

Transparent Debugging of Dynamically Instrumented Programs

Naveen Kumar
University of Pittsburgh
Pittsburgh, PA, U.S.A.
naveen@cs.pitt.edu

Ramesh Peri
Intel Corporation
Hillsboro, OR, U.S.A.
ramesh.v.peri@intel.com

Abstract

Dynamic instrumentation systems, used for program analysis, bug isolation, software security and simulations, are becoming increasingly popular. There exists a need to debug dynamically instrumented programs while keeping the presence of dynamic instrumentation system hidden from debug users. Existing debuggers use debug information in program binaries that have been generated by a compiler at static compile time, to provide their debug support. Since dynamic instrumentation systems generate program code at run-time, existing debuggers are not able to provide the same kind of debug support. The most comprehensive existing debug techniques that satisfy this need, used by Tdb, require modification of existing debuggers. This paper provides an OS level approach that silently intercepts the communication between a debugger and the OS and uses a set of debug specifications to provide Tdb's transparent debugging. As a result, any existing debugger can be used to debug dynamically instrumented programs. The proposed techniques have been implemented on x86/Linux platform for the dynamic instrumentation system Pin.

1. Introduction

Dynamic instrumentation is a technique for program analysis, software security and simulation, in which code is inserted and removed from programs during execution. Dynamic instrumentation systems have become quite popular in the last decade. Examples of such systems are Dyninst [2], Kerninst [7], Dynamo Rio [1], FIST [3] and Pin [5]. These systems fetch code from applications at run-time, on-demand, and generate instrumented code. The generated code is usually cached in a software managed *code cache* where it executes from. This caching mechanism helps to avoid the overhead of generating the same code again.

While the popularity of dynamic instrumentation has been increasing, dynamic instrumentation systems need to provide appropriate debug support for them to become widely accepted. In this paper, the debug support refers to the debuggability of the dynamically instrumented application and not the instrumented code. An argument against the need to debug dynamically instrumented programs is that a program may be debugged when it does not get instrumented and then the bug-free program be executed with dynamic instrumentation. This approach, however, may not always be feasible.

One motivation to debug instrumented programs is because dynamic instrumentation is often used by bug-isolation systems, such as Valgrind [6]. In order to reduce an extra software development cycle, it is important that bug-isolation uses of dynamic instrumentation be possible from within a traditional debugger. Another motivation for such debugging techniques is in the case of dynamic instrumentation systems that are present at

the operating system level. In this case, every program on the system gets instrumented before it can be run. Therefore, a developer is not able to debug programs unless appropriate debug techniques are available.

The difficulty with debugging arises due to dynamic code generation, which renders the static debug information available with a program inconsistent with the executing program. To visualize this problem, consider a breakpoint to be inserted at a program location. Since all parts of the program execute from the code cache, the breakpoint must be placed in the code cache now [4]. A debugger that is not aware of dynamic instrumentation may not be able to realize this. The presence of additional code (instrumentation code) and modifications to program code during run-time code generation create problems with debug commands that single-step execution and report values of variables.

Transparent debugging of dynamically translated (and instrumented) programs has been proposed by Tdb [4]. Tdb generates debug information when code is generated and instrumented in the dynamic instrumentation system and modifies a debugger to use this debug information. One drawback of Tdb's current approach is that it requires modifying existing debuggers. Currently Tdb uses a modified version of Gdb 5.3. For dynamic instrumentation systems and the debug techniques therein to be popular and widely used, it is important that users be able to use their own, unmodified, debuggers.

Debuggers typically need operating system support to debug a program running in a different address space than the debugger. This is the most common debug model used by programmers today and is the model we are assuming in this paper. This paper proposes a set of debug specifications at the operating system level in order to provide Tdb's debug support. The techniques proposed herein do not require modifying the underlying OS or even recompiling or restarting the OS. We intercept the communication between the debugger and the OS by injecting a dynamically shared library into the debugger before it is started. The library contains the code that performs alternative debug actions on the dynamically instrumented code instead of the original program, thereby completely hiding dynamic instrumentation from the debugger, and hence the user. The debug specifications are a set of algorithms that define which OS services need to be intercepted and what actions should be taken instead. The actions include using the native OS services as well as the debug engine of Tdb. The debug specifications are the crux of this paper.

