Vol. 40 No. 10 Transactions of Information Processing Society of Japan Oct. 1999

Regular Paper

Low-intrusion Cooperative Debugger for Multi-threaded Distributed Programs

NORIO SATO,[†,☆] DAG H. WANVIK,[††] HARALD BOTNEVIK,[††] TROND BØRSTING[††] and JON E. STRØMME[††]

A new debugger has been developed which reduces intrusion into the normal execution of a program under test (PUT), can be connected to one or more PUTs at the same time, and enables cooperative debugging. These novel features improve the productivity and quality of large, real-time, multi-threaded, distributed applications. The basic idea is to place a debug client on the host and debug servers on the target computers, then couple them asynchronously. The debug client can create sessions that are thread instances of command script interpreters. This makes it possible to debug more than one thread and more than one PUT at the same time, and to trace them in parallel (e.g., one session for each thread or PUT), or in combination (e.g., one session for several threads and/or PUTs). The debug client allows distributed co-clients to share its sessions, thus allowing programmer cooperation. The debug servers are dedicated threads running inside the PUTs (at low priority). They perform the commands coming from the debug client. Conditional tracing and breakpoints are evaluated by the servers, eliminating communication with the client. Such event filters enable tracing, breaking, and stepping of individual threads with a low level of intrusion into the PUT execution. The debug client can handle multiple source languages and different target processors at the same time. While the first versions of the client and debug server were developed to test PUTs written in CHILL running on a specific user-level thread library, later versions include support for PUTs written in C and C++ running on this library. By adapting similar debug servers for other runtime environments, the paradigm is applicable to testing of a wider class of real-time, multi-threaded, distributed applications.

1. Introduction

Real-Time Concurrent Programs and Testing

Telecommunication control processing, such as switching, requires large real-time concurrent programs to be embedded in target systems. Different parts of such programs are coded by different groups of people often in different programming languages. Telephone switching is a performance-critical application that can be categorized as having *medium real-time* requirements—for example, up to one million busy-hour call attempts (BHCA) must be handled with a dial-tone delay of not more than a couple of seconds—as well as a few *hard real-time*[4] requirements. Systems running these applications thus execute many threads concurrently inside a program; these threads share code. The programs are distributed among different nodes in the network and communicate with each other, via a signaling network within timing constraints.

Because the testing and debugging phases in such programs often account for 50 to 70 per cent of the total development effort[4], debugging tools are critical to reducing costs and saving time. Without them, debugging involves the tedious analysis of log dumps, static memory dumps, and memory-stack dumps.

Limitations of Traditional Interactive Symbolic Debuggers

Debuggers for testing UNIX processes, such as GDB[1] and DBX[3], provide a user interface at the source-language level, that is, a symbolic interface, and commands for tracing the sequential execution of the program under test (PUT)[☆☆]. The commands are executed by using operating system facilities such as *ptrace*[5]. Although DBX has been enhanced to support multi-threading, namely, the inspection and single-stepping of individual threads, it is still based on the traditional paradigm of completely stopping the PUT whenever an event occurs, which is the so-called "stop-the-world" model.

Some of the cross-debuggers used for embed-

† NTT Optical Network Systems Laboratories
†† Kvatro Telecom AS
☆ Presently with Kanazawa Institute of Technology

☆☆ In this paper, we define a "process" as a "program"; otherwise we use generally accepted terminology. In CHILL[15], a "thread" is called a "process", but "thread" is used here to avoid confusion.

ded programs provide a similar symbolic interface. They are combined with an in-circuit emulator (ICE) and connected to target processors to monitor and control the hardware registers. By downloading the executable files to the target processors or by sharing files between the host and target computers, sequential debugging similar to that for UNIX can be achieved.

As the clock frequencies in target processors become faster, an ICE cannot always be used; instead, built-in facilities that communicate with the host computers via a network are needed. This is partly solved by embedding read-only-memory code inside the target processor. Still, the traditional debuggers are unable to recognize the target operating system. To them, the code residing in the target memory is simply seen as a collection of machine instructions.

Real-Time Monitoring

Programs can sometimes run even with errors, particularly errors related to real-time processing and concurrency. However, these errors can seriously degrade performance, cause frequent timeouts, and block resources.

Without real-time monitoring, "post-mortem" analysis of the relevant log information is the only way to locate a problem. This requires the insertion of "instrumentation code" here and there in the application code or in the operating system (or thread library). To plug this code in and out of a program requires recompilation, relinking, reloading, and rerunning, which are inflexible and time-consuming.

Our Approach

The debugger presented in this paper was initially developed for CHILL systems[15),16)], but its basic mechanisms have turned out to be independent of the source language. Our debugger covers the paradigm of communicating threads and programs in general, and has been extended to handle programs written in other languages, such as C and C++, given thread libraries.

2. Requirements

2.1 Largely Unintrusive Monitoring and Debugging

Traditional debuggers are *highly intrusive.* Because they are *synchronously coupled* with the PUT, the execution of PUT is suspended each time a debug operation is performed. Therefore, they cannot support the debugging of real-time concurrent programs, which requires the following facilities:

Largely unintrusive break filtering: Because threads run by sharing code, the break events hit by only one thread should be filtered out so that they do not significantly affect the execution of other threads. The low intrusiveness requires an implementation technique such that this filtering is done inside the PUT (without stopping it).

Largely unintrusive stepping: Real-time debugging necessitates stepping the execution of one thread while executing the other threads normally so as to avoid unneeded timeouts and maintain the load, which may indirectly play a role in the problem being investigated. This requires an implementation technique such that the single-stepping of one thread does not significantly decrease the speed of other threads sharing the same code.

Largely unintrusive real-time monitoring: Both OS-level and user-level *instrumentation code* is required inside the PUT. Being able to plug this code in and out during debugging rather than during compilation would reduce monitoring intrusion into the performance in normal execution time, as well as providing more flexibility.

