Debugging Concurrent Programs

by

Jun Shih

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 1996

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Debugging concurrent programs is difficult because concurrent programs contain both sequential errors and additional concurrent errors. It is essential to have a symbolic debugger that truly understands concurrency to improve concurrent debugging capabilities and reduce debugging time. KDB was designed to be such a concurrent debugger, however it was far from complete.

This thesis presents extensions to KDB's functionality, usability, and portability. Restricted conditional breakpoints, attachment of KDB to a running application, behavioural groups, programmatic interface, and improved user interface have been added to KDB to extend its functionality. KDB has been modified to understand μC++ programs better so that inserted code is hidden from users improving KDB's usability. Finally, KDB has been ported to the i486 architecture on the Linux OS, increasing portability.

KDB is written in μC++ and is a concurrent application, so it was possible to test the extensions while debugging the development of the extensions. This direct feedback helped in understanding if an extension was actually helpful in debugging a concurrent program.

Acknowledgements

I would like to thank my supervisor, Peter Buhr, for his time, guidance, encouragement and patience throughout the development of this thesis. I also thank Professor Stephen Mann for his fast reading and valuable comments on my thesis. I would also like to thank professor Thomas Kunz for reading my thesis, as well as his useful suggestions. I thank Martin Karsten for answering questions about KDB and his suggestions. I would like to thank Donald Pazel from IBM for answering questions about their debugger. I want to thank Ivan Yu for his discussion. I also thank Robert Berks and Russell Mok for helping me prepare my presentation. I gratefully acknowledge the financial support I received from NSERC, ITRC and the UW Graduate Scholarship.

Special thanks to my family for their love, understanding and support. I also want to thank my friends, Stuart, Dave, Robert, Toby, Ed, Karsten, Philip, Yunping, Lisa, Pat, Ka Yee, Marc, Vivian and the others for non-technical support as well as bugging me. I had a wonderful time with you guys.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The focus of this thesis is on developing tools and techniques to aid in the debugging of concurrent programs. When debugging a sequential program, the process is synchronous between the debugger and the application. However, when debugging a concurrent program, these same control and manipulation mechanisms must be provided independently for every thread of control. Unfortunately, most debuggers do not work with concurrent programming languages or environments.

Of those that do deal with concurrency, most work only with kernel (operating-system) threads. Kernel threads are controlled and scheduled by the operating system not by the runtime environment of the application using them. While kernel threads are essential, so too are user threads [ABLL92], which subdivide a kernel thread's execution among user threads in an application. User threads have the potential to be significantly less expensive than kernel threads in many cases because the language runtime system has specific knowledge about the form of concurrency and its implementation.

Given that user threads are important, some mechanism must exist to debug concurrent programs using them. The reason is straightforward, each language and/or thread library is different, and hence, each requires individual debugging support. Furthermore, computer vendors cannot be expected to support all extant and future languages and thread libraries. The KDB debugger [Kar95, BKS96], developed by Martin Karsten, shows that it is possible to build very powerful and flexible debugging support for user threads.

KDB (kalli's debugger) is a multi-threaded debugger for debugging multi-threaded μC++ applications. μC++ [BS96] is an extended version of C++ providing light-weight tasking facilities using a shared-memory model. Both uniprocessor and multiprocessor μC++ applications may be debugged.

Unfortunately, KDB fell short in a number of ways due to time constraints in its design and implementation. This thesis presents the work I did in augmenting KDB to provide a significantly better concurrent debugger for user threads.

To extend KDB's functionality, I added fast conditional breakpoints, attachment of KDB to a running application, behavioural groups, programmatic interface, and several minor enhancements to KDB's user interface.

As well, I solved a difficult problem relating to hidden code inserted by the μC++ translator. Each μC++ statement is transformed into one or more C++ statements. The generated code is normally invisible to users, but becomes visible when debugging μC++ programs. Without knowledge of the inner workings of μC++, it was easy to step into inserted code during debugging and become lost. In particular, this problem confused and discouraged students from using KDB.

The solution was to make KDB more aware of μC++ so that generated C++ code remained hided during debugging, which substantially improved KDB's usability.

Finally, KDB was originally designed to support multiple architectures; however, Martin only had time to implement one architecture: SPARC. To determine if the original design is truly generalized, I ported KDB to the Intel 486 architecture on the Linux operating system. Linux/i486 was chosen because the i486 has a significantly different architecture from the SPARC, and it is a useful platform used by many students.

1.1 Definitions

The following definitions are used in this thesis:

Debugging is the process of locating, analyzing, and correcting suspected faults that cause a program to fail to perform its required function.

A *process* is a program component with its own thread.

A *task* (sometimes called a light-weight process) is similar to a process except reduced along some particular dimension. It is often the case that a process has its own memory address space, while tasks share a common memory. As well, a process's thread is usually scheduled by the OS, while a task's thread is usually scheduled within the application.

Parallelism occurs only when multiple processors (CPUs) are present in execution of a program.

Concurrency denotes when the execution of a program *appears* to occur in parallel.

A *critical section* is a group of instructions that must be performed atomically.

1.2 Debugging Issues in Concurrent Systems

1.2.1 Multi-threaded Systems

Machine architectures that support multi-threaded programming exist on:

- `uniprocessor machines`: programs are run concurrently via context-switching among different processes and tasks within processes.
- `multiprocessor machines`: programs are run in parallel using separate CPUs but sharing the same memory.
- `distributed systems`: programs are run in parallel on different machines using separate CPUs and separate memories.

There are two types of inter-process communication (IPC) used to enable communication among different threads of control:

- `shared memory`: memory can be accessed by more than one process or processor
- `message passing`: no shared memory among processors, so processors communicate by sending and receiving messages through communication channels

Multi-threaded applications consist of a set of concurrent threads that are distinguished by its creation:

- `kernel thread`: created and managed by the operating system kernel.
- `user-level thread`: created and managed by the runtime system of a programming language or a thread package within the process's address space.

On all systems, user-level threads are executed by kernel threads. The address space of an application is usually divided into code and data memory. If multiple threads share the same code memory, they usually also share the data memory. The reverse is not necessarily true.

1.2.2 Difficulty in Debugging Concurrent Applications

Concurrent programming is difficult due to the additional temporal dimension resulting from concurrency. This temporal dimension makes it significantly harder to debug current programs than sequential programs, since they contain both sequential errors and additional concurrent errors. Additional sources of error include deadlock [MR91], race conditions [NM92], and non-determinism [MH89], which are unknown in the sequential domain. Each kind of concurrent error is discussed below:

- `deadlock`: occurs when two or more tasks form a dependency cycle of holding resources and waiting for those same resources.
- `race condition`: occurs when access to memory by different threads is not properly synchronized, which causes non-atomic execution of a critical section. Race conditions exist in both shared-memory and message-passing operating systems.

- `non-determinism`: sequential programs are deterministic: given the same inputs, the result is always the same. However, concurrent programs do not always reproduce the same behaviour. This situation is particularly hard to deal with, since a programmer often has no control over it. Therefore, deterministically replaying the execution of a program is very important in debugging concurrent programs.

As a result of non-determinism in concurrent programs, there is a phenomenon called probe effect [Gai86], which occurs when a concurrent program is augmented with debugging or performance analysis probes, which increases or decreases the program's non-determinism, resulting in different program behaviour. The probe effect makes debugging and analysing concurrent programs very difficult, because the error or performance problem may not occur due to the probe effect.

1.3 Thesis Organization

Chapter 2 gives an overview of KDB. It presents the architecture and features of KDB. Chapter 3 discusses some parallel and distributed debuggers, which are contrasted with KDB. Chapters 4, 5 and 6 contain the contributions that were made in the form of extensions to KDB.

Chapter 4 discusses the enhancements to the functionality of KDB, which include fast restricted conditional breakpoints, behavioural groups, attachment of KDB to a program, programmatic interface and improvements to the user interface.

Chapter 5 discusses debugging translated code. Because μC++ is a translator and not a compiler, a μC++ statement may be translated into one or more C++ statements. This chapter present a method to make the inserted code invisible to users when debugging a μC++ program.

Chapter 6 discusses the implementation of KDB on an Intel 486 architecture in the Linux environment.

Chapter 2

KDB

KDB (Kalli's DeBugger) [BKS96] [Kar95] is a concurrent interactive source-level debugger running on UNIX based symmetric shared-memory multiprocessors that provides independent control of user-level threads. The design of KDB (but not the implementation) is applicable to both shared memory and distributed memory systems and is intended to be portable and inter-operable with other debugging tools.

2.1 KDB Design

2.1.1 Independent Control of User Level Threads

Currently, computer vendors providing shared-memory concurrency hardware are creating support for multiple kernel threads in an address space (UNIX process), such as Solaris Threads. Along with kernel threads comes additional support for debugging in the form of new capabilities to query and manage kernel threads, e.g., extensions in `/proc` for kernel-level threads.

However, user-level threads are essential [ABLL92], and will always exist. It is naive of operating system developers to assume that the support they provide today will encompass all extant or future concurrency paradigms. Therefore, it is important to have a debugger that works with user-level threads and controls each thread independently. Most existing UNIX debuggers use debugging primitives based on `ptrace` [Sunb] or `/proc` [Suna], which only allow synchronous interaction. This existing approach blocks IPC and therefore precludes independent control of user-level threads, and proper handling of multiple UNIX processes.

The mechanism used in KDB to achieve efficient asynchrony between the debugger and the target application, as well as independent control of user-level threads, is to distribute part of the debugger into the target application, called the *local debugger* (see Figure 2.1), and the *global debugger* using two independent communication channels with the target application:

- `Modification Channel`: which is synchronous and created by using traditional debugging primitives (`ptrace` or `/proc`). It is established only temporarily for the global debugger to modify the target application's code, e.g., inserting a breakpoint.

- `Control and Notification Channel`: which is asynchronous and established through a socket between local and global debugger. The local debugger catches target application debugging events, like encountering a breakpoint, and reports them to the global debugger. The global debugger then sends control commands, like continue a thread, to the local debugger. This communication channel does not block either target application or global debugger. Thus, it fulfills the requirements of independent control of user-level threads. In fact, if it was possible for the local debugger to modify the target's code directly, the synchronous modification channel could be merged into the asynchronous control and notification channel.

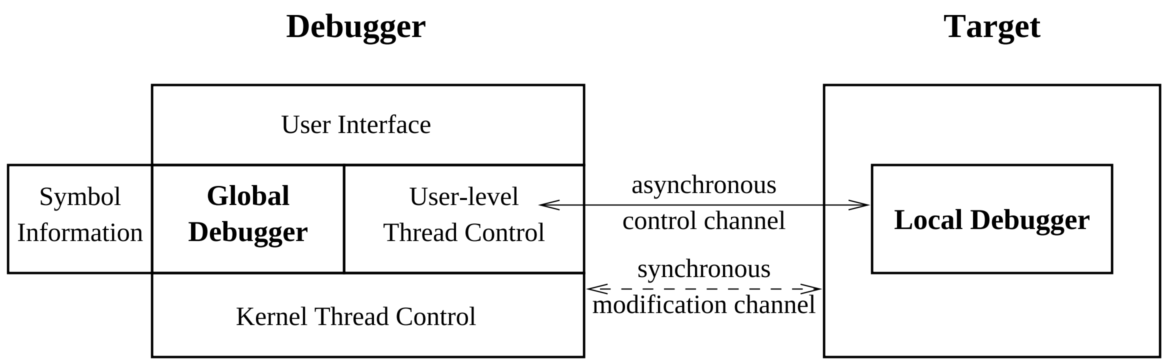

Figure 2.1: Distribution of Work between Debugger and Target

Portability

Portability across different operating systems and architectures, and for different source languages is important. Portability can be achieved by encapsulation of dependent code. The separation of the debugger into multiple modules to ensure certain portability aspects is shown in Figure 2.1.

The availability of any debugging hardware support makes implementation of a debugger easier and efficient. For example, some hardware has support for single stepping. However, a debugger can not rely on these features. Thus, the relevant code must be encapsulated so that it can be adapted to different architectures. In the case of little or no hardware support, these facilities must be emulated in software. For example, single stepping can be implemented by setting a breakpoint at a proper location and continuing execution.

In KDB, the following parts of the debugger have been encapsulated:

- hardware support
- communication with the operating system (e.g., control of kernel threads)
- communication with the runtime system (e.g., control of user-level threads)
- different executable file formats
- target application's global symbol table and code debugging information
- user interface

Inter-operability

Event collection and graphical visualization of the behaviour of a concurrent program can help significantly in understanding it. Thus integrating all these abilities into the debugging environment is a reasonable goal. Figure 2.2 shows a possible relationship of auxiliary and debugging tools.