This paper is organized as follow: Section 2 describes Tdb, which is important background information needed to understand this work; Section 3 describes each of the debug specifications proposed in this work; Section 4 describes some of the implementation challenges that we encountered and our solutions

to them. Note that this work was done using a different dynamic instrumentation system, Pin, and platform, x86/Linux, than Tdb that used FIST-Strata and Sparc/Solaris. Section 5 discusses some related work. Finally, section 6 concludes this paper.

2. Transparent Debugging using Tdb

Tdb is a modified version of the GNU debugger gdb, that allows transparent debugging of dynamically translated and instrumented programs. Transparency refers to hiding the modifications made by the dynamic translation system and the presence of dynamic translator itself.

Figure 1 shows the structure of Tdb. The debug engine is the critical functionality introduced by Tdb and it sits between the native debugger and the dynamic translation system. The debug engine consists of two repositories: the mapping repository for storing debug information and the breakpoint repository for storing information about active breakpoints. The debug information consists of debug mappings that relate "untranslated" code with the translated code. The engine also has three components: the mapping generator generates debug mappings based upon translation information received from the dynamic translator, the mapper uses the mappings generated by the mapping generator and gets invoked by the native debugger, and the breakpoint manager handles insertion, deletion and lookup of breakpoints.

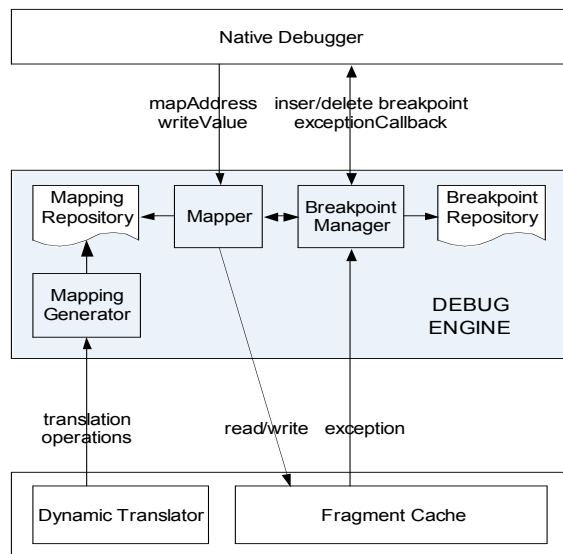


Figure 1: Debug Architecture of TDB [4]

The debug engine is also invoked by the native debugger, in this case gdb, whenever the debugger needs to perform an operation on the program being debugged (read/write data or instructions, insert/remove breakpoints). The debug engine in return performs similar operations on the translated program in the code cache.

In order to perform its debug actions, the debug engine generates and uses several types of mappings. The U-T mapping is generated between original program location of an instruction and its translated location, and is bi-directional in nature. U-T mappings are not generated for instrumentation code or any additional code generated by the dynamic translation system that cannot be related to "untranslated" code. Other mappings include

the T-T mappings, between translated locations, and T-U mappings, between translated and program locations. The T-T and T-U mappings are uni-directional and are used to roll-ahead execution of code that needs to be kept hidden from debug users.

In the existing implementation of Tdb, the mapping generator of the debug engine is in the address space of the dynamic translator, while the mapper and breakpoint managers are in the address space of the native debugger. The repositories are arranged as shared memory accessible to both the native debugger and the dynamic translator. In the proposed OS level implementation of Tdb, the work performed by the mapper and the breakpoint manager are inherently handled by the interceptor, a new component introduced in this paper and described in the following section. Therefore, the debug engine only consists of the mapping generator and the repositories.

3. Debug Specifications

Figure 2 shows the structure of our proposed debugger. Note the differences between this structure and the one proposed in Tdb. There are two new components: the OS module and the interceptor. The OS module is an operating system module, such as *ptrace*, that is used by common debuggers to access and control the program being debugged. This component is present in Tdb as well but not important since it is not modified. The interceptor is a component developed in this work, which intercepts the communication between the native debugger and the OS module. The interceptor uses a set of specifications that describe what communication is to be intercepted and what actions are to be taken instead. The debug specifications are shown in the Appendix.

