the trace data structure, then awakens the debugger. Depending on the command type, the child may reenter the trace state and wait for a new command or return from handling signals and resume execution. When the debugger resumes execution, the kernel saves the return value supplied by the traced process, unlocks the trace data structure, and returns to the user. So essentially *ptrace* based debugging provides the ability to

- Access the application's registers and memory.

- Read kernel information such as execution status about the application.
- Continue or single step the application via object code modification (section 4.1).
- Send signals (such as KILL) to the application.

The MpD system was written on top of a version of gdb developed by Deborah Caswell and David Black to run specifically under Mach [7]. The Caswell-Black version of gdb implements the *ptrace*/debugger extensions needed to deal with multithreaded programs.

Vanilla gdb[23] running in a normal Unix single execution-thread environment lacked several key features. Basically, the debugger lacked the internal data structures and process control logic for tracking and controlling the states of multiple threads. Furthermore, the standard Unix utility to trace and control a child process, *ptrace*, was inadequate because it implicitly assumed only one application thread.

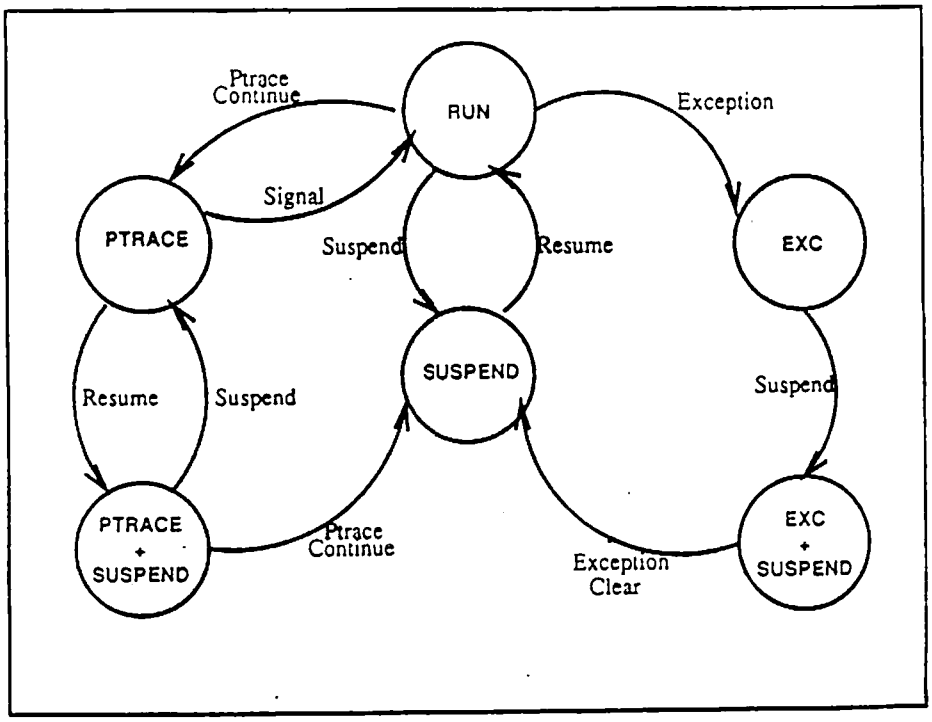

Figure 3.2: Application Threads State Diagram

The Caswell-Black gdb, henceforth referred to simply as gdb, dealt with these low-level thread control problems. It has enhanced thread control logic and an extended ptrace to handle individual threads in a task. Gdb interacts with the operating system to recognize multiple threads in a single task as does the enhanced ptrace. It also bases actions on exception handling; each exception generates a message that identifies its originating thread and task. Any exceptions not handled by the application-specific exception handlers, are sent to gdb's exception port. Then gdb, when awakened by the child (the debugged application), checks its exception port to read the status of its child. Concurrent exceptions (e.g., multiple threads hitting breakpoints simultaneously) are simply converted into multiple messages, a distinct advantage over the standard Unix paradigm where they become signals that are serialized. At this point, gdb merely dequeues the messages at the exception port and can determine the complete state of its traced application. In an interrupt based environment each signal would casuse the state to be saved, resulting in significant stack overhead.

Figure 3.2 describes the process control logic employed by gdb. It should be noted that the Mach suspend and resume mechanisms are based on reference counts (e.g., a twice suspended thread must be resumed twice). Also, task and thread mechanisms are connected; a thread may run only if both it and its task are not suspended.

A quick summary of the diagram (the detailed explanation of this diagram is found in the Caswell-Black paper) is that initially all application threads are in the run state. Exceptions move the application to the *exc* state, whereas a Unix signal moves the application to the *ptrace* state. Usually exceptions are caused by program errors (divide by zero, invalid memory reference, and etc.) whereas signals are messages (timer clock finished, thread received asynchronous input, and etc.). In the *exc* state all threads are suspended, putting the application in the *exc + suspend* state. Once these exception messages are cleared, the debugger moves to the *suspend* state waiting for user manipulation. Similarly, in *ptrace*, suspension of threads places the application in the *ptrace + suspend* state, and clearing the signal moves it to the *suspend* state. In the *suspend* state, the user specifies which threads should be resumed and arranges for the application to move back into the *run* state. A resumption can be a complete continue or simply a single step of one thread.

Mach and gdb were ported by Charlie Perkins to the IBM 8CE multiprocessor[12] which was the hardware platform for the MpD debug-

ger. A complete SCE machine consists of eight processing elements, each with an RT ROMP 125 CPU and 8MB local memory, and a 16MB global shared memory. The eight processing elements share a fast inter-processor-communication bus that can access the global as well as the local memories of each processor. A local memory reference takes longer to resolve; if a particular location is frequently referenced it is pinned to shared memory. An IBM RT 125 acts as a frontend to the SCE, providing all I/O facilities [12]. Mach maintains a single global execution queue in shared memory and threads are dispatched to a free processor. The RT frontend also runs Mach and uses a server process on a specialized AT bus to communicate with the SCE. The diagram (Figure 3.3) below shows the hardware scheme.

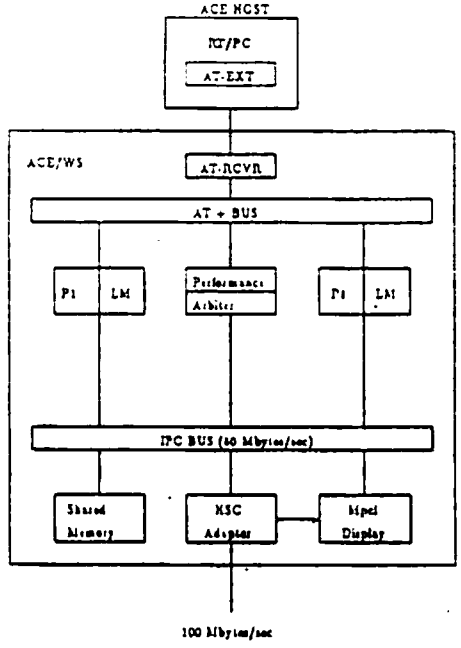

Figure 3.3: SCE Hardware Organization

Gdb running on the SCE under Mach uses the master/slave paradigm. One gdb execution image runs on the master processor and it controls the application via the operating system primitives explained earlier. The application runs in normal fashion, except wherever a breakpoint is inserted, the object code of the corresponding thread is modified. Since the code is shared among threads, inserted breakpoints affect all threads executing that portion. So as the threads of an application migrate to the various

processors, the breakpoint assertions encountered are reported to the debugger via software traps. The debugger gains control over the application via these traps and suspends the execution of the relevant threads. Similarly restarting a thread involves removing or disabling the active breakpoint and designating the thread as executable. Part of the work done by Caswell and Black was to add a mechanism to single out threads for manipulation.

This describes the complete background for the development of MpD. The Mach operating system abstractions and handling of distributed computation was extremely convenient. It had in place the mechanisms to develop parallel programs quickly and easily. The operating system provided a robust, friendly programming environment. The SCE also was a valuable machine, since it was a hands-on multiprocessor. It had several serious bugs which we will explore, but overall was a useful machine.

4 Overview of Implementation

To a large extent, debugging on a multiprocessor means being able to track multiple concurrent threads and present their actions in an understandable user interface. This is essentially a problem of modeling the program's execution flow. In our environment, a program begins with a single thread of execution, which can fork into a finite "bundle" of independent concurrent threads. Each child thread is capable of continuing the forking in a recursive manner, possibly causing an exponential branching of the original execution stream. We define a set of events $\{e_1, e_2, e_3, ...\}$, each in a separate thread $\{T_1, T_2, T_3, ...\}$, concurrent *if and only if* no ordering can be placed on their execution sequence with respect to each other. That is, whenever events are concurrent, we mean they are causally independent, not necessarily actually unrelated. Thus, events in concurrent threads can occur simultaneously on different processors or sequentially in any permutation on a shared processor. A program event stream is recursively defined to be finished when every one of its child threads ceases to be active.

For an example, we refer back to the earlier Program 1 on page 11-12. The program creates four separate Cthreads to compute a final value. Each cthread has a separate instantiation of the function `rtn()`. In each of these run-time copies (the object code, remember, is shared) the main computation, done within the for-loop, is completely independent of any other thread and thus executes concurrently. However, the initial `printf()`

contains a lock to serialize I/O (necessary on our machine since the RT frontend handled all I/O). MpD detects this causal dependence and warns the user with message indicating the threads executing have a dependency. The user can choose to ignore the warning statements, or instruct the debugger to disregard synchronizations involving a particular lock variable by using the skipon command - pr_lock in this case.

