KDB: A Multi-threaded Debugger for Multi-threaded Applications

Peter A. Buhr†, Martin Karsten‡ and Jun Shih†
† Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1
‡ Fakultät für Mathematik und Informatik, Universität Mannheim, Mannheim, Deutschland
E-mail: {pabuhr,mkarsten,jshih}@uwaterloo.ca

1 Introduction

Concurrent programs contain both sequential and concurrent errors. While deadlock and race conditions are unique to concurrent programs, there also exist algorithmic design errors, such as inhibiting concurrency, which are unknown in the sequential domain. Recently, there has been a large effort in debugging race conditions [16], both statically [8] and dynamically [7], and to a lesser extent, deadlock [13]. Our experience shows that concurrent errors occur with diminishing frequency in the order: traditional sequential errors, algorithmic design errors, deadlock, race conditions. However, the difficulty in determining and fixing these errors grows exponentially from sequential errors to race conditions. Our experience also shows that the frequency of deadlock and race-conditions diminishes significantly when high-level concurrency constructs (e.g., task, monitor, actor, etc.) are used, versus thread and lock programming.

We believe the best way to improve concurrent debugging capabilities and significantly reduce debugging time is to use high-level concurrency constructs with a symbolic debugger that truly understands it is debugging a concurrent program coupled with a cooperative concurrent run-time system that actively participates in the debugging process. Additionally, the debugger must provide independent and concurrent access to every thread of control in the target program. Such a debugger handles a large set of errors in concurrent programs, leaving esoteric errors to specialized debugging tools. Ultimately, a debugger and specialized tools must complement each other.

Our experience comes from designing high-level concurrent extensions for C++, called μC++ [5], using μC++ to build itself, a debugger and visualization toolkit [4], a database toolkit [3], and using μC++ to teach concurrency to undergraduate students. μC++ is a shared-memory user-level thread library that runs on symmetric multiprocessor architectures (e.g., SUN, DEC, SGI, Sequent); user-level threads are executed by multiple kernel threads associated with shared memory, which provides true parallelism when appropriate hardware is available. Furthermore, μC++ provides several high-level concurrency constructs, e.g., coroutines, monitors and tasks, for composing an application; hence, programmers do not work at the level of threads and locks. Over the last 5 years, we have experienced the full gamut of sequential and concurrent errors in building a reasonably large body of concurrent software, and especially working with students learning concurrent programming (≈1000 students).

SPDT'96, Philadelphia PA, USA
©1996 ACM 0-89791-846-0/96/05..$3.50

2 Related Work

Rather than attempting to list all concurrent debuggers, we have selected a few archetypical examples that cover the two main concurrent domains.

Distributed Memory: Examples of interactive source-level debugging of distributed memory applications are given by *Node Prism* [17] or *LPdbx* [18]. In these scenarios, target applications consist of multiple UNIX processes. A *slave debugger* is attached to each target process using the UNIX debugging primitives and it controls the process on behalf of a *master debugger*. Depending on the architecture of the master debugger, target processes can be controlled independently and aggregated into groups so that a single debug operation can be issued for a group of processes.

Shared Memory: Various approaches exist to build a debugger for multi-threaded shared-memory applications [6, 14]. Most are like, or built from, Gnu's Debugger, GDB [20], which provides support to debug multiple kernel threads sharing the address space of exactly one UNIX process. One prototype debugger [10] handles multiple user-level threads but only in a single UNIX process. Specifically, breakpoints may be set for individual threads, and the target application stops only if a particular thread encounters the breakpoint. However, these shared-memory debuggers allow interaction with only one thread of control at a time.

Our work concentrates on shared memory concurrency because it is a simpler domain in which to write and develop concurrent programs; in particular, μC++ is a shared memory thread library. However, much of our work is applicable to both the shared and distributed domain.

3 User-Level Threads

Currently, computer vendors providing shared-memory concurrency hardware are creating non-standard 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 the kernel threads, e.g., extensions in /proc for kernel-level threads.

However, we argue that user-level threads are essential [1], 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. Kernel threads must assume a worst case scenario for operations like context switch because there is little or no knowledge of the particular language and/or concurrency paradigm running within the address space. As more state is added to kernel threads, like UNIX signal delivery at the kernel thread level, the cost of fundamental operations only increases. Only the language system running within the address space knows what constitutes the execution state. For example, nano-threads have been suggested that are simple finite state machines with statically determinable state; it is possible to switch among nano-threads with as few as 2 or 3 register assignments. Kernel threads cannot achieve this level of performance nor should it be expected of them. Therefore, user-level threads have the potential to be significantly less expensive than kernel-level threads in many cases, because the language runtime system has specific knowledge about the concurrency paradigm and its implementation.

Given that user-level threads are important, some mechanism must exist to debug concurrent programs using them. To the best of our knowledge there are very few concurrent debuggers that work with user-level threads [9]. 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. It is the purpose of our work to show that it is possible to build very powerful and flexible debugging support for user-level threads, and the concrete demonstration system to illustrate these ideas is μC++.