2.2 KDB Features

KDB provided a number of powerful debugging and implementation features. Those features that are directly extended or have a related extension in this thesis are

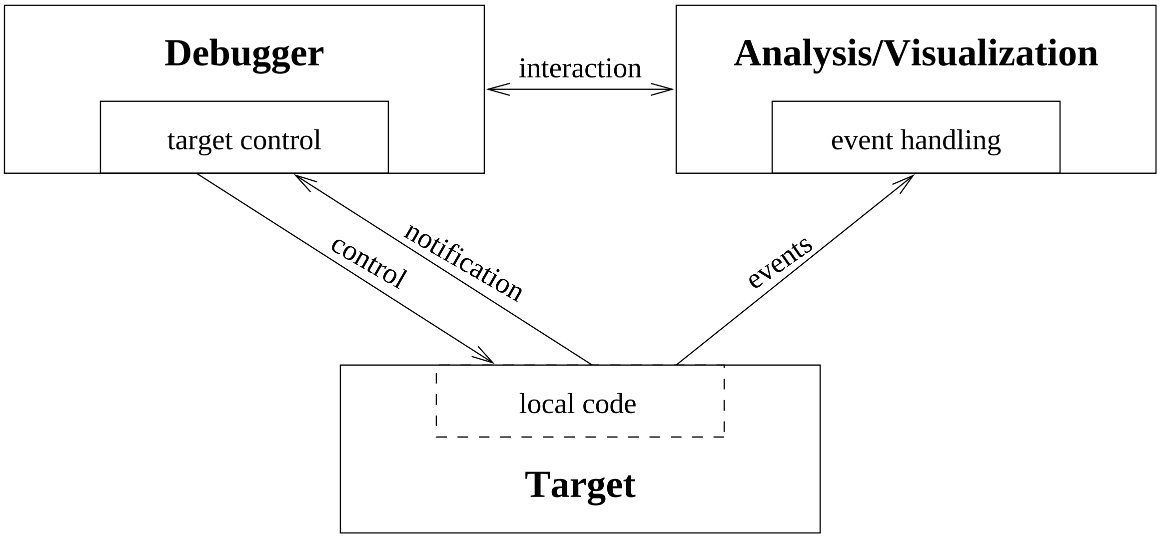

Figure 2.2: Interactive Debugging and Event Generation

discussed in detail.

2.2.1 Operational Group

An operational group is mainly a convenience facility for a user. Instead of issuing the same command multiple times for a group of threads, the command is issued once for the operational group. This convenience facility is an important feature for scaling to medium or large numbers of threads. Once groups of threads are formed, a user can easily perform common operations on the group. Instead of interacting with a larger number of individual threads, the user interacts with a small number of groups. Threads can be arbitrarily grouped into an operational group, but certain commands are meaningful to all threads in the group only if they all share the same source code. For instance, setting a common breakpoint would be meaningless for threads that do not share source code. A thread can belong to any number of operational groups.

2.2.2 Fast Breakpoints

Breakpoints in KDB are implemented by storing the breakpoint handling code at some convenient place in memory and inserting a branch to the breakpoint handling code in place of the original code. This is opposed to the traditional approach of inserting a TRAP instruction that requires interaction with the OS, and hence, is expensive. As well, performance can be increased if the breakpoint is handled in user code, since the number of kernel context switches between debugger and target is decreased. For example, a single breakpoint often applies only to a small number of threads. However, in an application with many threads sharing the same code image, a breakpoint can be triggered by other threads thousands of times. The check to see if the breakpoint is applicable to the current thread can be done quickly by the local breakpoint handler instead of the global debugger, and the global debugger is notified only if the breakpoint is applicable. It is shown in [Kar95] that checking the applicability of a breakpoint using a local breakpoint handler is approximately 2400 times faster than using the global debugger.

2.2.3 Using GDB Code

Since writing a debugger from scratch is very difficult, and to increase the portability of the debugger and avoid unnecessary implementation, parts of the GDB debugger [SP95] (a portable sequential debugger with source freely available) were re-used. Using GDB code also gives the advantage of possible benefit from future GDB development as well as its portability for different platforms.

GDB code is used to

- deal with different executable file formats, e.g., *a.out*, *elf* etc.
- access symbolic debug information, which is inserted into the application when it is compiled with the `-g` flag
- interpret application data, like stack and variable contents

To gain synchronous control of the target process, GDB, however, is not run as a separate process and all parts of the GDB code that can gain control over a UNIX process or modify the target process have been removed.

Chapter 3

Related Work

3.1 The Portable Parallel/Distributed Debugger

The Portable Parallel/Distributed Debugger [Hoo96], p2d2, was designed and built at NAS to accomplish the following two main objectives:

- portable enough to run on the varied collection of machines in NAS
- capable of controlling the execution of 256 processes, without requiring a window for each process.

p2d2 features a client-server architecture, see Figure 3.1. The debugger client consists of a user interface with which the user interacts, and a distribution manager that keeps track of processes and process groups. As well, a debugger server manages server operations communicated across the network to possibly multiple remote servers controlling multiple target processes.

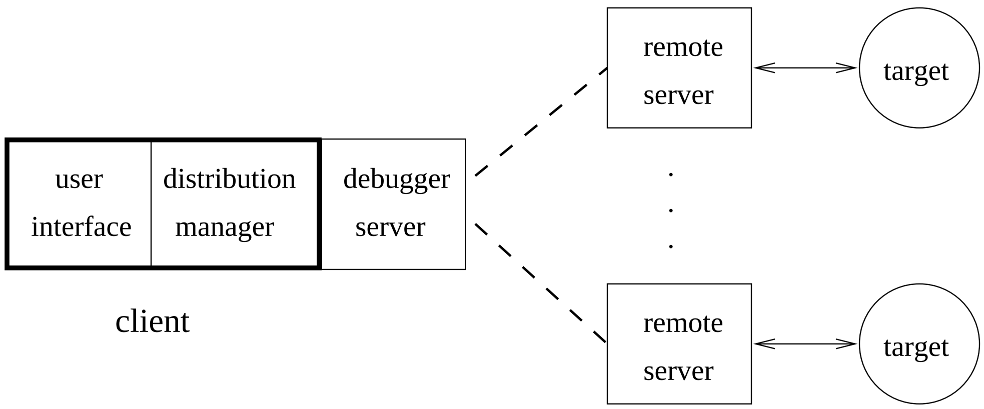

Figure 3.1: p2d2's Client-Server Architecture

Like KDB, the p2d2 debugger server is based on `gdb`, in fact, each remote server is an instance of GDB. The debugger server is responsible for sending commands to the appropriate instance of GDB and parsing any replies. Unlike KDB, GDB is used to control the target process. The p2d2 debugger was not designed to support user-level threads. On the other hand, KDB supports independent and asynchronous control of user-level threads.

p2d2 supports the following useful features:

- control set: similar to the operational group in KDB, which allows a user to send a command to a set of processes.

- focus group: displays more detailed information about a handful of processes, e.g., process's ID, machine name and executable name, state information.

3.2 The Mantis Parallel Debugger

Mantis [LC96], a graphical debugger for parallel programs, provides integrated support for the Split-C language [CDG+93]. The Mantis interface can also be used for sequential debugging in C, C++ Fortran, and other languages, allowing a single environment for both sequential and parallel debugging.

Mantis is also based on GDB with added language support. A debug session consists of a global debugger and multiple node debuggers. Both the global debugger and the node debugger are based on GDB with global language support and nodal language support respectively. Each node debugger interacts with a user process. The global debugger gathers information from the node debuggers and directs user requests to them. Mantis was not intended to support user-level threads.

3.3 IBM Distributed Debugger

The IBM Distributed Debugger [MMP+96] supports the debugging of multi-threaded, multi-process and multi-language applications that use multiple middlewares while executing in a heterogeneous distributed environment. The debugger consists of the following two parts:

- `front-end`: consists of the user interface views, the centralized logic required to control the distributed application, and an event monitor.

- `back-end`: uses a platform independent class library and executes on each of the machines on which the application executes.

A debug session consists of a front-end and multiple back-ends controlling a distributed application. The architecture is similar to that of the p2d2 debugger, except it is more sophisticated.

The client and the debugger server of the p2d2 debugger are similar to the front-end of the IBM distributed debugger, but the IBM distributed debugger has an event engine consisting of an `event collector` that collects and stores events that are generated by the distributed application and an `event monitor` that displays the events collected by the event collector. The event engine is based on David Taylor's event monitoring tool [Tay93] from the University of Waterloo.

The remote server of the p2d2 debugger is somewhat like the back-end of the IBM distributed debugger. Each remote server of the p2d2 debugger is merely an instance of GDB. The back-end of the IBM distributed debugger consists of a `debug demon` and a `debug engine`. The debug demon is assumed to be running on every host that a debug engine runs on. The purpose of the debug demon is to accept a request from the front-end and start up a debug engine. Once the debug engine is started, it communicates with the front-end directly. The debug engine contains an instrumentation and control library (ICL) and Middleware Control Services (MCS). The ICL provides the debugger developer with a platform-independent and language-independent class library for debugging single-process applications. The MCS serves as a mechanism needed by a debuggee to notify the debug engine asynchronously of the occurrence of certain events in the middleware. These notifications allow the debug engine to take appropriate event-specific action.

According to Donald P. Pazel (personal communication), the IBM distributed debugger provides independent control of kernel threads but currently does not provide support for user-level threads.

Chapter 4

Enhanced Features

While Karsten implemented the initial core components of KDB, a number of important features still remained unimplemented. These missing features are necessary to substantially enhance the usability and power of the debugger.

In this chapter, I discuss the following four major enhancements I added to KDB (fast conditional breakpoints, attachment of KDB to a running application, behavioural groups and programmatic interface), as well as several other minor enhancements.

4.1 Enhanced Breakpoint

The ability to set breakpoints is possibly the most important feature of a debugger. *Fast breakpoints* are essential to provide adequate performance with multiple tasks sharing code images. *Conditional breakpoints* are also very useful in debugging, for example, when debugging a looping structure, the ability to stop when the loop iterates N times is essential. The easiest and quickest way to achieve this is to set a conditional breakpoint because issuing repeated *continue* commands is unacceptable. In this section, fast breakpoints in KDB are discussed and how *fast restricted conditional breakpoints* were added.

4.1.1 Fast Breakpoints

Fast breakpoints implemented by Martin in KDB are similar to those discussed in [Kes90]. Breakpoints are implemented not by a traditional `trap` instruction, but by inserting a call to a dedicated breakpoint routine (breakpoint_handler_i()). The original instructions at the breakpoint are saved at a temporary location. When a task executes the inserted call, the called routine checks if the breakpoint applies to this thread. If the breakpoint does not apply to the current thread, the original instructions are executed at the temporary location. The original instructions are restored when the breakpoint is removed. Figure 4.1 shows the pseudo code for a breakpoint handling routine.

There are N breakpoint_handler_i() routines (where i = 0..N-1) created at compilation time, one for each breakpoint. A breakpoint is implemented by inserting a call to the appropriate breakpoint_handler_i() in the application's code. Since N is finite, when a breakpoint is removed, the code changes are undone and the breakpoint_handler_i() routine is re-used for another breakpoint. The global debugger cycles through the N breakpoint_handler_i() routines as breakpoints are set and cleared.

For breakpoint_handler_i() to decide whether the breakpoint applies to the cur-

```
bool breakpointHandler( int number ) {
    sendMessageToGlobalDebugger( number );
    bool breakpoint_removed = receiveContinueMessageFromGlobalDebugger();
    return breakpoint_removed;
}

void breakpoint_handler_i() {
    saveApplicationState();
    // U_THIS_TASK is a pointer to the currently executing user-level thread
    if ( U_THIS_TASK->bp_check[ i / 8 ] & (1 << (i % 8)) ) { // breakpoint set for thread
        if ( breakpointHandler( i ) ) {        // tell global debugger
            asm(" sub RA, -8, RA ");
            restoreApplicationState();
            return;
        }
    }
    restoreApplicationState();
    asm("
        ! reserve as many NOP instructions as needed to store the temporary code
        nop
        . . .
        ! final return to jump back into the application
        jmp address_after_breakpoint_code
    ");
}
```

Figure 4.1: Breakpoint Handling in Application Code

rent thread, an N-bit mask (bp_check in the pseudo code) is stored with each thread. The breakpoint is applicable to the current thread if the *ith* bit of the mask in the current thread is set, which corresponds to breakpoint_handler_i(). If the breakpoint is applicable to this thread, the routine breakpointHandler(i) is called, which notifies the global debugger that the breakpoint is encountered for this thread, and waits for a continue message from the global debugger.

When a continue message is received for this thread, its execution continues. If the breakpoint is removed at this time (i.e., breakpointHandler(i) returns true), the original instructions are already restored at the original location, thus, the return address needs to be adjusted backwards in order for the original instructions to be executed at the original location. If the breakpoint is not removed at this time, then the modified original instructions are executed at the temporary location.

Since breakpoint handling interacts with the runtime system, which handles the scheduling of the user-level threads, this kind of breakpoint can not be set in parts of the runtime system. However a traditional sequential debugger can be used to debug this part of the runtime system, if needed.

As mentioned earlier in Section 2.2.2, having the breakpoint applicability test checked locally in the application produced a speedup of 2400 times over using the traditional approach.

4.1.2 Fast Restricted Conditional Breakpoints

I augmented the existing fast breakpoints with a conditional check, e.g.,

```
break Philosopher.cc:82 if k1 == 9
```

which means the breakpoint is set for the current thread at line 82 of source file Philosopher.cc and the breakponit is only triggered when variable k1 is equal to 9. Like fast breakpoints, the breakpoint condition is checked and evaluated locally during breakpoint evaluation, rather than at the global debugger. The conditional breakpoint is restricted because only the following conditional forms are allowed:

- *integer* [== | != | >= | <= | > | <] *integer*
- *pointer* [== | != | >= | <= | > | <] *pointer*

It is infeasible to dynamically compile and insert code into the target application for arbitrary conditional commands presented dynamically to the global debugger. Therefore, all possible combinations of operand types and operators must be pre-compiled into the local debugger, and the appropriate operation is dynamically performed during breakpoint evaluation. Since most conditions involve only integers or pointers, the restrictions on the conditional expression are not particularly significant.