Timely event generation is of fundamental importance to the debugger. Users describe abstractly by the Data Path Expression (DPE) notation how they believe the program executes; these DPEs are translated by gdb into specific breakpoints by gdb (shown by the lines from DPE parser to GDB) which indicate when an event has occurred. A breakpoint is simply an interruption to the normal execution of a program whenever a specified condition is matched[17]. The conditions allowed by MpD are the same as those allowed by gdb. These include a trace variable reaching a prescribed value, a specific program line being reached, an if-condition being satisfied, and a monitored function being entered. For a single thread of execution, breakpointing is straightforward since multiple conditions are never satisfied simultaneously. Concurrently executing threads, however, complicate breakpointing by not only satisfying conditions simultaneously but also by passing information between them. Thus, the interrupt mechanism controlling a task with concurrent threads needs to recognize explicit concurrent breakpoints as well as those resulting from threads sharing information. Moreover, the breakpointing mechanism should be thread-specific. It is unnecessary to suspend all threads executing within a task, only those affected by the breakpoint should be stopped.

So we can say the *basic goal* of the MpD debugger is to compare actual concurrent execution of a program against the user's imagined model. To achieve this goal, MpD's implementation was broken down into several parts.

The initial step was gdb integration. In its original state, gdb did not work in the Mach environment on the RT and 8CE hardware. The various malfunctions, it was discovered, were due to compiler symbol table incompatibilities and kernel problems.

After getting basic gdb working, the second step was to specify a language set for the DPE recognizer. The set chosen was a subset of the DPE hierarchy powerful enough to model "safe concurrency"[15] and limited enough to reflect the finite resources of the development system.

The language chosen dictated the development of the Predecessor Automata (PA) and modeling power of the Recognizer. It also dictated the

specifics of the DPE parser. For each DPE specified, the parser constructs the corresponding predecessor automata. In addition, it attaches actions to the breakpoints wherever indicated. Several dynamic structures are created during execution to do resource monitoring (lock usage, message passing, and etc.) apart from DPE verification. Figure 4.1 shows the system.

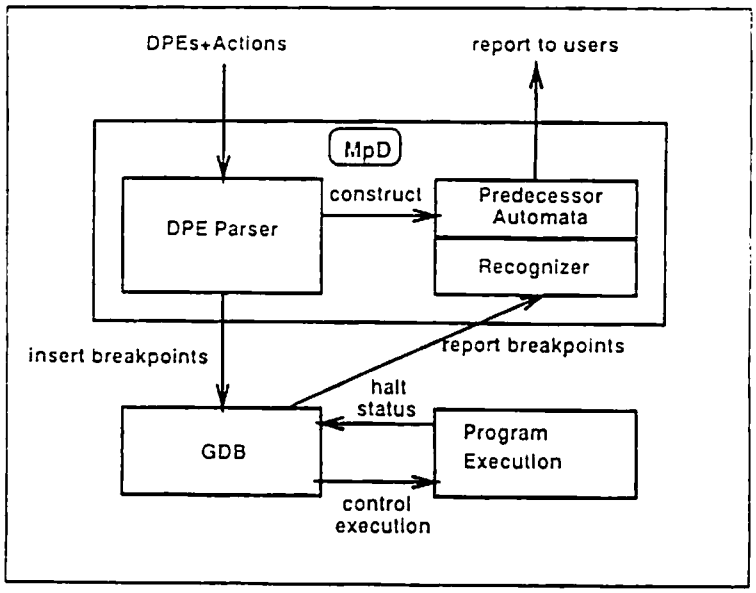

Figure 4.1: MpD System

The final phase involved building the Recognizer and linking it to the rest of the system. The Recognizer's function is to traverse the PA to see if the input event stream cause the PA to enter a final state. It also analyzes the program execution and gives a number of useful statistics about each trial run. While creating each of these portions, the user interface was developed. An on-line manual system was implemented for the system.

This was the logical breakdown of the problem. One of the early problems with this approach, however, was that the initial step took quite a bit more time than expected. The gdb-related bugs took a significant amount of time to track down. The fix in each case was not as difficult as identifying the originating circumstances. The end result of this was that a bit of each portion of the system got worked on rather than orderly completion of one segment after the other.

4.1 Gdb Integration

There were three major problems with gdb on the SCE. Two of these were related to the symbol table generated by the compiler and the third was a hardware bug that was specific to the SCE.

Gdb in its original state was unable to read the compiler's symbol table. The symbol table format gdb expected was the standard Common Object File Format (COFF), but the hc2[2] compiler generated a variation from it. As an aside, the hc2 compiler was used instead of the standard C-compiler (cc), because on the RT-125 Mach system cc produced floating point errors.

Refen Koh did the initial work in rewriting the symbol table reading utilities of gdb. Basically, in each data section of an object file, the virtual address of the variables was incorrectly computed. The fix was to initialize the offset pointer in some locations and to calculate an offset in other locations; this was a bit tricky to implement because the code was scattered across several large files with little documentation explaining the address arithmetic.

The second symbol table problem was also memory related. In this case the wrong address was being computed for all local variables in functions. This problem surfaced in two instances.

It was noticed that any time a local variable was accessed in the debugger, gdb would look at the address of the local variable *plus* an offset. The offset in each case turned out to be the size all the local variables occupied. So for example if a function had a single local integer variable, any time gdb referenced that variable (suppose the user requested to print the contents of the variable) gdb would look in the address of the variable plus four for the variable. The offset is four because an integer occupies four bytes on the SCE. This additional offset is of course wrong; gdb should simply look at the address of the variable. This behavior was due to the hc2 compiler's peculiar symbol table format because the standard C-compiler, cc, had no such problem.

The solution implemented for the problem involved reading the brackets "{}" of a function. Any time an open bracket "{" was read, a flag indicated how much of the local memory was used within the block. With the amount of space computed, the offset was used to correct the locations gdb had for the local variables. This patch works for functions without multiply embedded blocks declaring variables; a single block declares all

[2]The hc2 compiler is a trademark of MetaWare from Palo Alto, CA.

local variables used in the function.

Local variables in functions under two circumstances still cause problems. The first is when within a body of a function, a call to another function is made (could be itself). If the arguments associated with that function occupy more memory than four integers, the local variables in the caller function are erroneously computed by gdb. This is because on the RT Romp processor (the processor used on an SCE) 4 registers are available as parameters to a function call. Any arguments in excess of four are placed on the stack. The excess causes a mis-alignment in gdb. An example is given below:

```
foo()
{
    int loc_int;

    goo(1, 2, 3, 4, 5);
            .
            .
            .
}
```

Figure 4.1.1: Mis-Alignment Sample Program

The address of loc_int in function foo() would be computed by gdb to be actually 4 bytes less than it should be. Suppose gdb sees the local address for loc_int to be 0xde88, the actual address should be 0xde8c. This increase by 4 is due to the function call goo() whose call arguments take up 5 integers in memory. This problem persists because gdb has no obvious method of determining when a function call is occurring while reading the symbol table.

The second situation resulting in gdb misbehavior also relates to incorrect address calculation. Local variables that are declared within the context of a local block in a function can have an incorrect address if more than one local block exists. An example is given below:

```
foo()
{
    int loc_int;
    {
        int loc_loc_int1;
                .
                .
                .
    }
    {
        int loc_loc_int2;
                .
                .
                .
    }
}
```

Figure 4.1.2: Block Alignment Program

For the variable loc_loc_int1 gdb would have an incorrect address. The address would be offset down by the amount of space occupied by loc_loc_int2 (in our system, 4 bytes). This problem exists for blocks within blocks as well. It is always the variables in the outer blocks that are incorrectly offset while the variables in the innermost block have the correct addresses. Programmers rarely use multiple blocks in a single function, so we chose to ignore it. The problem was also tangential to our main interest.

The third gdb problem was related to controlling the debugged application on the SCE. Originally, after the first breakpoint was reached, attempting to continue or single step was impossible. The debugged application would resume, finish execution, and then hang. This was due to a hardware bug for which the kernel never implemented a software patch.

The INSTEP_BIT was the cause of failure. When this bit is turned on, the kernel is supposed to single instruction step the application. However, in our gdb this failed to occur—this manifested itself as a message being timed out.

The single instruction step is crucial for breakpoints since it is the basis on which they are implemented. Basically, each time a breakpoint is inserted, the actual machine code is modified and a special instruction

replaces it. This instruction, when encountered during execution, generates a software trap at which point the debugger gains control.

To continue after this point, the original contents of the machine code is put back and a single instruction step is done to proceed past this point. After this single instruction is finished, the breakpoint instruction is reinserted and execution continues normally. The motive for the double-stop is that the breakpoint encountered after the first breakpoint may be this very same breakpoint (the breakpoint could have been placed in a loop, recursive call, goto, etc.). This necessitates that a breakpoint be replaced immediately after its removal.

On the 8CE the failure of the single instruction step caused the debugger to lose control after the very first breakpoint. The software trap that was never generated manifested itself as a message timing out. Normally a message comes back to gdb indicating the child has halted in some state and is ready to be manipulated. Investigating this problem led us to the following comment in a kernel include file:

```
/* simulated, may be unreliable in hardware */
```

A patch was implemented for this fatal error. Our idea was to generate for every breakpoint, two adjacent breakpoints. When the first is encountered, it is removed and the second is inserted. After the second is reached, it is removed and the original is replaced. This "jitter-step" simulates the instruction step.

In our implementation, the jitter-step is implemented as a toggle. Basically, when a breakpoint is encountered, its shadow breakpoint replaces it. When the shadow breakpoint is reached, its shadow (the real breakpoint) replaces it. The diagram below shows this.

The solution proved to be effective for overcoming the hardware bug. It however had a slight defect that was again due to a hardware bug. If the user placed a breakpoint at a line of code calling a library function, the shadow breakpoint computation would not work. Instead of placing the shadow breakpoint 2 bytes ahead, we were forced to place it on the next executable line of code. The user had to specify this next line of code with an additional parameter to normal breakpoint command.

The last minor problem we had with gdb was that certain sequences of actions attached to breakpoints caused core dumps. Gdb would lose track of how many actions there actually were and reference an illegal address.