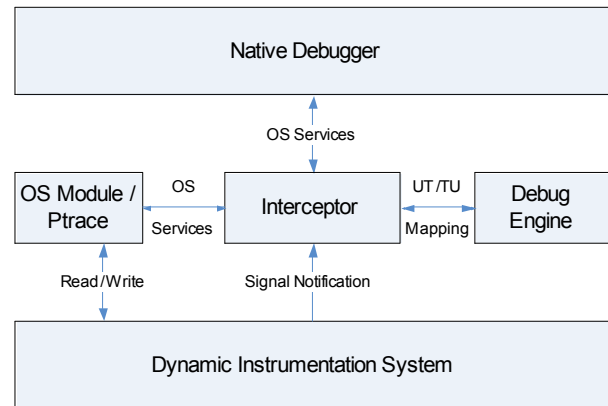


Figure 2: Proposed Debug Architecture

The insight for the proposed solution comes from the understanding that a debugger builds its features on top of services provided by the underlying operating system. There are five main services provided by the OS that are needed to access and control a child process (for debugging):

- getting a signal delivered on behalf of the application,
- reading from an application's address space/registers,
- writing to the application's address space/registers,
- requesting execution of exactly one instruction of the application (single-step),
- and continuing the execution of the stopped application.

The debug specifications describe how each of these services need to be modified in order to debug dynamically instrumented programs. The debug specifications are shown in the Appendix and described below.

1. Signal Handler

This specification describes what is done when a notification about a program being stopped is intercepted. On UNIX systems, this notification consists of information about the signal received by the program and is sent to the debugger. When a breakpoint/watchpoint hits in a program or the program stops after a single-step operation, a signal, SIGTRAP, is delivered to the debugger on a UNIX system. Therefore, this is the most important signal from the debugger's point of view and hence we focus on this signal. Note that the specification would be similar on non-UNIX systems, even if the notification mechanism (signal) might be different.

When a SIGTRAP is delivered to the program, the specification determines whether the signal occurred due to a breakpoint or not. This can be determined by inspecting the last instruction executed (PC-1) and seeing if PC-1 has a breakpoint trap instruction. In case the breakpoint was a hardware breakpoint (set in a debug register), the appropriate register can be inspected to determine whether execution stopped due to a breakpoint. If execution was not stopped due to a breakpoint, the interceptor decides whether to notify the debugger right away or to roll-ahead execution to a favorable point before notifying the debugger. The idea is to stop only at locations that can be related back to the original program (using the debug mappings).

Since any signal received by the program is instead delivered to the debugger, the interceptor must intercept all signals and decide whether the current location is a favorable stopping point. Although this is not shown in the debug specification for signal handler, the interceptor intercepts every signal (that can be intercepted) and decides whether to roll-ahead execution or stop right away. The difference between other signals and SIGTRAP is that a breakpoint/watchpoint can raise a SIGTRAP, in which case execution is always stopped immediately.

2. Read Register

A debugger reads register values of an executing program to inspect the current stopped location and values of variables in the program. The register containing the program counter (PC) value will hereafter be referred to as EIP. The debug specification for reading the EIP ensures that the debugger sees an untranslated program address, whenever possible. For this, the interceptor intercepts the debugger's command that reads EIP. The interceptor then reads the actual value in EIP, the PC, and tries to relate the PC with untranslated program addresses using the TU mappings of the debug engine. Note that TU mappings relate a translated location in the code cache with an untranslated program location, when possible. The interceptor then sets the value of EIP to be the mapped "untranslated" location, for use by the native debugger.

When stopped due to a breakpoint, the value in EIP is the current stopped location, which is the starting address of the breakpoint location offset by the size of the breakpoint. The value in EIP may not have a TU mapping, although the breakpoint location does. The debug specification, therefore, relates the location *PC - size of breakpoint* with the untranslated code and then offsets the

untranslated location by the size of the breakpoint (see Read Register in Appendix). This address is then returned as the value read by the debug command.

Execution can stop even if there is no breakpoint hit. This can happen when single-stepping or when the program receives a signal. In these circumstances, the execution is always rolled-ahead by the signal handler specification (see above) to stop the execution at a location which can be related back to untranslated code. Therefore, the read register command always results in a valid untranslated location.

3. Write Register

A debugger can modify contents of a register due to an explicit request by user to do so. A debugger, for instance, can modify the value of PC when a user wants to start execution from a different location than currently stopped at. A debugger also needs to reset the value of PC when continuing from a breakpoint, adjusting by the size of the breakpoint instruction. The interceptor intercepts commands from the debugger that modify EIP so that control does not exit from the code cache.