Data Structures

Since the breakpoint conditions are evaluated in the target application, the breakpoint conditions are stored in the local debugger. Tasks can share breakpoints but not conditions unless some technique is used to determine equality of expressions. Therefore the number of conditions that may be stored in the local debugger varies with the number of conditional breakpoints set by the user. An array of linked lists of size N is used to store the conditions, where N is the maximum number of break-

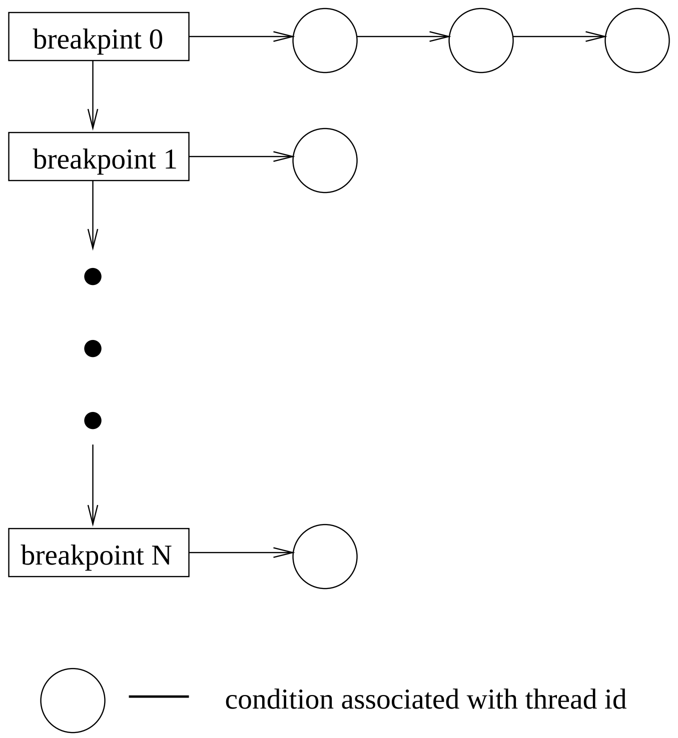

Figure 4.2: Queued Conditions identified by Breakpoint Number and Thread Id

points preset when the local debugger is compiled. The data structure is shown in Figure 4.2. Each condition is uniquely identified by the breakpoint number and thread id. If a limit is placed on the number of conditions (as for breakpoints), the data structure could be implemented with a fix sized two dimensional array with better performance. However, the overhead of dynamic memory allocation is small compared to the cost of evaluating the condition and is more flexible, using less memory. The disadvantage of this approach is that the local debugger interacts with the execution of the program more heavily, increasing the probe effect.

When a conditional breakpoint command is given to the global debugger, the expression string is stored in the global debugger without evaluating the operands.

This delay is necessary because the operands may not yet exist in the target; thus the addresses and other information for the variables in the breakpoint condition may not be retrievable at the time when the breakpoint is set. The global debugger (which implements the breakpoint by selecting a breakpoint handler routine) now tells the local debugger that the inserted breakpoint is associated with a condition. When the breakpoint handler is invoked for the first time, and if the breakpoint is associated with a condition, the local debugger sends a request for the condition to the global debugger. The global debugger then looks up the addresses and other information for the operands associated with the condition and sends this information to the local debuggers (this is done only once). The breakpointHandler routine in Figure 4.1 is modified to deal with the breakpoint condition, and shown in Figure 4.3.

Evaluation of Breakpoint Condition

As mentioned in Section 4.1.2, for efficiency the breakpoint condition is evaluated by the local debugger. The following types of operand are supported: integer or pointer. The operation can be one of the following:

<, <=, >, >=, !=, ==

An operand of a condition can be a constant or a variable. The evaluation of a constant is the constant itself. The evaluation of a variable depends on whether the variable is *local*, *static* or *register*. The variable can also be a member of a structure, e.g., this->id. There are four different forms allowed:

A, *A, P->A and S.A

```
bool breakpointHandler( int number ) {
    bool breakpoint_removed = false;
    ULThreadId thread_id = U_THIS_TASK;

    if (bpConditionMask[number] == true &&
        bpCondition[number].search(thread_id) == NULL) {
        sendRequestForConditionToGlobalDebugger( number );
        bpCondition[number] = receiveConditionMessageFromGlobalDebugger();
    }

    if (bpConditionMask[number] == false ||
        EvaluateCondition(bpCondition[number].search(thread_id)) == true) {
        sendMessageToGlobalDebugger( number );
        breakpoint_removed = receiveContinueMessageFromGlobalDebugger();
    }

    return breakpoint_removed;
}
```

Figure 4.3: Handling Breakpoint Condition

In general, an address of a variable is evaluated as:

Variable Form	Evaluation of the Address
A	absolute_address(A)
*A	contents(A)
P->A	contents(P) + offset(A)
S.A	absolute_address(S) + offset(A)

However, the absolute address of a local variable changes during execution, e.g., a local variable of a routine may have a different address for each invocation. Thus, the local debugger only keeps relative addresses of local variables, since relative addresses do not change. In the case of a register variable, the register number is stored. The local debugger calculates the absolute addresses needed for the conditions from the relative addresses:

- the absolute address of a *static variable* is the address of the static variable, which does not change during the execution of the program; thus, this address is kept at the local debugger.

- the absolute address of a *local variable* can be calculated as:

  address = fp + offset

  where address is the address of the local variable, fp is the frame pointer of the user function, and offset is the relative address of the local variable. Figure 4.4 shows the stack frame for the SPARC architecture. The fp of the user function can be retrieved through the fp from the breakpointHandler(i) function (see Figure 4.5), since the caller's fp is stored on the callee's stack. The offset of a variable on the stack is looked up by the global debugger and stored in the local debugger.

- For the SPARC architecture, when the routine breakpoint_handler_i() is called, the "local" and "in" registers are saved on the user's stack by the `save` instruction. The absolute address of a *register variable* is calculated as

  sp + register_offset[register_number]

  where sp is the stack pointer of the user function, register_offset contains the offsets for registers stored on the stack, which is different from architecture

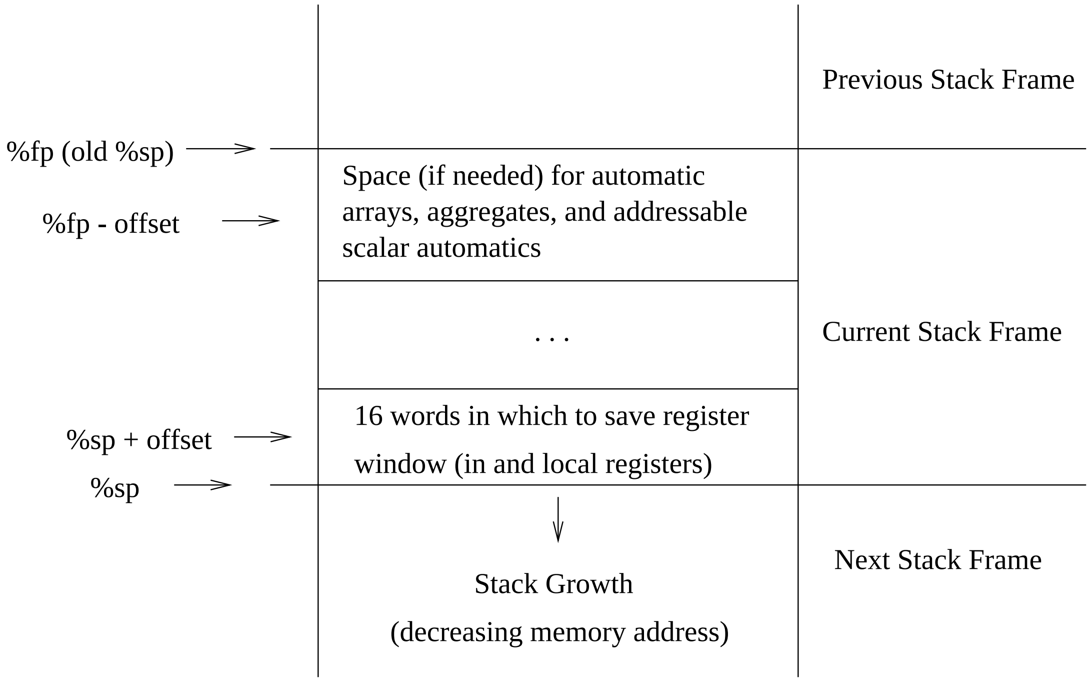

Figure 4.4: The SPARC User Stack Frame

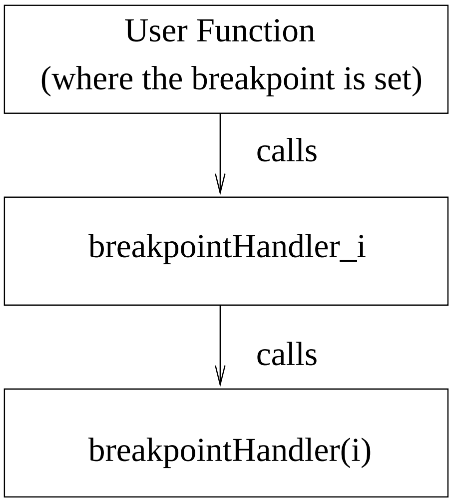

Figure 4.5: The Breakpoint Handler Call Stack

to architecture. The sp of the user function can be retrieved in the same way as the fp through the caller's stack (see Figure 4.4).

Note that for expressions like P->A, the offset for A is evaluated only once by the global debugger. If the value for P is changed later in the program, e.g., assigned to NULL, this can cause a run time error. Correctly handling this case is beyond the scope of this work. To fully support all possible breakpoint conditions, it is necessary to implement a small parser within the application, which parses and evaluates expressions. This approach is not appropriate for compiled languages, nor easily implemented. Although, the fast restricted conditional breakpoints are useful, it does not support all necessary conditions. Thus, a fallback option involving the global debugger evaluating more complex conditions is reasonable. However, it is not currently supported.

Performance

A simple experiment was conducted in which the same conditional breakpoint was set in a μC++ program using GDB and KDB. KDB was found to be approximately 30 times faster than GDB.

4.2 Attachment of the Debugger to a Running Application

Sequential programs are deterministic, errors can be regenerated by rerun the application. However, errors may not be regenerated easily for concurrent programs, since they are non-deterministic. Thus, the ability to attach a debugger to a running application is extremely useful, especially for debugging applications with an infinite loop or a deadlock. Therefore, I added this feature to the existing KDB.

To attach KDB to a running application, the application has to be compiled with the `debug` option, which is the default. This option links the application with the debug version of the μC++ unikernel or multikernel. The debug version performs runtime checks to help during the debugging phase of a μC++ program and links the local debugger modules with the program. As well, calls are inserted in certain routines to report particular events to the local debugger, which must be subsequently reported to the global debugger, such as the creation of a user-level thread.

The command for attaching KDB to an executable already running is:

attach *executable-file process-id*

This command is issued in the command prompt of the KDB main window (see Figure 4.6).

In traditional debugging, attaching to a running target application is straightforward. It is accomplished by using the system primitives `ptrace` or `/proc`, which are only suited for a sequential debugger. As discussed in Section 2.1.1, the local debugger is distributed into the target application, therefore using the PTRACE_ATTACH command of `ptrace` freezes the application and thus the local debugger. Although `/proc` does not stop the target process from continuing normal execution, it is impossible to asynchronously wait for events from the target application.

In order for KDB to attach to a running application, the global debugger has to establish a communication channel to the local debugger of the target application, then obtain information about the structure of the user-level thread system through the local debugger, which for μC++ is the clusters, processors (kernel threads) and user threads of the application.

When a target application starts, the μC++ kernel startup routine checks whether a global debugger exists. If it does, a local debugger is activated to communicate with the global debugger. If the global debugger does not exist when the target application starts, the local debugger is not activated, and hence, calls to its routines to report events simply return. Thus, attaching the global debugger to the target application involves the following steps:

- request kernel of the running target application to activate the local debugger.
- establish the asynchronous communication channel between the local and global debugger (See Figure 2.1).