4 Asynchronous Control

It is important to note that all UNIX debuggers must use the synchronous UNIX debugging primitives /proc or ptrace, making it impossible to interactively control a thread while other application threads execute; either the debugger process is active and the application process blocked or vice versa. These synchronous primitives preclude independent and asynchronous control of individual threads within a UNIX process and restrict a distributed debugger to independent control of one thread per UNIX process.

The effect this has on debugging a concurrent program is that it destroys the notion of concurrent execution. When a user is debugging a particular user-level thread, all other threads are quiesced, which violates the notion that threads execute independently. In other words, normally there is no expectation that stopping one thread should affect others unless the stopped thread is holding shared resources. While it maybe useful to stop all threads when one thread stops, this should not be the normal mode of operation nor should it be the only mode of operation. We cannot emphasize enough that the debugging model must match with the concurrency model. To force a sequential debugging model on a concurrent model is unacceptable. For example, after single stepping passed a lock release statement, other threads waiting for the lock should continue execution immediately, even though the thread being debugged is still stopped by the debugger awaiting further operations. If other threads do not make progress when the lock is released, how can a user test the concurrent nature of the program?

5 Basic Debugging

The most common errors in both sequential and concurrent programs are simple errors. The kind of errors depends on the particular programming language, e.g., uninitialized variable, invalid subscript, incorrect use of a pointer, improper expression or algorithm specification. A traditional symbolic debugger is extremely useful for dynamically tracing through a program to determine the cause of simple errors. However, most traditional debuggers fail for concurrent programs because they are unable to locate and display all the necessary context associated with concurrent execution. The problem is multiple thread stacks; a traditional debugger assumes a single stack, and therefore, cannot locate variables and routine invocations for other threads of control. Without this additional capability, locating simple errors is extremely difficult, often requiring manually tracing list structures and call frames. Independent recognition and control of each thread by the debugger allows simple errors in concurrent programs to be debugged in the usual manner.

For a concurrent debugger to know about multiple stacks, it must understand the runtime structure of the concurrent system, e.g., find all threads, and if applicable, where they are executing. Hence, the debugger needs to be kept informed about the dynamic structure of the application. This information can be displayed, in various ways, during execution to provide a simple monitoring tool that illustrates concurrent control flow. Such cheap visualization is often sufficient to identify many algorithmic errors, such as threads waiting unnecessarily or inhibiting concurrency.

Both livelock and deadlock errors can be located quickly with a concurrent debugger. If the debugger can stop, examine, and step through individual threads, it is possible to quickly locate which thread(s) are spinning and where, or by examining call stacks, which threads have formed call cycles and what resources they have allocated. It may still take some effort to reason backwards to determine why the livelock or deadlock formed, but the programmer is now reasoning with detailed information.

Our conclusion is that a concurrent debugger has the potential to deal reasonably well with a large number of sequential and concurrent errors. We have also observed that application developers, often us and especially students, do not want or have the time to deal with the complexity associated with many static and dynamic analysis tools. Nor do these tools deal with most of the errors that occur during the software development process. When judiciously used, these tools are invaluable; however, these tools alone are insufficient to debug a concurrent program because they do not address simple errors.

6 KDB

KDB (Kalli's DeBugger) [11] is a concurrent debugger running on UNIX based symmetric shared-memory multiprocessors that achieves our goal of independent control of user-level threads. The design for KDB (but not the present implementation) is applicable to both shared memory and distributed memory, and is intended to be highly efficient and to cooperate with tools that implement high-level analysis. KDB uses a large part of gdb to handle symbolic debug information and to interpret raw application data, like variable or stack contents, but gdb is not run as a separate process and never controls the execution of the target. KDB supports the μC++ execution environment, which shares all data, and has multiple kernel and user-level threads. Kernel threads provide parallelism on multiprocessors, and user threads refine that parallelism; user threads can be executed by any kernel thread. An unusual property of μC++ is that kernel threads are obtained from UNIX processes instead of using vendor specific kernel threads; a kernel thread is created by forking a UNIX process and mmaping all data to be shared with the parent process. (This approach is more portable than vendor specific kernel threads.) Unfortunately, the limitations of mmap preclude sharing the code image, which means there are multiple code images. These code images present a problem only to a debugger, which must set and reset breakpoints in each code image. Thus, KDB handles multiple code images in multiple UNIX processes, and multiple kernel and user-level threads are controlled.

The mechanism used to achieve asynchronous execution of an application and the debugger is similar to that used by distributed debuggers, but finer grain: part of the debugger, called the *local debugger*, is distributed into the target application (see Figure 1). The *global debugger* uses two different channels of communication with the application processes. The first channel is synchronous because it is implemented by UNIX debugging primitives; this channel is temporary, lasting only as long as necessary to modify an application's code, e.g., to set and reset breakpoints. The second channel is asynchronous via a socket between the global and local debugger; this channel is used to notify the global debugger about debugging events in the application, e.g., breakpoint encountered, or notify the local debugger about events generated by the user interacting with the global debugger, e.g., continue execution of a thread. Finally, the global debugger is itself a multi-threaded application, written in μC++, so its internal as well as external interactions are asynchronous, i.e., the debugger acts like a multi-threaded server for a client application. This includes communication with the X-window/Motif user interface so the debugger can continue even when there is no user interaction. (This required changes to X11R6 to work with μC++.) This structure forms the basis for KDB, providing the ability to independently control a dynamic number of UNIX processes running a dynamic number of light-weight μC++ threads and a non-blocking interactive user interface. It also means that KDB can be used to debug itself.