This was due to a subtle memory bug which was fixed. In addition, a new interface was added for attaching actions to breakpoints. Whereas originally the breakpoints had to be inserted, and then the actions had to be attached separately, this could all be done in one continuous stream. Multiple breakpoints (and their associated actions) can be entered with the "actions" command. Below we show breakpoints 1 and 2 being inserted in the file.c having print actions attached to them.

```
break file.c:34 {
   p foo
}
break file.c:555 {
   p goo
}
end
```

4.2 DPE Language Set

The MpD layer of gdb is based on the Data Path Expression[15] language. A DPE consists of up to three components: one or more program events, zero or more relations among events, and zero or more actions. Events and the relations among them specify the behavior of program execution, while actions are performed by the debugger when the particular behavior is recognized during execution. There may be multiple DPEs considered for the same execution.

A simple event is an assertion that a particular point in the program has been reached, for example, a particular subroutine has been called. More complex events may include conditions, such as a particular variable address is being accessed at that point. The types of conditions that can be expressed depend on the implementation. A set of operators, sequencing (;), exclusive selection (+), repetition (*), and concurrency (&) express the basic relationships among events. Another operator, concurrent closure (@), is included in the extended DPE language but is not implemented in MpD. The meaning of these operators is summarized in the following table (Figure 4.1.1).

Symbol	*Meaning*	*xpression*	*Description*
;	Sequential	$a;b$	Event a causally precedes event b.
+	Exclusive Or	$a+b$	Event a or event b occurs, not both.
*	Repetition	$a*$	$\epsilon + a + a;a + a;a;a + ...$
&	Concurrent	$a\&b$	Event a and b occur concurrently.

Figure 4.2.1: DPE operators

When the same event name appears multiple times in a DPE, it refers to a different instance of the event. Thus "A*" means zero or more sequential occurrences of "A," and "A & A" means two concurrent executions of distinct events named "A." In the DPE "(A; S; B) & (C; S; D)" the event "S" occurs twice, as opposed to a synchronization. To describe implicit synchronization, we use a special symbol "$" as a prefix to an event to indicate this event is a synchronization. The previous example with implicit synchronization would be "(A; $S; B) & (C; $S; D)." An explicit synchronization would be "(A & C); S; (B & C)."

The DPE language implemented is a subset of the DPE hierarchy, called safe DPEs[15].

(A;B) & (C;D)

(A;B;(C & D);E;F) & (G;H;K)

A & B & C & D & E

In MpD, an event is defined as a breakpoint inserted into the program execution image using gdb's breakpoint facilities. So numerical breakpoint labels rather than mnemonic event names are used. The DPEs are constructed as expressions on breakpoint numbers. To illustrate actual DPE usage in MpD the program below calculates the gradient of the function $f(x,y,z) = x^4z^2 - yx^3z^2 + 5$ using three Cthreads, one for each coordinate axis.

```
1  /* Program calculates gradient of f(x,y,z) = x^4z^2 - yx^3z^2 + 5 */
2
3  #include <cthreads.h> /* Cthreads include file */
4
5  mutex_t          pr_lock;
```

```
6  int
x_axis(), y_axis(), z_axis();  /* functions to be forked */
7  float           x_grad, y_grad, z_grad;
8
9   main()
10  {
11    cthread_t t1, t2, t3;
12
13    setbuf(stdout, NULL);
14    cthread_init();
15    pr_lock = mutex_alloc();
16 /* create 3 Cthreads by forking */
17
18    t1 = cthread_fork( x_axis, 33.2, 2.33, 20.4);
19    t2 = cthread_fork( y_axis, 33.2, 2.33, 20.4);
20    t3 = cthread_fork( z_axis, 33.2, 2.33, 20.4);
21
22    cthread_join(t1);      /* join the Cthreads after execution */
23    cthread_join(t2);
24    cthread_join(t3);
25    printf("x_grad=%6.2f  y_grad=%6.2f  z_grad=%6.2f.\n",
26       x_grad, y_grad, z_grad);
27  }
28
29
30  x_axis(xval, yval, zval)      /* evaluate partial with respect to x */
31    float  xval, yval, zval;
32  {
33    int ctr, ctr2;
34
35    mutex_lock(pr_lock);
36    printf("Thread x partial.\n");
37    mutex_unlock(pr_lock);
38    x_grad = 4 * xval * xval * xval * zval * zval
39  }
40
41
42  y_axis(xval, yval, zval)      /* evaluate partial with respect to y */
43    float  xval, yval, zval;
44  {
45    int ctr, ctr2;
```

```
46
47    mutex_lock(pr_lock);
48    printf("Thread y partial.\n");
49    mutex_unlock(pr_lock);
50    y_grad = -xval * xval * xval * zval * zval;
51  }
52
53
54  z_axis(xval, yval, zval)  /* evaluate partial with respect to z */
55     float  xval, yval, zval;
56  {
57     int ctr, ctr2;
58
59    mutex_lock(pr_lock);
60    printf("Thread z partial.\n");
61    mutex_unlock(pr_lock);
62    z_grad = 2 * xval * xval * xval * xval * zval -
63             2 * yval * xval * xval * xval * zval;
64  }
```

Program 2: Gradient Calculation Example

Now we insert a breakpoint at line numbers 15, 36, 38, 48, 50, 62, and 25 generating the breakpoint numbers 1, 2, 3, 4, 5, 6, and 7 respectively. The following are valid DPEs for this program provided the corresponding breakpoints are enabled:

- (2;3) & (4;5) — Confirms threads `x_axis` and `y_axis` are concurrent.
- 2 & 4 & 6 — Confirms threads `x_axis`, `y_axis`, and `z_axis` are concurrent.
- 1; (2 & 4 & 6); 7 — Confirms three threads are forked from a single thread and later re-join it.

Whenever gdb reaches a breakpoint, it calls MpD to inform it of its halting status. The status indicates the current breakpoint, thread, resources owned by the thread, and arguments to the breakpoint (a function call, for example, has parameters as arguments). MpD analyzes this information

and returns a value indicating whether gdb should execute the actions associated with that breakpoint. The actions may involve artificial variables (debugger as opposed to program variables), I/O, and thread manipulation.

The decision to execute the actions depends on whether or not MpD successfully matched the input breakpoint with a transition in the PA (PAs, remember, are used recognize DPEs). This process is similar to an FSA making a state to state move based on the input transition token. Actions can be assigned to more than just single breakpoints: they can be specified to be carried out only after an entire DPE is recognized—i.e., the associated PA is in the final state.

We will illustrate these concepts by attaching actions to the DPEs for Program 2. For example, in the DPE (2;3) & (4;5) and breakpoint 5 we can attach {`print y_grad`}. The value of `y_grad` will be printed when the program encounters breakpoint 5 provided the breakpoint fits the execution stream described by the DPE. That is, breakpoint 5 would have to occur sequentially after breakpoint 4. If breakpoints 2 and 3 are not current to 4 and 5 or breakpoints 2 and 3 do not occur sequentially, the DPE would not be matched and the user informed of this.

Let us now consider attaching the action {`print (x_grad + y_grad + z_grad)`} to just the DPE "2 & 4 & 6." In our implementation, to force this action to be done *only* after the entire DPE we use the `if_concurrent` command. We write the action as:

```
if_concurrent
   print x_grad + y_grad + z_grad
end
```

This action is attached to breakpoints 2, 4, and 6 since all three are possibly the last event completing the concurrent DPE. This awkward interface for attaching actions to DPEs containing the concurrent operator is due to breakpoints + actions being entered separately from the DPE. We attempted to reuse gdb code as much as possible and combining DPE notation with breakpoint + actions would have this difficult. It would have required completely re-writing the existing command parsing routines - a very time consuming task.

A DPE such as 2*{print foo} containing the repetition operator (*) causes an interesting dilemma. Actions attached to the breakpoint associated with the "*" are executed the very first time the breakpoint is seen

rather than the last time the event is seen. This policy stems from being unable to determine when an event has occurred for the last time. Continuing past a repetition breakpoint after its first occurrence without executing its attached actions could result in the actions never being executed. So executing the actions on the first and only instance of the breakpoint is a compromise (note that this 2^* operator in this case really performs as 2^+ operator in regular expression notation, so far as actions are concerned).

A user wanting to adhere to executing actions attached only to 2^*, for example, can use artificial variables to force actions to occur only on the last occurrence of event 2 if they are willing to execute the program twice. On the first run an artificial variable counter can be set up to see how many times the event 2 occurs; suppose this is n. Then the DPE can be reformed as 2; 2; 2; ... (n-times) and to the last "2" we attach the actions. This would have the effect of 2+ actions. Another method would be to simply re-run the program until the counter reaches this prescribed value, n, then execute the actions.

As an additional example we demonstrates how a user could test to see if a concurrent read and write was occurring in a program—an undesirable race condition. Suppose we have two functions, Read(X) and Write(X), that do our low-level reading and writing. We would like to write a DPE that tests if Read(X) and Write(X) occur concurrently on the same variable X.

1. Initialize two gdb artificial variables for later use.
2. Use gdb action commandfor defining events and corresponding actions.
3. Use the event command (alias for gdb's break) to define the event of entering Read function.
4. Associate an action with the individual Read event to save the address of the function argument.
5. Use the if_concurrent command added to gdb to perform the ensuing actions(s) only when the entire concurrent DPE this breakpoint is a member of is finished.
6. Repeat the above three steps for the Write function.
7. End gdb actions shell, which causes gdb to print the numbered labels assigned to the two breakpoints.