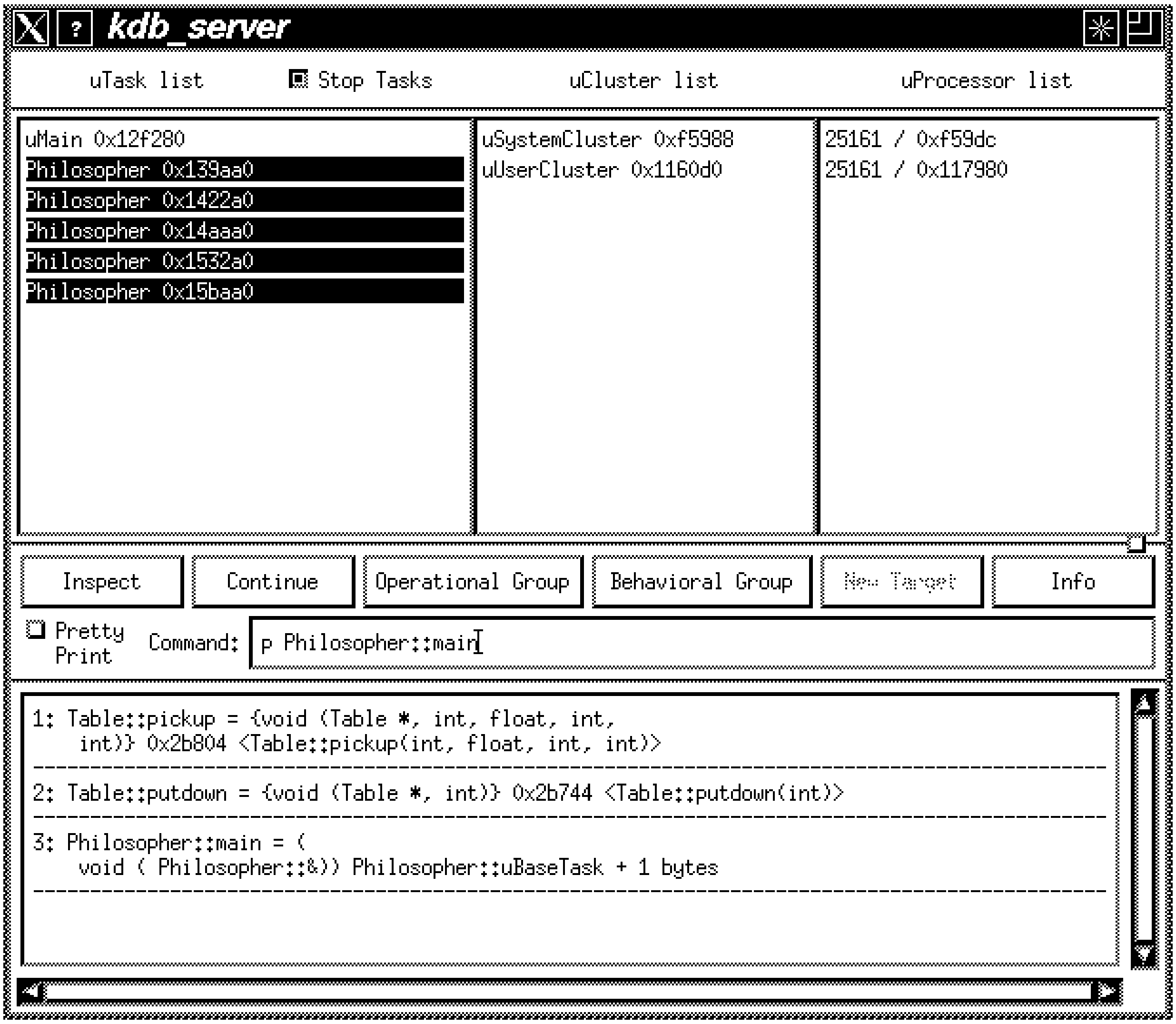

Figure 4.6: Main Window

- transfer the current execution environment, i.e., clusters, processors, and threads, from the application to the global debugger.

4.2.1 Activating the Local Debugger

Initially, some form of communication is needed between the global debugger and the application to inform the μC++ kernel to activate the local debugger. This initial communication is accomplished via a variable uKernelModule::uAttaching in the debugging version of the μC++ kernel, which is checked every time a context switch is done.

```
if ( uKernelModule::uAttaching ){
    uKernelModule::uAttaching = 0; // reset
    uAttachLocalDebuggerInstance = ActivateLocalDebugger();
}
```

To attach, the global debugger stops the application and modifies the variable uKernelModule::uAttaching using `ptrace` or `/proc`.

If the variable uKernelModule::uAttaching was shared among all processors executing an application, there would be a race condition as each processor noticed the variable set by the global debugger and activated the local debugger. However, this problem is easily dealt with by making uKernelModule::uAttaching private to each processor as opposed to shared among all processors, so that each processor has a private copy of the variable. To attach, any UNIX process id associated with the application can be specified in the KDB attach command, and its unique instance of uKernelModule::uAttaching is modified.

Note that certain actions involved in activating the local debugger require locks. However, locks can not be used by the kernel thread. Only user-level threads can use locks, since when a thread blocks acquiring a lock, it is put to sleep by the kernel when the next context switch is performed, and another thread is scheduled. However, the kernel thread can not put itself to sleep, since this would be a deadlock. Thus, the local debugger can not be activated directly from the kernel. Instead, the check for attaching is performed inside the context switch, just before restarting the next user-level thread. In effect, the activation of the local debugger is performed by the next user-level thread to be scheduled (i.e., piggy-backed on the restarting of the thread), and that thread is allowed to block if necessary.

One additional problem is that the application can be in a state where the variable uKernelModule::uAttaching may not be checked, since the μC++ kernel does perform context switches for the two following cases:

- There is no work to do, so the μC++ kernel thread puts itself to sleep, thus no context switching is performed.
- In the uniprocessor case, if the application turns off time slicing, no context switching is done.

To solve this problem, the global debugger always sends a SIGALRM signal to the target application immediately after the variable uKernelModule::uAttaching is modified. This signal either wakes up the μC++ kernel thread and/or forces the μC++ kernel to perform a context switch.

4.2.2 Establish the Asynchronous Communication Channel

Since KDB currently does not handle distributed applications, when a target application starts, the μC++ kernel activates the local debugger if the global debugger exists by checking for the existence of certain shell variables. If the shell variables exist, the local debugger then finds the INET communication port number from a file whose name is in the environment variable _KALLIS_DEBUGGER_PORT_, which is set by the KDB command. Unfortunately, an environment variable created after an application starts running is not accessible. Instead, the global debugger port number is supplied by the global debugger by writing the port number into the variable uKernelModule::uAttaching. Since the port number is always positive, it is possible for the μC++ kernel to know when the variable has been modified by the global debugger.

4.2.3 Transferring the Current Execution Environment

Once the local debugger of KDB is activated, there is still the problem of transferring the particular aspects of the application's execution state needed by the global debugger. In essence, the application needs to be stopped, with respect to the actions that are synchronized between it and the global debugger, for example, the global debugger is informed about the creation and destruction of each cluster, processor and task. There are two approaches to solving this problem.

The first solution is for the local debugger to inform the global debugger of its activation via the communication channel, and the global debugger then stops the application and traverses all the pertinent runtime data structures to acquire the information it needs about the application. Unfortunately, this approach is difficult because the interface for reading data from the application can be as small as 4 byte words of data. It also makes the code performing the traversal dependent on the structure of the runtime kernel, which makes maintenance difficult when the runtime system changes. As well, if KDB is to handle different user-level thread libraries, it must have a detailed knowledge of the data structures in each.

The second solution is to have the local debugger, which is already thread library specific, stop the application and transfer the data it would normally have been sending as events occurred. To do this requires traversing the runtime data structures to locate the necessary information, and clearly, these data structures can not be changing during the traversal. A safe traversal is accomplished by making the local debugger a monitor, which serializes access. This serialization is already needed during normal usage of the local debugger because it has internal state.

Once the activation routine of the local debugger is called by the activating thread, the runtime data structures can be safely traversed because all calls that report events to the local debugger wait until after the traversal is complete and the local debugger is fully activated. In fact, an application can continue to execute during the attachment if it is not generating new events for the local debugger.

4.3 Behavioural Groups

Behavioural groups was proposed by Martin, but not implemented. The idea of a behavioural group is to allow a user to debug a program at a higher level, so the expected and actual behaviour of the program can be compared. A similar and more sophisticated approach is found in [Bat95].

In [BKS96] and [Kar95], a behavioural group has been defined as a set of arbitrary tasks whose behaviour is linked to some event. If an event occurs for any task in a behavioural group, an action is applied to all the tasks in the group, e.g., when one task triggers a breakpoint, all tasks in the group are stopped. Hence, a behavioural group must have an event and operation associated with it. Given an `event` and a corresponding `action`, the behavioural group performs the following:

- the event operation is applied to the group if the event is a breakpoint
- wait for the event to occur
- when the event occurs, the action operation is applied to the group.

In order to examine the idea, I implemented the initial version of behavioural groups. To use a behavioural group, a user selects a group of threads from the main window (see Figure 4.6), and presses the "Behavioural Group" button. A behavioural group window (see Figure 4.7) pops up with the group of threads selected from the main window. The initial version restricted the event commands to breakpoint commands, and operation commands to the stop command. Figure 4.7 shows a behavioural group window where a breakpoint is about to be set for each Philosopher task, and each task will be stopped when one of the tasks reaches the breakpoint. Finally, no thread can belong to more than one behavioural group of threads, because different behavioural groups could have comflicting opertations such as stop and continue.

An attempt was made to generalize the initial version by allowing other events

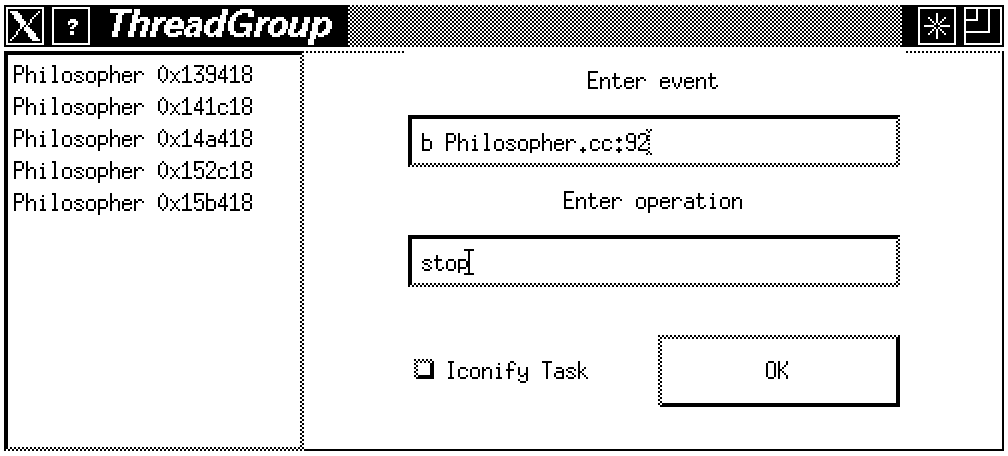

Figure 4.7: A Behavioural Group Window

and operations. What events other than break and what operations other than stop could provide useful capabilities for debugging? Unfortunately, the answer to this question appears to be none. For example,

event = stop and operation = stop

would mean when one of the tasks in the group is stopped by a user (by pressing the `stop` button on the task's thread interface, see Figure 4.9), then the debugger stops the rest of the tasks in the group. However, this can be achieved more efficiently by an operational group to stop all tasks in the group. The reason for this is that most events like stop or continue are predictable and the only unpredictable event is encountering a breakpoint. All predictable events can be handled by an operational group.

The conclusion is that the restricted (initial) version of behavioural groups is extremely useful. The unpredictable occurrence of encountering a breakpoint and stopping all tasks in the group can be used for detecting race conditions, which are very difficult to locate. Currently, there are no other unpredictable events in KDB other than breakpoint. One possible way of making a behavioural group more general is to allow some form of external events. This extension might also allow threads to be in more than one behavioural group. However, this idea is not implemented.

4.4 Programmatic Interface

Event collection and graphical visualization of the interactions among threads of a concurrent program can help significantly in understanding and debugging the program. Thus, it is useful to integrate all these abilities into a debugging environment. Normally debugger input is usually tied to keyboard input. However a programmatic interface to a debugger allows input from another program that wants to use the debugger for some purpose, e.g., an event visualization tool. Thus, KDB was extended with a programmatic interface.

Often a programmatic interface is piggy-backed on top of the user interface. For example, debuggers with a command line interface can be accessed programmatically by redirecting input and output. However, this approach is not always satisfactory because the communication link may not be general enough, and generating user-level inputs and parsing user-level outputs to find required information is tedious.

Instead, I augmented KDB with a separate communication channel and protocol for the communication on the channel. The channel is an INET socket so that remote (distributed) interaction is possible. The protocol is a simple machine level protocol for controlling KDB and obtaining output. Clearly, it would be best if there was a standard protocol for communicating with a debugger.

Two additional command line options are introduced:

- `-api`: enable the programmatic interface, an INET socket is created and the port number is printed on the terminal.

- `-nointerface`: disable the Motif debugger interface, the global debugger is started without the Motif window interface.