2.2 Concurrent Debugging of Multiple PUTs

Traditional debuggers cannot be connected to more than one PUT at the same time, so they are not well suited to debugging multiple process applications. Using two or more debuggers for such applications is unattractive, because it is almost impossible to synchronize the related events caught by the different debuggers within reasonable real-time constraints. Using one debugger would allow the coupling of two or more debugger functions in command scripts.

2.3 Multiple Language Support

Because the PUT code may be written in different languages, one debugger should be able to analyze the debug information for multiple languages. In our setting, the PUTs are to be written mainly in CHILL and partly in other languages such as C and C++.

2.4 Debugging of PUTs on Multiple Target Processors

It is sometimes necessary to debug PUTs running on different target processors at the same time. Traditional debuggers cannot do this because they include many operations that depend on the target processor and kernel.

2.5 Cooperative Debugging

Different parts of the PUT code may have been written by different teams at geographically separated sites. The members of such teams should be able to work together as a group in the same debugging session.

3. Proposed Debugger

3.1 Fundamental Features

Splitting of the debugger into client and server: As shown in **Fig. 1**, we have split the debugger functions into those for the *debug client* and those for the *debug servers*, enabling them to work concurrently and to be distributable. The client interacts with the users and interprets the debug commands. The servers filter out the break events without waiting for communication with the client. The client can be connected to one or more servers at the same time. This splitting also liberates the client from target processor dependency to a great extent, facilitating its connection to PUTs on different target processors.

Multi-threaded debug client: We made the debug client multi-threaded, thereby hiding the communication latency caused by waiting for thread events coming from many PUTs, the interaction with the user, and the symbol information loading (for the effect of latency hiding by threads, see for example, Ref. 9)). The command interpreter is the central part of the debug client. As shown in **Fig. 2**, it is multi-threaded and instantiated. The thread instances provide *multiple sessions*. Each session offers a user interface consisting of input, output, and source-code sub-windows. A session is thus realized as a thread that interprets a sequence of command scripts.

This makes it possible to observe and control different parts (i.e., different threads or related PUTs) concurrently. In the debug client, the sessions share *events*, *debugger variables*, and PUT information, so they can easily cooperate.

Distribution of debugger clients: We extended the communication with the user so that the debug client can allow *co-debugger clients* (running as processes on possibly different workstations) to share the session. This allows several geographically separated team members to participate in the same debugging session.

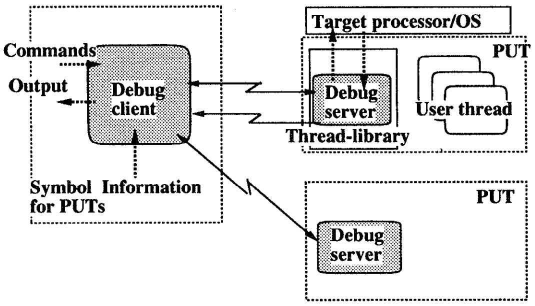


Fig. 1 Splitting of the debugger into client and server.

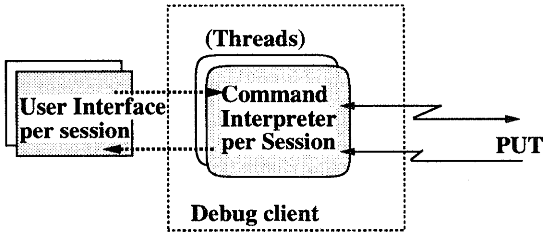


Fig. 2 Enabling multiple debugging sessions by using multi-threading.

Extension of symbol information and permissibility of incremental loading: Because symbol information used for debugging is bound to individual tool-chains☆, we designed a new debug symbol format, the *Debug Information Language* (DIL)☆☆. This language is represented in LISP-like character-string format, and is therefore extensible and portable. It allows for incremental loading of individual modules, so the startup time is very short and independent of program size. This makes it possible to debug very large systems, such as telephone-call-processing programs, which can run into millions lines of code. DIL can be generated by compilers (for CHILL), or extracted from the executable files (for C and C++).

We call the debug client the *Pilot*, the co-debugger clients the *Co-pilots*, and the debug server the *Program Execution Control Server* (PXCS). The overall structure is shown in **Fig. 3**. A Co-pilot is a remote process that exchanges session commands and session output with a session thread in the Pilot.

☆ A "tool-chain" means a set of tools, such as a compiler, linkage editor, loader, and debugger.

☆☆ We call this new symbol format a "language", since it is readable in ASCII. But it has no dynamic semantics, so it can also be called a "format".

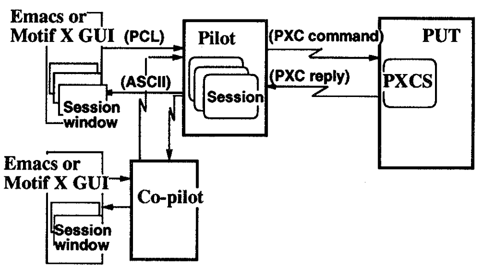


Fig. 3 Overall structure of the Pilot and the PXCS.

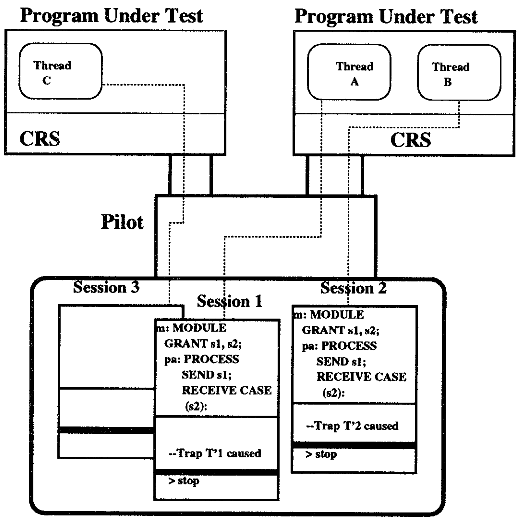


Fig. 4 User interface, Pilot sessions, and PUT.

Figure 4 shows an intuitive view of the user interfaces, sessions, and program(s) under test. The Pilot sessions are debugger threads which shadow PUT threads.