8. Use the dpe command to enter the dpe command shell.

9. Construct the "10 & 11" DPE, which implicitly constructs the corresponding PA.

10. Quit the dpe shell and run the program.

11. Use our state command to request information about DPE analysis.

```
(gdb) set $read_var = 0
(gdb) set $write_var = 1
(gdb) actions
event Read {
set $read_var = X
if_concurrent
  if ($read_var == $write_var) echo "concurrent access"
}
event Write {
set $write_var = X
if_concurrent
  if ($read_var == $write_var) echo "concurrent access"
}
end
Breakpoint 10 0x123:  file Demo.c, line 37
Breakpoint 11 0x456:  file Demo.c, line 99
(gdb) dpe
DPE> dpe 10 & 11
DPE> quit
(gdb) run
(gdb) state
```

Figure 4.2.2: DPE Breakpont Shell

The addition of actions, in general, to the DPE syntax increases its expressive power in the Chomsky language hierarchy[14]. For example, we can recognize $a^n b^n$, a classic example of a non-regular set. Here we can assume that "a" and "b" correspond to two distinct events, say that of entering two different functions. So in testing for $a^n b^n$ we are testing to see if the functions were accessed an equal number of times during execution. We proceed by having two artificial variables, one attached to the event "a" and the other to the event "b." The artificial variables are incremented

by one each time we see its respective event. At the end of execution we can test to see if the two artificial variables are equal. If equal, we accept, otherwise reject. This testing can also be done while incrementing. A message can be printed whether the two variables are equal each time either variable is incremented; the last message indicates the final status.

Since gdb allows artificial variables, artificial arrays, calling of global functions, and general arithmetic computation, the actions essentially make the language a Turing machine equivalent. This is obvious if we consider the Turing machine equivalent, a two counter. A two-counter is an off-line Turing machine whose storage tapes are semi-infinite, and whose tape alphabets contain only two symbols, Z and B (blank). The symbol Z, which serves as a bottom of stack marker, appears initially on the cell scanned by the tape head and may never appear on any other cell. An integer i can be stored by moving the tape head i cells to the right of Z. A stored number can be incremented or decremented by moving the tape head right or left. We can test whether a number is zero by checking whether Z is scanned by the head, but we cannot directly test whether two numbers are equal. A two counter can be simulated by using two artificial arrays and two artificial variables. The variables can be indices into the arrays, completing the two-counter. Obviously, the semi-infinite tape requirement fails in the real implementation, but theoretically the modeling power is apparent. Gdb actions are capable of doing computation, I/O, and determining a program's state.

4.3 Predecessor Automata Construction

DPEs are recognized during execution by traversing the corresponding Predecessor Automata (PA) they generate. If one or more DPE is active during execution, `MpD` will inform the user which DPEs were matched, if any, during the application's execution. PAs are analogous to FSAs, but PAs recognize concurrent partially ordered event streams expressible as DPEs while FSAs recognize only sequential totally ordered event streams expressible as regular expressions. Note that all regular expressions are also DPEs, but those containing any concurrent operators are not regular expressions.

An FSA makes a transition from state to state when the input event matches the event labeling the transition. Certain states are designated as final states, and if the input ends when the FSA is in a final state, or a final state can be reached by null transitions, the corresponding regular

expression has been matched by the input event stream, otherwise not.

Similarly, a PA makes a transition from state to state when its input event plus an encoding of the set of events causally preceding the current event match the (event, predecessor set) pair label of the transition. Note that only the immediately preceding event in each thread of control is contained in the predecessor set, not the full history of events; there are multiple predecessors when the event reflects a synchronization among multiple threads. As with FSAs, there are final states, and the partially ordered event stream has been matched if and only if the PA is in the final state when the input ends, or a final state can be reached via null transitions.

As an example consider the DPE "1:(2 & 4 & 6);7" described in section 4.2. Event 1 has no predecessors. Events 2, 4, and 6 all occur sequentially after event 1 and thus event 1 is a predecessor for each of them. Furthermore, events 2, 4, and 6 are concurrent to each other. Although these events do not belong to the same thread as event 1, they occur sequentially after it because event 1's thread created their respective threads. Now event 7 has as its predecessors events 2, 4, and 6, since it occurs after these threads joined it. In general for a DPE, the PA it creates defines the execution relationships and run-time checks in the Recognizer verify these relationships.

PAs are constructed using the DPE parse tree. The DPE parser, written using Lex and Yacc, creates a binary parse tree from the user's DPE. Afterwards, a postorder traversal of the tree constructs the PA. The relation operators (;, *, +, and &) used in the DPE form the internal nodes of the tree, and the leaves are the breakpoint numbers used in the DPE. The DPEs "a ; b", "a*", "a + b + c", and "a & b" create the following trees (Figure 4.3.1):

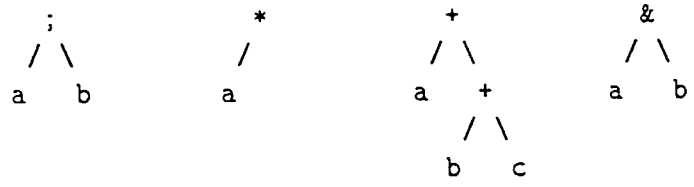

Figure 4.3.1: DPE Parse Tree

A basic idea of the postorder traversal is to create for each node a left PA and a right PA. The two sub-PAs are then fused together according to

the operator at the internal node. Since this algorithm is done in postorder, it begins at the bottom of the tree and carries up recursively resulting in one final PA.

For the sequential operator (;) we simply concatenate the left and right PA, making the final state of the left PA the start state of the right PA. For the selection operator (+) we make them share a common starting state, leaving the transitions to their respective second states intact. The resulting PA branches into the two sub-PAs and then converges again on a common final state which is gotten to by null transitions. For the repetition operator (*) a loop as well as a null transition are added. A null transition from the start to final state is added as is another null transition from the final to start state. For the concurrent operator (&), the PAs are *not* combined in an interleaved fashion. Rather they are fused together as a group.

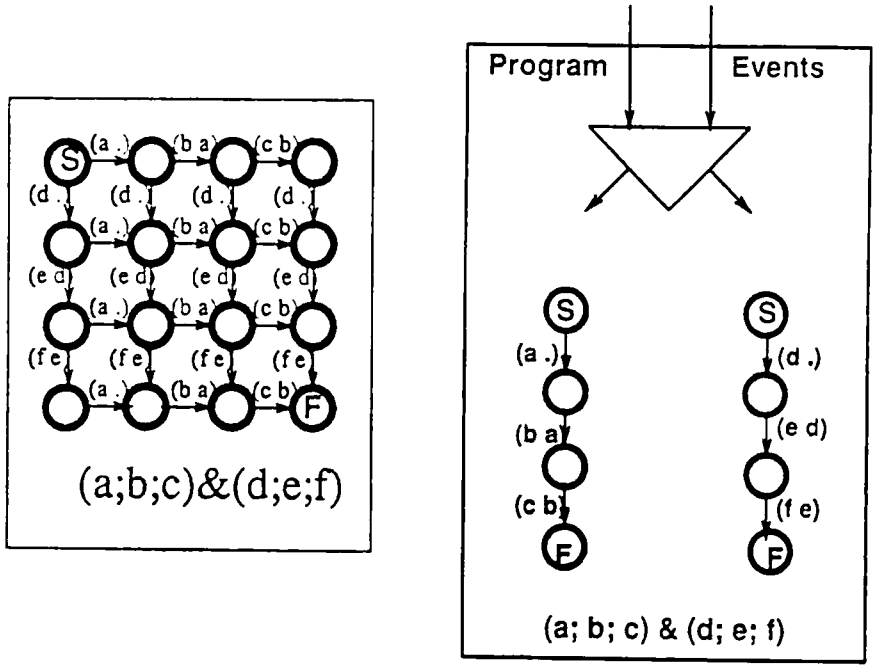

Figure 4.3.3: Interleaved vs. Implemented version of Concurrent Operator

When the Recognizer encounters such a group, it knows to finish processing it before proceeding. That is, each member of the group must be in a final state before any transition beyond the group can be taken. For example, in the DPE "((a ; b) & (c ; d)) ; e" the Recognizer will not take the transition "e" until the sub-PAs "(a ; b)" and "(c ; d))" are both in a final state (and they occurred concurrently with respect to each other). If

"e" comes to early the PA fails to move and the user is informed the DPE did not match.

The motive for not interleaving is simple. Consider the example (a;b;c) & (d;e;f). The total number of events in this DPE is six. In diagram 4.3.3 we compare the interleaving states vs. our implemented approach. Our implemented model saves memory at the cost of adding logic (and thus overhead) at the initial concurrent set state. This state decides, when given an input transition, which machine receives it. They both describe a path length of six with $(3+3)!/(3!*3!)$ possible execution paths.

In general, for such situations, if we have A sequential events on the left side of a concurrent operator (sub-expressions such a*, (a+b), etc. are treated as one event), B sequential events on the right side, all the possible orderings would be $(A+B)!/(A!*B!)$. This derivation describes the composition of the automaton, A=$a_1;a_2;a_3;...a_m$ and B=$b_1;b_2;b_3;...;b_n$, results in an ordering of length m+n that preserves the original ordering $a_1;a_2;a_3;...;a_m$ and $b_1;b_2;b_3;...b_n$ (Figure 4.3.4). The 3 automaton case A=$a_1...a_m$,B=$b_1...b_n$, and C=$c_1...c_p$ of path lengths respectively A, B, and C, we have $(A+B+C)!/(A!+B!+C!)$ and the composition forms a rectangular solid with start and final states at the opposite ends of the longest diagonal. The complete generalization where we have $A_1\&A_2\&A_3...\&A_n$ the composition becomes an n-dimensional solid with $(A_1+A_2+A_3...+A_n)!/(A_1!*A_2!*A_3!*...*A_n!)$ possible paths.