When a μC++ application is compiled using -debug and -g flags, the local debugger's code is linked in and symbolic debugging information is generated. During application startup, the local debugger checks for the presence of the global debugger by checking for a shell variable created by the global debugger. The μC++ runtime system cooperates during debugging by reporting certain events to the local debugger, which in turn reports them to the global debugger, if necessary.

The local debugger is both a strong and weak point of KDB. The strong point is that it provides an ideal location for generalization and optimization. The local debugger can be specific to particular kernel-threads and/or user-level concurrency library but have a standard protocol for communication with the global debugger. As well, the local debugger can perform a significant amount of work on behalf of the global debugger, e.g., looking up data/code and performing certain checks, significantly reducing the probe effect. For example, if multiple threads execute the same code image, but a breakpoint is set for only a subset of the threads, each thread triggers the breakpoint but most triggerings are inadvertent. Checking whether a breakpoint applies to a thread is done by the local debugger so no trap or kernel context switch is necessary to continue execution. Experimental results show a speedup factor of roughly 2,400 over the traditional method where this check is performed in a different address space. Finally, the local debugger can interact with other debugging tools, such as event tracers. The major weak point is that the local debugger can be corrupted by a runaway application. We believe additional redundancy can be added to recover from almost all data corruption. As well, part of the local debugger is implemented as a μC++ task to provide independent execution, which introduces a recursive dependency that precludes debugging parts of the μC++ kernel using KDB. As a result, μC++ implementers must use print statements and traditional debuggers at the lowest internal level of the μC++ kernel. While this structure for a concurrent debugger is reasonably intuitive [19], we know of no debugger that provides completely independent access to and manipulation of shared-memory user-level threads executed cooperatively by a group of UNIX processes.

7 KDB Functionality

Most user interactions occur through the two windows shown in Figure 2.

Main Window: When the debugger starts, the main window appears and has 2 panes (see Figure 2 (a)). The top pane contains dynamic tables of all tasks, clusters and processors (UNIX processes) currently active in the application. (Clusters and processors are artifacts of μC++ and not discussed here.) When connected to the debugger, the default action is for a task to block after creation, unless the Stop Tasks button is toggled. It is possible to select one or more tasks in the task list by clicking on a task and dragging to select a group (several Philosopher tasks are currently selected); groups provide a mechanism to control a number of tasks in some related way (see Section 8). The bottom pane contains controls for global debugger interactions. Most of the operations at this level manage or query task groups. Individual task information and control is provided through a task window (discussed next), which is created by selecting a task(s) in the main window and clicking on Inspect, or implicitly pops up when a breakpoint is encountered for a task. The group operation, Continue, is directly available to continue execution of all tasks of a selected group. All other group operations are accessed through the popup group windows available from buttons Operational Group and Behavioural Group. The Command area is for typing in the following commands:

- print *expression*: print global expression values.
- attach *executable-file process-id*: attach KDB to the executable that is already running.

Where appropriate, command results appear in the bottom of the pane. If the Pretty Print button is toggled, complex data is shown in an easier to read structured way but results in longer output. Each output in the bottom pane is numbered on the left and separated with a row of dashes, so a history of all output can be scrolled through using the scroll bar on the right of the bottom pane. Finally, the inactive button, New Target, is used to connect another program to the debugger when debugging of the current program is complete, which allows consecutive debugging of programs without restarting the debugger.

Task Window: A task window has 2 panes (see Figure 2 (b)). The top pane displays information about source code.