3.2 Pilot Command Language (PCL) and Interpretation

As shown in **Fig. 5**, the *Pilot command language* (PCL) is a general scripting language. As opposed to commands in traditional debuggers, PCL is programmable. The programmability is useful for producing "regression test scripts", especially since PCL reflects source code syntax, by allowing both CHILL and C/C++ expressions, statements, and control structures to be used, facilitating easy cutting and pasting from program source code into a debugging session. The PCL allows for full access to the address space of the connected PUTs, and to the *debugger variables*☆ that are pre-defined or created in Pilot sessions.

```
 1 > -- Session 1 --
 2 >CONNECT 'my_clock@target.net';
   =>Program connected x'1
 3 >$echo_commands:=TRUE;
 4 -- Change context (CC).
   >CC 'clock_m|write_seconds';
   => the source code text appears in the source
   window, procedure "write_seconds" in
   module "clock_m" is highlighted.
 5 >clear_screen(); --Call procedure.
   => procedure "clear_screen" in program x'1
   is called.
 6 --Start a new session (see line 19-21)
   > START SESSION;
   => New session started s'2.
 7 -- Set a break trap denoted as "t'4" or "s'1|t'4"
   >TRAP (write_seconds)WHEN(between)
   >END;
   => Trap created t'4, disabled.
 8 >CC between; -- Go to filter procedure.
   => the source code for procedure "between" appears.
   => low_limit and high_limit below are PUT variables
 9 >low_limit := 30; -- Set up filter.
   => Variable low_limit is assigned by 30.
10 >high_limit := 40; -- Set up filter.
   => Variable low_limit is assigned by 40.
11 >ENABLE t'4;
   => Trap t'4 enabled.
   => Trap hit t'4, by thread i'10.
12 >RECEIVE t'4; -- Synchronize, possible wait
   => The breaked line appears in the source window
13 >STEP;
14 >STEP INTO;
15 >DISABLE t'4; --Disable two traps.
   => Trap disabled t'4,
16 >DELETE t'4
   =>Trap deleted t'4
17 >RESUME; -- Resume thread suspended;
   =>Thread i'10 resumed.
18 >DISCONNECT x'1;
   =>Program x'1 disconnected.
   ...
19 > -- Session 2 --
20 > $cl:=CONNECT 'cradle_clock';
   =>Program connected x'2.
21 >CC main; -- Set context to C++ function.
   => The source code for main appears.
```

Fig. 5 Example of PCL (Pilot command language) script.

Connecting and disconnecting: Pilot can be connected to and disconnected from running programs, without disturbing their execution (lines 2, 18, and 20).

Monitoring and changing PUT states: The Pilot can monitor threads (e.g., "cc i'10;" to set the session context to thread i'10) and change their running states, (e.g., "suspend i'10;" "resume i'10;"), setting breaks and causing suspension (lines 7) or resumption (line 15). It can change the state of the

☆ Debugger variables are denoted as $*idenfier*. It is recommended that they be used rather than *debugger literals* such as x'1 (a program number), t'1 (trap number), i'10 (thread number) to write reusable regression-test scripts.

PUT as a whole☆.

Creating sessions: A Pilot user can create concurrent sessions (line 6), enabling multiple views of the PUT(s). In this example, the PCL scripts on lines 6–18 and 19–21 are input and interpreted concurrently in different sessions.

Setting context: Each session has a *session context* indicating the location of the PUT. A context is specified as a set of objects of the PUT, such as a program, source modules or source files, line numbers, threads, procedures, blocks, and even an incarnation level of recursive procedures. Typically, users will specify a context explicitly to bring up the source code in the window in order to set breakpoints or to investigate variables (lines 4, 8, and 21). In most cases, however, the context is set automatically, for example, when the interpretation must be synchronized with some event coming from the PUT(s), such as *receive* (lines 12) or *step* (lines 13 and 14).

Accessing PUT variables: As in traditional debuggers, the PUT variables can be read or updated (lines 9 and 10). Any active local variables can be accessed, whether the thread is running or suspended.

Stepping: Stepping is either for a single thread (lines 13 and 14) or the PUT as a whole, depending on the state of the PUT.

Traps: A trap (line 7) is a "breakpoint" with a difference, namely, the ability to specify a *filter procedure* (e.g., the procedure name "between" in line 7), a *trace data supplier procedure*, and *PCL actions.* The first two are called inside the PUT without waiting for communication with the Pilot (for more specifics, see Section 4.3.3). The last is interpreted in the Pilot if the filter procedure returns "true".

A Co-pilot has a similar interface; it enables other users, possibly working on different workstations, to join the debug session. Once connected to the Pilot, they can watch the work being done, invoke commands, or split off into different sessions.

In the Pilot, more than one event may arrive from the PUT(s) concurrently with the PCL interpretation. Instead of disturbing the interpretation, these events are "queued" inside the Pilot. This situation never occurs in the traditional "synchronous-break" paradigm. The *receive* picks up one of the events: the one arriving first, one hit by a specified trap (line 12), or one hit by a specified thread. The completion of the *receive* causes the proper setting of the session context, which is indicated to the user by highlighting of the related source code.

☆ "RELOAD $c1;" will cause the reloading of program x'2 to be in the program suspended state. "RESUME ($this);" will cause x'2 to run.

4. Implementation

4.1 Communication Infrastructure

The Pilot and PXCS communicate with each other by using the distributed architecture, CDPS (CHIPSY☆☆ Distributed Processing System)[17], developed for CHILL programs. This architecture is similar to that for RPC[7] and CORBA[21], that is, binding by a name service creates *proxy threads* on both sides, and asynchronous messages can be passed to and received from these proxies. The underlying protocol is scalable. Inter-process communication[8] is used among processes within one computer, and TCP/IP and so on are used across computers.

The CDPS architecture differs from the RPC and CORBA architectures in several ways: The CDPS model is based on communicating threads, not objects. The communication process is fully transparent and seamless at the application level. It uses *CHILL signals*☆☆☆ that convey a list of (data) typed values from one thread to another asynchronously; these signals are selectively received[15]. CHILL signals are suitable for spontaneous message passing in both directions, in contrast to the query/answer-type messages of RPC.