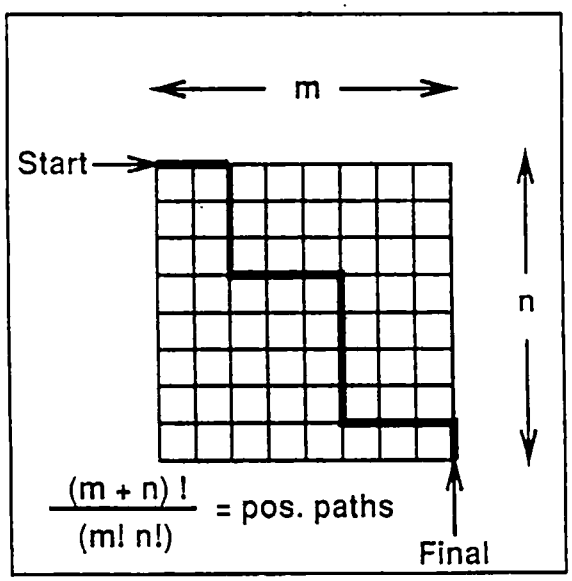

Figure 4.3.4: All Concurrent Execution Possible Paths

The memory space required to assemble such solids is a function of their dimension. For the simple case A=$a1; a2; ...; an$ and B=$b1; b2; b3; ...; bm$. A & B, n*m cells are needed. Memory for the general case occupies $A_1 * A_2 * A_3 * ... * A_n$ cells. The cell notation can be represented in programming a corresponding n-dimensional array. Each state occupies one element in each cell and requires n transitions.

For the concurrent operator we did *not* choose the interleaved method because it was more memory intensive than the "stringing" together of concurrent PAs approach. Also, the interleaving method was a bit more complex to implement. Early on, we did an analysis of the two methods and execution speed difference between the two was negligible.

4.4 Recognizer Construction

The *Recognizer* processes each breakpoint gdb gives it by attempting to match a transition in the PA. Its basic function is to verify the relationships defined by the PA and execute actions as transitions are matched. Although the PA construction is independent of the particular concurrent programming language used, the run-time support provided by the *Recognizer* is dependent on the implementation. The ability of PAs to distinguish between concurrency and interleaving is inherently tied to the synchronization primitives used in the operating system. In our case, these were the functions provided by the Cthreads library.

When events are described as concurrent in a DPE, the PA recognizer must check two conditions. It must first check that events occur on different threads, since of course there is causal dependency between events in the same thread. This check does not suffice. There may be causal dependency caused via message passing or locking mechanisms. So the second item is to check for dependencies resulting in implicit or explicit synchronization.

On our system the dependencies were through the Cthreads functions `condition_wait`, `mutex_lock`, `mutex_unlock`, and `condition_signal`. Any time a thread accesses one of these functions, `MpD` updates some internal data structures to determine the point of synchronization. The user can verify expected synchronizations and uncover others that may be unwanted. The dependency and synchronization checks are implemented as an option the user can turn off to reduce debugger overhead.

In synchronization involving `mutex_lock` and `mutex_unlock`, a thread locks a resource, forcing other threads to wait until it is finished. If the

locking mechanisms are not properly structured, then multiple threads executing in parallel can attempt to access the same resource, which often results in undesirable race conditions.

MpD detects this situation by identifying in each thread those `mutex_locks` in different threads that are competing against each other. A special run-time table is created internally in the debugger for each thread. Each time a `mutex_lock` is attempted and a thread waits on a resource, the thread that currently owns that resource is identified. A predecessor table keeps track of which events belong to the lock owner's stream and which to the current thread's stream. The predecessor table can be accessed easily to identify quickly whether two events are concurrent or not. An example of explicit synchronization such as this is two threads trying to lock an I/O resource. One acquires access to the resource, forcing the other to wait until the owner relinquishes it.

Causal dependency is detected in an analogous manner. Whenever a thread attempts to `mutex_lock` on a variable that another thread currently owns, forcing it to wait, all `mutex_lock` waits of the owner thread are passed along via a table update to the lock-seeking thread. An example of such a dependency involving two resources and three threads is: Events in thread T_1 and thread T_2 are described as concurrent. Thread T_1 locks resource R_1, a resource needed by thread T_3. Meanwhile thread T_2 is waiting for thread T_3 to unlock resource R_2, which it cannot until it obtains R_1. Thus T_2 indirectly waits on T_1 through T_3. MpD detects this type of waiting. When T_2 waits on T_3, the resources T_3 was waiting on are inherited by T_2. Thus, we see T_1 is being waited on, so we know that events in T_1 and T_2 are not concurrent.

The Cthreads library synchronization primitives `condition_wait` and `condition_signal` can also induce causal dependency. If two (or more) threads have events specified in a DPE as concurrent, but one waits for the other due to a `condition_wait` (directly or indirectly through another unspecified thread), the concurrent relationship does not hold. MpD informs the user with messages indicating the thread waiting and location in code.

In similar fashion, the PA recognizer enforces dynamic checks for sequential relationships. One of four conditions must be satisfied.

1. The events and its predecessor are in the same thread.
2. The event and its predecessor are in different threads, but serialized via locking or waiting.

3. The event and its predecessor are in different threads and the predecessor's thread *created* the event's thread.

4. The event and its predecessor are in different threads, but the predecessor's thread has *exited* or *joined* the event's thread.

Examples of conditions three and four are the DPEs "a; (b & c)" and "(b & c); a." In the first example the thread to which event "a" belongs created the threads to which events "b" and "c" belong. In the second example events "b" and "c" belong to different threads that eventually join the thread event "a" belongs. The DPE "a & b & c" is an example of condition one. All events are described to be on different threads and concurrent. If serialization occurs during execution then these threads, although on different threads, would be sequentially executing - illustrating condition two.

Along with DPE support for multiprocess programming, specific facilities were included in MpD to aid Cthreads programmers in the Mach environment (we could have included specific support for other programming libraries). In particular, several Cthreads functions were targeted to check for common mistakes.

The most useful Cthreads-specific support included in MpD is automatic detection of deadlocks caused by an improper sequence of `mutex_locks`. Each time a thread attempts to lock a resource using `mutex_lock`, the debugger's internal causal dependency table of what resources each thread holds/wants is updated. As this table is updated, a graph of dependencies is created. If the graph forms a cycle leading back to the current thread, the user is warned of the impending deadlock with a line by line printout of the culprits. Any general semaphore lock/unlock can use this mechanism.

MpD does numerous other checks for common Cthreads programming errors. For example, the function `cthread_init` must be called prior to using any of the other Cthreads routines. Failure to do so results in corruption of memory and core dumps. MpD checks whether this function is called prior to any other Cthreads function. A cthread, once created, can be joined (via `cthread_join`) or detached (via `cthread_detach`) only once during execution. This is checked by monitoring all Cthreads to make sure only one or the other is done. Furthermore, any cthread attempting to join itself (via `cthread_join`) results in deadlock; this condition is detected and the user is warned.

Appendix A at the end contains the listings of programs illustrating

these errors. These are actual programs in which the debugger detects errors. The examples illustrate detection of deadlocks, join/detach errors, and improper use of `mutex_lock`. The programs use the Cthreads constructs available and the comments explain the program flaws.

The full set of commands available within MpD's dpe command shell are in Appendix B at the end. This shell is entered from within gdb by typing the "dpe" command.

5 Using MpD to Debug Two Concurrent Applications

In addition to using MpD on various in-house test programs, we used it to debug two externally developed applications. These applications were requested informally on the mach information newsgroup on Internet and were in no way written with MpD in mind. Nevertheless, our debugger proved to be useful in debugging the applications.

5.1 Application 1: Parallel Logic Simulator

This application was supplied by Robert Mueller-Thuns of the University of Illinois at Urbana-Champaign. It is a Cthreads implementation of a parallel logic simulator, and consists of about 1,900 lines of C code. The application ran correctly on the SCE's RT front end using the co-routines implementation of the Cthreads library (libco_threads.a). However, using the real parallel implementation of the Cthreads package on the SCE co-processor (libthreads.a), the application would run briefly then suspend itself in some unknown state. This occurred every time the application was tested; note that our approach to debugging here assumes reproducibility.

Debugging this behavior began with the construction of a DPE to determine how far along in the calling sequence the application reached before hanging. The code fragment below (Sample Code from Application 1) from the file main.c and figure 5.1.1 show the code and the breakpoints we inserted into the code to represent interesting program events.

```
54   if ( argc < 3 ) {
55      fprintf ( stderr, "[pecs]: too few arguments\n") ;
56      fprintf ( stderr, "(pecs <model file> <input file>)\n") ;
57      quit (" " ) ;
58   }
59
60   timer_mark () ;
61   initMACH () ;
62   fprintf(stderr,"[boss]: MACH initialized in %d ms\n",timer_dall());
63   ++ argv ;
64   if ( ( inFile = fopen ( *argv, "r" ) )  == NULL )
65     quit ( "cannot open model file " ) ;
66  /*
67   readModel ( *argv ) ;
68  */
69   fscanf ( inFile , "%d", &nProc ) ;
70   fscanf ( inFile , "%d", &cthread_debug ) ;
71   fclose ( inFile ) ;
72   printf("[boss]: # processes %d ( debug = %d )\n",nProc,cthread_debug);
73   ++ argv ;
74   strcpy ( fileName, *argv ) ;
```

Sample Code from Application 1

```
event main.c:54{
cont
}
event timer.c:timer_mark{ # entering function timer_mark() line 60
cont
}
event main.c:38{   # inside the function initMACH() line 61
cont
}
event main.c:74{
cont
}
end
```

Figure 5.1.1: Inserted Breakpoints

The DPE used to describe the execution was "1;2;3;4". That is, the events were expected to occur sequentially in the order specified. This DPE was confirmed.