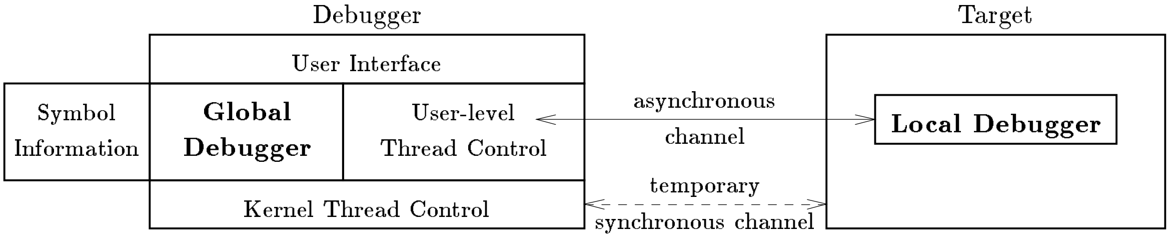


Figure 1: Debugger Design

The highlighted line marks the current statement being executed when stepping through a program, or by clicking on a line, that line becomes the operand for a command like setting or clearing a breakpoint. A task window begins by showing the last known position of the corresponding task by giving the source file name and line number (top right of the top pane), and displaying the source code (bottom of the pane) with the particular line highlighted. Button Stopped Position returns to the last known execution position from the current selection. Button Source Files pops up a list of all source files for the program, which can be selected for display in the top pane. The bottom pane contains controls for sequential debugging of a thread; these buttons are largely self-explanatory. The Breakpoints button pops up a list of all breakpoints with additional controls for enabling, disabling or removing them. The up and down arrows move up and down a task's stack frame, respectively, which also causes the appropriate source code for the stack frame to be displayed in the top pane. The Command area is for typing in the following commands, print, break and clear, that are evaluated in the scope of the current stack location. Where appropriate, results are displayed in the bottom of the pane. The Stop button is inactive when a task is stopped, and all buttons, except Stop and Backtrace, are inactive when a task is running. Pressing BackTrace for a running task prints a snapshot of a task's stack frame at the bottom of the pane, which is useful for monitoring execution, but it is not possible to move up and down this stack frame because it is changing dynamically as the task executes.

8 Thread Groups

Different kinds of possibly overlapping groups can be formed from selected tasks. However, grouping tasks together does not affect the ability to control tasks separately; furthermore, task reactions to the commands issued to a group, such as setting a breakpoint or continuing execution, become visible in each task's window.

Operational Group: Tasks can be grouped together and operations issued on the group of tasks as a user convenience, rather than entering multiple commands for each task (as in *Node Prism* [17]). Multiple operational group windows can be created, each defining a different group. A task may appear in any number of operational groups. When an operational group window is closed, that group is terminated.

Clicking on Operational Group forms a group of all tasks currently selected in the main window (see Figure 2 (a)), and pops up a window (see Figure 3 (a)) where commands can be issued on all tasks that belong to the group. The following commands, break, clear, stop, cont, next, step, can be entered in this window and the corresponding operation is performed on each task in the group. In general, if a command is not applicable to one of the tasks, e.g., stop for an already stopped task, the command is silently ignored. The upper right area of the window shows a history of commands entered for this operational group. Figure 3 (a) shows an operational group window where a breakpoint was previously set for each philosopher task (upper right), and each task is about to be continued.

Behavioural Group: A behavioural group is a set of tasks whose behaviour is linked to some event. In other words, 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. Furthermore, a task can appear in only one behavioural group at a time because actions in one group might cause inconsistent behaviour in another group.

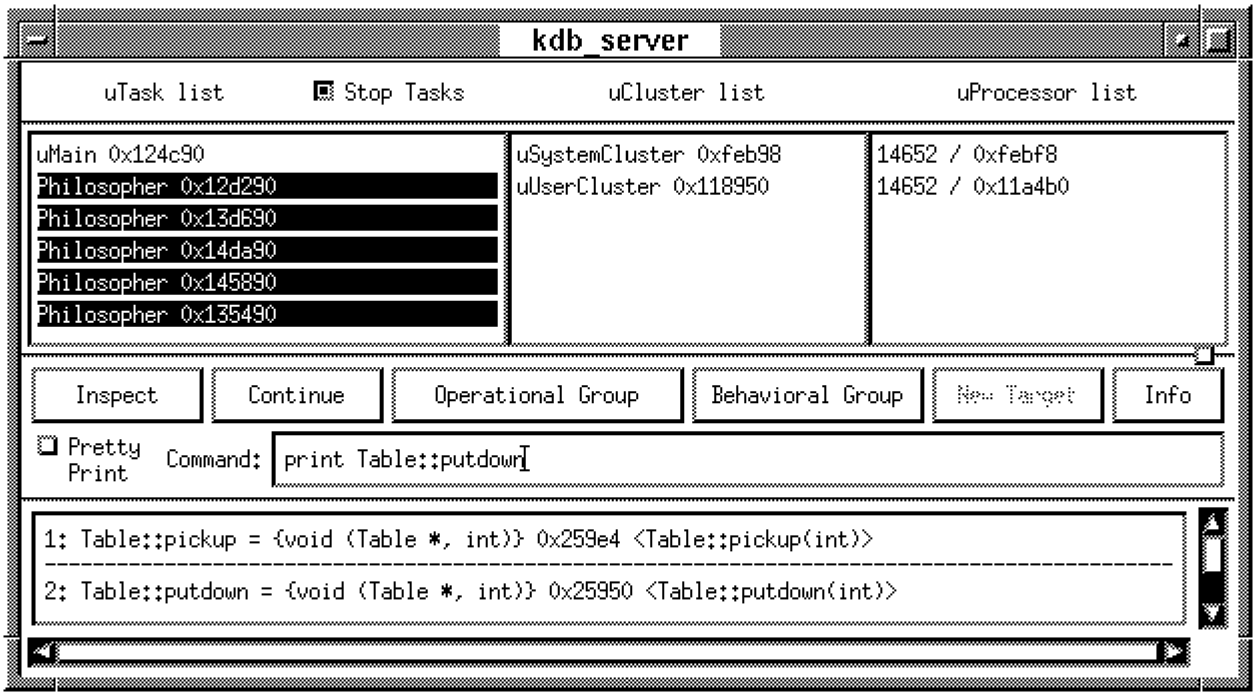
Clicking on Behavioural Group forms a group of all tasks currently selected in the main window (see Figure 2 (a)), and pops up a window where an event and action can be issued for all tasks that belong to the group (see Figure 3 (b)). Currently, the only event command is break, and the only operation command is stop. Figure 3 (b) shows a behavioural group window where a breakpoint is about to be set for each philosopher task, and each task in the group will be stopped when one of the tasks reaches the breakpoint.