4.2 Multi-threading in the Pilot

As shown in **Fig. 6**, the Pilot is multi-threaded. The threads are for user interaction, PCL interpretation, symbol information (DIL) analysis, and communication with the PUT(s). These threads are bound to synchronous I/O, but work concurrently.

A pair of parser and interpreter threads are created for each session. This enables the concurrent execution of sessions. Each PCL command is parsed into an internal tree (IT). Inside the Pilot, the messages carry a pointer indicat-

☆☆ CHILL Integrated Programming System developed by Kvatro Telecom AS.

☆☆☆ In CHILL, messages are called "signals", so "CHILL signals" means "messages" passed from a thread to another thread.

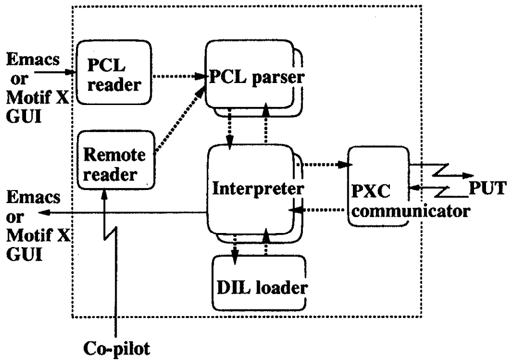


Fig. 6 Threads in the Pilot.

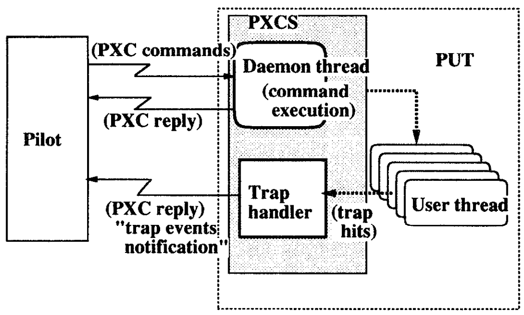

PXCS
(PXC commands)
Daemon thread (command execution)
PUT
Pilot
(PXC reply)
Trap handler
(trap hits)
User thread
(PXC reply) "trap events notification"

Fig. 7 Overall structure of the PXC Server (PXCS).

ing IT nodes, so message passing is very quick.

The evaluation of IT nodes requires fetching of the values that reside in the PUT. This may cause considerable latency, so the evaluation of an IT is performed bottom-up in parallel: for example, for "*pointer*$->$*.field(index)*"☆, the values of "*pointer*" and "*index*" are fetched in parallel.

4.3 PXC Server (PXCS) Implementation

4.3.1 Interaction with the Pilot

One generic Pilot runs on the host computers, while the PXCS is target-OS-specific. In CHIPSY, the PXCS is embedded in a thread library denoted *CHIPSY Real-Time Operating System* (CRS)[17]. As depicted in **Fig. 7**, it consists of a daemon thread and a trap handler. The former executes the commands coming from the Pilot, while the latter detects the trap hits in the PUT.

The Pilot affects the PUT by sending *PXC commands* that execute PCL at the PXC level. These commands are for *reading* or *writing* variables, setting *break-points*, *stepping*, and so on. Conversely, by sending *PXC replies*, the PXCS indicates the completion of a command and reports events that have occurred inside the PUT.

4.3.2 Session Context and PXC Commands

In the Pilot, DIL is used to convert the session context to the place where PXC commmands are applied and to convert from PXC replies to the session context. From the PXC commands, the PXCS learns of the place, such as the thread and absolute address (or stack frame and offset). Conversely, the Pilot sets the session context correctly based on the thread indication and program counter in the PXC replies, using the instruction ranges contained in DIL.

For example, a break-hit report from the PXCS to the Pilot identifies the thread that hit the break and the *program counter* of the break. For PCL *backtrace*, *up*, and *down* (stack-frame) commands, the Pilot requires PXCS to provide with sequence of return addresses of the dynamically nested procedures stored in the stack frames of the thread in question.

Stack variables can be accessed as follows: First, the PXCS must know the stack frame structure, such as dynamic links (i.e., saved "frame pointers")☆☆. Next, the life-time of the stack variables must be known. For this purpose, the Pilot sends the thread indication, frame number, and instruction range of the block where the variable is defined. If the thread is not *extinct* (*terminated*) and its current program counter is within the specified range, the variable is accessible.

4.3.3 Trap Handling in PXCS

We attached additional actions to a breakpoint to serve as *trap actions*. They are *filter procedure*, *trace data supplier procedure*, and *PCL sequence*, as shown in **Fig. 8**. The first two are user-specific procedures called inside the PUT so as to reduce the intrusiveness. The filter procedure returns a boolean value. If the value is true, the trap hit is effected, and the trace data supplier procedure returning the values of interest, if specified, is executed, followed by a trap-event report to the Pilot. The PCL sequence, if present, is interpreted when the trap event is received by the Pilot session; it serves as (*soft real-time*[4]) instrumentation code that is not pre-planned at compilation time.

☆ The symbol "$->$" means a dereferencing operator, "." is a field selector (of structure types), and "()" means array element denotation ("[]" in C syntax).

☆☆ This includes the possibility of a leaf procedure on a SPARC workstation.

```
-- where to break
TRAP("break point")
        -- when breaks, what to suspend
        -- (optional)
        SUSPENDS (PROGRAM [LOCK]
                        | THREAD)
        -- called on target (optional)
        WHEN("filter procedure")
        -- on target trace (optional)
        TRACES("trace data supplier procedure")
        -- interpreted in the Pilot (optional)
        "PCL sequence"
END;
```

Fig. 8 Trap command.