Since we were debugging unfamiliar code, our first step was to determine how many threads were being created by the application. So in the Cthreads routine that was being forked, the following event was defined using normal gdb facilities. (setupMach.c is one of the application program files, not part of the Mach implementation.)

```
(gdb) set $i=0 #dummy counter to count number of threads
(gdb) actions
event setup { # setup is the name of forked routine (in setupMach.c)
$i = $i + 1
cont
}
end
```

The number of threads being forked was determined by printing the value of the artificial variable $i, which was 4. In the forked routine `setup()`, the C library routines, `fopen` and `printf` were called.

```
if (!(dbgf = fopen (dbgName, "w"))){ /* line 298 of setupMach.c */
    printf("[dbg] %s :", dbgName); quit("cannot open debug file");
    }
```

We defined an event for reaching this code, to which gdb assigned the breakpoint number 1.

```
event setupMACH.c:298 {
# line calling fopen and prior to calling printf
cont
}
end
```

We then defined the DPE "1 & 1 & 1 & 1", representing four concurrent threads executing this same statement causally independently. The program was executed, and this DPE was confirmed. Thus, `fopen` (as

well as `printf`) was being invoked by four threads concurrently, without synchronization.

We suspected that these functions were executed concurrently (without synchronization). The Mach Cthreads[8] manual recommends that functions in libc.a should never be called without `mutex_lock/mutex_unlock` protecting them. The library simply does not support concurrent access (the code is not reentrant). So we modified the program to protect all calls to functions in libc.a with `mutex_locks/mutex_unlocks`. The program was recompiled and re-run, and execution proceeded normally with correct results.

5.2 Application 2: Grobner Basis

This application was written by Stephen Schwab of Carnegie Mellon University. It is a Cthreads implementation of a Grobner Basis, and consists of about 5,600 lines of C code. This application in its original form did not work at all when we compiled it for the SCE; it would core dump.

The core dump, it appeared, stemmed from de-referencing a nil pointer. The first DPE was constructed to check whether or not the relevant events leading up to the point where the core dump occurred were purely sequential. The breakpoints picked were random points in the code leading up to the core dump location. The initial DPE was "1; 2; 3; 4", using the breakpoint numbers assigned by gdb for the program events shown below.

```
event Main.c:311{ # breakpoint 1
cont
}
event Gbasis.c:233{ # breakpoint 2
cont
}
event Pairs.c:51{ # breakpoint 3
cont
}
event Pairs.c:165{ # breakpoint 4, print the stack hierarchy
where
}
end
```

Figure 5.2.1: Inserted Breakpoints

This DPE was confirmed and the core dump was due to a null pointer being de-referenced. The statement below was responsible with the variable pt being null.

```
while ((pt->old==1)||(pt==p))
    pt = pt->next;
```

So the statement was changed to:

```
while ((pt && pt->old==1)||(pt==p))
    pt = pt->next;
```

This testing of pt before de-referencing was not sufficient to solve the original problem. The core dump simply passed from one location to another. A second DPE plotted the path to the new core dump location, using the events illustrated below. This was expressed as "1; 2; 3; 4; 5" for the five new breakpoints defined using gdb (the same breakpoint numbers were reused because we recompiled and restarted gdb). This DPE was matched, so we could again rule out concurrency as the source of the problem.

```
event Main.c:111{ # breakpoint 1
cont
}
event Gbasis.c:233{ # breakpoint 2
cont
}
event Pairs.c:58{ # breakpoint 3
cont
}
event Pairs.c:217{ # breakpoint 4
cont
}
event MExpo.c:111{ # breakpoint 5, location of new core dump
where          # where prints contents of stack
}
end
```

Figure 5.2.2: Inserted Breakpoints

Upon closer scrutinization of the code, a logic error was discovered in the original location. The code was corrected so that instead of passing along a nil pointer, a duplicate of the original pointer was passed on instead.

```
while ((pt && pt->old==1)||(pt==p))
    pt = pt->next;

        if (pt == NULL)
    pt = p;
```

With this correction, the application worked correctly. Note that for this case, the application did not execute at all even when run sequentially. The error we corrected was before the program split into various threads, and so no concurrency related bugs were found.

6 Evaluation

The gdb version we used as our basis was very useful. It had most of the sequential thread code written, tested, and working. Features such as breakpoint insertion and variable accessing were already in place. With the enhancements of Caswell-Black, we were able to concentrate on the higher-level debugging mechanisms. This made system development significantly faster. In addition, the on-line manual system with the shell interface, although not optimal, was handy. The user simply had to type "help" followed by the command name to ask for manual pages about the command. A list commands could be displayed by typing "help."

Now the down-side was that some of the basic needs of our project were not provided. To put them in place would have required major code re-writing of the debugger and compiler. A major system limitation was using breakpoints as the basis for event generation. The breakpoints had to be in place for MpD to know when an event occurred – no non-invasive monitoring facility to ascertain statements had been executed was available. Consequently, to properly analyze a program, the user was forced to give some thought to breakpoint placement.

In addition the breakpoint mechanism could conceivably alter execution, so a debugging session did not necessarily reflect an actual execution sequence. We minimized the impact of this by warning the user of all

possible locations where threads may interact. Hence, threads that synchronized but produced the correct results would still generate a warning that in future runs the results may differ due to the synchronization. Similarly threads that are supposed to synchronize but do not are also pointed out by the debugger.

Our system also places restrictions on the ability to deal with concurrent breakpoints. Gdb still contains code that does not support multiple threads. No mechanism exists to save the state of every thread, which would be useful for handling multiple breakpoints with a single continue, that is be able to start, stop, and re-start groups of threads. Saving state would also support conditional breakpoints[7]. It would eliminate problems such as hitting breakpoints already removed. This occurs when multiple threads hit breakpoints, and one thread removes the other breakpoints. But since the debugger does not save the state of every thread these "removed" breakpoints still appear. We felt these were low-level issues that we would ignore in order to implement high-level debugging such as event relationships.

Another aspect we chose to ignore was the MpD user interface. It lacks features such as mnemonic tags for breakpoints, automatic saving/reading of breakpoint files, editing facilities for actions, and a unified DPE/breakpoint shell. Currently users enter DPEs separately from the breakpoints and actions. The `if_concurrent` command links the two shells together. This command, as we explained earlier, is awkward to use. Improving these constructs would make a simpler more user-friendly debugging environment.

A serious weakness in our system is that references to arbitrary pointer structures and addresses cannot be detected. The only data references MpD can currently detect are breakpoint generated traps at static source code locations. These include entering a function or reaching a specific line number in the code. An arbitrary address reference cannot be detected by the debugger because there is no mechanism to test when a physical location in memory is being accessed. This limitation can be overcome, however, if hardware support is provided. A bus monitor that recognizes data addresses would be useful for tracing memory accesses. It would also lay the groundwork to developing a system that provides playback built on address references. Some of the preliminary design work for such a bus monitor has been completed by an independent party[20], but it is unclear at what point this will be integrated with the rest of the system.

The lack of a bus monitor also contributed to MpD containing machine

dependent code. On several occasions we had to resort to reading registers and stack locations. This was a result of not being able to access parameters of function calls any other way. Parameters are pushed onto the stack or into specific registers which `MpD`, on the RT Romp processor, knows how to read.

One of the most useful aspects of our system, we believe, is the way the cthreads library code is examined to check for impending deadlocks. Theoretically any standard library code could have been examined to check for improper usage. Not only could the debugger check to see if library code is mis-used, but also it could point out the precise locations. It is important to realize that what we propose is more than a static function interface checking (e.g., lint). What we suggest is that the debugger be built with the ability to check the logic constructs dynamically created by the program execution. These are not limited to deadlocks; monitoring resource queues, pinpointing message generators, and memory management are just a few other examples. Ideally, the user could select library functions (or functions) to monitor and describe the logical framework of the call. The framework would describe how the functions are to interact with each other or perhaps what state the data should be left after finishing the function. The debugger could then monitor execution to see if the framework was violated.

Consider for example monitoring the C memory allocation/deallocation routines `malloc()` and `free()`. If the debugger has access to each invocation of these system calls via an interface layer, it can keep an updated free list for the application. The list can be useful in a number of ways. If some memory access is out of range, the debugger can display the legal valid range for that piece of memory since it has the list of memory blocks and their sizes. Similarly a `free()` with bad arguments could immediately be identified since it does not map to a valid allocated piece of memory. Furthermore, we can at any time get an accurate picture of current memory usage for the application. This is particularly useful if we are attempting to detect memory leakages. It would be an interesting feature to display the memory usage as an application goes through various stages of its execution.

The system interface layer should not be based on a breakpoint paradigm. The compiler and debugger should work together to signal the monitoring process when key events occur. The signals could be written to a special file from which the debugging monitor reads. The compiler can have this an option, similar to the "-g" option most C compilers have

to preserve symbol table line numbers for debuggers. The user could still generate other events by simply using the standard breakpoint mechanism.

Incorporating and extending the ideas of Agrawal[1] would also be useful. In his work, also developed on top of gdb, the Alok debugger can generate a backward list of statements affecting the current breakpoint halted on. For example, if the debugger stopped on an assignment statement, static and dynamic analysis allows the debugger to pinpoint prior statements affecting the assignment. His debugger, however, has no notion of concurrency or relationships among breakpoints. It does a straightforward data-flow-analysis on the program utilizing a compiler-generated flow graph. A similar type of analysis would be very useful in our system, particularly if it also incorporated our mechanisms to deal with concurrency. The data-flow-analysis suggested by Agrawal, coupled with our event detection plan, would make an extremely useful debugger.

The debugger is usable in its present form on the SCE; prior to this there was no symbolic debugger available on the SCE. The concurrent mechanisms we believe are useful for things such as deadlock and race condition detection. The concurrency mechanisms are also useful to check if a parallel program's threads are executing concurrently or restrictions are causing excessive waiting. Using MpD, a better understanding of a program's concurrent behavior can gained since it identifies resource sharing and waiting among multiple threads in a program.

7 Conclusion

The primary contribution of this research was to show that the Data Path Expression debugger language and the corresponding Predecessor Automata recognizer mechanism could be supported in a real system. We have shown that the MpD paradigm is useful for detecting errors in a concurrent programming environment. We also described the difficulties we encountered implementing it. MpD so far has been used to debug two externally developed applications.