The proposed specification handles four different cases as observed by the interceptor:

(a) the debugger could be writing PC to EIP; the interceptor determines this is true if the value being written can be mapped to the value in EIP; the interceptor then sets the value being written to be the value in EIP;

(b) the debugger could be adjusting the PC to restart execution after hitting the breakpoint; the debug engine determines this by mapping the value being written with the *"actual" PC - size of breakpoint*; if so, a new value, *actual PC - (size of breakpoint)*, is written into EIP;

(c) the debugger does not write any other value to EIP, unless it is user-directed; the interceptor does its best to keep execution within the code cache by trying to map the value being written to code cache locations and choosing an arbitrary location from one of those locations;

(d) if the interceptor cannot map the value being written to code cache locations, the original value is written to PC. Note that the option (d) of the debugger is likely going to make the control exit from the code cache. The interceptor assumes that user-directed jumps to random program locations are not safe and can provide an appropriate warning to the user.

4. Write Memory

A debugger needs to write into a program's address space for one of three reasons. The interceptor handles these cases using the Write Memory debug specification shown in Appendix. The three cases and the corresponding debug specifications are discussed below.

(a) Insertion of a breakpoint involves writing a breakpoint trap instruction into a program. When a debugger tries to write a breakpoint, the interceptor looks up the U-T mapping for the breakpoint location and inserts a breakpoint at each of the corresponding translated locations in the code cache. The

interceptor also updates the breakpoint information in the breakpoint repository of the debug engine.

(b) Removal of a breakpoint leads to insertion of the original instruction that existed before a breakpoint was inserted in the breakpoint location. The interceptor can determine whether a breakpoint is being removed, if the location and data being written exist in the breakpoint repository. If so, it removes all corresponding breakpoints from the code cache.

(c) A debug user can instruct modification of instructions or data in a program's address space. The native debugger does not take any action in these cases and lets the write happen normally. The writes to an application's instructions, directed by user, may not reflect accurately in code that has already been translated in the code cache. The interceptor does not help alleviate this problem. Therefore, user-directed changes to PC are unsafe and we do not recommend them. This issue exists in Tdb as well.

Note that systems supporting hardware breakpoints insert and remove breakpoints by writing values into appropriate registers. In that case, the Write Register specification for those registers would provide the functionalities shown in the first two cases of Write Memory above.

5. Continue

A continue command is issued by the native debugger to request execution of a stopped program. The interceptor does not need to do anything special for this.

6. Single-step

A single-step command allows executing a single instruction in the program. The interceptor does not intercept this command. A single-step issued by the debug user can still turn into a set of single-steps due to execution roll-ahead performed by the interceptor, as shown in the signal handler debug specification.

7. Other commands

Other commands such as Read Memory, or Kill, Attach etc., do not lead to any new functionality and do not have any specifications.

To understand how the proposed system works using these specifications, consider a scenario when a program is stopped at a location in the code cache and a single-step command is issued by the debug user. The single-step command (`stepi` in `gdb`) allows execution of exactly one instruction in a stopped program. In the proposed debugger, the interceptor gets invoked instead of the OS service, which then requests the OS to single-step execution of the program (see the Single-step specification above). After the single-step is complete, the execution stops and the interceptor is notified instead of the native debugger (see the Signal Handler specification above). The interceptor uses the debug engine and its mappings to determine whether it needs to roll-ahead execution. Execution would be rolled-ahead, for instance, when current PC is instrumentation code that needs to be hidden from the native debugger. The interceptor issues another single-step and waits for another notification.

4. Implementation

We used the dynamic instrumentation Pin and the platform x86/Linux for our implementation. We verified our implementation using the GNU debugger `gdb`.

The GNU/Linux operating system provides the `ptrace` utility for debuggers. This utility controls and monitors a process and is similar to those provided by other UNIX flavors. The debugger can `wait` on a process until a signal is delivered to the process or the process exits. The reason why the process stopped or exited is passed via arguments to `wait` which can be analyzed by a debugger. The GNU/Linux operating system also allows a shared library to be preloaded into a program by means of the environment variable `LD_PRELOAD`.

We implemented our interceptor as a shared library, which gets injected into a debugger during startup using `LD_PRELOAD`. The interceptor then overrides the existing `ptrace` utility and gets invoked whenever a debugger calls `ptrace`. The debug engine consists of repositories containing active breakpoints and debug mappings. Pin is modified to provide translation information in the form of tuples containing untranslated and code cache locations for each translated instruction. The interceptor also overrides the `wait` system call so that it gets invoked before the native debugger and can take appropriate actions as per the Signal Handler debug specification.