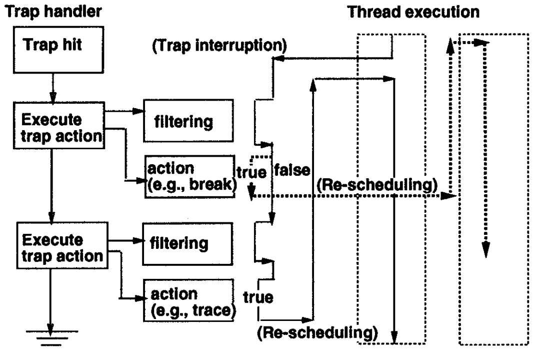


Fig. 9 Trap handler and trap actions in PXCS.

An example is shown in **Fig. 9**, where two PCL traps sharing the same breakpoint are set. The first filtering (*filter procedure*) returns false, so no action is executed. The second filtering returns true, so the corresponding *trace data supplier procedure* is called and the value is sent to the Pilot☆. Because this is tracing, the trapped thread continues its execution.

If the first filtering returns true and the action is thread-break, the trapped thread is suspended, rescheduling is done, and the second filtering is not entered. While the trapped thread is suspended☆☆, the other threads continue their normal execution☆☆☆.

4.3.4 Stepping Techniques

As in traditional debuggers, stepping by source code line is supported. The modifiers *over*, *into*, and *return* can be attached to the step command. The Pilot sends a step command containing *from-address*, *to-address*, and a thread indication. In the case of "step into" and "step over", the from- and to-address indicate the current and next lines. In the case of "step return", they indicate the address range of the procedure code. Stepping by this range of program counters can be achieved by repetitive trapping of single instructions until the trapped program counter exceeds the specified range.

☆ To minimize intrusion, the value is sent via the PXC daemon thread, which is running at a low priority.

☆☆ When a PCL resume is given for the trapped thread, the second filtering is entered, the trace is possibly done, and normal execution is immediately followed.

☆☆☆ If a trace action is followed by a break action, the trace is completed and the break takes effect immediately after the trace.

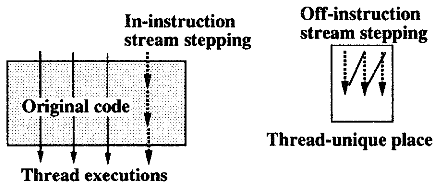

In-instruction stream stepping
Off-instruction stream stepping
Original code
Thread-unique place
Thread executions

Fig. 10 In- and off-instruction stream stepping.

The efficiency with which this is done depends on the target processors. In most CISC (complex instruction set computer) processors, efficient hardware support is available. We can use a *trace bit* in the *status register* that is set for each thread at context switch time. In most RISC (reduced instruction set computer) processors, however, single-stepping must be done, using repetitive breaks implemented by replacing the original code with a breakpoint instruction (i.e., a trap instruction). As shown in the left half of **Fig. 10**, if this technique is applied to threaded systems, the trap catches any thread that executes that code.

Therefore, we developed an "off-instruction-stream" technique in the PXCS for SPARC$^{26)}$. As shown in the right half of Fig. 10, instructions are loaded one by one from the original stepped code, placed in a thread-unique place, and executed with a trap to catch the completion. The result of the stepping, namely, adjustment of the program counter, is then recalculated as if were being done "in-stream". This technique frees the other threads from ever having to execute breakpoint instructions.

4.3.5 Handling of the Execution States of Threads and Programs

Working together, the PXCS and the CRS kernel set the proper thread and program state. The states of individual threads and the program state are independent. The program executing state is either *program loaded*, *program executing*, *program suspended*, or *program locked*.

In the program executing state, stepping and breaks for individual threads can be performed with little intrusion into the program. If a break occurs, and a thread-level break is required, rescheduling is done. If stepping is done, the thread being stepped executes many trap interrupts, slowing down its own execution. Rescheduling is still normal, so the stepping may cause another thread to receive CPU time.

In the suspended state, thread rescheduling is locked, while external interrupts are not masked and the "execution queue" is accessible. Hence, the state of each thread can be manipulated by the Pilot. Users can choose which threads will start competing for the CPU when execution is resumed or stepping is started. In the locked state, interrupts are masked and the "execution queue" is locked, so it is not possible to manipulate threads.

Working together, the CRS and PXCS support program breaks and program steps in addition to thread breaks and thread steps. The (highly intrusive) program break and stepping are very useful when complex thread synchronization is to be debugged.

The PXCS identifies the type of breakpoint when a trap is hit. It may use the explicit breakpoint type (thread or program break) to determine the next action, or it may use the execution state in which the program break occurred, for instance, if the break occurs while an interrupt is being handled or while code is being executed inside a synchronized kernel procedure. This last situation is called an implicit program break.

The same situation is observed when a thread steps into a CRS procedure. When stepping over the rescheduling procedure calls, the stepping automatically leaves the thread-stepping mode and enters program-stepping mode. After stepping over the context switch procedure, thread-stepping resumes. The program state after a step or when a breakpoint is hit is thus determined by several factors.

5. Discussion

5.1 Intrusiveness into Real-Time PUTs

The Pilot is a non-trivial superset of traditional debuggers, because it acts as an ordinary debugger, if one specifies that all execution should be suspended ("program suspend") when a trap is hit. If only some threads are stopped, new debugging situations and problems may arise; for example, global data may be transient. New capabilities can also be obtained, such as debugging for a realistic load, and testing during operations.

The intrusiveness of the Pilot into the PUT has at least two dimensions: (A) the time duration of the PUT suspension and (B) the suspension range, that is, the number of programs, threads, events, or resources that are blocked.

5.1.1 Low Intrusiveness in Terms of PUT Suspension

The filter procedure and the trace data supplier procedure reduce intrusion in terms of dimension (A). Naturally, there is a context switch inside the PUT, but the intrusion it causes is much less than waiting for communication with the debugger, that is, via a signal in UNIX or via a network, which involves process-level context switches[☆]

Soft realtime is handled, since PUT does not necessarily stop[☆☆]. Threads, when traced, are delayed only by the amount of time needed to send the response to the debugger. As shown in **Fig. 11**, however, the PXCS is designed so that the PXCS messages are asynchronized. This works nicely if the I/O does not overflow (i.e., if there are not too many PXC messages in a short interval).