The following messages are supported by the programmatic interface, where each message name has a corresponding #define in the programmatic interface include file. The following constants and types are defined:

```
#define MAX_CMD_LEN   256
#define MAX_VAR_LEN   64
#define MAX_COND_LEN 128
#define MAX_PATH_LEN 256
#define MAX_PRINT_LEN 1024
typedef enum {BP_SET, ..., TERMINATE} MessageType;
typedef int          NotifyMsg;
typedef void*        ThreadId;
typedef void*        ListId;
```

BP_SET

Set a breakpoint in the specified user-thread. The message specifies the user thread id and the breakpoint location, which is parsed by KDB, for example,

- 82 *// breakpoint at line 82*
- Table::pickup *// breakpoint in the function pickup*
- Philosopher.cc:82 if k1 <= 10

The structure of the message from the controlling program is shown below:

```
struct {
  MessageType message_type;
  ThreadId thread_id;
  char break_cmd[MAX_CMD_LEN+MAX_COND_LEN];
};
```

The structure of a notification message from the global debugger is shown below:

```
struct {
  MessageType message_type;
  NotifyMsg   notify_msg;
};
```

On success, notify_msg is set to 0. In the case of failure, notify_msg is set to one of the following:

1. thread does not exist

2. thread is not in stopped state

3. breakpoint command error

4. other error

BP_CLEAR

Clear a breakpoint in the specified user thread. The message specifies the user thread id and the breakpoint location, which is parsed by KDB. Clear commands are similar to breakpoint commands except there is no conditional clause. For example:

- 82
- Table::pickup
- Philosopher.cc:82

The structure of the message from the controlling program is shown below:

```
struct {
    MessageType message_type;
    ThreadId thread_id;
    char clear_cmd[MAX_CMD_LEN];
};
```

The structure of a notification message from the global debugger is shown below:

```
struct {
  MessageType message_type;
  NotifyMsg   notify_msg;
};
```

On success, notify_msg is set to 0. In the case of failure, notify_msg is set to one of the following:

1. thread does not exist
2. thread is not in stopped state
3. clear command error
4. other error

CONTINUE

Continue a user thread, which may be previously stopped or has encountered a breakpoint. The message specifies the user-thread id of the user-thread to be continued. The structure of the message from the controlling program is shown below:

```
struct {
    MessageType message_type;
    ThreadId thread_id;
};
```

The structure of a notification message from the global debugger is shown below:

```
struct {
  MessageType message_type;
  NotifyMsg   notify_msg;
};
```

On success, notify_msg is set to 0. In the case of failure, notify_msg is set to one of the following:

1. thread does not exist

2. thread is not in stopped state (thread is in running already)

STOP

Stop a specified user thread. The message specifies the user thread id to be stopped. The structure of the message from the controlling program is shown below:

```
struct {
    MessageType message_type;
    ThreadId thread_id;
};
```

The structure of a notification message from the global debugger is shown below:

```
struct {
  MessageType message_type;
  NotifyMsg   notify_msg;
};
```

On success, notify_msg is set to 0. In the case of failure, notify_msg is set to one of the following:

1. thread does not exist

2. thread is in stopped state (thread is stopped already)

PRINT

Print a variable in the specified user thread. The message specifies the user thread id and the variable name. The result of print is sent back to the controlling program. The structure of the message from the controlling program is shown below:

```
struct {
    MessageType message_type;
    ThreadId thread_id;
    char var_name[MAX_VAR_LEN];
};
```

The structure of the notification message from the global debugger is shown below:

```
struct {
    MessageType message_type;
    NotifyMsg notify_msg;
    char print[MAX_PRINT_LEN];
};
```

On success, notify_msg is set to 0. The raw output is put in field print. In the case of failure, notify_msg is set to one of the following:

1. thread does not exist

2. thread is not in stopped state

ATTACH

Attach the debugger to the specified process. The message specifies the process id and the relative or absolute path of the executable. The structure of the message is shown below:

```
struct {
    MessageType message_type;
    int  pid;
    char path[MAX_PATH_LEN];
};
```

The structure of a notification message from the global debugger is shown below:

```
struct {
  MessageType message_type;
  NotifyMsg  notify_msg;
};
```

On success, notify_msg is set to 0. In the case of failure, notify_msg is set to one of the following:

1. process does not exist
2. executable file does not exist
3. executable file is not compiled with -debug option

CLUSTER_LIST, THREAD_LIST, PROCESSOR_LIST

Send the current cluster list, thread list, or processor list, depending on the message_type, to the controlling program. The structure of the message is shown below:

```
struct {
    MessageType message_type;
};
```

The structure of the notification message from the global debugger is shown below:

```
struct {
    MessageType message_type;
    NotifyMsg notify_msg;
    char list_name[MAX_VAR_LEN];
    ListId id;
};
```

Members of a list are sent one at a time. notify_msg is set to 0, if current list requested is completely sent. notify_msg is set to 1, if more list items are to be sent.

BP_HIT

Notification only message. Notify the controlling program that a breakpoint is encountered in a thread. The structure of the message is shown below:

```
struct {
    MessageType message_type;
    int thread_id;
};
```

PROGRAM_TERMINATED

Notification only message. Notify the controlling program that the program has terminated. The structure of the message is shown below:

```
struct {
    MessageType message_type;
};
```

TERMINATE

Close the connection. KDB removes any outstanding breakpoints from the application and terminates. At this point, the application continues execution normally. The structure of the message is shown below:

```
struct {
    MessageType message_type;
};
```

There is no notification.

4.5 Other Enhanced Features

4.5.1 Updating GDB

Although using GDB code has advantages, as mentioned in Section 2.2.3, there are a few disadvantages as well:

- the design of GDB is tightly integrated, hence it is non-trivial to use only the necessary parts
- the need to configure and compile GDB before KDB can be built
- GDB has to be constantly upgraded to work with the most recent version of the GNU compilers and it often lags behind. For example, gdb-4.14 can not find addresses of local variables for programs compiled with g++-2.7. This problem has been solved in gdb-4.16 on most platforms, however, it still exists on Solaris.

Figure 4.8 shows how GDB code is used with the rest of KDB. A library, `libgdb`, is constructed containing the parts of the GDB code that are needed by KDB and APIs for interacting with the GDB code. Modules that are not needed by KDB are removed (for example, language extensions for Fortran). The APIs were written to make access to GDB code easier. For example, the function accessGDB_getCallingFrame(prgreg_t* reg_set, int nth) creates and returns the nth stack frame backwards from the current frame. Finally, because GDB code is not thread safe, access to the APIs is covered by a monitor, `AccessGDB`, to serialize access.

The libraries, `libbfd`, `libiberty`, `libopcodes` and `libreadline`, are part of the GDB source distribution. Each library provides support for GDB. For example, `libbfd` (Binary File Descriptor) deals with different formats for executables and core files. These libraries use the μC++ `malloc` function, since the default system malloc is not safe for μC++ threads. This is achieved by turning on the flag -DNO_MMALLOC when the libraries are compiled.

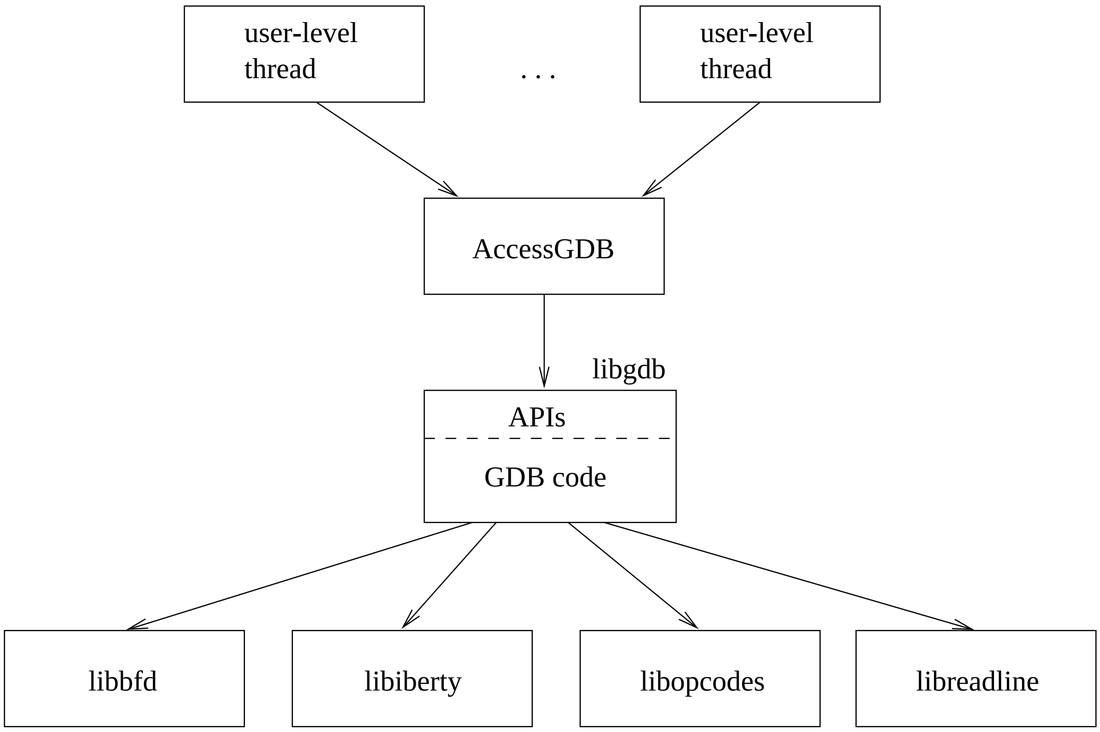

Figure 4.8: Access GDB utilities

During the development of KDB, GDB has been updated twice, from version 4.14 to 4.15 and 4.15 to 4.16, to keep compatibility with newer versions of g++.

For each update, the following enhancements were performed:

- check whether the GDB functions called from the APIs still exist and have the same functionality. Unfortunately, this check must be done by browsing GDB source code, reading comments and writing small programs to test the functionality.
- GDB contains a data structure, target_ops, that contains callbacks for a variety of actions, for example, reading a target's memory. This structure must be checked to determine if the declarations for callback functions have changed. If target_ops is extended, some additional callback functions have to be provided.
- update the patch files to reflect new locations of changes.
- remove newly added GDB object files that are not used by KDB, e.g., ser-tcp.o.

The APIs to GDB have also been augmented for implementing KDB on the i486 architecture. This is discussed in Section 6.4.

4.5.2 User Interface

I made a few changes to the KDB interface based on experience gained through usage.

Click to Print

One of the most common operations performed during debugging is printing the value of a variable. Normally, printing is performed by typing the entire print command in the command prompt of the thread window. To simplify this operation, the interface was modified so that double clicking on a variable or function in the source text pane of the thread window prints the value of the variable or the function address. The longest C-style variable/expression not separated by whitespace is selected, e.g., clicking anyway on "state[me]" selects the entire string. As well, the print command for this operation is also inserted into the command prompt verifying what command is executed. This command is also helpful for printing slight variants of the variable, for example, to print *NoPhils after printing NoPhils. Figure 4.9 shows a double clicking on the variable state[me], and the output is shown in the output pane. Finally, cut and paste capabilities were added to many of the windows.

Buffering Output

The previous interface flushed all output from the output panes of the main and Thread windows. However, previous output is very useful. Therefore the interface was changed to buffer output in the output panes, each logical block of output is separated by dashed lines. Figure 4.9 has 3 blocks of output.

Reducing Window Clutter

When debugging a large number of threads, a computer screen does not have enough space to show many thread windows, especially when working with operational/behavioural groups with a large number of threads. When a breakpoint is encountered in a thread, the default action is to create a thread window on the screen. This semantic can be very inconvenient if a breakpoint hit occurs in multiple threads within a group, causing multiple windows to pop-up and flood the screen. Therefore, this semantic was modified so that windows can be created in iconic state if one does not already exist, by clicking the toggle button in the group interface window.

Reorganizing Buttons on the Thread and Main Windows

Some of the buttons for the main and Thread windows were moved to make interacting easier, for example, common operations were moved closer together.

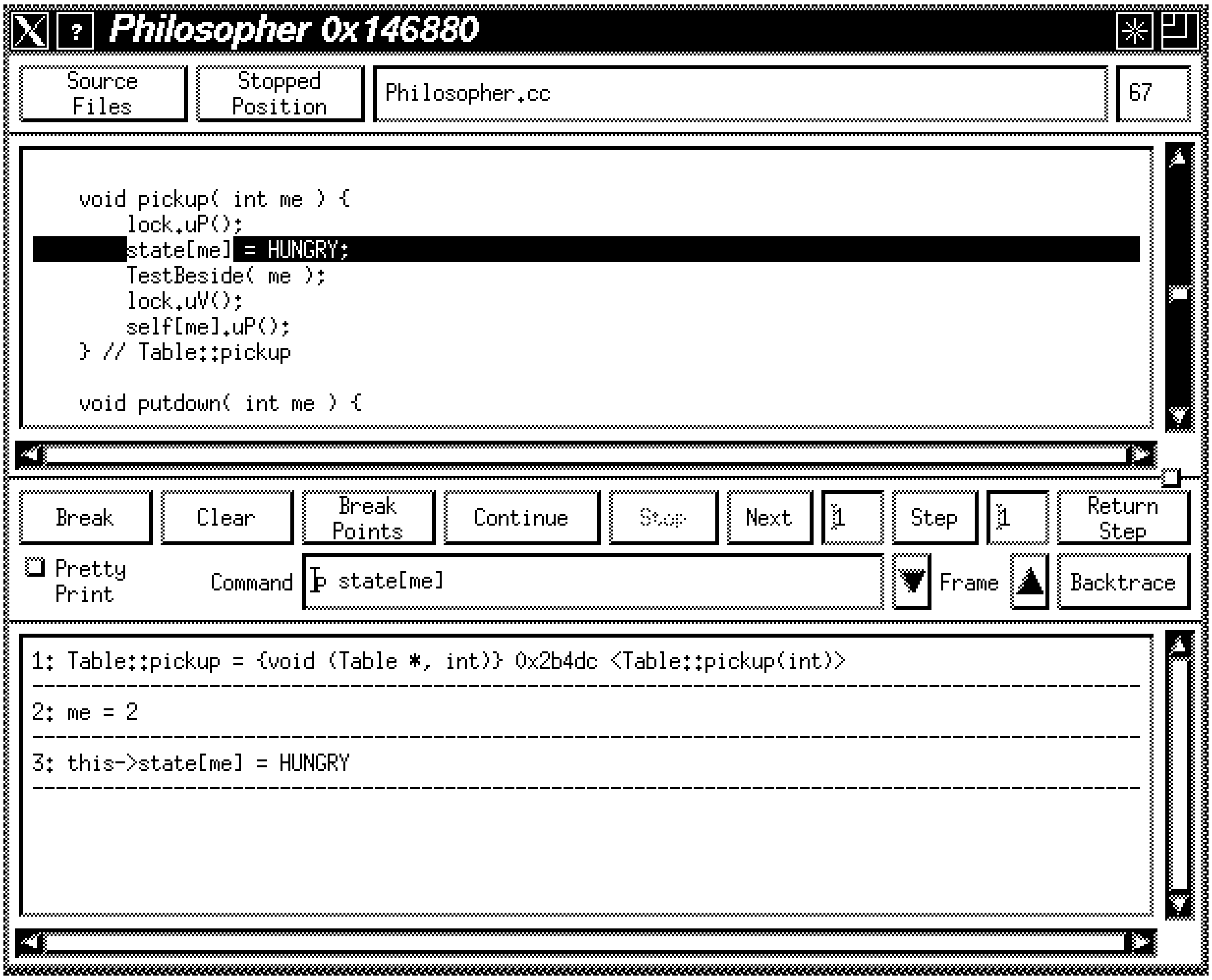

Figure 4.9: The Thread Window

Chapter 5

Debugging Translated Code

μC++ is a translator[1], which reads a program containing concurrent language extensions and translates each extension into one or more C++ statements. The translated program is then compiled by an appropriate C++ compiler and linked with a concurrency runtime library, called the μC++ kernel. Normally, the inserted code is invisible to programmers. Unfortunately the inserted code becomes visible when debugging at the statement level, since the inserted code is composed of one or more language statements. Currently, there is no facility in C++ or gdb to tag a group of statements as being 0 or possibly 1 statement. This same problem occurred as far back as `ratfor` [Ker75] and more recently in the C++ compiler (AT&T C front) [Str91], both of which were translators and not true compilers. In fact, a similar problem occurs when debugging optimized code because the program execution may not follow the source code.