9 Debugging with KDB

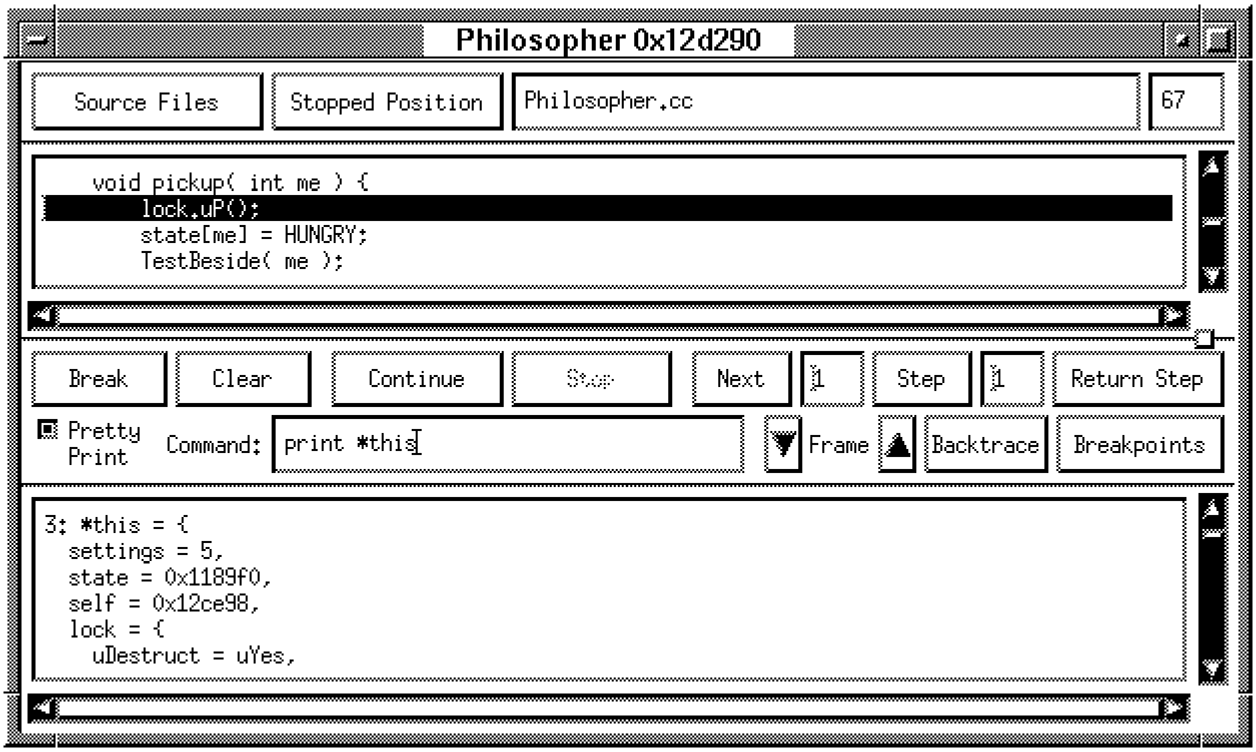
This section explains how the different features of KDB can be used to debug a concurrent μC++ application.

Most importantly, simple errors in tasks are handled by traditional debugging methods applied at the user thread level. The ability to individually control and examine each user-level thread through a task window allows the programmer to precisely follow control flow and examine execution state. A user is never in a situation of having information about only one task at a time; all tasks can be queried at all times. When an application fails, e.g., segmentation fault, the local debugger catches the signal and informs the global debugger, which stops all other UNIX processes running light-weight threads as quickly at possible. The user can then examine all threads to investigate the error; however, no further execution is possible. Nevertheless, a programmer can examine all the existing execution states in a symbolic manner.

The top pane in the main window and individual task windows present a cheap dynamic visualization of the concurrency in an application, making it easy to spot major algorithmic errors. Simply watching the order that tasks are created and/or destroyed in the main window can be extremely informative. When a failure occurs, knowing which tasks have completed and which still exist conveys substantial information.