The trace point will generate trace messages which will be received and delivered by the PXC thread and output via asynchronized I/O. Since the user threads run, the pool set aside for this purpose might fill rapidly, depending on how often the trace point is reached. Since the I/O operation will take some time to complete, there will be a delay before the PXC thread consumes and writes the next trace message.

The trace pool will be completely filled. At that point, the PXC will revert to *program locked mode*; that is, user threads will be stopped and thus unable to deliver any more trace messages until they are set free again.

☆ Quantitative measurements are difficult with such a new debugger. However, it is evident that the latency caused by thread switches inside the PUT is by far (by an order of three or so) faster than the latency caused by the communication between different UNIX processes, or those running different computers (debugger and PUT) which possibly involve transmission latency and human interaction.

☆☆ Note that other interactive debuggers of the "stop the world model" always stop the execution of the PUT, so real-time debugging cannot be handled in the first place.

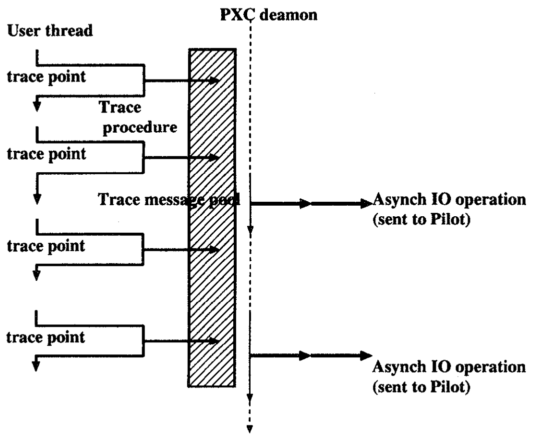


Fig. 11 Sending trace date from PXCS to the Pilot.

The PXC will then empty the pool down to THRESHOLD % filling degree. When that many trace messages have been sent, the PUT's threads are set free again.

Thus overflow never occurs, but uncritical trace generation will seriously influence the PUT's real-time behavior; that is, it will cause *high intrusion.*

If *medium intrusion* is desired, the user should make sure that the pool is never full by one or more of the following means: (a) place trace points in fewer places; (b) attach a filter to the trace point, so as to trace only what is really desired (Since the *filter procedure* is executed in PUT, this is very cheap); (c) allow the filter to inspect the pool-filling level and skip the traces when the pool is full or above a desired threshold. In this way, the *program locked mode* of PXC operation is never used, but some trace might be lost.

For a really *low intrusion mode*, the user must make sure that the PXC thread is at a lower priority than the user threads. This will need to be used in conjunction with tricks (a), (b) and (c) above. If this technique is used, the real-time behavior of PUT is little affected, since only "idle time" in PUT is used for sending trace messages.

PUT execution is perturbed only by the fact that, in UNIX, another program (e.g., the Pilot) is executing and some process switching occurs between the kernel and the user space (for write operations).

Contrast this with the delay in a conventional debugger, where (a) all threads stop; (b) user input is awaited; and then (c) PUT is run again. Even if no user input is required, PUT cannot proceed until the debugger has executed for a while. The latter is similar to a "Pilot trap body" with a body containing a "resume". The "Pilot traces option" allows speed-up relative to this conventional debugger scenario.

5.1.2 Low Intrusiveness in Terms of the Suspension Range

Low intrusiveness in terms of dimension (B) means not stopping the PUTs. If the threads in the PUT are so tightly coupled that a thread can timeout if another thread does not respond, stopping a thread or a subset of threads may not be "largely unintrusive". In such a case, a "highly intrusive scheme" may be more suitable.

On the other hand, if knowledge of the application enables the programmer to stop and manipulate one or more threads without affecting the system execution as a whole, typically, the performance may suffer slightly, or access to resources may be slightly reduced for a while. This situation is likely in telephone call processing—for example, the processing of call handling threads and administrative threads.

5.2 Location Transparency

Distributing the debugger client has two key advantages. One is that, whereas ordinary debuggers can connect to remote targets at most one by one, the Pilot makes it possible to debug several (related) target programs at the same time. The other is that it allows several groups in several places to join the same debugging session. Thus, the Pilot provides location transparency in two ways.

5.3 Applicability to Various Platforms

The Pilot itself is transparent to thread libraries, so the idea of having debugging support in a thread library is general.

Applications using thread libraries supplied by OS vendors, such as those for Solaris 2.x[12] and POSIX (pThread)[10],[11], in which we cannot embed a PXCS, require an "out-of-process" solution.

We are now studying the idea of using a "PXC agent process" that operates a PUT from outside. With this approach, the benefit of lower intrusiveness in the sense of dimension (A) is less applicable. The applicability of filters and trace data supplier procedures is yet to be determined. Still, if the OS allows stopping of one or more threads, low intrusiveness in terms of dimension (B) is expected. Furthermore, by using the "/proc" utilities[6], the agent process allows inspection of global data in any

Solaris process without stopping it. This is not possible with other debuggers.

6. Application Examples

6.1 CHILL Applications

We used the Pilot for both host and target testing of CHILL applications. As a host platform, we used a SPARC workstation with SunOS, and as our first commercial switching target, we used a G-micro[23] platform. We implemented the G-micro version of CRS in order to interface the CHILL code with the underlying operating system. The overall environment is shown in **Fig. 12**. Our goals and approaches were as follows:

- Run the CHILL code on the SPARC workstation with SunOS and apply the Pilot for host testing: This was achieved by translating the intermediate code and sharing the CHIPSY back-end compiler between the CHIPSY and NTT front-ends.
- Run the CHILL code on a G-micro switching platform:
  - Enhance the NTT version of the CHILL compiler to generate DIL: this work has been completed.
  - Develop a PXCS on the G-micro switching platform: G-micro is a CISC processor with debugging facilities similar to those of other CISC processors, such as the i486[24] and MC680x0[25]. The OS supports a TCP/IP protocol stack and signal deliveries to processes. The technique has already been proven, so the development of the PXCS is now being planned.