[1]It is a "translator" because μC++ programs are partially parsed and symbol tables are constructed.

The problem this causes during debugging is that a programmer can easily step into translated code, which can be very confusing to novice programmer and even difficult to deal with for experienced programmers due to the complexities of μC++. If the translated code could be made invisible during debugging as it is at other times, then the debugging model would match the execution model, making life simpler for the programmers.

In this chapter, I discuss the methods that I developed to address the problem of debugging translated code. Both for normal inserted functions and for special situation associated with coroutine functions.

5.1 Normal Functions

A μC++ program and its translated C++ code are shown in Figure 5.1 (some editing of the translated code has been performed for presentation reasons). The translator preserves line numbers between the original and translated programs by generating long lines in the translated output (which is why the example in Figure 5.1 was edited). This approach means that compilation errors generate correct line numbers and stepping from line to line (called nexting) using the debugger hides the translated code, but not stepping into routines invoked within a line (called stepping). The translated code includes a number of declarations, including a default constructor and destructor for the task Dummy, which are inserted before the statement int x. In detail, each μC++ task has the following calls inserted: uBaseTask::uBaseTask, uSerial::uSerial, uQueue<uBaseTaskSL>::uQueue, uSerialConstuctor::uSerialConstuctor, uTaskConstructor::uTaskConstructor, and

μC++ code	Generated C++ code
uTask Dummy{	class Dummy : public uBaseTask {
	private:
	uAction uDestruct;
	Dummy(Dummy&);
	Dummy& operator = (Dummy&);
	protected:
	uSerial uSerialInstance;
	uBaseTaskQueue uEntry00;
	public:
	~ **Dummy(){{**
	uSerialDestructor u0007(uDestruct,uSerialInstance,uEntry00,0x00);{
	uTaskDestructor u0008(uDestruct, (uBaseTask&) *this);
	} }
	}
	Dummy (uAction uConstruct = uYes) { { uDestruct = uConstruct; {
	uSerialConstructor u0009(uConstruct, uSerialInstance); {
	uTaskConstructor u0010(uConstruct,*this,”Dummy”,uSerialInstance);
	} } }
	}
	private:
int x;	int x ;
void main() {	void main ( ) { **uTaskMain u0006 (uSerialInstance) ; {**
int i = 8;	int i = 8 ;
x = 8 * i;	x = 8 * i ;
}	} }
};	};
void uMain::main(){	void uMain :: main ( ) {
	uTaskMain u0011 (uSerialInstance) ; {
Dummy dummy;	Dummy dummy ;
int z = 9;	int z = 9 ;
int y = 10;	int y = 10 ;
}	} }

Figure 5.1: A Sample C++ Program and its Translated C++ Code

uSerialConstructor::~uSerialConstructor.

During a debugging session, a user would expect a stepping operation for the declaration

```
Dummy dummy;
```

to transfer to line

```
int z = 9;
```

because they did not write a constructor for the task. However, it transfers to the first line in task Dummy

```
int x;
```

since the inserted constructor for Dummy is defined on that line (remember the source in Figure 5.1 has been edited). This transfer can not be avoided, since there is no way to know if the constructor was provided by the user or the μC++ translator. Therefore, users must simply get use to this additional transfer.

However, an additional step operation by the user does not return after the declaration of dummy. Instead, it transfers into the function uBaseTask::uBaseTask, which is the base class constructor for any uTask class (more inserted code). This transfer of control is now very confusing for users in debugging their programs and also requires some knowledge of the implementation details of μC++. A similar situation occurs when a task exits, as the inserted destructor of the task is called.

There are two possible solutions to this problem. In order to discuss them, it is first necessary to look at how single stepping is implemented in KDB.

5.1.1 Single Stepping

In KDB, Martin implemented the traditional approach for single stepping of a user-level thread by inserting temporary breakpoints in the target application, continuing the application, and removing the breakpoints afterwards. As mentioned, there are two operations for controlled advancing of execution: *next*, which does not step into subroutines, and *step*, which does. For each kind of operation, multiple breakpoints may be inserted and removed afterwards.

A default breakpoint is set at the beginning of the next line in the source code. Additional breakpoints are inserted if any or all of the instructions are detected in the current source line:

- `call`: if the operation is *step*, a breakpoint is set at the target address of the call.
- `return`: a breakpoint is set in the caller's function after the call.
- `branch`: if the target address is beyond the scope of the current line, a breakpoint is set at the target address.

After the breakpoints are inserted, the thread is continued until one of the breakpoints is encountered, and then all temporary breakpoints are removed.

5.1.2 Hiding Translated Code

There are two possible solutions for dealing with translated code involving normal functions. The first solution is more general. The debugger checks if it is stepping into an inserted routine, like uBaseTask::uBaseTask. If it is, the single stepping is modified to:

step, return step, step

The first *step* steps into uBaseTask::uBaseTask, and the *return step* steps out of uBaseTask::uBaseTask. The last *step* steps into the next inserted routine in this case, uSerial::uSerial. And, a similar check is done for uSerial::uSerial, and a similar action is taken. This action is done for all inserted code defined on that line. When the last function is exited, the last *step* moves to the next execution point. Although this method works, it is very time consuming, since for each function called, multiple steps are taken.

The second solution is to examine the translated code for groups of consecutive calls; and next over the entire group. For example, none of the inserted functions at the start of a task need to be entered. Therefore, nexting over the single line inserted by the translator for all the calls in that group moves to the next execution point, i.e.,

```
int z = 9;
```

for the program in Figure 5.1. This action is accomplished by simply changing the single stepping operation into a nexting operation, which prevents stepping into any function defined on the translated line (see Section 5.1.1). The disadvantage of this approach is that the action is dependent on the particular code sequences generated by the translator. This solution is a refinement of the first solution, and does not generalize to all situations. Thus, I used the second solution wherever possible, and used the first solution otherwise.

5.1.3 Scheduling Statements

μC++ supports internal and external scheduling within objects that provide mutual exclusion. The statements for internal scheduling are uSignal and uWait. The statement for external scheduling is uAccept with clauses uWhen, uOr and uElse. Each of these statements is translated into one or more function calls at translation time. Clearly, for these μC++ language statements, control should not transfer into the translated functions generated for them by a step operation. Thus, a step operation should be equivalent to a next operation. A uAccept statement is translated into functions that have unique names, for example, the statement,

```
uAccept (z);
```

is translated to the line (shown here as multiple lines),

```
unsigned int uHere ;
uSerialInstance . uAcceptStart ( uHere ) ;
if ( uSerialInstance . uAcceptTry ( uEntry02 , 0x02 ) ) {
    u0013 : ;
. . .
```

Thus, for each step operation on a scheduling function, the debugger changes it to a next operation instead. Again, this works because the translated code is a single line of C++ code.

However this scheme fails for complex statements translated into multiple statements and routine calls, for example,

```
uWhen ( fred() ) uAccept (z);
```

which is translated to the line (shown here as multiple lines)

```
unsigned int uHere ;
uSerialInstance . uAcceptStart ( uHere ) ;
if ((fred ( )) && uSerialInstance.uAcceptTry( uEntry02, 0x02 )) {
  u0014 : {
...
```

The scheme described above skips to the next line, and never enters the user function fred(), while step should enter the function.

Thus, the *step* operation into the function uAcceptStart is changed to:

step, return step, step

which ensures control steps into the user function fred(). If there is no user function, this stops in the function uAcceptTry, which is a special function and the *step* operation is handled accordingly.

5.1.4 Event Trace

When a μC++ application is compiled with the option `-trace`, additional runtime code is inserted in objects to generate events for particular μC++ operations. See [BK95] for information about displaying event traces. With the `-trace` option, the statement

```
uAccept (z);
```

is translated to the line (shown here as multiple lines)

```
unsigned int uHere ;
uSerialInstance . uAcceptStart ( uHere ) ;
uTraceAcceptStart ( uTraceInstance ) ;
if ( uSerialInstance . uAcceptTry ( uEntry02 , 0x02 ) )
```

All inserted trace code, like uTraceAcceptStart, is also handled using similar techniques as the ones discussed in this section.

5.2 Coroutine Functions

5.2.1 What is a Coroutine?

The flow control of coroutine functions are very different from those of normal functions. A *coroutine* is an object with its own execution state so its execution can be suspended and resumed. Execution of a coroutine is suspended as control leaves it, only to carry on from that point when control returns at some later time. This means that a coroutine is not restarted at the beginning on each activation and that its local variables are preserved. Hence, a coroutine solves the class of problems associated with finite-state machines and push-down automata, which are logically characterized by the ability to retain state between invocations.

Coroutines can be used in two slightly different ways:

- A `semi-coroutine` is characterized by the fact that it always activates its caller, as in the producer-consumer example of Figure 5.2. Notice the explicit call from Prod's main routine to delivery and then the return back when delivery completes. delivery always activates its coroutine, which subsequently activates delivery.

- A `full-coroutine` is characterized by the fact that it never activates its caller; instead, it activates another coroutine by invoking one of its member routines. Thus, full coroutines activate one another often in a cyclic fashion, as in the producer-consumer example of Figure 5.2. Notice the uResume statements in routines payment and delivery. The uResume in routine payment activates the execution-state associated with Prod::main and that execution-state continues in routine Cons::delivery. Similarly, the uResume in routine delivery activates the execution-state associated with Cons::main and that execution-state continues in Cons::main initially and subsequently in routine Prod::payment.

5.2.2 Debugging Coroutines

While it is possible to debug coroutines using GDB because there is no concurrency, once a *step* occurs into the translated code for uResume and uSuspend, it does not come out in the resumed coroutine because control must first transfer through a context switch; using *next* skips the important control flow into the other coroutine. In effect, uResume and uSuspend need to be handled in a similar way to routine calls. That is, a step at a uResume or uSuspend ends up at the correct execution

Producer

```
uCoroutine Prod {
    Cons *c;
    int N, status;

    void main() {
        int p1, p2;
        // 1st resume starts here
        for ( int i = 1; i <= N; i += 1 ) {
            ... // generate a p1 and p2
            status = c->delivery( p1, p2 );
            if ( status == ... ) ...
        } // for
        c->delivery( -1, 0 );
    }; // main
  public:
    Prod( Cons *c ) { Prod::c = c; };
    void start( int N ) {
        Prod::N = N;
        uResume;          // restart Prod::main
    }; // start
}; // Prod
```

Consumer

```
uCoroutine Cons {
    int p1, p2, status;

    void main() {
        // 1st resume starts here
        while ( p1 >= 0 ) {
            // consume p1 and p2
            status = ...;
            uSuspend;    // restart Cons::delivery
        } // while
    }; // main
  public:
    int delivery( int p1, int p2 ) {
        Cons::p1 = p1;
        Cons::p2 = p2;
        uResume;         // restart Cons::main
        return status;
    }; // delivery
}; // Cons

int main() {
    Cons cons;             // create consumer
    Prod prod( &cons );    // create producer

    prod.start( 10 );      // start producer
} // main
```

Figure 5.2: Semi-Coroutine Producer-Consumer

Producer

```
uCoroutine Prod {
    Cons *c;
    int N, money, status, receipt;

    void main() {
        int p1, p2;
        // 1st resume starts here
        for ( int i = 1; i <= N; i += 1 ) {
            ... // generate a p1 and p2
            status = c->delivery( p1, p2 );
            if ( status == ... ) ...
        } // for
        c->delivery( -1, 0 );
    }; // main
  public:
    int payment( int money ) {
        Prod::money = money;
        ... // process money
        uResume; // restart prod in Cons::delivery
        return receipt;
    }; // payment
    void start( int N, Cons *c ) {
        Prod::N = N;
        Prod::c = c;
        uResume;
    }; // start
}; // Prod
```

Consumer

```
uCoroutine Cons {
    Prod *p;
    int p1, p2, status;

    void main() {
        int money, receipt;
        // 1st suspend starts here
        while ( p1 >= 0 ) {
            // consume p1 and p2
            status = ...
            receipt = p->payment( money );
        } // while
    }; // main
  public:
    Cons( Prod *p ) { Cons::p = p; };
    int delivery( int p1, int p2 ) {
        Cons::p1 = p1;
        Cons::p2 = p2;
        uResume;
        // restart cons in Cons::main 1st time
        // and cons in Prod::payment afterwards
        return status;
    }; // delivery
}; // Cons

int main() {
    Prod prod;              // create producer
    Cons cons( &prod );     // create consumer

    prod.start( 10, &cons );// start producer
} // main
```

Figure 5.3: Full-Coroutine Producer-Consumer

point with respect to coroutine control flow, just as a step at a call or return ends up at the correct execution point with respect to subroutine control flow. Initially, KDB was worse than GDB with respect to following coroutine execution because breakpoints can not be set in some parts of the μC++ kernel as mentioned in Section 4.1.1, thus it was impossible to even *step* into the context switch routine to get to the other coroutine, which is possible with GDB.