There were a number of interesting issues that we encountered during the implementation of this system. We briefly describe these below.

(a) Handling breakpoint insertion in untranslated code.

Pin generates code on-demand and leads to code duplication in traces [5]. The debug specifications in the Appendix assume that code translations exist for the breakpoint location when a breakpoint is being inserted. This may not be the case, since Pin generates code on-demand and the fresh translations for the same code can occur (often leading to code duplication). One plausible solution is for the interceptor to insert breakpoints in native code and force retranslation of the code every time a breakpoint is inserted or removed. This approach is used in debugging programs translated by Dynamo-RIO [1]. However, this approach has its own drawbacks. Issues arise when a breakpoint is to be inserted in the same fragment of code (or trace) which is executing. In addition, Pin performs register re-allocation. The allocation on the traces containing the breakpoint and the real instruction may differ leading to possible debug inconsistencies.

Another solution is the one used in Tdb [4], when a breakpoint is inserted in native program location in addition to available translated locations. When the dynamic translator detects a breakpoint trap instruction during instruction decode phase, it consults the breakpoint repository and translates the real instruction. After that, the dynamic translator inserts the breakpoint. This solution involves some modifications to the dynamic translation system.

Our goal in this work was to modify Pin as little as possible. When the native debugger tries to set a breakpoint, we do not insert it in the untranslated code. When new translations occur and translation information is sent to the debug engine, the debug engine consults its breakpoint repository to determine if

breakpoint needs to be placed in the freshly translated location. When a breakpoint is hit, in order to provide the same state to the debugger as one without dynamic instrumentation, the interceptor pre-emptively inserts the breakpoint at the PC that would be reported to the native debugger. This approach does have performance implications. However, most of the execution of an instrumented program remains within the code cache, and the check for breakpoint to be inserted is only made during new code generation. Therefore, the performance impact is insignificant.

(b) Intel x86 architecture has a flag register *Eflags* that contains a *Trap Flag*, TF, bit. When this bit is set, it results in an exception being raised after executing the next instruction. This is the basis of the single-step command of debuggers on x86. The code generated by Pin often needs to save and restore application context, including the *Eflags* register. When the interceptor is rolling-ahead execution, one of such saves may be encountered. If single-stepping (and roll-ahead) is stopped before the register has been restored, the program would get an unwanted exception. This issue can be handled by appropriately setting the TF bit in the saved *Eflags* location.

(c) When a debugger writes some data into a variable, likely being directed by the debug user, the data could have the same encoding as a breakpoint instruction. On our platform, the interceptor does not know whether it is the data segment being written to or the text segment. Further, the variable length instruction set makes it difficult to determine if a breakpoint is indeed being inserted, since the breakpoint instruction is only 1 byte and the Write Memory calls to *ptrace* are word aligned and 4 bytes in size.

Our solution is to consider all potential breakpoints as actual breakpoints. Since data addresses would not have appropriate debug mappings, we would never insert incorrect breakpoints in the code cache. We do have a memory penalty in the breakpoint repository if enough data writes are perceived incorrectly as breakpoint insertions.

5. Related Work

The closest work to this paper is Tdb. Section 2 has described Tdb in detail. Other dynamic instrumentation systems have had some support for debugging. Dynamo RIO allows debugging using existing debuggers, by retranslating at breakpoint insertions and deletions and providing custom signal handlers. The debug support has been described as not robust [1] and more work using debug mappings has been suggested.

Valgrind [6] provides support to attach the instrumented program to a debugger at a given point for inspection of program state. However, it is not possible to continue execution of a stopped program from within a debugger.

Parasoft Chaperon [8] is a memory debugger, similar in structure and functionality to Valgrind. Chaperon allows limited debugging of dynamically instrumented programs. In particular, a user cannot continue execution of a stopped program with the *continue* command.

6. Conclusions

This paper provides an OS level approach to transparently debug dynamically instrumented programs. The work builds upon

existing techniques of Tdb. We provide debug specifications and methodologies to intercept communication between an existing debugger and a program being debugged. The intercepted communication is then modified to use Tdb's debug engine and hide the presence and actions of dynamic instrumentation system. This work demonstrates that it is possible to transparently debug dynamically instrumented programs using existing, unmodified debuggers.