6.2 ACOOL Applications

PLATINA (Platform for Telecommunication and Information Network Applications)[18] is a fully distributed architecture that consists of *nodes* (*computers*), *domains* (*processes*), and *active objects* (*threads*). All the domains share memory space for quick communication that is basically asynchronous, as are CHILL signals. PLATINA is implemented on MC68030[25] and MIPS[27] computers.

PLATE (a PLATINA emulator) is a thread library on SPARC workstations with SunOS; it provides the same API as PLATINA. Applications on PLATE and PLATINA can share the same code and communicate with each other.

A new programming language called ACOOL (A Concurrent Object Oriented Language)[19] was designed for programming PLATINA/PLATE applications. The ACOOL compiler is a front-end add-on to the GCC compiler[2]☆. Our goals and approaches for applying Pilot to PLATINA and PLATE were as follows:

- Generate DIL for ACOOL: Because ACOOL is still an experimental language, "DIL for C" was used instead, in order to reduce costs.
- Apply the Pilot to PLATE applications: One solution is to implement the PXCS inside the PLATE thread library. Another is to reuse the (PXCS of) CRS for the SPARC/SunOS workstation. In view of imminent upgrade to Solaris 2.x threads, the latter approach was taken. First, the PLATE thread library was replaced by the CRS with the PXCS inside. Next, a wrapper-layer library was implemented so that the CRS is seen as a set of PLATINA system calls☆☆. Each PLATE thread (active object) is mapped to a CHILL thread, and PLATINA messages are implemented by using CHILL signals.
- Apply Pilot to PLATINA applications: The overall design, which is shown in **Fig. 13**, has been completed. First, because the Pilot and the PXCS are coupled using CHILL signals, a gateway was implemented to convert them to and from TCP/IP packets. Second, a Pilot daemon

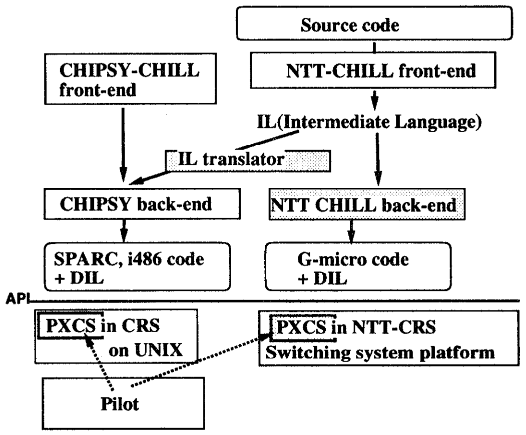


Fig. 12 Integration of CHIPSY and the NTT-CHILL support environment.

☆ This is the normal way of adding a new language to GCC. A similar approach is the implementation of uC++, which includes extensive additions of concurrency to C++[20].

☆☆ The major work was to convert the (CHILL-style) exceptions thrown by the CRS API into the (C-style) return code of the PLATINA API.

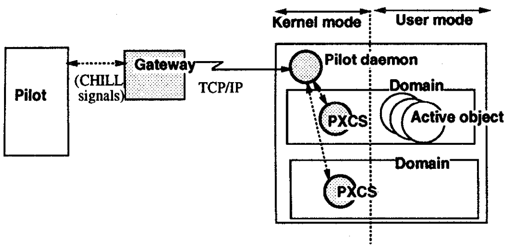


Fig. 13 Application of the Pilot to PLATINA.

thread is being implemented in PLATINA, which communicates the PXC commands via PLATINA messages. Third, the PXCS for PLATINA will be embedded in each PLATINA process (domain).

6.3 C and C++ Applications

Because DIL for C and C++ has already been defined and can be handled in Pilot, C and C++ applications are part of the support, as long as they are running on top of a thread library with PXCS☆ inside.

7. Conclusion

Debuggers that use different paradigms from those in traditional symbolic interactive debuggers are needed to monitor and debug real-time, multi-threaded, distributed applications. Specially needed are largely uninstrusive traps with on-target action and concurrent interpretation of command script sequences. We have proposed a debugger that uses a synchronous one-to-many coupling of a multi-threaded debug client and debug servers inside the tested program.

The chief advantages of this architecture are *low intrusiveness*, *programmability*, and *multi-PUT capability*, which differentiate the Pilot from many other debuggers. The debug client provides an interface at the source level for multiple source languages, multiple sessions, and cooperative debugging. Although the source code of the client is now more than a 100 K lines, there are few dependencies on the target processors. More work is needed to install debug servers in the target platforms we want to support, while the techniques for debug support used in the thread library are well proven. The code for the debug server is dependent on the target processors and operating systems but, in our experience, is in the order of 10 K lines.

One direction for the future is to provide a debug server for the thread libraries supplied by OS vendors. We have presented the concept of a debugging agent process and have developed an initial prototype. We are now studying the feasibility of supporting communicating objects, as in the CORBA[21] and TINA-C[22] architectures.

☆ To facilitate portable multi-thread programming in C++, a *wrapper-layer class library* was added on top of the CRS. This allows CHILL and C++ to exchange messages. We call this library *Cradle*[17]. For C++ wrapper-layer class libraries on top of other UNIX thread libraries, see Refs. 13) or 14).

Acknowledgments

The largely unintrusive debugger technologies we described in this paper were proposed by Kvatro Telecom AS and developed through cooperation projects with NTT labs. The contributions of the following people should be noted: Kristen Rekdal for project management, Bernt M. Johnsen for the DIL extractor for C++, Bjorn Kastnes for Pilot testing, Staale Deraas and Ulf Lunde for the DIL and DIL extractor for CHILL, Petter Moe for PXCS and Cradle, Jan Wedvik for Pilot TUI (GUI), Minoru Kubota for PLATINA-kernel-related work, and Shigeki Yamada for reading and commenting on earlier drafts of this paper.