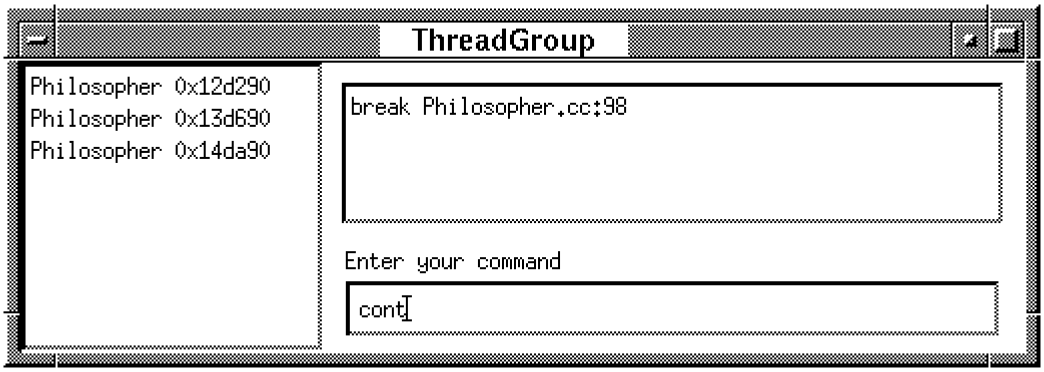


(a) Main Window

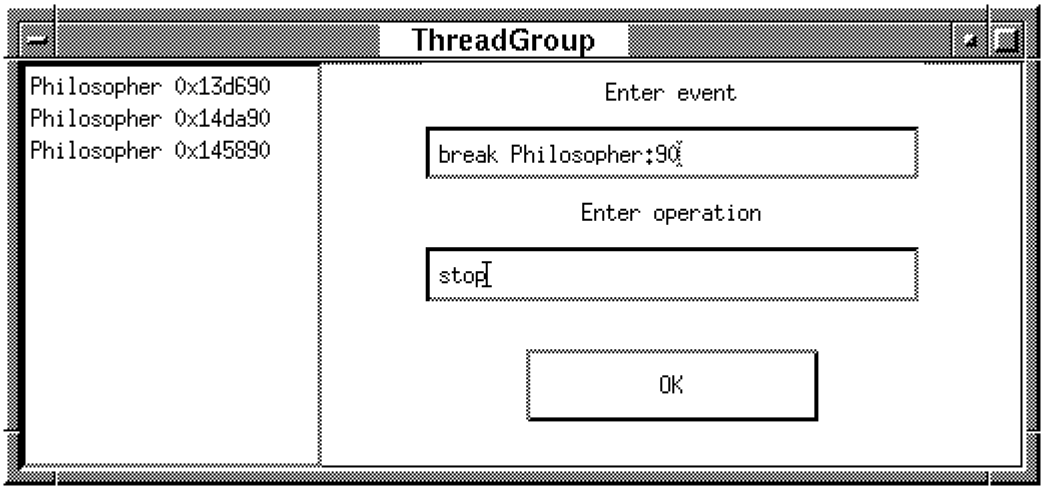


(b) Task Window

Figure 2: Debugger Interface



(a) Operational Group Window



(b) Behavioural Group Window

Figure 3: Thread Groups

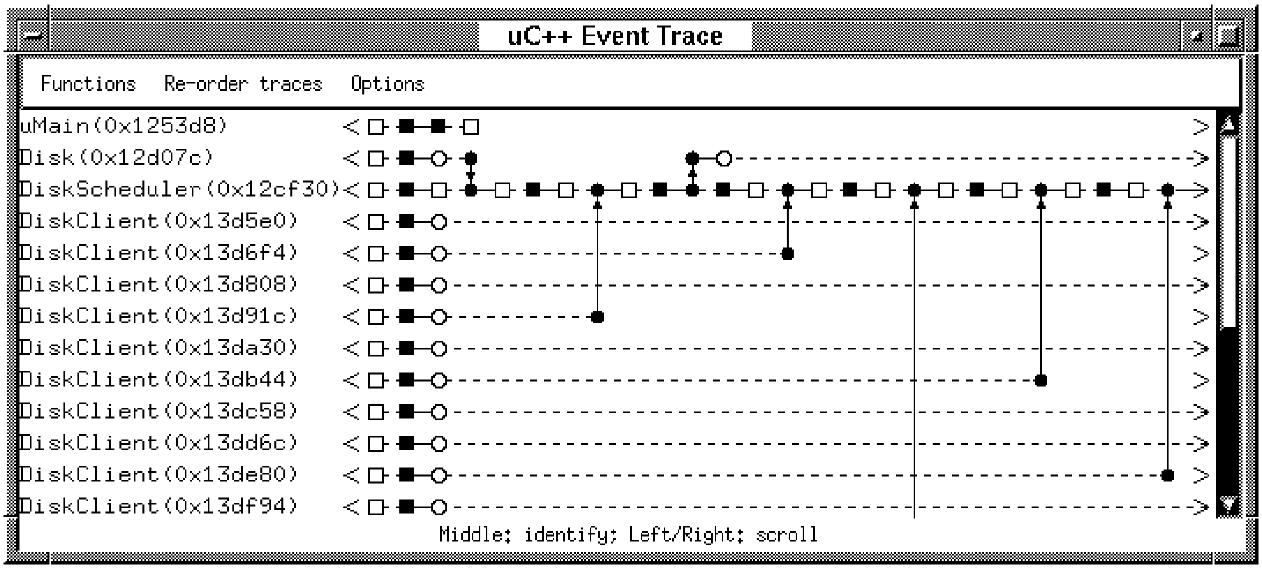


Figure 4: Event Tracing

The ability to operate on groups of tasks simplifies working with large numbers of tasks. While an operational group is merely a convenience capability, its usage cannot be underestimated when working with non-trivial numbers of tasks (e.g., tens or hundreds of tasks). Starting and stopping a group of related tasks can significantly simplify certain debugging operations. Behavioural groups are more powerful. For example, a behavioural group can provide the capability of debuggers that do not provide independent execution of threads; these debuggers stop all threads when one thread encounters a breakpoint. This capability is accomplished by forming a behavioural group of all tasks, setting the event to break and the operation to stop.

Livelock can be discovered by stopping tasks and stepping through them to determine if a task is spinning. If a deadlock occurs, tasks are still running from the debugger's perspective, but blocked from the thread system's perspective. Double clicking on the Backtrace button for a task enables the stack arrows, making it possible to move up and down a running task's stack frame. *This operation is safe only when a task is application blocked.* In this way, it is possible to find call cycles that lead to the deadlock.

10 Fast Breakpoints

Fast breakpoints are essential to provide adequate performance with multiple tasks sharing code images and are implemented using the following fast breakpoint scheme (similar to [12]). The maximum number of breakpoints, N, is preset when μC++ is compiled (default is 64). Each task has an N bit mask, and N handler routines are generated in the local debugger, each containing some nop instructions to reserve space. The global debugger cycles through the N handler routines as breakpoints are set and released. A breakpoint is implemented by selecting a handler routine, M, turning on the Mth bit in a task's mask, copying the instructions (with possible modifications for relative addressing) from the breakpoint into the handler, and inserting a call to the handler at the breakpoint. When handler M is called by a task, it checks if bit M is set in the task's mask. If the breakpoint is not applicable, the original instructions are executed at their temporary location in the handler and control transfers back to the breakpoint (approximately 25 instructions). If the breakpoint is applicable, the local debugger is called, which blocks the task and informs the global debugger through the asynchronous channel. This mechanism places some restrictions on where breakpoints can be set because some architectures restrict where calls may occur, but it is not a practical problem when debugging at the statement level. When a task is continued by the user, the global debugger informs the local debugger, and the local debugger reschedules the task. Notice that the local debugger must have knowledge of the user-level runtime system to manipulate tasks and runtime data structures, and therefore, is specific to that particular user-level thread system. Finally, implementing and removing a breakpoint for a task must be done in the code image of each UNIX process eligible to execute the task (see Section 6).