Acknowledgements

We would like to acknowledge the comments and feedback from Prof. Bruce Childers and Prof. Mary Lou Soffa, at the University of Pittsburgh. We would also like to thank Srinivas Doddapaneni and Jasper Kamperman, at Intel Corporation, for their motivation that led to this work. Finally, we would like to thank the Pin team for their immense help in understanding the dynamic instrumentation system and the anonymous reviewers for their helpful comments.

References

- [1] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.D. Thesis Massachusetts Institute of Technology, 2004.
- [2] J. Hollingsworth, B. Miller and J. Cargille. *Dynamic Program Instrumentation for Scalable Performance Tools*. SHPCC, May 1994.
- [3] N. Kumar, J. Misurda, B. Childers, and M. L. Soffa, *FIST: A framework for instrumentation in software dynamic translators*, Department of Computer Science, Technical Report TR-03-106, 2003, <http://www.cs.pitt.edu/~copa/FIST.pdf>.
- [4] N. Kumar, B. Childers, and M. L. Soffa. *Source-level debugging of dynamically translated programs*. In Proc. of the Automated and Analysis-Driven Debugging (AADEBUG), 2005.
- [5] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. *Pin: building customized program analysis tools with dynamic instrumentation*. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, 2005.
- [6] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. Ph.D. Thesis, University of Cambridge, 2005.
- [7] A. Tamches and B. Miller. *Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels*. Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI), 1999.
- [8] *Automating C/C++ Application Testing with Parasoft Insure++*, White paper, Parasoft Corporation, <http://www.parasoft.com>.

Appendix: Debug Specifications

<p>Invariant: DSignalHandler: //signal delivered via wait</p> <pre> if signal = SIGTRAP then ptrace (READ_REG, eip, PC) ptrace (READ_MEM, PC-1, ins) if ins = BREAKPOINT then StoppedAtBreakpoint ← TRUE original_signal_handler () else // not stopped at breakpoint; in single-step StoppedAtBreakpoint ← FALSE if ∃ TU (PC) then original_signal_handler () else // keep on single-stepping ptrace (SINGLE_STEP) else original_signal_handler () </pre>	<p>ptraceDE (CONTINUE) ptrace (CONTINUE)</p> <hr/> <p>ptraceDE (SINGLE_STEP) ptrace (SINGLE_STEP)</p> <hr/> <p>ptraceDE (READ_MEM, addr, data) ptrace (READ_MEM, addr, data)</p> <hr/> <p>ptraceDE (READ_REG, eip, PC) ptrace (READ_REG, eip, PC) if (StoppedAtBreakpoint = TRUE) then if ∃ (x ← TU(PC - BP_SIZE)) then // PC - BP_SIZE mapped back to untranslated code PC ← x + BP_SIZE else if ∃ (x ← TU (PC)) then PC ← x</p>
<p>ptraceDE (WRITE_REG, eip, data) ptrace (READ_REG, eip, PC) if ∃ x ∈ UT (data) : x = PC then // writing the value of PC back to PC ptrace (WRITE_REG, PC, x) else if ∃ x ∈ UT (data) : x = PC - BP_SIZE ∧ StoppedAtBreakpoint = TRUE then // rewinding PC after hitting breakpoint by BP_SIZE ptrace (WRITE_REG, eip, x) else if x ∈ UT (data) : // best-effort mapping: set PC to any x in UT(data) ptrace (WRITE_REG, eip, x) else // cannot map. Set PC to what debugger wanted ptrace (WRITE_REG, eip, data)</p>	<p>ptraceDE (WRITE_MEM, addr, data) if data = BREAKPOINT then ∀x ∈ UT (addr): // set bp at all translated locations ptrace(READ_MEM, x, m) // if there is already a bp, union handles it BR ← BR ∪ (x, m) ptrace (WRITE_MEM, x, data) ptrace(READ_MEM, addr, m) BR ← BR ∪ (addr, m) else if (addr, data) ∈ BR then ∀y ∈ UT (addr) : // remove bp from all translated locations if ∃ (q, n) ∈ BR : q = y then // remove only if there exist a bp ptrace (WRITE_MEM, y, n) BR ← BR - (x, m) ptrace (WRITE_MEM, addr, data)</p>