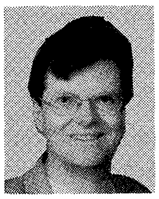
References

1) Stallmann, R.M. and Support, C.: *GDB Manual*, Free Software Foundation (1988–1995).
2) Stallmann, R.M.: Using and Porting GNU CC, Published by Free Software Foundation, for version 2.6 (July 1994).
3) Debugger Users Guide, SPARCworks^{TM} 3.1, SunSoft (1996). Original: Debugging a Program, Part No.802-3517-10, Revision A, Sun Microsystems (1996).
4) Tsai, J.J.P. and Yang, S.J.H. (Eds): *Monitoring and Debugging of Distributed Real-Time Systems*, IEEE Computer Press (1995).
5) Bach, M.J.: *The Design of the UNIX Operating System*, Prentice-Hall (1986).
6) Faulkner, R. and Gomes, R.: The Process File System and Process Model in UNIX System V, USENIX-Winter'91 (1991).
7) Bloomer, J.: *Power Programming with RPC*, O'Reilly & Associates (1992).
8) Stevens, W.R.: *UNIX Network Programming*, Prentice-Hall International (1991).
9) Strumpen, V.: Software-Based Communication Latency Hiding for Commodity Workstation Networks, *Proc. IEEE International Conference on Parallel Processing*, pp.146–153 (1996).
10) Northrup, C.J.: *Programming with UNIX Threads*, John Wiley & Sons (1996).

11) POSIX Threads Management Specification, ISO/ICE 9945-1 (1996), ANSI/IEEE Standard 1003.1 (1996).
12) Solaris 2.6 Reference Manual, Sun (1997).
13) Hughes, C. and Huges, T.: *Object-Oriented Multithreading Using C++*, Wiley Computer Publishing (1997).
14) Schmidt, D.C.: An OO Encapsulation of Lightweight OS: Concurrency Mechanisms in ACE Toolkit, Technical Report of the Department of Computer Science, Washington University, http://www.cs.wustl.edu/schmidt/.
15) CCITT High-Level Language (CHILL), ITU-T, Geneva (Recommendation Z.200) (1980, 1984, 1988, 1992, 1996).
16) Rekdal, K.: CHILL – the International Standard Language for Telecommunications Programming, Telektronikk, Information Systems (Mar. 1993).
17) CHIPSY Reference Manual, version 15, Kvatro Telecom AS, (1993), see also http://www.kvatro.no.
18) Kubota, M., Maruyama, K., Osaki, K. and Yamada, S.: Distributed Processing Platform for Switching Systems: PLATINA, *Proc. XIV International Switching Symposium*, pp.415–419 (Oct. 1992).
19) Maruyama, K.: Concurrent Object-Oriented Language COOL, *IPSJ Trans.*, Vol.34, No.5, pp.963–972 (1993).
20) Bugr, P.A., Ditchfield, G., Stroobosscher, R.A. and Younger, B.M.: uC++: Concurrency in the Object-Oriented Language C++, *Software Practice and Experience*, Vol.22, No.2, pp.137–172 (1992).
21) Object Management Group: The Common Object Request Broker. Architecture and Specification, Revision 2.0 (July 1996).
22) TINA-C: Overall Concepts and Principles of TINA, Document No.TB_MDC.018_2.0_94 (Dec. 1994).
23) G-micro Hitachi 32-bit Microprocessor H32/500 User's Manual, Hitachi Ltd.
24) 386^{TM} DX Microprocessor Programmer's Reference Manual (2nd edition), Intel Corporation (1989).
25) MC68030 Enhanced 32-bit Microprocessor User's Manual, Second Edition, Motorola Inc.
26) The SPARC Architecture Manual, version 8, SPARC International, Prentice-Hall (1992).
27) Kane, G.: MIPS RISC ARCHITECTURE, Prentice-Hall.

(Received June 26, 1998)
(Accepted July 1, 1999)

Norio Sato graduated from Faculty of Mathematical Engineering and Information Physics, the University of Tokyo in 1972. He joined NTT Telecommunication Laboratories, where he engaged in the development of the telephone switching system programs, and led the the CHILL language processor projects. He has a long experience in the standardization activities of the programming language CHILL at CCITT (now ITU). He was a Senior Research Engineer, Supervisor, Distributed Network Systems Laboratory, NTT Optical Network Systems Laboratories. He received Ph.D. in computer science from Ritsumeikan University in 1999.



Dag H. Wanvik has a M.Sc. from the Norwegian Institute of Technology (NTH, now NTNU) in Computer Science, 1979 and is now Senior Project Engineer with Kvatro Telecom AS. After research work on compilers and development tools at SINTEF, Trondheim, he joined Kvatro Telecom AS in 1987. He has a wide experience in systems programming, including real-time applications and operating systems. Among other hobbies, he is a Linux enthusiast.

Harald Botnevik has a M.Sc. from Norwegian Institute of Technology (NTH, now NTNU), in Computer Science, 1975 and is now Development Manager of Kvatro Telecom AS. Kvatro Telecom AS is a Norwegian software company developing and marketing CHIPSY, an advanced toolset for CHILL program development. Mr. Botnevik has been working with compilers and toolchains for CHILL since 1977 and has extensive experience also on quality assurance, project management and languages and tools for telecom software development. He is a member of ACM.

Trond Børsting has a M.Sc. from Norwegian Institute of Technology (NTH, now NTNU), in Computer Science, 1975 and is now Senior Project Engineer in Kvatro Telecom AS. Kvatro Telecom AS is a Norwegian software company developing and marketing CHIPSY, an advanced toolset for CHILL program development. Mr. Borsting has been working with real-time operating systems since 1975 and is the main responsible for CRS, the CHIPSY Real-time Operating System, including distributed services and thread debugging services.

Jon E. Strømme has a M.Sc. from the Norwegian Institute of Technology (NTH, now NTNU) in Computer Science, 1981 and is now Senior Project Engineer with Kvatro Telecom AS. After research work on compilers and data networks at SINTEF, Trondheim, he joined Kvatro Telecom AS in 1987. He has a wide experience in systems programming, including real-time applications, operating systems and debuggers.