The only scheme available to a programmer using the old version of KDB was to explicitly set breakpoints after each uSuspend and uResume to catch control flow after the context switch. Clearly, this approach is tedious and error-prone. To solve this problem, it is possible to calculate the address in the other coroutine where it is re-activated from its stack information, then implement a *step* operation by setting a breakpoint at that address and continuing the execution. However, calculating the re-activation address is a non-trivial task, because the location of the return address on the stack is architecture dependent. An alternative approach, which I subsequently adopted, is to look at the locations where the execution ends up after executing uResume and uSuspend. There are three possible locations for uResume:

- initiates coroutine's main routine, which only occurs once when uResume is called for the first time. The first routine encountered on starting a coroutine is uCoroutineMain::uCoroutineMain, which is inserted by the translator.

- re-activates the coroutine's main at the last uSuspend statement. The translated routine that is restarted is uBaseCoroutine::uCoSuspend.

- re-activates another coroutine's main at the last uResume. The translated routine that is restarted is uBaseCoroutine::uCoResume.

There is one possible location for uSuspend:

- transfer to last uResume statement. The translated routine that is restarted is uBaseCoroutine::uCoResume.

Stepping into the translated code for uResume or uSuspend is then modified into:

1. Setting temporary breakpoints at the end of the functions uCoroutineMain::uCoroutineMain, uBaseCoroutine::uCoSuspend and uBaseCoroutine::uCoResume for uResume, or at the end of the function uBaseCoroutine::uCoResume for uSuspend. Then a continue operation is performed.

2. A normal single stepping operation: *step*

After the continue operation, the program has executed through the context switch to the other coroutine and stops at the last line in one of the translated routines, and the second stepping operation transfers to the expected location in the other coroutine. The temporary breakpoints are then removed.

Thus, it is now possible to follow coroutine execution with KDB as easily as it is to follow subroutine execution.

5.3 Implementation Problems

In this chapter, I presented techniques for debugging translated code involving both normal functions and coroutine functions. In order for these techniques to work, it is essential to find the exact addresses of the inserted functions. However, some of these functions may be inline, and hence there may be multiple copies of the inline functions in the executable. For example, the gnu C++ compiler version 2.6 has this problem, but it is resolved in version 2.7.

These techniques also depend on the implementation of μC++. Any changes related to the inserted code (e.g., changing the name of a function) requires the debugger to be modified as well. However, this problem is mitigated somewhat by abstracting special inserted function names into a table with their corresponding actions. Therefore, it is very easy to change the names of special functions, or add new special functions and their actions. The special function table is shown in Table 5.1.

Finally, there is a cost associated with scanning this table during stepping, but the cost of searching this table is insignificant compared to the cost of setting and removing breakpoints needed for these debugger commands.

Special Inserted Function	Stepping Action
uCoroutineMain::~uCoroutineMain	next
uBaseCoroutine::uBaseCoroutine	next
uCoroutineDestructor::uCoroutineDestructor	next
uSerialMember::uSerialMember	step, return step
uSerialMember::~uSerialMember	step, return step
uSerialDestructor::uSerialDestructor	next
uSerialConstructor::~uSerialConstructor	next
uSerial::uSerial	next
uTaskMain::uTaskMain	step, return step
uBaseTask::uBaseTask	next
uTaskDestructor::uTaskDestructor	next
uTaskMain::~uTaskMain	next
uCondition::uS	next
uCondition::uW	next
uSerial::uAcceptStart	step, return step, step
uSerial::uAcceptEnd	next
uSerial::uAcceptTry	next
uSerial::uAcceptTestMask	next
uSerial::uAcceptElse	next
uTraceInit::uTraceInit	next
uTraceSuspend::uTraceSuspend	step, return step, step
uTraceResume::uTraceResume	step, return step, step
uTraceResume::~uTraceResume	step, return step
uTraceDestructor::uTraceDestructor	next
uTraceMain::~uTraceMain	next
uTracePetition::uTracePetition	next
uTraceAcceptStart::uTraceAcceptStart	step, return step, step

Table 5.1: Special Inserted Code with Actions

Chapter 6

Implementing KDB on Linux

While KDB was originally designed to support multiple architectures, it was only implemented on the SPARC architecture in a UNIX environment. Thus, I implemented KDB on the Intel 486 architecture in a Linux environment for three reasons:

- to determine if the original design is truly generalized to support multiple architectures,
- to get KDB running on an important, useful platform used by many students: Linux, and
- because learning how to implement user-level breakpoints on different architectures helps to extend KDB to run in a heterogeneous distributed environment in the future.

Intel 80x86 computers are CISC machines which are very different from RISC machines such as SPARC computers. Thus, a significant amount of work was involved in porting KDB to the Intel 486 architecture. Also this difference required additional generalization in the architecture of this component of KDB.

6.1 Process Control under Linux

In KDB, implementing and removing breakpoints in the target application is accomplished by using `ptrace` or `/proc`. In Linux, the `ptrace` system call is supported for process control. Linux does provide a `/proc` file system; however, it is different from the UNIX SVR4 `/proc` file system. The SVR4 `/proc` (process file system) is a file system that provides access to the image of each process in the system. The Linux `/proc` (process information pseudo-filesystem) is a pseudo-filesystem used as an interface to kernel data structures rather than reading and interpreting `/dev/kmem`.

The Linux `ptrace` call is very similar to that found in SunOS, but with fewer operations. The following operations supported by SunOS are not supported by Linux:

```
PTRACE_GETREGS,         /* 12, get all registers */
PTRACE_SETREGS,         /* 13, set all registers */
PTRACE_GETFPREGS,       /* 14, get all floating point regs */
PTRACE_SETFPREGS,       /* 15, set all floating point regs */
PTRACE_READDATA,        /* 16, read data segment */
PTRACE_WRITEDATA,       /* 17, write data segment */
PTRACE_READTEXT,        /* 18, read text segment */
PTRACE_WRITETEXT,       /* 19, write text segment */
```

```
PTRACE_GETFPAREGS,      /* 20, get all fpa regs */
PTRACE_SETFPAREGS,      /* 21, set all fpa regs */
PTRACE_DUMPCORE,        /* 25, dump process core */
PTRACE_SETWRBKPT,       /* 26, set write breakpoint */
PTRACE_SETACBKPT,       /* 27, set access breakpoint */
```

In Linux, `PTRACE_PEEKUSR` and `PTRACE_POKEUSR` must be used to get and set a process's registers. The code for getting and setting a process's registers in KDB was rewritten for Linux.

6.2 Setting and Resetting a Breakpoint

Intel 80x86 computers are CISC machines and instructions are of variable length. Thus, it is much harder to decode 80x86 instructions than to decode SPARC instructions. All instruction encodings are subsets of the general instruction format that consists of optional prefixes, one or two primary opcode bytes, possibly an address specifier consisting of the ModR/M byte and the SIB (Scale Index Base) byte, a displacement, if required, and an immediate data field, if required. The shortest instruction is 1 byte in size, and the longest instruction without redundant prefixes is 15 bytes long for the Intel 486.

6.2.1 Saving and Restoring the Local State

The breakpoint handler must save the state of a user-level thread, so when execution continues, the same state can be restored. This action is achieved by saving all registers and loading them back immediately before execution continues, which is shown by the calls saveApplicationState() and restoreApplicationState() in Figure 4.1. On the Intel 486, the Stack-Frame Base Pointer (EBP) register is automatically saved by pushing it on to the stack, at the start of each function, as shown below:

```
pushl %ebp
movl %esp,%ebp
```

The additional registers are then saved by:

```
pushal          # save all 32-bit general registers
pushf           # save the EFLAGS register
```

Note that pushal and pushf push 9 four-byte registers onto the stack (see Figure 6.1).

Since the callee is responsible for saving the general registers, they must be saved in the breakpoint handler. Condition codes (e.g., carry, sign, overflow) and mode bits are kept in the 32-bit register EFLAGS, which must be saved as well.

If the breakpoint is applicable to the current thread, the breakpoint handler calls the local debugger to notify the global debugger. This call automatically saves the registers as part of the standard calling convention. When the thread is continued, the reverse instructions are executed, before control transfers back into the application or before the temporary code is executed. This restore is done by the call restoreApplicationState() shown in Figure 4.1 and is shown below:

```
movl %ebp,%esp
subl $36,%esp
popf
popal
```

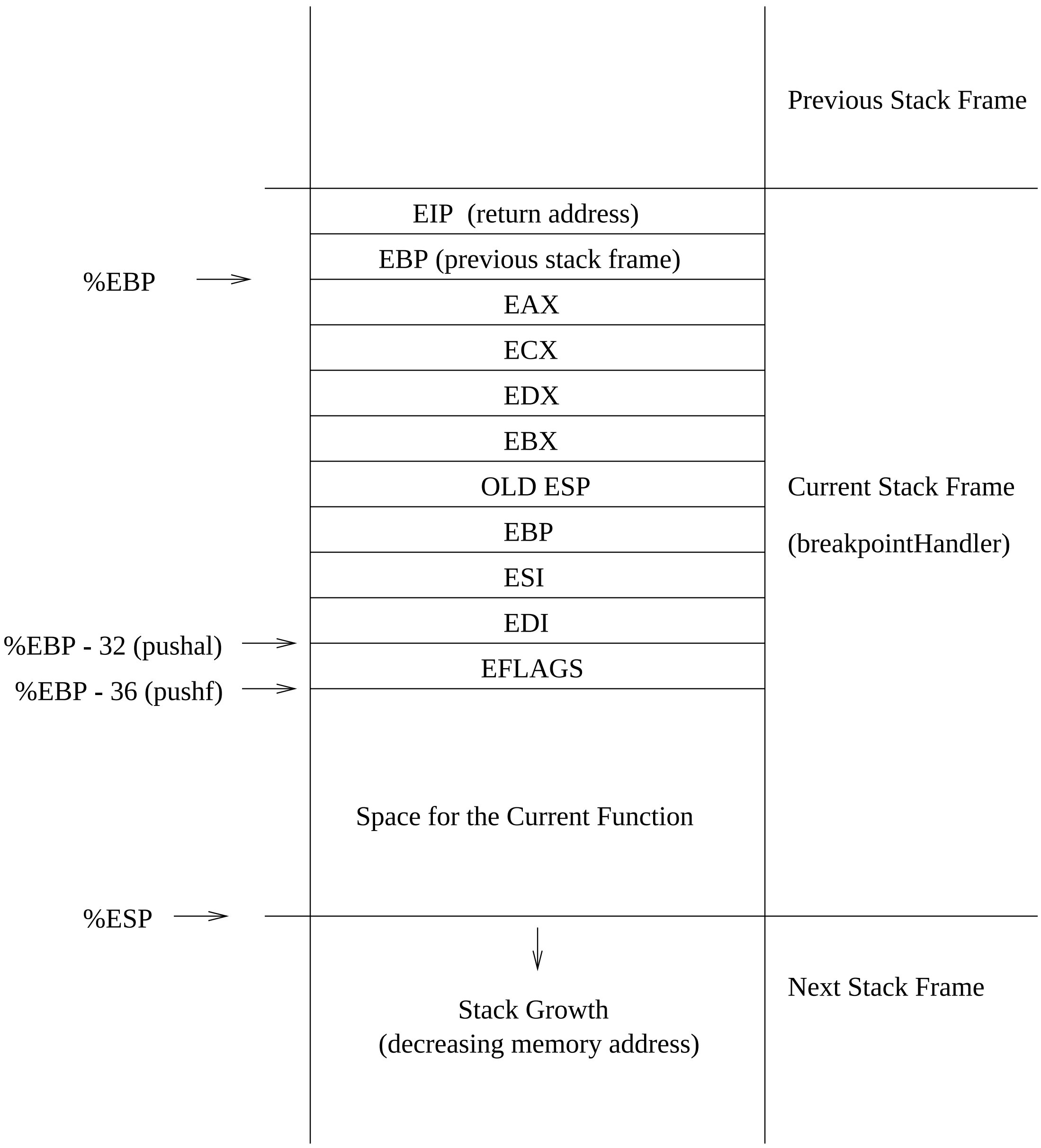

Figure 6.1: The breakpointHandler Stack Frame

6.2.2 Creating the Temporary Instructions

On the SPARC (and most other RISC machines), a fast breakpoint (see Section 2.2.2) is implemented by replacing the original instructions with a `call` instruction to a breakpoint handler, followed by a nop instruction (the delay slot). Since CISC machines do not require delay slots after transfers, only the `call` instruction is inserted at the address of the breakpoint. The Intel 486 `call` instruction is 5 bytes long, which spans the entire 32 bit address space. However, the number of bytes to be replaced can be up to 19 bytes long (e.g., a 4 byte instruction followed by a 15 byte instruction). Thus, some nop instructions have to be inserted after the `call` instruction if more than 5 bytes of original instructions are replaced.