In addition to fast breakpoints, we have implemented restricted fast conditional breakpoints, e.g.,

```
break Philosopher.cc:98 if i == 10
```

The conditional breakpoint is "fast" because it is evaluated by the local debugger during breakpoint evaluation, rather than the global debugger. The conditional breakpoint is restricted because the form of the conditional is only:

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

because there is no way to dynamically compile and insert code into the application as commands are given to the global debugger. Therefore, all possible combinations of operand types and operators must be pre-compiled into the local debugger, and the appropriate version selected dynamically during breakpoint evaluation. Even with this restriction, the conditional breakpoint is invaluable in controlling loop execution by placing conditionals on the loop index. We plan to add pointer data types, because many architectures now have different sized pointers and integers, and possibly floating point types. This small set of data types should handle 95% of all conditional breakpoint needs, while providing extremely fast conditional breakpoint execution.

One interesting problem in implementing fast breakpoints is that most RISC architectures require replacing multiple instructions at a breakpoint site (a call and a nop, because of the delay slot). This requirement results in a race condition when setting or resetting a breakpoint as a task may have executed at least one of the original instructions at the breakpoint site but become "ready" because of a time slice. When the task runs again, it executes incorrect code at the breakpoint. The local debugger deals with this problem by checking the execution location of all tasks before allowing the global debugger to set a breakpoint. If a task is ready but executing at the breakpoint location, the global debugger retries setting the breakpoint after a short delay. Since the check is one compare and done locally, the cost is very low, even for a large number of tasks. We are willing to pay a higher cost to set and reset a breakpoint, which is normally done as part of a slow user interaction, to obtain a very fast breakpoint applicability check.

11 Future Work

As mentioned in the introduction, a complete debugging environment must provide additional tools for event collection, analysis and replay. We are aware of this requirement and have begun to work on this issue. μC++ programs can already be compiled with event tracing for all accesses to coroutines, monitors, and tasks [2]. The events are currently displayed by a powerful event tracing viewer [21] (see Figure 4). The next step is to connect KDB, the local debugger, and the event tracer to provide debugging support for deadlock and race conditions. KDB will be extended with a programmatic interface, so that its functionality can be accessed by commands sent through a socket instead of through the user interface. Then the event tracer can use KDB to control individual threads to force execution to follow a previous event trace, and hence, produce replay of previous executions [15].