There are several improvements that could be made to the current system. The user interface and event detection scheme are two areas which, if improved, would add the most functionality. Another improvement would be to syntactically reduce DPE expressions to minimize backtracking. This is essentially equivalent to writing a regular expression reducer. The end result would be a more compact PA that does recognition more efficiently.

The Recognizer internally has several algorithms such as deadlock detection and race condition checking that are not fully optimized. Rewriting this code with faster algorithms that occupy less memory is possible. Also the final status of programs is not represented in the most readable manner. Currently for each thread we list the events that occurred during its execution. It would be nice, however, to have a graphical interface for this. Each thread could have its event stream execution drawn as interconnected events while interactions with other threads, such as synchronizations, could be highlighted. The graphs drawn in this manner, though, may run into display difficulties when trying to draw large non-planar graphs.

Another useful feature might be to have the actions that were deemed to be useful inserted into the program source code directly. Having actions be mini-programs means potentially debugging them (the actions) as well as the main program[19]. This can be particularly aggravating in situations where the action incorrectly alters the execution of the program, causing the programmer to mistakenly conclude the program is wrong when in fact it is the action.

Several important results were learned from this project. First is that a debugger can be programmed to detect a significant portion of concurrent/sequential errors in parallel programs. Reading system call stubs and keeping track of resource usage can identify problems such as deadlocks. We have also shown concurrency can be affected by the synchronization primitives available to the programmer. When these are clearly identified, the programmer can localize them to create more independent threads. Lastly we have argued, in a fully integrated system, using the operating system, compiler, and extended debugger monitor we can create a more automated environment for detecting programming errors.

8 Acknowledgements

The implementation of MpD on the 8CE was undertaken as part of a joint study with Dr. Colin G. Harrison of IBM T.J. Watson Research Center. Part of this work was done while the author was a co-op student employed by IBM in 1989-1990. The joint-study is covered under IBM agreement numbers 14640056, 1461056 and 14642053. We would like to thank Wenwey Hseush, Krish Kannan, Doug Kimmelman, P.R. Kumar, Chuck Marvin, Travis Winfrey, and Charlie Perkins for their suggestions and technical expertise.

Prof. Gail E. Kaiser's Programming Systems Laboratory is supported by National Science Foundation grants CCR-9000930, CDA-8920080 and CCR-8858029, by grants from AT&T, BNR, DEC, IBM, SRA and Xerox, by the New York State Center for Advanced Technology on Computer and Information Systems and by the NSF Engineering Research Center for Telecommunications Research.

References

[1] H. Agrawal, R. DeMillo, and E. Spafford *Efficient Debugging with Slicing and Backtracking.* IEEE Software, vol. 3 no. 8, pp. 21-28 (May 1991).

[2] R. Baron, D .Black, W. Bolosky, J. Chew, R. Draves, D. Golub, R. Rashid, A. Tevanian, and M. Young. *Mach Kernel Interface Manual.* Carnegie Mellon University Mach Manual (1988).

[3] P. Bates and J. Wileden. *Distributed Debugging Tools for Heterogeneous Distributed Systems.* ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, Madison,WI., pp. 11-22 (May 1988).

[4] D. Black, D. Golub, K. Hauth, A. Tevanian, and R. Sanzi. *The Mach Exception Handling Facility.* ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, Madison,WI, pp. 45-56 (May 1988).

[5] B. Bruegge. *Adaptability and Portability of Symbolic Debuggers.* Ph.D Thesis, Carnegie Mellon University CMU-CS-850174 (1985).

[6] B. Bruegge and P. Hibbard. *Generalized Path Expressions: A High-Level Debugging Mechanism.* Journal of Systems Software vol. 3, pp. 265-276 (April 1983).

[7] D. Caswell and D. Black. *Implementing a Mach Debugger for Multithreaded Applications.* USENIX Conference Proceddings, Washington DC, pp.25-39 (January 1990).

[8] E. Cooper. *Mach Cthreads.* Carnegie Mellon University Mach Manual (1988).

[9] S. Deodhar, S. Jain, and D. Basu. *Global Debugging for Mulitiprocessor Systems.* Tencon, Seoul, South Korea, pp. 796-800 (1987).

[10] Dennis L. Doubleday. *The Durra Application Debugger/Monitor* Technical Report CMU/WEI-89-TR-32 ESD-TR-89-43 (September 1989).

[11] I.J. Elshoff. *A Distributed Debugger for Amoeba.* SIGPLAN Parallel and Distributed Debugging,Madison,WI, pp. 1-10 (May 1988).

[12] A. Garcia, D. Foster, and R. Freitas. *The Advanced Computing Environment Multiprocessor Workstation.* IBM Technical Report RC 14491 (1989).

[13] J. Hennesy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann (1990).

[14] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley (1979).

[15] W. Hseush and G. Kaiser. *Modeling Concurrency in Parallel Debugging.* 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seattle, WA, pp. 11-20 (March 1990).

[16] P. Lauer and R. Campell. *Formal Semantics of a Class of High-Level Primitives for Coordinating Concurrent Processes.* Acta Informatica, vol. 5, pp. 297-332 (1975).

[17] B. Lazzerini and L. Lopriore. *Abstraction Mechanisms for Event Control in Program Debugging.* IEEE Transactions on Software Engineering, vol. 15, no. 7, pp. 890-901 (July 1989).

[18] B. Miller and J. Choi. *Breakpoints and Halting in Distributed Programs.* 8th International Conference on Distributed Computing Systems, San Jose, CA, pp. 316-323 (September 1988).

[19] R. Olsson, R. Crawford, and W. Ho. *Dalek: A GNU Improved Programmable Debugger.* USENIX Summer Conference, Anaheim, CA, pp. 221-231 (June 1990).

[20] C. Harrison. Private Communication, (May 1990).

[21] K. Ponamgi,W. Hseush, and G. Kaiser. *Debugging Multi-Threaded Programs with* `MpD`. IEEE Software, vol. 8, no. 3, pp. 37-43 (May 1991).

[22] R. Rosen. *Parasight: A Concurrent Debugger* IEEE Software Engineering Technologies, pp. 145-162 (May 1988)

[23] R. Stallman. *GDB Manual: The GNU Source-Level Debugger.* V2.5 Free Software Foundations (1988).

[24] A. Tanenbaum. *Computer Networks.* Prentice Hall (1988).

```
/*
        This program contains a deadlock. T..  eads are
    are attempting to share resources, but one gets  while
    the other R2. They keep their respective resour..  and
    continue to wait for the other thread to give up.
*/
#include  <stdio.h>
#include  <cthreads.h>
#include  "gdb_interface.h"

#define  OFF   0
#define  ON    1
#define  ACKNOWLEDGED 3

mutex_t pr_lock;
mutex_t mem_lock;

int    shared_variable=OFF;
int    thread1(), thread2();
```

main

```
main()
{
            cthread_t  t1,t2;

            Cthread_init();
            pr_lock = mutex_alloc();
            mem_lock = mutex_alloc();

            printf("%d: Ready?", getpid());
            getchar();

            t1 = Cthread_fork(thread1, getpid());
            t2 = Cthread_fork(thread2, getpid());

            Cthread_join(t1);
            Cthread_join(t2);

            printf("\nAll threads done.\n");
}
```

thread1

```
thread1(parent_pid)
int    parent_pid;
{

            Cthread_begin();

            mutex_lock(pr_lock);      /* resource 1 */
            cthread_yield();
            mutex_lock(mem_lock);

            printf("\tSender's parent's pid:  %d.\n", parent_pid);
            printf("\tSending Message (setting shared variable).\n");
            shared_variable = ON:

            mutex_unlock(mem_lock);
            mutex_unlock(pr_lock);
}
```

thread2

```
thread2(parent_pid)
int      parent_pid;
{
            Cthread_begin();
```

deadlock.c

...thread2

```
        mutex_lock(pr_lock);      /* resource 2 */
        cthread_yield();
        mutex_lock(mem_lock);

        printf("Receiver's parent's pid:  %d.\n", parent_pid);
        printf("Receiver got the message via shared variable.\n");
        printf("Receiver sending an acknowledgement.\n");
        shared_variable = ACKNOWLEDGED:

        mutex_unlock(mem_lock);
        mutex_unlock(pr_lock);
}
```

join_err.c

```
/*
        A thread attempts to joins itself.  This results in
    deadlock.  MpD detects where this occurs and tells the user.
*/

#include        <stdio.h>
#include        <cthreads.h>
#include        "gdb_interface.h"

mutex_t pr_lock;

int             rtn0(), rtn1(), rtn2(), rtn3();
cthread_t       thread;
```

main

```
main()
{
        cthread_t       t1, t2, t3, t4;
        string_t        name33;

        setbuf(stdout, NULL);

        Cthread_init();
        pr_lock = mutex_alloc();

        printf("%d: Ready?", getpid());
        getchar();

        thread = t1 = Cthread_fork( rtn0, 'a');
        t2 = Cthread_fork( rtn1, 'b');
        t3 = Cthread_fork( rtn2, 'c');
        t4 = Cthread_fork( rtn3, 'd');

        Cthread_set_name(t1,"thread1");
        name33 = cthread_name(t1);
        printf("the address of cthread name33:0x%x\n",name33);
        Cthread_set_name(t2,"thread1");
        Cthread_set_name(t3,"thread1");
        Cthread_set_name(t4,"thread1");

        Cthread_join(t1);
        Cthread_join(t2);
        Cthread_join(t3);
        Cthread_join(t4);

        printf("Done.\n");
        printf("Done.\n");
}
```

rtn0

```
rtn0(idc)
        char idc;
{
        int     ctr;

        Cthread_begin();
        Cthread_join(thread);                   /* Thread joining itself */

        for(ctr = 0; ctr < 1; ctr++)
                {
                mutex_lock(pr_lock);
                printf("this is thread 000\n");
                ctr = ctr;
                mutex_unlock(pr_lock);
```

join_err.c

...rtn0

```
                cthread_yield();
                }
        Cthread_exit();
}
```