Intel 486 conditional jump instructions (Jcc) can be 2 or 6 bytes long. For the 2 byte long instruction, the displacement address ranges from 128 bytes before the instruction to 127 bytes after the instruction. Thus, if a 2 byte Jcc instruction is replaced at the breakpoint address, it must be changed to a corresponding 6 byte long Jcc instructions so it executes correctly at the temporary location in the breakpoint handler. An unconditional jump instruction (jmp) can also be 2 or 5 byte long, thus a 2 byte jmp instruction at the breakpoint address must also be changed to a 5 byte long jmp instruction. Therefore, at least 27 bytes (e.g., 2 two-byte jump instructions followed by a 15 byte instruction) must be reserved in the breakpoint handler to store the worst case temporary instructions (see Figure 4.1).

There is a problem if a `branch` points to the middle of some inserted breakpoint instructions. For example, in Table 6.1, a breakpoint is set at line 10. Since the instruction on line 10 is 4 bytes long, the next instruction on line 11 also needs to be replaced since the call instruction is 5 bytes long. In this case, 7 bytes are replaced to implement the breakpoint instruction. If a branch instruction points at line 11 or address 0x800057a, and is taken while the breakpoint is implemented, it executes at the $5th$ byte of the call instruction. This problem is similar to the situation on the SPARC architecture where a `branch` points to the address right after the breakpoint address (the nop after the call). To handle this situation on the SPARC, the code of the current function is checked for branches to this address before the temporary code is generated, and if a branch to this address is detected, the address of the breakpoint is adjusted by +1. For the i486, if a branch to this address is detected, the address of the breakpoint is adjusted to the beginning of the next source line.

Original			**Breakpoint**	
Line	**Address**	**Instruction**	**Address**	**Instruction**
10	0x8000576	movb $0x61,0xffffffff(%ebp)	0x8000576	call _bphandler_0
11	0x800057a	movb 0xffffffff(%ebp),%al		
			0x800057b	nop
			0x800057c	nop

Table 6.1: A Potential Problem

The temporary code in the breakpoint handler for the general case is shown in Figure 6.2. The instructions

```
movl %ebp,%esp      // redo function save
popl %ebp
addl $4, %esp       // skip return address
```

restore the local state of the application. Since the call instruction pushes the return address onto the stack, this address has to be removed from the stack in order to execute the original instructions correctly. The correct return address is pushed onto the stack after the original instructions are executed (shown on the second last line).

```
movl %ebp,%esp       // redo function save
popl %ebp
addl $4, %esp        // skip reuturn address

original instruction 1
original instruction 2
.....
original instruction i              // at least 5 bytes

pushl ReturnAddress
retl
```

Figure 6.2: Temporary Code

`jump` Instructions

The argument of jcc and jmp instructions are adjusted when moved to the temporary location, since the target address is calculated relative to the program counter.

`call` Instruction

Like the `jump` instructions, the argument of call has to be adjusted when it is moved to the temporary location. However, there are two problems with this. First, a backtrace of the current function calls would include breakpoint_handler_i() instead of the caller from the original code (see Figure 4.1). For example, a backtrace would look like the one shown in Figure 6.3 instead of the one shown in Figure 6.4. breakpoint_handler_i() should never be seen by the user. Second, in the example shown in Figure 6.3, if the user steps through the function g, execution ends up in the function breakpoint_handler_i() as the user steps out of function g. This situation causes an error, since breakpoints can not be inserted in a breakpoint handler.

```
#0  g () at handler.cc:8
#1  0x804beb2 in breakpoint_handler_i ()
#2  0x8089726 in uMachContext::uInvoke ()
```

Figure 6.3: Incorrect Backtrace

```
#0  g () at handler.cc:8
#1  0x8049ed2 in uMain::main (this=0x80e9c88) at handler.cc:12
#2  0x8089726 in uMachContext::uInvoke ()
```

Figure 6.4: Correct Backtrace

On the SPARC architecture, these problems are avoided by placing a restore instruction right after the call instruction. The return address is set by restore instead of call. On the Intel 486 architecture, the call instruction is not moved into the breakpoint handler. Instead, it is replaced by a jmp instruction as shown below:

```
pushl return_address        // 5 bytes
jmp   function_address      // 5 bytes
```

This substitution of the jmp for the call fixes both of the previous problems.

6.3 Conditional Breakpoints

There is no difference in evaluating static variables on the i486 and SPARC architectures. There is some difference in evaluating register and local variables. As mentioned in Section 4.1.2, variables in conditional expressions for conditional breakpoints may be in registers, which must be accessed by the local debugger. On an i486, as described in Section 6.2.1, the registers are saved when the breakpoint handler is called, i.e., at the beginning of the breakpoint handler (see Figure 6.1). On a SPARC, registers are saved at the end of the user function (see Figure 4.4). The address of a register variable on an i486 is calculated as

```
sp - register_offset[register_number]
```

where sp is the stack pointer of the user function (see Figure 4.5), and register_offset is the offsets for registers stored on a stack.

The address of a local variable on an i486 is calculated similar to the SPARC case,

```
address = EBP + offset
```

where EBP (fp on SPARC) is the stack-frame base pointer of the user function, and offset is the relative address of the local variable. The EBP value of the previous stack frame is directly pointed to by the current EBP on the stack. The fp value of the previous stack frame on SPARC is saved on the stack.

6.4 Augmenting GDB Code

As mentioned in Section 2.2.3, one advantage of using GDB code is that GDB is highly portable. However, some changes to the GDB APIs had to be made:

- remove the GDB object files not used by KDB.
- modify the Makefile to generate the dummy functions on Linux for functions in the GDB object files that are removed.
- architecture and OS specifics defined in APIs are changed appropriately, e.g., matching the system definition and GDB definition for the register set, and modifying the target_ops data structure, which stores platform dependent data and callback functions such as reading register values. Some of these functions were rewritten.
- check that the GDB functions called from the APIs still exist and have the same functionality as those on the SPARC. This check is similar to that for updating GDB discussed in Section 4.5.1.

Chapter 7

Conclusions

Concurrent programming is difficult, since concurrent programs contain both sequential errors and additional concurrent errors. The complexity of concurrent programming can be reduced by using high-level concurrency constructs through programming languages such as μC++. A symbolic debugger that understands the concurrency constructs can improve debugging capabilities and reduce debugging time significantly. KDB is a concurrent symbolic debugger that provides independent control of user-level threads in a shared-memory environment. This thesis presents several ways to improve KDB and make it a more powerful debugger.

7.1 Summary

Chapter 4 describes features I added to KDB. These features include fast restricted conditional breakpoints, where the condition is evaluated by the local debugger; behavioral groups that allow a user to debug the application at a higher level;

attachment of the debugger to a running application, which allows KDB to debug a running application; programmatic interface, which allows KDB to be controlled by a program through a socket; and enhancements to the user interface.

Chapter 5 presents the approach I used to make μC++ translator code invisible to programmers during debugging. The strategy adopted to hide the code is to give special treatment to μC++ inserted routines when stepping into them. By generalizing the translated routines into a table with appropriate actions, the approach is reasonably flexible. Portability to other user-level thread libraries requires a table for each library.

Chapter 6 gives the implementation details of KDB on Linux for the Intel 486 architecture. KDB was designed to support multiple architectures, but only two UNIX versions on SPARC architectures were supported. Since the Intel 486 is a CISC machine, the implementation of user-level breakpoints is very different from that of SPARC. The process control is also different in Linux as in SunOS and SVR4. These differences required additional generalization in the architecture of this component of KDB.

7.2 Experience

Peter Buhr and I have used both the old and new KDB to debug many μC++ applications, including KDB itself. As well, the old KDB has been used by students in the CS342 concurrency course taught at the University of Waterloo. Unfortunately, the new KDB was not ready for student testing during this term so I cannot report on student experiences with the new KDB. The new KDB will be used the next time the CS342 course is taught.

Nevertheless, many of the new KDB extensions were made because of feedback from students using the old KDB. In particular, complaints were voiced about stepping into μC++ translator code and being unable to follow coroutine execution. Hiding μC++ translator code and dealing with coroutine control flow does simplify debugging. Both Peter and I noticed an improvement once this feature was available. I believe this improvement will help both teaching and learning μC++ concepts, especially coroutines, which are very different from normal routines. As well, both Peter and I found the simple click-to-print enhancement extremely useful during debugging, especially for long variable names.

While the other enhancements are not used as often, I would say that I used each enhancement at some point in debugging a concurrent program, and without that enhancement it would have been extremely difficult to have found the error. For example, attaching KDB to a running application was used to find several difficult race and deadlock conditions. These conditions did not occur often, so once the program failed, it was essential to extract as much information as possible directly from the failed execution. Once KDB was attached to the program, the ability to examine each task in detail was invaluable in locating the error.

While each enhancement is useful, I believe it is the cumulative effect of the set of enhancements that is most important. Users will use two enhancements to locate one problem and three different ones to locate another. Therefore, no particular enhancement stands out, but each one is needed to deal with the diverse kinds of errors found in a concurrent program.

7.3 Future Work

Although, the fast restricted conditional breakpoints are useful, it does not support all necessary conditions. It could be desirable to have a fallback option where the global debugger evaluates more complex conditions but at a significantly higher cost. Also behavioural groups can be made more general by accepting asynchronous external events rather than just breakpoint events in the application.

As mentioned in section 2.1, event collection and graphical visualization is a very useful tool in understanding concurrent programs. A more powerful debugging environment may be built by integrating these tools and the debugger. KDB is now extended with a programmatic interface, so its functionality can be accessed by commands via a socket instead of the user interface. Thus, KDB may be integrated with some event engine in a similar way to [Yu96].

The implementation of KDB on Linux supports the claim [Kar95] that the design of KDB can be generalized to multiple architectures, although there was more work than originally expected. More ports should be done to further test the portability of KDB and general applicability of the concept for user-level breakpoint handling.

Finally, the design of KDB is applicable to both shared-memory and distributed memory, but it is only implemented on shared-memory architectures. Ports to more architectures would allow KDB to run in a heterogeneous distributed environment. As discussed in [Kar95], a stub program would run on each machine, which performs some requests, like implementing a breakpoint, on behalf of the global debugger. A daemon, similar to [MMP+96], must also run on each machine to invoke the stub program.

Bibliography

[ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[Bat95] Peter C. Bates. Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior. *ACM Transactions on Computer Systems*, 13(1):1–31, February 1995.

[BK95] Peter A. Buhr and Martin Karsten. μC++ monitoring, visualization and debugging annotated reference manual, version 1.0. Technical Report Unnumbered: Available via ftp from plg.uwaterloo.ca in pub/MVD/Visualization.ps.gz, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, September 1995.

[BKS96] Peter A. Buhr, Martin Karsten, and Jun Shih. A multi-threaded debugger for multi-threaded applications. In *Proceedings of SPDT'96: SIG-

METRICS Symposium on Parallel and Distributed Tools, pages 80–87, Philadelphia, Pennsylvania, U. S. A., May 1996. ACM Press.

[BS96] Peter A. Buhr and Richard A. Stroobosscher. μC++ annotated reference manual, version 4.6. Technical Report Unnumbered: Available via ftp from plg.uwaterloo.ca in pub/uSystem/uC++.ps.gz, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, July 1996.

[CDG+93] D. Culler, A. Dusseau, S. Goldstein, A. Kerishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in split-c. In *Proceedings of Supercomputing '93*, pages 262–273, Portland, Oregon, Nov 1993.

[Gai86] Jason Gait. A probe effect in concurrent programs. *Software - Practice and Experience*, 16(3):225–233, March 1986. [Same as Gait1x].

[Hoo96] Robert Hood. The p2d2 project: Building a portable distributed debugger. In *Proceedings of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 127–136, Philadelphia, Pennsylvania, U. S. A., May 1996. ACM Press.

[Kar95] Martin Karsten. A Multi-Threaded Debugger for Multi-Threaded Applications. Diplomarbeit, Universität Mannheim, Mannheim, Deutschland, August 1995.

[Ker75] B.W. Kernighan. Ratfor - a preprocessor for a rational fortran. *Software Practice and Experience*, 5(10):395–406, October 1975.

[Kes90] Peter B. Kessler. Fast breakpoints: design and implementation. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 78–84, White Plains, NY, June 1990.

[LC96] Steven S. Lumetta and David E. Culller. The Mantis Parallel debugger. In *Proceedings of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 118–126, Philadelphia, Pennsylvania, U. S. A., May 1996. ACM Press.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[MMP+96] Michael S. Meier, Kevan L. Miller, Donald P. Pazel, Josyula R. Rao, and James R. Russell. Experiences with building distributed debuggers. In *Proceedings of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 70–79, Philadelphia, Pennsylvania, U. S. A., May 1996. ACM Press.

[MR91] Stephen P. Masticola and Barbara G. Ryder. A model of ada programs for static deadlock detection in polynomial time. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 26(12):97–107, December 1991.

[NM92] Robert H. B. Netzer and Barton P. Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1:74–88, March 1992.

[SP95] Richard M. Stallman and Roland H. Pesch. *Debugging with GDB*. Free Software Foundation, 675 Massachusetts Avenue, Cambridge, MA 02139 USA, 1995.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, second edition, 1991.

[Suna] Sun. proc(4). manual page.

[Sunb] Sun. ptrace(2). manual page.

[Tay93] David J. Taylor. The use of process clustering in distributed-system event displays. In *Proceedings of CASCON '93*, pages 505–512, Toronto, Oct 25-28 1993.

[Yu96] Ivan Yan-Kit Yu. Integrating event visualization with sequential debugging. Master's Essay, The University of Waterloo, May 1996.