We are still learning about behavioural groups. What events other than break might provide useful capabilities for debugging? For example, event stop and operation stop, might stop all tasks in the group when stop is pressed by the user for any task in the group. Finally, we need to make the local debugger's data more robust from inadvertent memory assignments. One possible solution is to write protect the page containing the local debugger's data, except when the local debugger needs to make changes.

12 Conclusion

While deadlock and race conditions are the bane of all concurrent programs, they represent only a small portion of the total errors. It is crucial to provide an environment that allows, at the very least, the ability to get to the complex errors. Many concurrent programmers are thwarted by simple errors long before they reach the difficult ones, making the presence of sophisticated concurrency tools moot.

We argue that a concurrent debugger, with adequate knowledge of the concurrent runtime system, should be the focal point to support programmers through initial and intermediate errors, and provide an engine for supporting other analysis tools (e.g., like *Dynascope* [19]). We have shown that such a debugger can be built for user-level threads, which we believe will become the most common form for providing concurrency to application developers in the future. As well, a user-level thread debugger can provide independent control of user-level threads, which preserves the concurrent execution model. Preserving this model is essential or the debugging environment does not reflect the normal execution environment. Finally, it is possible to move all user-thread specific code into the local debugger, making the largest debugger component, the global debugger, user-thread independent but still kernel-thread dependent.

References

[1] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism". *ACM Trans. Comput. Syst.*, 10(1):53–79, Feb. 1992.

[2] Buhr, P. A., Coffin, M. H., Jacobs, R. A., Larson, J., and Zinn, R. S. "Concurrent Monitoring, Visualization and Debugging". In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 178–180, San Diego, California, May 1993. [Extended abstract].

[3] Buhr, P. A. and Goel, A. K. "μDatabase Annotated Reference Manual, Version 1.0". Technical Report Unnumbered: Available via ftp from plg.uwaterloo.ca in pub/uDatabase/uDatabase.ps.gz, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, June 1995.

[4] Buhr, P. A. and Karsten, M. "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, Sept. 1995.

[5] Buhr, P. A. and Stroobosscher, R. A. "μC++ Annotated Reference Manual, Version 4.4". 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, Sept. 1995.

[6] Caswell, D. and Black, D. "Implementing a Mach Debugger for Multithreaded Applications". Technical Report CMU-CS-89-154, CMU, Nov. 1989.

[7] Damodaran-Kamal, S. K. and Francioni, J. M. "Nondeterminacy: Testing and Debugging in Message Passing Parallel Programs". In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 118–128, San Diego, California, May 1993.

[8] Emrath, P. A., Ghosh, S., and Padua, D. A. "Detecting Nondeterminacy in Parallel Programs". *IEEE Software*, 7(1):69–77, January 1992.

[9] IBM Canada Ltd. Laboratory, Information Development, 844 Don Mills Road, North York, ONT, Canada. M3C 1V7. *OS/2 Developer C/C++ Toolkit.*

[10] Jacobs, R. A. "A Debugger for Multi-Threaded Applications". Master's thesis, The University of Waterloo, July 1995.

[11] Karsten, M. "A Multi-Threaded Debugger for Multi-Threaded Applications". Diplomarbeit, Universität Mannheim, Mannheim, Deutschland, Aug. 1995.

[12] Kessler, P. B. "Fast Breakpoints: Design and Implementation". In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation, published in ACM SIGPLAN Notices*, volume 25, pages 78–84, June 1990.

[13] Masticola, S. P. and Ryder, B. G. "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.

[14] Nasser, F. and Stumm, M. "Extending gdb for Functional and Performance Debugging of Parallel Programs on Shared-Memory Multiprocessors". In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 214–216, San Diego, California, May 1993. [Extended abstract].

[15] Netzer, R. H. B. "Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs". In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 1–11, San Diego, California, May 1993.

[16] Netzer, R. H. B. and Miller, B. P. "What are Race Conditions? Some Issues and Formalizations". *ACM Letters on Programming Languages and Systems*, 1:74–88, March 1992.

[17] Sistare, S., Allen, D., Bowker, R., Jourdenais, K., Simons, J., and Title, R. "A Scalabe Debugger for Massively Parallel Message-Passing Programs". *IEEE Parallel & Distributed Technology*, 1(2):50–56, Summer 1994.

[18] Sorel, P. E., Fernandez, M., and Gosh, S. "A Dynamic Debugger for Asynchronous Distributed Algorithms". *IEEE Software*, 11(1):69–76, January 1994.

[19] Sosič, R. "The Dynascope Directing Server: Design and Implementation". *Computing Systems*, 8(2):107–134, Spring 1995.

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

[21] Taylor, D. "A Prototype Debugger for Hermes". In *Proceedings of the 1992 CAS Conference*, volume 1, pages 29–42, Toronto, Ont., Canada, November 1992. IBM Canada Ltd. Laboratory, Centre for Advanced Studies.