rtn1

```
rtn1(idc)
        char idc;
{
        int     ctr;

        Cthread_begin();
        for(ctr = 0; ctr < 10; ctr++)
                {
                mutex_lock(pr_lock);
                printf("this is thread 111\n");
                ctr = ctr;
                mutex_unlock(pr_lock);
                cthread_yield();
                }
        Cthread_exit();
}
```

rtn2

```
rtn2(idc)
        char idc;
{
        int     ctr;

        Cthread_begin();
        for(ctr = 0; ctr < 1; ctr++)
                {
                mutex_lock(pr_lock);
                printf("this is thread 222\n");
                ctr = ctr;
                mutex_unlock(pr_lock);
                cthread_yield();
                }
        Cthread_exit();
}
```

rtn3

```
rtn3(idc)
        char idc;
{
        int     ctr;

        Cthread_begin();
        for(ctr = 0; ctr < 1; ctr++)
                {
                mutex_lock(pr_lock);
                printf("this is thread 333\n");
                ctr = ctr;
                mutex_unlock(pr_lock);
                cthread_yield();
                }
        Cthread_exit();
}
```

doub_mutex.c

```
/*
        Mis-matched mutex_locks cause a deadlock.  MpD detects
    the line where this occurs and prints it out.  This error
    occurs in rtn1().
*/
#include            <stdio.h>
#include            <cthreads.h>
#include            "gdb_interface.h"

mutex_t pr_lock;

int                 rtn0(), rtn1(), rtn2(), rtn3();
```

main

```
main()
{
        cthread_t           t1, t2, t3, t4;
        string_t            name33;

        setbuf(stdout, NULL);

        Cthread_init();
        pr_lock = mutex_alloc();

        printf("%d: Ready?", getpid());
        getchar();

        t1 = Cthread_fork( rtn0, 'a');
        t2 = Cthread_fork( rtn1, 'b');
        t3 = Cthread_fork( rtn2, 'c');
        t4 = Cthread_fork( rtn3, 'd');

        Cthread_set_name(t1,"thread1");
        name33 = cthread_name(t1);
        printf("the address of cthread name33:0x%x\n",name33);
        Cthread_set_name(t2,"thread1");
        Cthread_set_name(t3,"thread1");
        Cthread_set_name(t4,"thread1");

        Cthread_join(t1);
        Cthread_join(t2);
        Cthread_join(t3);
        Cthread_join(t4);

        printf("Done.\n");
        printf("Done.\n");
}
```

rtn0

```
rtn0(idc)
        char idc;
{
        int     ctr;

        Cthread_begin();

        for(ctr = 0; ctr < 1; ctr++)
                {
                mutex_lock(pr_lock);
                printf("this is thread 000\n");
                ctr = ctr;
                mutex_unlock(pr_lock);
                cthread_yield();
                }
        Cthread_exit();
```

doub_mutex.c

...rtn0

```
}
```

rtn1

```
rtn1(idc)
        char idc;
{
        int     ctr;

        Cthread_begin();

        for(ctr = 0; ctr < 10; ctr++)
                {
                mutex_lock(pr_lock);                    /* ERROR, embedded locks */
                mutex_lock(pr_lock);
                printf("this is thread 111\n");
                ctr = ctr;
                mutex_unlock(pr_lock);
                mutex_unlock(pr_lock);
                cthread_yield();
                }
        Cthread_exit();
}
```

rtn2

```
rtn2(idc)
        char idc;
{
        int     ctr;

        Cthread_begin();
        for(ctr = 0; ctr < 1; ctr++)
                {
                mutex_lock(pr_lock);
                printf("this is thread 222\n");
                ctr = ctr;
                mutex_unlock(pr_lock);
                cthread_yield();
                }
        Cthread_exit();
}
```

rtn3

```
rtn3(idc)
        char idc;
{
        int     ctr;

        Cthread_begin();
        for(ctr = 0; ctr < 1; ctr++)
                {
                mutex_lock(pr_lock);
                printf("this is thread 333\n");
                ctr = ctr;
                mutex_unlock(pr_lock);
                cthread_yield();
                }
        Cthread_exit();
}
```

join_det.c

```
/*
        A Cthread attempts to join a detached thread.  This
    is the main routine.  MpD detects this error and points
    out the line where this occurs.
*/
#include            <stdio.h>
#include            <cthreads.h>
#include            "gdb_interface.h"

mutex_t pr_lock;

int                 rtn0(), rtn1(), rtn2(), rtn3();
```

main

```
main()
{
        cthread_t           t1, t2, t3, t4;
        string_t            name33;

        setbuf(stdout, NULL);

        Cthread_init();
        pr_lock = mutex_alloc();

        printf("%d: Ready?", getpid());
        getchar();

        t1 = Cthread_fork( rtn0, 'a');
        t2 = Cthread_fork( rtn1, 'b');
        t3 = Cthread_fork( rtn2, 'c');
        t4 = Cthread_fork( rtn3, 'd');

        Cthread_detach(t1);                         /* Detaching thread1 */

        Cthread_set_name(t1,"thread1");
        name33 = cthread_name(t1);
        printf("the address of cthread name33:0x%x\n",name33);
        Cthread_set_name(t2,"thread1");
        Cthread_set_name(t3,"thread1");
        Cthread_set_name(t4,"thread1");

        Cthread_join(t1);                   /* ERROR joining thread1 */
        Cthread_join(t2);
        Cthread_join(t3);
        Cthread_join(t4);

        printf("Done.\n");
        printf("Done.\n");
}
```

rtn0

```
rtn0(idc)
        char idc;
{
        int         ctr;

        Cthread_begin();

        for(ctr = 0; ctr < 1; ctr++)
                {
                mutex_lock(pr_lock);
                printf("this is thread 000\n");
                ctr = ctr;
                mutex_unlock(pr_lock);
```

join_det.c

...rtn0

```
                cthread_yield();
                }
        Cthread_exit();
}
```

rtn1

```
rtn1(idc)
        char idc;
{
        int     ctr;

        Cthread_begin();

        for(ctr = 0; ctr < 10; ctr++)
                {
                mutex_lock(pr_lock);
                printf("this is thread 111\n");
                ctr = ctr;
                mutex_unlock(pr_lock);
                cthread_yield();
                }
        Cthread_exit();
}
```

rtn2

```
rtn2(idc)
        char idc;
{
        int     ctr;

        Cthread_begin();

        for(ctr = 0; ctr < 1; ctr++)
                {
                mutex_lock(pr_lock);
                printf("this is thread 222\n");
                ctr = ctr;
                mutex_unlock(pr_lock);
                cthread_yield();
                }
        Cthread_exit();
}
```

rtn3

```
rtn3(idc)
        char idc;
{
        int     ctr;

        Cthread_begin();

        for(ctr = 0; ctr < 1; ctr++)
                {
                mutex_lock(pr_lock);
                printf("this is thread 333\n");
                ctr = ctr;
                mutex_unlock(pr_lock);
                cthread_yield();
                }
        Cthread_exit();
}
```

Appendix B

```
Usage:
help

help <commands>


Effect:


The word "help" by itself displays this file.  Typing help followed
by a specific command gives help on usage and effect of the command.
Available commands are:
```

dpe Enters dpe into dpe table and creates PA (predecessor automata) for it.

display Displays the dpe table or specific dpe statistics.

quit Exits dpe mode and re-enters gdb mode.

delete Deletes specified dpe(s) from dpe table.

active Makes specified dpe from dpe table active for execution modeling.

renum Renumbers entries in dpe table from 1 upwards.

skipon Don't stop at mutex lock/unlock specified.

skipoff Stop at mutex lock/unlock if encountered.

receive Receive messages from a specified port.

ignore Ignore messages from a specified port.

help Displays this file.

```
Usage:
active <number>
```

Effect:

Current active DPE for execution modeling becomes entry specified by number in the DPE table. Old active DPE switches places with new active DPE in the DPE table. The DPE in entry 1 in the DPE table is ALWAYS the currently active one. To see the DPE table use command "display."

Usage:
delete <number(s)>

Effect:

Deletes DPE(s) corresponding to the number(s) in the DPE table.

Example:

delete 4 5 7

This will delete DPEs 4, 5 and 7.

Usage:
display

display <number>
display locks
display ports

Effect:

Displays the dpe table. If given a specific entry in dpe table, detailed statistics about the dpe are printed. With the key word "locks" it displays current list of mutex lock/unlock variables being ignored. The key word "ports" lists currently monitored ports to appear.

Usage:
dpe <dpe expression>

Effect:

Creates predecessor automata (pa) out of specified dpe. A data path expression (dpe) tries to model the execution flow of the breakpoints set in gdb. To specify the execution flow the operators used are:

; Sequential.

+ Exclusive or.

* Repetition.

& Concurrent.

Examples of dpes modeling execution flow would be:

dpe 1;2 means breakpoint 1 is seen then breakpoint 2.

dpe 1&2 means breakpoint 1 is concurrently with breakpoint 2.

dpe (1;2)*;(3+4);5 means breakpoint 1 followed by breakpoint 2 can occur 0 to any number of times, then either breakpoint 3 or 4 occurs, followed by breakpoint 5.

The operators may be grouped by parentheses to any depth. In addition, several dpes may co-exist with several active (i.e. gdb events are compared to them) at any given time. The DPEs are placed

into a "DPE table" which can be manipulated by various commands while in the DPE parser (see help for list of available commands).

When the program executes in gdb the actual order of the breakpoints occurring is compared with the specified DPEs. At the end of execution the command "state" provides information comparing the DPE models against actual execution.

Usage:
ignore <port>

Effect:

All messages sent to specified port via the msg_send() function will be ignored when we receive_msg(). This is usually used to turn off a port the receive command specified earlier.

Example:

ignore port1 (Where port1 is a Mach port declared in the program.)

Usage:
quit

Effect:

User exits dpe manipulation mode and re-enters gdb mode.
